June 2004

Microsoft®

SQL Server 2000 Driver for JDBC

User's Guide and Reference

Table of Contents

Preface

The SQL Server 2000 Driver for JDBC is a Type 4 driver that is compliant with the JDBC specification.

Using This Book

This book assumes that you are familiar with your operating system and its commands, the definition of directories, and accessing a database through an end-user application.

This book contains the following information:

- Chapter 1 "Quick Start" on page 11 provides information about connecting with your SQL Server 2000 Driver for JDBC.
- Chapter 2 "Using the SQL Server 2000 Driver for JDBC" on page 15 provides information about using JDBC applications with the SQL Server 2000 Driver for JDBC.
- Chapter 3 "SQL Server 2000 Driver for JDBC" on page 25 provides detailed information specific to the driver.
- Appendix A "JDBC Support" on page 33 provides information about developing JDBC applications for SQL Server 2000 Driver for JDBC environments.
- Appendix B "SQL Server 2000 Driver for JDBC GetTypeinfo" on page 65 provides results returned from the method DataBaseMetaData.getTypeinfo for the SQL Server 2000 Driver for JDBC.

- Appendix C "Designing JDBC Applications for Performance Optimization" on page 77 provides information about enhancing the performance of your application by optimizing its code.
- Appendix D "SQL Escape Sequences for JDBC" on page 93 describes the scalar functions supported for the SQL Server 2000 Driver for JDBC. Your data store may not support all of these functions.

Typographical Conventions

This book uses the following typographical conventions:

Convention	Explanation
italics	Introduces new terms that you may not be familiar with, and is used occasionally for emphasis.
bold	Emphasizes important information. Also indicates button, menu, and icon names on which you can act. For example, click **Next**.
UPPERCASE	Indicates the name of a file. For operating environments that use case-sensitive file names, the correct capitalization is used in information specific to those environments. Also indicates keys or key combinations that you can use. For example, press the ENTER key.
`monospace`	Indicates syntax examples, values that you specify, or results that you receive.
`monospaced italics`	Indicates names that are placeholders for values you specify; for example, *`filename`*.
forward slash /	Separates menus and their associated commands. For example, Select File / Copy means to select Copy from the File menu.
vertical rule |	Indicates an OR separator to delineate items.

Convention	Explanation
brackets [ ]	Indicates optional items. For example, in the following statement: SELECT [DISTINCT], DISTINCT is an optional keyword.
braces { }	Indicates that you must select one item. For example, {yes | no} means you must specify either yes or no.
ellipsis . . .	Indicates that the immediately preceding item can be repeated any number of times in succession. An ellipsis following a closing bracket indicates that all information in that unit can be repeated.

About SQL Server 2000 Driver for JDBC Documentation

The SQL Server 2000 Driver for JDBC library consists of the following books:

- *SQL Server 2000 Driver for JDBC Installation Guide* details requirements and procedures for installing the SQL Server 2000 Driver for JDBC.
- *SQL Server 2000 Driver for JDBC User's Guide and Reference* provides both general and driver-specific information about using the SQL Server 2000 Driver for JDBC, as well as about enhancing driver performance.

The SQL Server 2000 Driver for JDBC online documentation is provided in PDF format, which allows you to view it online or print it. You can view the online documentation using Adobe Acrobat Reader, 3.*x* or greater. Using Acrobat Reader 3.*x* or greater with Search allows you to take advantage of full-text search across both the SQL Server 2000 Driver for JDBC online books.

HTML-based online help is placed on your system during normal installation of SQL Server 2000 Driver for JDBC. It is installed in a directory named Help beneath the product installation directory.

To access help from a Windows environment, you must have Internet Explorer 5.*x* or higher, or Netscape 4.*x* or higher installed. Open the program group for SQL Server 2000 Driver for JDBC and click the help icon.

To access help from a UNIX environment, you must have Netscape 4.*x* or higher installed. At a command prompt, enter:

```
netscape_exe install_dir/help/wwhelp/js/html/frames.htm
```

where *netscape_exe* is the name of the Netscape executable and *install_dir* is the path to the directory in which the SQL Server 2000 Driver for JDBC is installed.

1 Quick Start

The following basic information enables you to connect with your SQL Server 2000 Driver for JDBC immediately after installation. To take full advantage of the features of the driver, however, you should read Chapter 2 “Using the SQL Server 2000 Driver for JDBC” and Chapter 3 “SQL Server 2000 Driver for JDBC” for details.

NOTE: For installation instructions for SQL Server 2000 Driver for JDBC, see the *SQL Server 2000 Driver for JDBC Installation Guide*.

Connecting to a Database

Once the driver is installed, you can connect from your application to your database in two ways: with a connection URL through the JDBC driver manager, or with a JNDI data source. This quick start explains how to establish your database connection using a connection URL. See Chapter 2 “Using the SQL Server 2000 Driver for JDBC” for details on using data sources.

You can connect through the JDBC driver manager with the method DriverManager.getConnection. This method uses a string containing a URL. Use the following steps to load the driver from your JDBC application.

1. Setting the Classpath

The SQL Server 2000 Driver for JDBC needs to be defined in your CLASSPATH variable. The CLASSPATH is the search string that your Java Virtual Machine (JVM) uses to locate the JDBC drivers on your computer. If the drivers are not on your CLASSPATH, you receive the error "class not found" when trying to load the driver. Set your system CLASSPATH to include the following entries, where *install_dir* is the path to your SQL Server 2000 Driver for JDBC installation directory:

```
install_dir/lib/msbase.jar
install_dir/lib/msutil.jar
install_dir/lib/mssqlserver.jar
```

Windows Example

```
CLASSPATH=.;c:\Microsoft SQL Server 2000 Driver for
JDBC\lib\msbase.jar;c:\Microsoft SQL Server 2000 Driver for
JDBC\lib\msutil.jar;c:\Microsoft SQL Server 2000 Driver for JDBC
\lib\mssqlserver.jar
```

UNIX Example

```
CLASSPATH=.;/home/user1/mssqlserver2000jdbc/lib/msbase.jar;/home/user1/
mssqlserver2000jdbc/lib/msutil.jar;/home/user1/mssqlserver2000jdbc/lib/
mssqlserver.jar
```

2. Registering the Driver

Registering the driver tells the JDBC driver manager which driver to load. When loading a driver using class.forName(), you must specify the name of the driver:

com.microsoft.jdbc.sqlserver.SQLServerDriver

For example:

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
```

3. Passing the Connection URL

After registering the driver, you must pass your database connection information in the form of a connection URL. The following is a template URL for the SQL Server 2000 Driver for JDBC. Substitute the values specific to your database. (For instructions on connecting to named instances, see "Connecting to Named Instances" on page 28 in the SQL Server 2000 Driver for JDBC chapter.)

```
jdbc:microsoft:sqlserver://server_name:1433
```

For example, to specify a connection URL that includes the user ID "username" and the password "secret":

```
Connection conn = DriverManager.getConnection
  ("jdbc:microsoft:sqlserver://server1:1433","username","secret");
```

NOTES:

The *server_name* is an IP address or a host name, assuming that your network resolves host names to IP addresses. You can test this by using the ping command to access the host name and verifying that you receive a reply with the correct IP address.

The numeric value after the server name is the port number on which the database is listening. The values listed here are sample defaults. You should determine the port number that your database is using and substitute that value.

You can find the complete list of Connection URL parameters in "Connection String Properties" on page 26 of this book.

2 Using the SQL Server 2000 Driver for JDBC

The Type 4 SQL Server 2000 Driver for JDBC provides JDBC access through any Java-enabled applet, application, or application server. It delivers high-performance point-to-point and *n*-tier access to Microsoft SQL Server across the Internet and intranets. The driver is optimized for the Java environment, allowing you to incorporate Java technology and extend the functionality and performance of your existing system.

About the SQL Server 2000 Driver for JDBC

The SQL Server 2000 Driver for JDBC is compliant with the JDBC 2.0 specification. The driver also supports a subset of the JDBC 2.0 Optional Package, which provides the following functionality:

- Java Naming Directory Interface (JNDI) for naming data sources
- Connection Pooling

Connecting Through the JDBC Driver Manager

One way of connecting to a database is through the JDBC driver manager using the method DriverManager.getConnection. This method uses a string containing a URL. The following is an example of using the JDBC driver manager to connect to Microsoft SQL Server 2000 while passing the user name and password:

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
Connection conn = DriverManager.getConnection
  ("jdbc:microsoft:sqlserver://server1:1433;User=test;Password=secret");
```

URL Examples

The complete connection URL format used with the driver manager is:

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

where:

hostname	is the TCP/IP address or TCP/IP host name of the server to which you are connecting. NOTE: Untrusted applets cannot open a socket to a machine other than the originating host.
port	is the number of the TCP/IP port.
property=value	specifies connection properties. See "Connection String Properties" on page 26 for a list of connection properties and their values.

The following example shows a typical connection URL:

```
jdbc:microsoft:sqlserver://server1:1433;user=test;password=secret
```

Connecting Through Data Sources

A SQL Server 2000 Driver for JDBC data source is a DataSource object that provides the connection information needed to connect to an underlying database. The main advantage of using a data source is that it works with the Java Naming Directory Interface (JNDI) naming service, and it is created and managed outside of the applications that use it. Because the connection information is outside of applications, the time it takes to reconfigure your infrastructure when a change is made is minimal. For example, if the underlying database is moved to another server and uses another port number, the administrator must change only the relevant properties of the SQL Server 2000 Driver for JDBC data source (a DataSource object). The applications using the underlying database do not need to change because they only refer to the logical name of the SQL Server 2000 Driver for JDBC data source.

How SQL Server 2000 Driver for JDBC Data Sources Are Implemented

Microsoft ships a data source class for the SQL Server 2000 Driver for JDBC. See Chapter 3 “SQL Server 2000 Driver for JDBC” on page 25 for the name of the class.

The SQL Server 2000 Driver for JDBC data source class provided implements the following interfaces defined in the JDBC 2.0 Optional Package:

- javax.sql.DataSource
- javax.sql.ConnectionPoolDataSource, which enables you to implement connection pooling

NOTE: You must include the javax.sql.* and javax.naming.* classes to create and use SQL Server 2000 Driver for JDBC data sources. The SQL Server 2000 Driver for JDBC provides all the necessary JAR files that contain the required classes and interfaces.

Calling a Data Source in an Application

Applications can call a SQL Server 2000 Driver for JDBC data source using a logical name to retrieve the javax.sql.DataSource object. This object loads the SQL Server 2000 Driver for JDBC and can be used to establish a connection to the underlying database.

Once a SQL Server 2000 Driver for JDBC data source has been registered with JNDI, it can be used by your JDBC application as shown in the following example:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup(“jdbc/EmployeeDB”);
Connection con = ds.getConnection(“matt”, “wwf”);
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the SQL Server 2000 Driver for JDBC data source (EmployeeDB). The Context.lookup() method returns a reference to a Java object, which is narrowed to a javax.sql.DataSource object. Finally, the DataSource.getConnection() method is called to establish a connection with the underlying database.

Using Connection Pooling

Connection pooling allows you to reuse connections rather than create a new one every time the SQL Server 2000 Driver for JDBC needs to establish a connection to the underlying database. Connection pooling manages connection sharing across different user requests to maintain performance and reduce the number of new connections that must be created. For example, compare the following transaction sequences.

Example A: Without Connection Pooling

1 The client application creates a connection.

2 The client application sends a data access query.

3 The client application obtains the result set of the query.

4 The client application displays the result set to the end user.

5 The client application ends the connection.

Example B: With Connection Pooling

1 The client checks the connection pool for an unused connection.

2 If an unused connection exists, it is returned by the pool implementation; otherwise, it creates a new connection.

3 The client application sends a data access query.

4 The client application obtains the result set of the query.

5 The client application displays the result set to the end user.

6 The client application returns the connection to the pool.

NOTE: The client application still calls "close()", but the connection remains open and the pool is notified of the close request.

The pool implementation creates "real" database connections using the getPooledConnection() method of ConnectionPoolDataSource. Then, the pool implementation registers itself as a listener to the PooledConnection. When a client application requests a connection, the pool implementation (Pool Manager) assigns one of its available connections. If there is no connection available, the Pool Manager establishes a new connection and assigns it to that application. When the client application closes the connection, the Pool Manager is notified by the driver through the ConnectionEventListener interface that the connection is free and available for reuse. The pool implementation is also notified by the ConnectionEventListener interface when the client somehow corrupts the database connection, so that the pool implementation can remove that connection from the pool.

Once a SQL Server 2000 Driver for JDBC data source has been registered with JNDI, it can be used by your JDBC application as shown in the following example, typically through a third-party connection pool tool:

```
Context ctx = new InitialContext();
ConnectionPoolDataSource ds =
(ConnectionPoolDataSource)ctx.lookup("jdbc/EmployeeDB");
pooledConnection pcon = ds.getPooledConnection("matt", "wwf");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the JDBC data source (EmployeeDB). The Context.lookup() method returns a reference to a Java object, which is narrowed to a javax.sql.ConnectionPoolDataSource object. Finally, the ConnectionPoolDataSource.getPooledConnection() method is called to establish a connection with the underlying database.

NOTE: JDBC drivers do not manage connection pooling. You must use an external connection pool manager.

Specifying Connection Properties

You can specify connection properties using the JDBC driver manager or SQL Server 2000 Driver for JDBC data sources. See "URL Examples" on page 16 for information about specifying properties through the driver manager. See "Connecting Through Data Sources" on page 17 for information about data sources.

See "Connection String Properties" on page 26 for the list of the connection properties specific to the driver.

Using the SQL Server 2000 Driver for JDBC on a Java 2 Platform

When using the SQL Server 2000 Driver for JDBC on a Java 2 Platform with the standard security manager enabled, you must give the driver some additional permissions. Refer to your Java 2 Platform documentation for more information about the Java 2 Platform security model and permissions.

You can run an application on a Java 2 Platform with the standard security manager using:

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

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

Web browser applets running in the Java 2 plug-in are always running in a Java Virtual Machine with the standard security manager enabled. To enable the necessary permissions, you must add them to the security policy file of the Java 2 Platform. This security policy file can be found in the jre/lib/security subdirectory of the Java 2 Platform installation directory.

To use JDBC data sources, all code bases must have the following permissions:

```
// permissions granted to all domains
grant {
// DataSource access
permission java.util.PropertyPermission "java.naming.*", "read,write";
// Adjust the server host specification for your environment
permission java.net.SocketPermission "*.microsoft.com:0-65535", "connect";
};
```

To use insensitive scrollable cursors, and perform client-side sorting of some DatabaseMetaData ResultSets, all code bases must have access to temporary files. If access to temporary files is not available, the driver may throw an exception indicating that it is unable to set up a static cursor cache.

For JDK 1.1 environments, access to "current working directory" must be granted.

For Java 2 environments, access to the temporary directory specified by the VM configuration must be granted.

The following is an example of permissions being granted for the C:\TEMP directory:

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

Error Handling

The SQL Server 2000 Driver for JDBC reports errors to the calling application by throwing SQLExceptions. Each SQLException contains the following information:

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

SQL Server 2000 Driver for JDBC Errors

An error generated by the SQL Server 2000 Driver for JDBC has the following format:

```
[Microsoft][SQL Server 2000 Driver for JDBC]message
```

For example:

```
[Microsoft][SQL Server 2000 Driver for JDBC]Timeout
expired.
```

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

Database Errors

An error generated by the database has the following format:

```
[Microsoft][SQL Server 2000 Driver for JDBC][SQL Server]
message
```

For example:

```
[Microsoft][SQL Server 2000 Driver for JDBC][SQL Server]
Invalid Object Name.
```

Use the native error code to look up details about the possible cause of the error. For these details, refer to your database documentation.

SQL Server 2000 Driver for JDBC Directory Structure

Table 2-1 shows the SQL Server 2000 Driver for JDBC directory structure and provides a description of files and directories. All of the following files and directories are located in the SQL Server 2000 Driver for JDBC installation directory.

Table 2-1. SQL Server 2000 Driver for JDBC Directory and Files

Directories and Files	Description
uninstall.class	Executable that uninstalls the SQL Server 2000 Driver for JDBC (Windows only).
/books/	Directory that contains all of the SQL Server 2000 Driver for JDBC online documentation.
/Help/	Directory that contains the SQL Server 2000 Driver for JDBC online help.
/lib/mssqlserver.jar	Jar file containing the SQL Server 2000 Driver for JDBC and data source classes, specifically: com.microsoft.jdbc.sqlserver.SQLServerDriver and com.microsoft.jdbcx.sqlserver.SQLServerDataSource as well as other SQL Server driver-specific classes. This Jar file must be on your CLASSPATH to use the SQL Server 2000 Driver for JDBC.
/lib/msbase.jar	Jar file containing classes that are used by the SQL Server 2000 Driver for JDBC. This Jar file must be on your CLASSPATH to use the SQL Server 2000 Driver for JDBC.
/lib/msutil.jar	Jar file containing classes that are used by the SQL Server 2000 Driver for JDBC. This Jar file must be on your CLASSPATH to use the SQL Server 2000 Driver for JDBC.
/SQLServer JTA/instjdbc.sql /SQLServer JTA/sqljdbc.dll	Files used for installing JTA stored procedures for SQL Server 2000.

3 SQL Server 2000 Driver for JDBC

The SQL Server 2000 Driver for JDBC (the "SQL Server driver") supports the SQL Server 2000 database system available from Microsoft.

To use JDBC distributed transactions through JTA, you must install stored procedures for SQL Server. See "Installing Stored Procedures for JTA" on page 32 for details.

NOTE: Although the SQL Server driver supports JTA, JTA support is not available when the connection method SelectMethod is set to direct. See "SelectMethod" on page 26 for details.

Data Source and Driver Classes

The data source class for the SQL Server driver is:

com.microsoft.jdbcx.sqlserver.SQLServerDataSource

For information on SQL Server 2000 Driver for JDBC data sources, see "Connecting Through Data Sources" on page 17.

The driver class for the SQL Server driver is:

com.microsoft.jdbc.sqlserver.SQLServerDriver

Connection String Properties

You can use the following connection properties with the JDBC driver manager or SQL Server 2000 Driver for JDBC data sources.

Table 3-1 lists the JDBC connection properties supported by the SQL Server driver, and describes each property. The properties have the form:

```
property=value
```

NOTE: All connection string property names are case-insensitive. For example, PortNumber is the same as portnumber.

Table 3-1. SQL Server Connection String Properties

Property	Description
DatabaseName OPTIONAL	The name of the SQL Server database to which you want to connect.
HostProcess OPTIONAL	The process ID of the application connecting to SQL Server 2000. The supplied value appears in the "hostprocess" column of the sysprocesses table.
NetAddress OPTIONAL	The MAC address of the network interface card of the application connecting to SQL Server 2000. The supplied value appears in the "net_address" column of the sysprocesses table.
Password	The case-insensitive password used to connect to your SQL Server database.
PortNumber OPTIONAL	The TCP port (use for DataSource connections only). The default is 1433.
ProgramName OPTIONAL	The name of the application connecting to SQL Server 2000. The supplied value appears in the "program_name" column of the sysprocesses table.
SelectMethod	SelectMethod={cursor | direct}. Determines whether database cursors are used for Select statements. Performance and behavior of the driver are affected by the SelectMethod setting.

***Table 3-1. SQL Server Connection String Properties** (cont.)*

Property	Description
SelectMethod *(cont.)*	Direct—The direct method sends the complete result set in one request to the driver. It is useful for queries that only produce a small amount of data that you fetch completely. You should avoid using direct when executing queries that produce a large amount of data, as the result set is cached completely on the client and constrains memory. In this mode, each statement requires its own connection to the database. This is accomplished by "cloning" connections. Cloned connections use the same connection properties as the original connection; however, because transactions must occur on a single connection, auto commit mode is required. Due to this, *JTA is not supported* in direct mode. In addition, some operations, such as updating an insensitive result set, are not supported in direct mode because the driver must create a second statement internally. Exceptions generated due to the creation of cloned statements usually return an error message similar to “Cannot start a cloned connection while in manual transaction mode.” Cursor—When the SelectMethod is set to cursor, a server-side cursor is generated. The rows are fetched from the server in blocks. The JDBC Statement method setFetchSize can be used to control the number of rows that are fetched per request. The cursor method is useful for queries that produce a large amount of data, data that is too large to cache on the client. Performance tests show that the value of setFetchSize has a serious impact on performance when SelectMethod is set to cursor. There is no simple rule for determining the value that you should use. You should experiment with different setFetchSize values to find out which value gives the best performance for your application. The default is direct.

Table 3-1. SQL Server Connection String Properties *(cont.)*

Property	Description
SendStringParameters AsUnicode	SendStringParametersAsUnicode={true | false}. Determines whether string parameters are sent to the SQL Server database in Unicode or in the default character encoding of the database. True means that string parameters are sent to SQL Server in Unicode. False means that they are sent in the default encoding, which can improve performance because the server does not need to convert Unicode characters to the default encoding. You should, however, use default encoding only if the parameter string data that you specify is consistent with the default encoding of the database. The default is true.
ServerName	The IP address (use for DataSource connections only). To connect to a named instance, specify *server_name\instance_name* for this property, where *server_name* is the IP address and *instance_name* is the name of the instance to which you want to connect on the specified server.
User	The case-insensitive user name used to connect to your SQL Server database.
Wsid	The workstation ID. Typically, this is the network name of the computer on which the application resides (optional). If specified, this value is stored in the master.dbo.sysprocesses column hostname and is returned by sp_who and the Transact-SQL HOST_NAME function.

Connecting to Named Instances

Microsoft SQL Server 2000 supports multiple instances of a SQL Server database running concurrently on the same server. An instance is identified by an instance name.

To connect to a named instance using a connection URL, use the following URL format:

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

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

where:

server_name is the IP address or hostname of the server.

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

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

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

To connect to a named instance using a data source, specify the ServerName connection property as described in "Connection String Properties" on page 26.

Data Types

Table 3-2 lists the data types supported by the SQL Server driver and how they are mapped to the JDBC data types.

Table 3-2. SQL Server 2000 Data Types

SQL Server Data Type	JDBC Data Type
bigint	BIGINT
bigint identity	BIGINT
binary	BINARY
bit	BIT

***Table 3-2. SQL Server 2000 Data Types** (cont.)*

SQL Server Data Type	JDBC Data Type
char	CHAR
datetime	TIMESTAMP
decimal	DECIMAL
decimal() identity	DECIMAL
float	FLOAT
image	LONGVARBINARY
int	INTEGER
int identity	INTEGER
money	DECIMAL
nchar	CHAR
ntext	LONGVARCHAR
numeric	NUMERIC
numeric() identity	NUMERIC
nvarchar	VARCHAR
real	REAL
smalldatetime	TIMESTAMP
smallint	SMALLINT
smallint identity	SMALLINT
smallmoney	DECIMAL
sql_variant	VARCHAR
sysname	VARCHAR
text	LONGVARCHAR
timestamp	BINARY
tinyint	TINYINT
tinyint identity	TINYINT
uniqueidentifier	CHAR
varbinary	VARBINARY
varchar	VARCHAR

SQL Escape Sequences

See Appendix D "SQL Escape Sequences for JDBC" on page 93 for information about the SQL escape sequences supported by the SQL Server driver.

Isolation Levels

SQL Server supports isolation levels Read Committed, Read Uncommitted, Repeatable Read, and Serializable. The default is Read Committed.

Using Scrollable Cursors

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

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

Installing Stored Procedures for JTA

To use JDBC distributed transactions through JTA, the system administrator should use the following procedure to install SQL Server JDBC XA procedures. This must be repeated for each SQL Server installation that will be involved in a distributed transaction.

1 Copy the file sqljdbc.dll from the SQL Server 2000 Driver for JDBC installation directory to the *SQL_Server_Root*/binn directory of the database server for SQL Server.

2 From the server, use the ISQL utility to run the instjdbc.sql script. The system administrator should back up the master database before running instjdbc.sql.

At a command prompt, use the following syntax to run instjdbc.sql:

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

where:

sa_password is the password of the system administrator.

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

location is the full path to instjdbc.sql. This script is located in the SQL Server 2000 Driver for JDBC installation directory.

3 The instjdbc.sql script generates many messages. In general, these messages can be ignored; however, you should scan the output for any messages that indicate an execution error. The last message should indicate that instjdbc.sql ran successfully. The script fails when there is not enough space available in master database to store the JDBC XA procedures or to log changes to existing procedures.

A JDBC Support

This appendix provides information about JDBC compatibility and developing JDBC applications for the SQL Server 2000 Driver for JDBC environments.

JDBC Compatibility

Table A-1 shows compatibility between the JDBC application versions, Java Virtual Machines, and the SQL Server 2000 Driver for JDBC.

Table A-1. JDBC Compatibility

JDBC Version Used*	JRE/JDK	Compatible?	Comments
1.22	1.0.2	No	The SQL Server 2000 Driver for JDBC does not support Java Virtual Machine 1.0.2.
1.22	1.1.8	Yes	
1.22	1.2	Yes	
2.0	1.0.2	No	The SQL Server 2000 Driver for JDBC does not support Java Virtual Machine 1.0.2.
2.0	1.1.x	No	A JDBC 2.0 application requires the JDBC 2.0 classes.
2.0	1.2	Yes	
2.0	1.3	Yes	

*Refers to whether the application is using JDBC 1.22 or JDBC 2.0 features.

Supported Functionality

The following tables list functionality supported for each JDBC object.

Array Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Core	No	Array objects are neither exposed, nor taken as input.

Blob Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Core	No	Blob objects are neither exposed, nor taken as input.

Clob Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Core	No	Clob objects are neither exposed, nor taken as input.

CallableStatement Object **Methods**	**Version Introduced**	**Supported**	**Comments**
void addBatch ()	2.0 Core	Yes	
void addBatch (String)	2.0 Core	No	Throws “invalid method call” exception.
void cancel ()	1.0	Yes	
void clearBatch ()	2.0 Core	Yes	

CallableStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void clearParameters ()	1.0	Yes	
void clearWarnings ()	1.0	Yes	
void close ()	1.0	Yes	
boolean execute ()	1.0	Yes	
boolean execute (String)	1.0	No	Throws “invalid method call” exception.
int [] executeBatch ()	2.0 Core	Yes	
ResultSet executeQuery ()	1.0	Yes	
ResultSet executeQuery (String)	1.0	No	Throws “invalid method call” exception.
int executeUpdate ()	1.0	Yes	
int executeUpdate (String)	1.0	No	Throws “invalid method call” exception.
Array getArray (int)	2.0 Core	No	Throws “unsupported method” exception.
BigDecimal getBigDecimal (int)	2.0 Core	Yes	
BigDecimal getBigDecimal (int, int)	1.0	Yes	
Blob getBlob (int)	2.0 Core	No	Throws “unsupported method” exception.
boolean getBoolean (int)	1.0	Yes	
byte getByte (int)	1.0	Yes	
byte [] getBytes (int)	1.0	Yes	
Clob getClob (int)	2.0 Core	No	Throws “unsupported method” exception.
Connection getConnection ()	1.0	Yes	

CallableStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
Date getDate (int)	1.0	Yes	
Date getDate (int, Calendar)	2.0 Core	Yes	
double getDouble (int)	1.0	Yes	
int getFetchDirection ()	2.0 Core	Yes	
int getFetchSize ()	2.0 Core	Yes	
float getFloat (int)	1.0	Yes	
int getInt (int)	1.0	Yes	
long getLong (int)	1.0	Yes	
int getMaxFieldSize ()	1.0	Yes	
int getMaxRows ()	1.0	Yes	
ResultSetMetaData getMetaData ()	2.0 Core	Yes	
boolean getMoreResults ()	1.0	Yes	
Object getObject (int)	1.0	Yes	
Object getObject (int, Map)	2.0 Core	Yes	Map ignored.
int getQueryTimeout ()	1.0	Yes	Always returns 0.
Ref getRef (int)	2.0 Core	No	Throws “unsupported method” exception.
ResultSet getResultSet ()	1.0	Yes	
int getResultSetConcurrency ()	2.0 Core	Yes	
int getResultSetType ()	2.0 Core	Yes	
short getShort (int)	1.0	Yes	
String getString (int)	1.0	Yes	
Time getTime (int)	1.0	Yes	

CallableStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
Time getTime (int, Calendar)	2.0 Core	No	Throws “unsupported method” exception.
Timestamp getTimestamp (int)	1.0	Yes	
Timestamp getTimestamp (int, Calendar)	2.0 Core	Yes	
int getUpdateCount ()	1.0	Yes	
SQLWarning getWarnings ()	1.0	Yes	
void registerOutParameter (int, int)	1.0	Yes	
void registerOutParameter (int, int, String)	2.0 Core	Yes	String/typename ignored.
void registerOutParameter (int, int, int)	1.0	Yes	
void setArray (int, Array)	2.0 Core	No	Throws “unsupported method” exception.
void setAsciiStream (int, InputStream, int)	1.0	Yes	
void setBigDecimal (int, BigDecimal)	1.0	Yes	
void setBinaryStream (int, InputStream, int)	1.0	Yes	
void setBlob (int, Blob)	2.0 Core	No	Throws “unsupported method” exception.
void setBoolean (int, boolean)	1.0	Yes	
void setByte (int, byte)	1.0	Yes	
void setBytes (int, byte [])	1.0	Yes	
void setCharacterStream (int, Reader, int)	2.0 Core	Yes	

CallableStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void setClob (int, Clob)	2.0 Core	No	Throws "unsupported method" exception.
void setCursorName (String)	1.0	No	
void setDate (int, Date)	1.0	Yes	
void setDate (int, Date, Calendar)	2.0 Core	Yes	
void setDouble (int, double)	1.0	Yes	
void setEscapeProcessing (boolean)	1.0	Yes	Ignored.
void setFetchDirection (int)	2.0 Core	Yes	
void setFetchSize (int)	2.0 Core	Yes	
void setFloat (int, float)	1.0	Yes	
void setInt (int, int)	1.0	Yes	
void setLong (int, long)	1.0	Yes	
void setMaxFieldSize (int)	1.0	Yes	
void setMaxRows (int)	1.0	Yes	
void setNull (int, int)	1.0	Yes	
void setNull (int, int, String)	2.0 Core	Yes	
void setObject (int, Object)	1.0	Yes	
void setObject (int, Object, int)	1.0	Yes	
void setObject (int, Object, int, int)	1.0	Yes	
void setQueryTimeout (int)	1.0	Yes	
void setRef (int, Ref)	2.0 Core	No	Throws "unsupported method" exception.

CallableStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void setShort (int, short)	1.0	Yes	
void setString (int, String)	1.0	Yes	
void setTime (int, Time)	1.0	Yes	
void setTime (int, Time, Calendar)	2.0 Core	Yes	
void setTimestamp (int, Timestamp)	1.0	Yes	
void setTimestamp (int, Timestamp, Calendar)	2.0 Core	Yes	
void setUnicodeStream (int, InputStream, int)	1.0	No	Throws “unsupported method” exception.
boolean wasNull ()	1.0	Yes	

Connection Object **Methods**	**Version Introduced**	**Supported**	**Comments**
void close ()	1.0	Yes	When a connection is closed while there is an active transaction, that transaction is rolled-back.
void commit ()	1.0	Yes	
Statement createStatement ()	1.0	Yes	
Statement createStatement (int, int)	2.0 Core	Yes	ResultSet.TYPE_SCROLL_SENSITIVE downgraded to TYPE_SCROLL_INSENSITIVE
boolean getAutoCommit ()	1.0	Yes	
String getCatalog ()	1.0	Yes	Support is driver-specific.

Connection Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
DatabaseMetaData getMetaData ()	1.0	Yes	
int getTransactionIsolation ()	1.0	Yes	
Map getTypeMap ()	2.0 Core	Yes	Always returns empty java.util.HashMap.
SQLWarning getWarnings ()	1.0	Yes	
boolean isClosed ()	1.0	Yes	
boolean isReadOnly ()	1.0	Yes	
String nativeSQL (String)	1.0	Yes	Always returns same String as passed in.
CallableStatement prepareCall (String)	1.0	Yes	
CallableStatement prepareCall (String, int, int)	2.0 Core	Yes	ResultSet.TYPE_SCROLL_SENSITIVE downgraded to TYPE_SCROLL_INSENSITIVE
PreparedStatement prepareStatement (String)	1.0	Yes	
PreparedStatement prepareStatement (String, int, int)	2.0 Core	Yes	ResultSet.TYPE_SCROLL_SENSITIVE downgraded to TYPE_SCROLL_INSENSITIVE
void rollback ()	1.0	Yes	
void setAutoCommit (boolean)	1.0	Yes	
void setCatalog (String)	1.0	Yes	Support is driver-specific.
void setReadOnly (boolean)	1.0	Yes	
void setTransactionIsolation (int)	1.0	Yes	
void setTypeMap (Map)	2.0 Core	Yes	Ignored.

ConnectionPoolDataSource Object **Methods**	**Version Introduced**	**Supported**	**Comments**
PrintWriter getLogWriter ()	2.0 Optional	No	
int getLoginTimeout ()	2.0 Optional	Yes	
PooledConnection getPooledConnection ()	2.0 Optional	Yes	
PooledConnection getPooledConnection (String, String)	2.0 Optional	Yes	
void setLogWriter (PrintWriter)	2.0 Optional	No	
void setLoginTimeout (int)	2.0 Optional	Yes	

DatabaseMetaData Object **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean allProceduresAreCallable ()	1.0	Yes	
boolean allTablesAreSelectable ()	1.0	Yes	
boolean dataDefinitionCausesTransaction Commit ()	1.0	Yes	
boolean dataDefinitionIgnoredInTransactions ()	1.0	Yes	
boolean deletesAreDetected (int)	2.0 Core	Yes	
boolean doesMaxRowSizeIncludeBlobs ()	1.0	Yes	
ResultSet getBestRowIdentifier (String, String, String, int, boolean)	1.0	Yes	
String getCatalogSeparator ()	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
String getCatalogTerm ()	1.0	Yes	
ResultSet getCatalogs ()	1.0	Yes	
ResultSet getColumnPrivileges (String, String, String, String)	1.0	Yes	
ResultSet getColumns (String, String, String, String)	1.0	Yes	
Connection getConnection ()	2.0 Core	Yes	
ResultSet getCrossReference (String, String, String, String, String, String)	1.0	Yes	
String getDatabaseProductName ()	1.0	Yes	
String getDatabaseProductVersion ()	1.0	Yes	
int getDefaultTransactionIsolation ()	1.0	Yes	
int getDriverMajorVersion ()	1.0	Yes	
int getDriverMinorVersion ()	1.0	Yes	
String getDriverName ()	1.0	Yes	
String getDriverVersion ()	1.0	Yes	
ResultSet getExportedKeys (String, String, String)	1.0	Yes	
String getExtraNameCharacters ()	1.0	Yes	
String getIdentifierQuoteString ()	1.0	Yes	
ResultSet getImportedKeys (String, String, String)	1.0	Yes	
ResultSet getIndexInfo (String, String, String, boolean, boolean)	1.0	Yes	
int getMaxBinaryLiteralLength ()	1.0	Yes	
int getMaxCatalogNameLength ()	1.0	Yes	
int getMaxCharLiteralLength ()	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
int getMaxColumnNameLength ()	1.0	Yes	
int getMaxColumnsInGroupBy ()	1.0	Yes	
int getMaxColumnsInIndex ()	1.0	Yes	
int getMaxColumnsInOrderBy ()	1.0	Yes	
int getMaxColumnsInSelect ()	1.0	Yes	
int getMaxColumnsInTable ()	1.0	Yes	
int getMaxConnections ()	1.0	Yes	
int getMaxCursorNameLength ()	1.0	Yes	
int getMaxIndexLength ()	1.0	Yes	
int getMaxProcedureNameLength ()	1.0	Yes	
int getMaxRowSize ()	1.0	Yes	
int getMaxSchemaNameLength ()	1.0	Yes	
int getMaxStatementLength ()	1.0	Yes	
int getMaxStatements ()	1.0	Yes	
int getMaxTableNameLength ()	1.0	Yes	
int getMaxTablesInSelect ()	1.0	Yes	
int getMaxUserNameLength ()	1.0	Yes	
String getNumericFunctions ()	1.0	Yes	
ResultSet getPrimaryKeys (String, String, String)	1.0	Yes	
ResultSet getProcedureColumns (String, String, String, String)	1.0	Yes	
String getProcedureTerm ()	1.0	Yes	
ResultSet getProcedures (String, String, String)	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
String getSQLKeywords ()	1.0	Yes	
String getSchemaTerm ()	1.0	Yes	
ResultSet getSchemas ()	1.0	Yes	
String getSearchStringEscape ()	1.0	Yes	
String getStringFunctions ()	1.0	Yes	
String getSystemFunctions ()	1.0	Yes	
ResultSet getTablePrivileges (String, String, String)	1.0	Yes	
ResultSet getTableTypes ()	1.0	Yes	
ResultSet getTables (String, String, String, String [])	1.0	Yes	
String getTimeDateFunctions ()	1.0	Yes	
ResultSet getTypeInfo ()	1.0	Yes	
ResultSet getUDTs (String, String, String, int [])	2.0 Core	No	Always returns empty ResultSet.
String getURL ()	1.0	Yes	
String getUserName ()	1.0	Yes	
ResultSet getVersionColumns (String, String, String)	1.0	Yes	
boolean insertsAreDetected (int)	2.0 Core	Yes	
boolean isCatalogAtStart ()	1.0	Yes	
boolean isReadOnly ()	1.0	Yes	
boolean nullPlusNonNullIsNull ()	1.0	Yes	
boolean nullsAreSortedAtEnd ()	1.0	Yes	
boolean nullsAreSortedAtStart ()	1.0	Yes	
boolean nullsAreSortedHigh ()	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean nullsAreSortedLow ()	1.0	Yes	
boolean othersDeletesAreVisible (int)	2.0 Core	Yes	
boolean othersInsertsAreVisible (int)	2.0 Core	Yes	
boolean othersUpdatesAreVisible (int)	2.0 Core	Yes	
boolean ownDeletesAreVisible (int)	2.0 Core	Yes	
boolean ownInsertsAreVisible (int)	2.0 Core	Yes	
boolean ownUpdatesAreVisible (int)	2.0 Core	Yes	
boolean storesLowerCaseIdentifiers ()	1.0	Yes	
boolean storesLowerCaseQuoted Identifiers ()	1.0	Yes	
boolean storesMixedCaseIdentifiers ()	1.0	Yes	
boolean storesMixedCaseQuoted Identifiers ()	1.0	Yes	
boolean storesUpperCaseIdentifiers ()	1.0	Yes	
boolean storesUpperCaseQuoted Identifiers ()	1.0	Yes	
boolean supportsANSI92EntryLevelSQL ()	1.0	Yes	
boolean supportsANSI92FullSQL ()	1.0	Yes	
boolean supportsANSI92Intermediate SQL ()	1.0	Yes	
boolean supportsAlterTableWith AddColumn ()	1.0	Yes	
boolean supportsAlterTableWith DropColumn ()	1.0	Yes	
boolean supportsBatchUpdates ()	2.0 Core	Yes	
boolean supportsCatalogsInData Manipulation ()	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean supportsCatalogsInIndex Definitions ()	1.0	Yes	
boolean supportsCatalogsInPrivilege Definitions ()	1.0	Yes	
boolean supportsCatalogsInProcedure Calls ()	1.0	Yes	
boolean supportsCatalogsInTable Definitions ()	1.0	Yes	
boolean supportsColumnAliasing ()	1.0	Yes	
boolean supportsConvert ()	1.0	Yes	
boolean supportsConvert (int, int)	1.0	Yes	
boolean supportsCoreSQLGrammar ()	1.0	Yes	
boolean supportsCorrelatedSubqueries ()	1.0	Yes	
boolean supportsDataDefinitionAndData ManipulationTransactions ()	1.0	Yes	
boolean supportsDataManipulation TransactionsOnly ()	1.0	Yes	
boolean supportsDifferentTableCorrelation Names ()	1.0	Yes	
boolean supportsExpressionsIn OrderBy ()	1.0	Yes	
boolean supportsExtendedSQLGrammar ()	1.0	Yes	
boolean supportsFullOuterJoins ()	1.0	Yes	
boolean supportsGroupBy ()	1.0	Yes	
boolean supportsGroupByBeyondSelect ()	1.0	Yes	
boolean supportsGroupByUnrelated ()	1.0	Yes	
boolean supportsIntegrityEnhancement Facility ()	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean supportsLikeEscapeClause ()	1.0	Yes	
boolean supportsLimitedOuterJoins ()	1.0	Yes	
boolean supportsMinimumSQLGrammar ()	1.0	Yes	
boolean supportsMixedCaseIdentifiers ()	1.0	Yes	
boolean supportsMixedCaseQuoted Identifiers ()	1.0	Yes	
boolean supportsMultipleResultSets ()	1.0	Yes	
boolean supportsMultipleTransactions ()	1.0	Yes	
boolean supportsNonNullableColumns ()	1.0	Yes	
boolean supportsOpenCursorsAcross Commit ()	1.0	Yes	
boolean supportsOpenCursorsAcross Rollback ()	1.0	Yes	
boolean supportsOpenStatementsAcross Commit ()	1.0	Yes	
boolean supportsOpenStatementsAcross Rollback ()	1.0	Yes	
boolean supportsOrderByUnrelated ()	1.0	Yes	
boolean supportsOuterJoins ()	1.0	Yes	
boolean supportsPositionedDelete ()	1.0	Yes	
boolean supportsPositionedUpdate ()	1.0	Yes	
boolean supportsResultSetConcurrency (int, int)	2.0 Core	Yes	
boolean supportsResultSetType (int)	2.0 Core	Yes	
boolean supportsSchemasInData Manipulation ()	1.0	Yes	

DatabaseMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean supportsSchemasInIndex Definitions ()	1.0	Yes	
boolean supportsSchemasIn PrivilegeDefinitions ()	1.0	Yes	
boolean supportsSchemasInProcedure Calls ()	1.0	Yes	
boolean supportsSchemasInTable Definitions ()	1.0	Yes	
boolean supportsSelectForUpdate ()	1.0	Yes	
boolean supportsStoredProcedures ()	1.0	Yes	
boolean supportsSubqueriesIn Comparisons ()	1.0	Yes	
boolean supportsSubqueriesInExists ()	1.0	Yes	
boolean supportsSubqueriesInIns ()	1.0	Yes	
boolean supportsSubqueriesIn Quantifieds ()	1.0	Yes	
boolean supportsTableCorrelationNames ()	1.0	Yes	
boolean supportsTransactionIsolationLevel (int)	1.0	Yes	
boolean supportsTransactions ()	1.0	Yes	
boolean supportsUnion ()	1.0	Yes	
boolean supportsUnionAll ()	1.0	Yes	
boolean updatesAreDetected (int)	2.0 Core	Yes	
boolean usesLocalFilePerTable ()	1.0	Yes	
boolean usesLocalFiles ()	1.0	Yes	

DataSource Object **Methods**	**Version Introduced**	**Supported**	**Comments**
Connection getConnection ()	2.0 Optional	Yes	
Connection getConnection (String, String)	2.0 Optional	Yes	
PrintWriter getLogWriter ()	2.0 Optional	No	
int getLoginTimeout ()	2.0 Optional	Yes	
void setLogWriter (PrintWriter)	2.0 Optional	No	
void setLoginTimeout (int)	2.0 Optional	Yes	

Driver Object **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean acceptsURL (String)	1.0	Yes	
Connection connect (String, Properties)	1.0	Yes	
int getMajorVersion ()	1.0	Yes	
int getMinorVersion ()	1.0	Yes	
DriverPropertyInfo [] getPropertyInfo (String, Properties)	1.0	Yes	
boolean jdbcCompliant ()	1.0	Yes	

PooledConnection Object **Methods**	**Version Introduced**	**Supported**	**Comments**
void addConnectionEventListener (ConnectionEventListener)	2.0 Optional	Yes	

PooledConnection Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void close()	2.0 Optional	Yes	
Connection getConnection()	2.0 Optional	Yes	A particular PooledConnection object can have only one Connection object open, and that is the one most recently created. The purpose of allowing the server (PoolManager implementation) to invoke the method getConnection a second time is to give that application server a way to take a connection away from an application and give it to someone else. This is rare, but the capability is there. The driver does not support this "reclaiming" of connections and will throw a SQLException "Reclaim of open connection is not supported."
void removeConnectionEvent Listener (ConnectionEventListener)	2.0 Optional	Yes	

PreparedStatement Object **Methods**	**Version Introduced**	**Supported**	**Comments**
void addBatch ()	2.0 Core	Yes	
void addBatch (String)	2.0 Core	No	Throws "invalid method call" exception.

PreparedStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void cancel ()	1.0	Yes	
void clearBatch ()	2.0 Core	Yes	
void clearParameters ()	1.0	Yes	
void clearWarnings ()	1.0	Yes	
void close ()	1.0	Yes	
boolean execute ()	1.0	Yes	
boolean execute (String)	1.0	No	Throws “invalid method call” exception.
int [] executeBatch ()	2.0 Core	Yes	
ResultSet executeQuery ()	1.0	Yes	
ResultSet executeQuery (String)	1.0	No	Throws “invalid method call” exception.
int executeUpdate ()	1.0	Yes	
int executeUpdate (String)	1.0	No	Throws “invalid method call” exception.
Connection getConnection ()	1.0	Yes	
int getFetchDirection ()	2.0 Core	Yes	
int getFetchSize ()	2.0 Core	Yes	
int getMaxFieldSize ()	1.0	Yes	
int getMaxRows ()	1.0	Yes	
ResultSetMetaData getMetaData ()	2.0 Core	Yes	
boolean getMoreResults ()	1.0	Yes	
int getQueryTimeout ()	1.0	Yes	Always returns 0.
ResultSet getResultSet ()	1.0	Yes	

PreparedStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
int getResultSetConcurrency ()	2.0 Core	Yes	
int getResultSetType ()	2.0 Core	Yes	
int getUpdateCount ()	1.0	Yes	
SQLWarning getWarnings ()	1.0	Yes	
void setArray (int, Array)	2.0 Core	No	Throws "unsupported method" exception.
void setAsciiStream (int, InputStream, int)	1.0	Yes	
void setBigDecimal (int, BigDecimal)	1.0	Yes	
void setBinaryStream (int, InputStream, int)	1.0	Yes	
void setBlob (int, Blob)	2.0 Core	No	Throws "unsupported method" exception.
void setBoolean (int, boolean)	1.0	Yes	
void setByte (int, byte)	1.0	Yes	
void setBytes (int, byte [])	1.0	Yes	
void setCharacterStream (int, Reader, int)	2.0 Core	Yes	
void setClob (int, Clob)	2.0 Core	No	
void setCursorName (String)	1.0	No	Throws "unsupported method" exception.
void setDate (int, Date)	1.0	Yes	
void setDate (int, Date, Calendar)	2.0 Core	Yes	
void setDouble (int, double)	1.0	Yes	

PreparedStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void setEscapeProcessing (boolean)	1.0	Yes	Ignored.
void setFetchDirection (int)	2.0 Core	Yes	
void setFetchSize (int)	2.0 Core	Yes	
void setFloat (int, float)	1.0	Yes	
void setInt (int, int)	1.0	Yes	
void setLong (int, long)	1.0	Yes	
void setMaxFieldSize (int)	1.0	Yes	
void setMaxRows (int)	1.0	Yes	
void setNull (int, int)	1.0	Yes	
void setNull (int, int, String)	2.0 Core	Yes	
void setObject (int, Object)	1.0	Yes	
void setObject (int, Object, int)	1.0	Yes	
void setObject (int, Object, int, int)	1.0	Yes	
void setQueryTimeout (int)	1.0	Yes	
void setRef (int, Ref)	2.0 Core	No	Throws “unsupported method” exception.
void setShort (int, short)	1.0	Yes	
void setString (int, String)	1.0	Yes	
void setTime (int, Time)	1.0	Yes	
void setTime (int, Time, Calendar)	2.0 Core	Yes	
void setTimestamp (int, Timestamp)	1.0	Yes	

PreparedStatement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void setTimestamp (int, Timestamp, Calendar)	2.0 Core	Yes	
void setUnicodeStream (int, InputStream, int)	1.0	No	Throws “unsupported method” exception.

Ref Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Core	No	

Referenceable Object **Methods**	**JDBC Version Introduced**	**Supported**	**Comments**
Reference getReference()	javax.naming	Yes	Implemented by DataSource classes.

ResultSet Object **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean absolute (int)	2.0 Core	Yes	
void afterLast ()	2.0 Core	Yes	
void beforeFirst ()	2.0 Core	Yes	
void cancelRowUpdates ()	2.0 Core	Yes	
void clearWarnings ()	1.0	Yes	
void close ()	1.0	Yes	
void deleteRow ()	2.0 Core	Yes	
int findColumn (String)	1.0	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
boolean first ()	2.0 Core	Yes	
Array getArray (String)	2.0 Core	No	Throws “unsupported method” exception.
Array getArray (int)	2.0 Core	No	Throws “unsupported method” exception.
InputStream getAsciiStream (String)	1.0	Yes	
InputStream getAsciiStream (int)	1.0	Yes	
BigDecimal getBigDecimal (String)	2.0 Core	Yes	
BigDecimal getBigDecimal (int)	2.0 Core	Yes	
BigDecimal getBigDecimal (String, int)	1.0	Yes	
BigDecimal getBigDecimal (int, int)	1.0	Yes	
InputStream getBinaryStream (int)	1.0	Yes	
InputStream getBinaryStream (String)	1.0	Yes	
Blob getBlob (int)	2.0 Core	No	Throws “unsupported method” exception.
Blob getBlob (String)	2.0 Core	No	Throws “unsupported method” exception.
boolean getBoolean (String)	1.0	Yes	
boolean getBoolean (int)	1.0	Yes	
byte getByte (int)	1.0	Yes	
byte getByte (String)	1.0	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
byte [] getBytes (String)	1.0	Yes	
byte [] getBytes (int)	1.0	Yes	
Reader getCharacterStream (int)	2.0 Core	Yes	
Reader getCharacterStream (String)	2.0 Core	Yes	
Clob getClob (String)	2.0 Core	No	Throws “unsupported method” exception.
Clob getClob (int)	2.0 Core	No	Throws “unsupported method” exception.
int getConcurrency ()	2.0 Core	Yes	
String getCursorName ()	1.0	No	Throws “unsupported method” exception.
Date getDate (int)	1.0	Yes	
Date getDate (String)	1.0	Yes	
Date getDate (String, Calendar)	2.0 Core	No	Throws “unsupported method” exception.
Date getDate (int, Calendar)	2.0 Core	Yes	
double getDouble (String)	1.0	Yes	
double getDouble (int)	1.0	Yes	
int getFetchDirection ()	2.0 Core	Yes	
int getFetchSize ()	2.0 Core	Yes	
float getFloat (int)	1.0	Yes	
float getFloat (String)	1.0	Yes	
int getInt (int)	1.0	Yes	
int getInt (String)	1.0	Yes	
long getLong (int)	1.0	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
long getLong (String)	1.0	Yes	
ResultSetMetaData getMetaData ()	1.0	Yes	
Object getObject (int)	1.0	Yes	
Object getObject (String)	1.0	Yes	
Object getObject (int, Map)	2.0 Core	Yes	Map ignored.
Object getObject (String, Map)	2.0 Core	Yes	Map ignored.
Ref getRef (int)	2.0 Core	No	Throws "unsupported method" exception.
Ref getRef (String)	2.0 Core	No	Throws "unsupported method" exception.
int getRow ()	2.0 Core	Yes	
short getShort (String)	1.0	Yes	
short getShort (int)	1.0	Yes	
Statement getStatement ()	2.0 Core	Yes	
String getString (int)	1.0	Yes	
String getString (String)	1.0	Yes	
Time getTime (int)	1.0	Yes	
Time getTime (String)	1.0	Yes	
Time getTime (String, Calendar)	2.0 Core	Yes	
Time getTime (int, Calendar)	2.0 Core	Yes	
Timestamp getTimestamp (int)	1.0	Yes	
Timestamp getTimestamp (String)	1.0	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
Timestamp getTimestamp (String, Calendar)	2.0 Core	Yes	
Timestamp getTimestamp (int, Calendar)	2.0 Core	Yes	
int getType ()	2.0 Core	Yes	
InputStream getUnicodeStream (int)	1.0	No	Throws "unsupported method" exception.
InputStream getUnicodeStream (String)	1.0	No	Throws "unsupported method" exception.
SQLWarning getWarnings ()	1.0	Yes	
void insertRow ()	2.0 Core	Yes	
boolean isAfterLast ()	2.0 Core	Yes	
boolean isBeforeFirst ()	2.0 Core	Yes	
boolean isFirst ()	2.0 Core	Yes	
boolean isLast ()	2.0 Core	Yes	
boolean last ()	2.0 Core	Yes	
void moveToCurrentRow ()	2.0 Core	Yes	
void moveToInsertRow ()	2.0 Core	Yes	
boolean next ()	1.0	Yes	
boolean previous ()	2.0 Core	Yes	
void refreshRow ()	2.0 Core	Yes	
boolean relative (int)	2.0 Core	Yes	
boolean rowDeleted ()	2.0 Core	Yes	
boolean rowInserted ()	2.0 Core	Yes	
boolean rowUpdated ()	2.0 Core	Yes	
void setFetchDirection (int)	2.0 Core	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void setFetchSize (int)	2.0 Core	Yes	
void updateAsciiStream (String, InputStream, int)	2.0 Core	Yes	
void updateAsciiStream (int, InputStream, int)	2.0 Core	Yes	
void updateBigDecimal (int, BigDecimal)	2.0 Core	Yes	
void updateBigDecimal (String, BigDecimal)	2.0 Core	Yes	
void updateBinaryStream (String, InputStream, int)	2.0 Core	Yes	
void updateBinaryStream (int, InputStream, int)	2.0 Core	Yes	
void updateBoolean (int, boolean)	2.0 Core	Yes	
void updateBoolean (String, boolean)	2.0 Core	Yes	
void updateByte (String, byte)	2.0 Core	Yes	
void updateByte (int, byte)	2.0 Core	Yes	
void updateBytes (String, byte [])	2.0 Core	Yes	
void updateBytes (int, byte [])	2.0 Core	Yes	
void updateCharacterStream (String, Reader, int)	2.0 Core	Yes	
void updateCharacterStream (int, Reader, int)	2.0 Core	Yes	
void updateDate (int, Date)	2.0 Core	Yes	
void updateDate (String, Date)	2.0 Core	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void updateDouble (String, double)	2.0 Core	Yes	
void updateDouble (int, double)	2.0 Core	Yes	
void updateFloat (int, float)	2.0 Core	Yes	
void updateFloat (String, float)	2.0 Core	Yes	
void updateInt (int, int)	2.0 Core	Yes	
void updateInt (String, int)	2.0 Core	Yes	
void updateLong (String, long)	2.0 Core	Yes	
void updateLong (int, long)	2.0 Core	Yes	
void updateNull (String)	2.0 Core	Yes	
void updateNull (int)	2.0 Core	Yes	
void updateObject (String, Object)	2.0 Core	Yes	
void updateObject (int, Object)	2.0 Core	Yes	
void updateObject (String, Object, int)	2.0 Core	Yes	
void updateObject (int, Object, int)	2.0 Core	Yes	
void updateRow ()	2.0 Core	Yes	
void updateShort (int, short)	2.0 Core	Yes	
void updateShort (String, short)	2.0 Core	Yes	
void updateString (String, String)	2.0 Core	Yes	

ResultSet Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
void updateString (int, String)	2.0 Core	Yes	
void updateTime (int, Time)	2.0 Core	Yes	
void updateTime (String, Time)	2.0 Core	Yes	
void updateTimestamp (String, Timestamp)	2.0 Core	Yes	
void updateTimestamp (int, Timestamp)	2.0 Core	Yes	
boolean wasNull ()	1.0	Yes	

ResultSetMetaData Object **Methods**	**Version Introduced**	**Supported**	**Comments**
String getCatalogName (int)	1.0	Yes	
String getColumnClassName (int)	2.0 Core	Yes	
int getColumnCount ()	1.0	Yes	
int getColumnDisplaySize (int)	1.0	Yes	
String getColumnLabel (int)	1.0	Yes	
String getColumnName (int)	1.0	Yes	
int getColumnType (int)	1.0	Yes	
String getColumnTypeName (int)	1.0	Yes	
int getPrecision (int)	1.0	Yes	
int getScale (int)	1.0	Yes	
String getSchemaName (int)	1.0	Yes	

ResultSetMetaData Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
String getTableName (int)	1.0	Yes	
boolean isAutoIncrement (int)	1.0	Yes	
boolean isCaseSensitive (int)	1.0	Yes	
boolean isCurrency (int)	1.0	Yes	
boolean isDefinitelyWritable (int)	1.0	Yes	
int isNullable (int)	1.0	Yes	
boolean isReadOnly (int)	1.0	Yes	
boolean isSearchable (int)	1.0	Yes	
boolean isSigned (int)	1.0	Yes	
boolean isWritable (int)	1.0	Yes	

RowSet Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Optional	No	

Serializable Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(N/A)	java.io	Yes	Implemented by DataSource classes.

Struct Object Methods	Version Introduced	Supported	Comments
(all)	2.0	No	

Statement Object Methods	Version Introduced	Supported	Comments
void addBatch (String)	2.0 Core	Yes	
void cancel ()	1.0	Yes	
void clearBatch ()	2.0 Core	Yes	
void clearWarnings ()	1.0	Yes	
void close ()	1.0	Yes	
boolean execute (String)	1.0	Yes	
int [] executeBatch ()	2.0 Core	Yes	
ResultSet executeQuery (String)	1.0	Yes	
int executeUpdate (String)	1.0	Yes	
Connection getConnection ()	2.0 Core	Yes	
int getFetchDirection ()	2.0 Core	Yes	
int getFetchSize ()	2.0 Core	Yes	
int getMaxFieldSize ()	1.0	Yes	
int getMaxRows ()	1.0	Yes	
boolean getMoreResults ()	1.0	Yes	
int getQueryTimeout ()	1.0	Yes	
ResultSet getResultSet ()	1.0	Yes	
int getResultSetConcurrency ()	2.0 Core	Yes	
int getResultSetType ()	2.0 Core	Yes	
int getUpdateCount ()	1.0	Yes	

Statement Object *(cont.)* **Methods**	**Version Introduced**	**Supported**	**Comments**
SQLWarning getWarnings ()	1.0	Yes	
void setCursorName (String)	1.0	No	Throws “unsupported method” exception.
void setEscapeProcessing (boolean)	1.0	Yes	Ignored.
void setFetchDirection (int)	2.0 Core	Yes	
void setFetchSize (int)	2.0 Core	Yes	
void setMaxFieldSize (int)	1.0	Yes	
void setMaxRows (int)	1.0	Yes	
void setQueryTimeout (int)	1.0	Yes	

XAConnection Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Optional	Yes	

XADataSource Object **Methods**	**Version Introduced**	**Supported**	**Comments**
(all)	2.0 Optional	Yes	

B SQL Server 2000 Driver for JDBC GetTypeinfo

The following table provides results returned from the method DataBaseMetaData.getTypeinfo for the SQL Server 2000 Driver for JDBC. The table is alphabetical first by TYPE_NAME, and then by parameter.

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC

TYPE_NAME = bigint

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

TYPE_NAME = bigint identity

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

***Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC** (cont.)*

TYPE_NAME = binary

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

TYPE_NAME = bit

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

TYPE_NAME = char

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

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC (cont.)

TYPE_NAME = datetime

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

TYPE_NAME = decimal

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

TYPE_NAME = decimal() identity

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

***Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC** (cont.)*

TYPE_NAME = float

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

TYPE_NAME = image

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

TYPE_NAME = int

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

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC (cont.)

TYPE_NAME = int identity

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

TYPE_NAME = money

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

TYPE_NAME = nchar

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

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC (cont.)

TYPE_NAME = ntext

AUTO_INCREMENT = NULL	MINIMUM_SCALE = NULL
CASE_SENSITIVE = false	NULLABLE = 1
CREATE_PARAMS = NULL	NUM_PREC_RADIX = NULL
DATA_TYPE = -1 (LONGVARCHAR)	PRECISION = 1073741823
FIXED_PREC_SCALE = false	SEARCHABLE = 1
LITERAL_PREFIX = N’	SQL_DATA_TYPE = NULL
LITERAL_SUFFIX = ’	SQL_DATETIME_SUB = NULL
LOCAL_TYPE_NAME = ntext	UNSIGNED_ATTRIBUTE = NULL
MAXIMUM_SCALE = NULL	

TYPE_NAME = numeric

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

TYPE_NAME = numeric() identity

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

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC (cont.)

TYPE_NAME = nvarchar

AUTO_INCREMENT = NULL	MINIMUM_SCALE = NULL
CASE_SENSITIVE = false	NULLABLE = 1
CREATE_PARAMS = max length	NUM_PREC_RADIX = NULL
DATA_TYPE = 12 (VARCHAR)	PRECISION = 4000
FIXED_PREC_SCALE = false	SEARCHABLE = 3
LITERAL_PREFIX = N’	SQL_DATA_TYPE = NULL
LITERAL_SUFFIX = ’	SQL_DATETIME_SUB = NULL
LOCAL_TYPE_NAME = nvarchar	UNSIGNED_ATTRIBUTE = NULL
MAXIMUM_SCALE = NULL	

TYPE_NAME = real

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

TYPE_NAME = smalldatetime

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

***Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC** (cont.)*

TYPE_NAME = smallint

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

TYPE_NAME = smallint identity

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

TYPE_NAME = smallmoney

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

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC (cont.)

TYPE_NAME = sql_variant

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

TYPE_NAME = sysname

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

TYPE_NAME = text

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

***Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC** (cont.)*

TYPE_NAME = timestamp

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

TYPE_NAME = tinyint

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

TYPE_NAME = tinyint identity

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

Table B-1. GetTypeinfo for the SQL Server 2000 Driver for JDBC (cont.)

TYPE_NAME = uniqueidentifier

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

TYPE_NAME = varbinary

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

TYPE_NAME = varchar

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

C Designing JDBC Applications for Performance Optimization

Developing performance-oriented JDBC applications is not easy. JDBC drivers do not throw exceptions to say that your code is running too slow.

These guidelines were compiled by examining the JDBC implementations of numerous shipping JDBC applications. The guidelines discuss using database metadata materials, retrieving data, selecting JDBC objects and methods, designing JDBC applications, and updating data.

The following table summarizes some common JDBC system performance problems and suggests some possible solutions.

Problem	Solution	See guidelines in...
Network communication is slow	Reduce network traffic	"Using Database Metadata Methods" on page 78
Evaluation of complex SQL queries on the database server is slow and might reduce concurrency	Simplify queries	"Using Database Metadata Methods" on page 78 "Selecting JDBC Objects and Methods" on page 83
Excessive calls from the application to the driver decrease performance	Optimize application-to-driver interaction	"Retrieving Data" on page 81 "Selecting JDBC Objects and Methods" on page 83
Disk input/output is slow	Limit disk input/output	"Designing JDBC Applications" on page 86

Using Database Metadata Methods

Because database metadata methods that generate Resultset objects are slow compared to other JDBC methods, their frequent use can impair system performance. The guidelines in this section will help you to optimize system performance when selecting and using database metadata.

Minimizing the Use of Database Metadata Methods

Compared to other JDBC methods, database metadata methods that generate Resultset objects are relatively slow. Applications should cache information returned from result sets that generate database metadata methods so that multiple executions are not needed.

While almost no JDBC application can be written without database metadata methods, you can improve system performance by minimizing their use. To return all result column information *mandated* by the JDBC specification, a JDBC driver may have to perform complex queries or multiple queries to return the necessary result set for a single call to a database metadata method. These particular elements of the SQL language are performance-expensive.

Applications should cache information from database metadata methods. For example, call getTypeInfo once in the application and cache away the elements of the result set that your application depends on. It is unlikely that any application uses all elements of the result set generated by a database metadata method, so the cache of information should not be difficult to maintain.

Avoiding Search Patterns

Using null arguments or search patterns in database metadata methods results in generating time-consuming queries. In addition, network traffic potentially increases due to unwanted results. Always supply as many non-null arguments to result sets that generate database metadata methods as possible.

Because database metadata methods are slow, applications should invoke them as efficiently as possible. Many applications pass the fewest non-null arguments necessary for the function to return success.

For example:

```
ResultSet WSrs = WSc.getTables (null, null, "WSTable", null);
```

should be:

```
ResultSet WSrs = WSc.getTables ("cat1", "johng", "WSTable", "TABLE");
```

Sometimes, little information is known about the object for which you are requesting information. Any information that the application can send the driver when calling database metadata methods can result in improved performance and reliability.

Using a Test Query to Determine Table Characteristics

Avoid using getColumns to determine characteristics about a table. Instead, use a test query with getMetadata.

Consider an application that allows the user to choose the columns that will be selected. Should the application use getColumns to return information about the columns to the user or instead prepare a test query and call getMetadata?

Case 1: GetColumns Method

```
ResultSet WSrc = WSc.getColumns (... "UnknownTable" ...);
// This call to getColumns will generate a query to
// the system catalogs... possibly a join
// which must be prepared, executed, and produce
// a result set
. . .
WSrc.next();
string Cname = getString(4);
. . .
// user must retrieve N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: GetMetadata Method

```
// prepare test query
PreparedStatement WSps = WSc.prepareStatement
    (... "SELECT * from UnknownTable WHERE 1 = 0" ...);
// query is never executed on the server -
// only prepared
ResultSetMetaData WSsmd=wsps.getMetaData();
int numcols = WSrsmd.getColumnCount();
...
int ctype = WSrsmd.getColumnType(n)
...
// result column information has now been obtained
// Note we also know the column ordering within the
// table!  This information cannot be
// assumed from the getColumns example.
```

In both cases, a query is sent to the server, but in Case 1 the query must be evaluated and form a result set that must be sent to the client. Clearly, Case 2 is the better performing model.

To somewhat complicate this discussion, let us consider a DBMS server that does not natively support preparing a SQL statement. The performance of Case 1 does not change but Case 2 increases minutely because the test query must be evaluated instead of only prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and should execute without accessing table data. For this situation, method 2 still outperforms method 1.

Retrieving Data

To retrieve data efficiently, return only the data that you need, and choose the most efficient method of doing so. The guidelines in this section will help you to optimize system performance when retrieving data with JDBC applications.

Retrieving Long Data

Unless it is necessary, applications should not request long data because retrieving long data across a network is slow and resource-intensive.

Most users don't want to see long data. If the user does want to see these result items, then the application can query the database again, specifying only the long columns in the select list. This method allows the average user to retrieve the result set without having to pay a high performance penalty for network traffic.

Although the best method is to exclude long data from the select list, some applications do not formulate the select list before sending the query to the JDBC driver (that is, some applications `select * from <table name> ...`). If the select list contains long data, then some drivers must retrieve that data at fetch time even if the application does not bind the long data in the result set. When possible, the designer should attempt to implement a method that does not retrieve all columns of the table.

Additionally, although the getClob and getBlob methods allow the application to control how long data is retrieved in the application, the designer must realize that in many cases, the JDBC driver emulates these methods due to the lack of true locator support in the DBMS. In such cases, the driver must retrieve all of the long data across the network before exposing the getClob and getBlob methods.

Sometimes long data must be retrieved. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen.

Reducing the Size of Data Retrieved

To reduce network traffic and improve performance, you can reduce the size of any data being retrieved to some manageable limit by calling setMaxRows, setMaxFieldSize, and the driver-specific SetFetchSize. Another method of reducing the size of the data being retrieved is to decrease the column size. If the driver allows you to define the packet size, use the smallest packet size that will meet your needs.

In addition, be careful to return only the rows you need. If you return five columns when you only need two columns, performance is decreased, especially if the unnecessary rows include long data.

Choosing the Right Data Type

Retrieving and sending certain data types can be expensive. When you design a schema, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the wire protocol.

Processing time is shortest for character strings, followed by integers, which usually require some conversion or byte ordering. Processing floating-point data and timestamps is at least twice as slow as integers.

Selecting JDBC Objects and Methods

The guidelines in this section will help you to optimize system performance when selecting and using JDBC objects and methods.

Using Parameter Markers as Arguments to Stored Procedures

When calling stored procedures, always use parameter markers for the argument markers instead of using literal arguments. JDBC drivers can call stored procedures on the database server either by executing the procedure as any other SQL query, or by optimizing the execution by invoking a Remote Procedure Call (RPC) directly into the database server. Executing the stored procedure as a SQL query results in the database server parsing the statement, validating the argument types, and converting the arguments into the correct data types. Remember that SQL is always sent to the database server as a character string, for example, "{call getCustName (12345)}". In this case, even though the application programmer might assume that the only argument to getCustName is an integer, the argument is actually passed inside a character string to the server. The database server would parse the SQL query, isolate the single argument value 12345, then convert the string '12345' into an integer value.

By invoking an RPC inside the database server, the overhead of using a SQL character string is avoided. Instead, the procedure is called only by name with the argument values already encoded into their native data types.

Case 1

Stored Procedure cannot be optimized to use a server-side RPC. The database server must parse the statement, validate the argument types, and convert the arguments into the correct data types.

```
CallableStatement cstmt = conn.prepareCall ("call getCustName (12345)");
ResultSet rs = cstmt.executeQuery ();
```

Case 2

Stored Procedure can be optimized to use a server-side RPC. Because the application calls the procedure by name and the argument values are already encoded, the load on the database server is less.

```
CallableStatement cstmt = conn.prepareCall ("Call getCustName (?)");
cstmt.setLong (1,12345);
ResultSet rs = cstmt.executeQuery();
```

Using the Statement Object Instead of the PreparedStatement Object

JDBC drivers are optimized based on the perceived use of the functions that are being executed. Choose between the PreparedStatement object and the Statement object depending on the planned use. The Statement object is optimized for a single execution of a SQL statement. In contrast, the PreparedStatement object is optimized for SQL statements that will be executed two or more times.

The overhead for the initial execution of a PreparedStatement object is high. The advantage comes with subsequent executions of the SQL statement.

Choosing the Right Cursor

Choosing the appropriate type of cursor allows maximum application flexibility. This section summarizes the performance issues of three types of cursors.

A forward-only cursor provides excellent performance for sequential reads of all of the rows in a table. However, it cannot be used when the rows to be returned are not sequential.

Insensitive cursors used by JDBC drivers are ideal for applications that require high levels of concurrency on the database server and require the ability to scroll forwards and backwards through result sets. The first request to an insensitive cursor fetches all of the rows and stores them on the client. Thus, the first request is very slow, especially when long data is retrieved. Subsequent requests do not require any network traffic and are processed quickly. Because the first request is processed slowly, insensitive cursors should not be used for a single request of one row. Designers should also avoid using insensitive cursors when long data is returned, because memory can be exhausted. Some insensitive cursor implementations cache the data in a temporary table on the database server and avoid the performance issue.

Sensitive cursors, sometimes called keyset-driven cursors, use identifiers, such as a ROWID, that already exist in your database. When you scroll through the result set, the data for the identifiers is retrieved. Because each request generates network traffic, performance can be very slow. However, returning nonsequential rows does not further affect performance. Sensitive cursors are the preferred scrollable cursor model for dynamic situations, when the application cannot afford to buffer the data from an insensitive cursor.

Designing JDBC Applications

The guidelines in this section will help you to optimize system performance when designing JDBC applications.

Managing Connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple statement objects, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. For example, some applications establish a connection and then call a method in a separate component that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the established connection object to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to perform SQL statements. Connection objects can have multiple statement objects associated with them. Statement objects, which are defined to be memory storage for information about SQL statements, can manage multiple SQL statements.

You can improve performance significantly with connection pooling, especially for applications that connect over a network or through the World Wide Web. Connection pooling lets you reuse connections. Closing connections does not close the physical connection to the database. When an application requests a connection, an active connection is reused, thus avoiding the network I/O needed to create a new connection.

Connection and statement handling should be addressed before implementation. Spending time and thoughtfully handling connection management improves application performance and maintainability.

Managing Commits in Transactions

Committing transactions is extremely disk I/O intensive and slow. Always turn off autocommit by using the following setting: `WSConnection.setAutoCommit(false)`.

What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is not a sequential write but a searched write to replace existing data in the table. By default, Autocommit is on when connecting to a data source, and Autocommit mode usually impairs performance because of the significant amount of disk I/O needed to commit every operation.

Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for long times, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.

Choosing the Right Transaction Model

Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network I/O necessary to communicate between all the components involved in the distributed transaction. Unless distributed transactions are required, avoid using them. Instead, use local transactions whenever possible.

For the best system performance, design the application to run under a single Connection object.

Updating Data

This section provides general guidelines to help you to optimize system performance when updating data in databases.

Using updateXXX Methods

Although programmatic updates do not apply to all types of applications, developers should attempt to use programmatic updates and deletes. Using the updateXXX methods of the ResultSet object allows the developer to update data without building a complex SQL statement. Instead, the developer simply supplies the column in the result set that is to be updated and the data that is to be changed. Then, before moving the cursor from the row in the result set, the updateRow method must be called to update the database as well.

In the following code fragment, the value of the Age column of the Resultset object rs is retrieved using the method getInt, and the method updateInt is used to update the column with an int value of 25. The method updateRow is called to update the row in the database that contains the modified value.

```
int n = rs.getInt("Age");
// n contains value of Age column in the resultset rs
. . .
rs.updateInt("Age", 25);
rs.updateRow();
```

In addition to making the application more easily maintainable, programmatic updates usually result in improved performance. Because the database server is already positioned on the row for the Select statement in process, performance-expensive operations to locate the row to be changed are not needed. If the row must be located, the server usually has an internal pointer to the row available (for example, ROWID).

Using getBestRowIndentifier()

Use getBestRowIndentifier() to determine the optimal set of columns to use in the Where clause for updating data. Pseudo-columns often provide the fastest access to the data, and these columns can only be determined by using getBestRowIndentifier().

Some applications cannot be designed to take advantage of positional updates and deletes. Some applications might formulate the Where clause by using all searchable result columns by calling getPrimaryKeys(), or by calling getIndexInfo() to find columns that might be part of a unique index. These methods usually work, but might result in fairly complex queries.

Consider the following example:

```
ResultSet WSrs = WSs.executeQuery
     ("SELECT first_name, last_name, ssn, address, city, state, zip
        FROM emp");
// fetchdata
...
WSs.executeQuery ("UPDATE EMP SET ADDRESS = ?
     WHERE first_name = ? and last_name = ? and ssn = ?
     and address = ? and city = ? and state = ?
     and zip = ?");
// fairly complex query
```

Applications should call getBestRowIndentifier() to retrieve the optimal set of columns (possibly a pseudo-column) that identifies a specific record. Many databases support special columns that are not explicitly defined by the user in the table definition but are "hidden" columns of every table (for example, ROWID and TID). These pseudo-columns generally provide the fastest access to the data because they typically are pointers to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from getColumns. To determine if pseudo-columns exist, call getBestRowIndentifier().

Consider the previous example again:

```
...
ResultSet WSrowid = getBestRowIndentifier()
   (.... "emp", ...);
...
WSs.executeQuery ("UPDATE EMP SET ADDRESS = ?
     WHERE first_name = ? and last_name = ? and ssn = ?
     and address = ? and city = ? and state = ?
     and zip = ?");
// fastest access to the data!
```

If your data source does not contain special pseudo-columns, then the result set of getBestRowIndentifier() consists of the columns of the most optimal unique index on the specified table (if a unique index exists). Therefore, your application does not need to call getIndexInfo to find the smallest unique index.

Conclusion

With thoughtful design and implementation, the performance of JDBC applications can be improved. By the appropriate use of DatabaseMetaData methods, retrieving only required data, selecting functions that optimize performance, and managing connections and updates, your applications can run more efficiently and generate less network traffic.

D SQL Escape Sequences for JDBC

A number of language features, such as outer joins and scalar function calls, are commonly implemented by DBMSs. The syntax for these features is often DBMS-specific, even when a standard syntax has been defined. JDBC defines escape sequences that contain standard syntaxes for the following language features:

- Date, time, and timestamp literals
- Scalar functions such as numeric, string, and data type conversion functions
- Outer joins
- Procedure calls

The escape sequence used by JDBC is:

```
{extension}
```

The escape sequence is recognized and parsed by the SQL Server 2000 Driver for JDBC, which replaces the escape sequences with data store-specific grammar.

Date, Time, and Timestamp Escape Sequences

The escape sequence for date, time, and timestamp literals is:

```
{literal-type 'value'}
```

where *literal-type* is one of the following:

literal-type	Description	Value Format
d	Date	`yyyy-mm-dd`
t	Time	`hh:mm:ss [1]`
ts	Timestamp	`yyyy-mm-dd hh:mm:ss[.f...]`

Example:

```
UPDATE Orders SET OpenDate={d '1995-01-15'}
WHERE OrderID=1023
```

Scalar Functions

You can use scalar functions in SQL statements with the following syntax:

```
{fn scalar-function}
```

where *scalar-function* is a scalar function supported by the Microsoft SQL Server 2000 Driver for JDBC, as listed in Table D-1.

Example:

```
SELECT {fn UCASE(NAME)} FROM EMP
```

Table D-1. Scalar Functions Supported

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
SQL Server 2000	ASCII	ABS	DAYNAME	DBNAME
	CHAR	ACOS	DAYOFMONTH	IFNULL
	CONCAT	ASIN	DAYOFWEEK	USERNAME
	DIFFERENCE	ATAN	DAYOFYEAR	
	INSERT	ATAN2	EXTRACT	
	LCASE	CEILING	HOUR	
	LEFT	COS	MINUTE	
	LENGTH	COT	MONTH	
	LOCATE	DEGREES	MONTHNAME	
	LTRIM	EXP	NOW	
	REPEAT	FLOOR	QUARTER	
	REPLACE	LOG	SECOND	
	RIGHT	LOG10	TIMESTAMPADD	
	RTRIM	MOD	TIMESTAMPDIFF	
	SOUNDEX	PI	WEEK	
	SPACE	POWER	YEAR	
	SUBSTRING	RADIANS		
	UCASE	RAND		
		ROUND		
		SIGN		
		SIN		
		SQRT		
		TAN		
		TRUNCATE		

Outer Join Escape Sequences

JDBC supports the SQL92 left, right, and full outer join syntax. The escape sequence for outer joins is:

```
{oj outer-join}
```

where *outer-join* is:

```
table-reference {LEFT | RIGHT | FULL} OUTER JOIN
{table-reference | outer-join} ON search-condition
```

where:

table-reference is a table name.

search-condition is the join condition you want to use for the tables.

Example:

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID,
Orders.Status
  FROM {oj Customers LEFT OUTER JOIN
        Orders ON Customers.CustID=Orders.CustID}
  WHERE Orders.Status='OPEN'
```

Table D-2 lists the outer join escape sequences supported by the Microsoft SQL Server 2000 Driver for JDBC.

Table D-2. Outer Join Escape Sequences Supported

Data Store	Outer Join Escape Sequences
SQL Server 2000	Left outer joins Right outer joins Full outer joins Nested outer joins

Procedure Call Escape Sequences

A procedure is an executable object stored in the data store. Generally, it is one or more SQL statements that have been precompiled. The escape sequence for calling a procedure is:

```
{[?=]call procedure-name[([parameter][,[parameter]]...)]}
```

where:

procedure-name specifies the name of a stored procedure.

parameter specifies a stored procedure parameter.

Index