Mimer JDBC

Driver Guide

December 2021

Produced and published by Mimer Information Technology AB, Uppsala, Sweden
Phone +46(0)18 780 92 00

Mimer SQL Web Sites:
https://developer.mimer.com
https://www.mimer.com

Contents

Chapter 1 Introduction 1
About this Guide 1
Definitions, Terms and Trademarks 1
Requirements 2
Environment 3
Logging 3
Chapter 2 Using the Mimer JDBC Driver 5
Loading a Driver 5
Connecting the Traditional Way 6
Connecting With URL 7
URL Syntax 7
Connecting the J2EE Way 9
Deploying Mimer JDBC in JNDI 9
Deploying Mimer JDBC in a Connection Pool 10
Deploying Mimer JDBC in Distributed Transaction Environments 10
Error Handling 10
The Class SQLException 11
The Class SQLWarning 11
Viewing Driver Characteristics 12
The mimcomm JNI library 12
Java Program Examples 13
JDBC Application Example 13
JDBC Application Example for J2EE 14
Using the Driver from Applets 15
Executing the Java Applet Example 16
Chapter 3 Programming With JDBC 17
Examples in this Chapter 17
Transaction Processing 17
JDBC Transactions 17
Auto-commit Mode 17
Manual-commit Mode 18

Setting the Transaction Isolation Level 19
Executing an SQL Statement 19
Using a Statement Object 19
Using a PreparedStatement Object 19
Using a CallableStatement Object 20
Batch Update Operations 20
Enhancing Performance 21
Result Set Processing 22
Scrolling in Result Sets 23
Positioning the Cursor 23
Result Set Capabilities 24
Holdable cursors 24
Updating Data 24
User-Defined Types 25
Default Type Mapping 25
Custom Java Classes With Type Mapping 26
Programming Considerations 27
Interval Data 27
Closing Objects 27
Increasing Performance 27

Appendix A Change History 29

New Functions 29
New Functions in 3.42 29
New Functions in 3.41 29
New Functions in 3.39 29
New Functions in 3.38 30
New Functions in 3.35 30
New Functions in 3.31 30
New Functions in 3.30 30
New Functions in 3.28 31
New Functions in 3.26 31
New Functions in 3.25 32
New Functions in 3.24 and 2.24 32
New Functions in 3.18, 2.18 and 1.18 32
New Functions in 3.17, 2.17 and 1.17 32
New Functions in 3.16, 2.16 and 1.16 32
New Functions in 3.15 32
New Functions in 2.9 32
New Functions in 2.8 33
New Functions in 2.7 33
New Functions in 2.5 33
New Functions in 2.4 33
New Functions in 2.3 33
New Functions in 2.0 33
New Functions in 1.9 33
New Functions in 1.7 34
New Functions in 1.2 34

Changed Functions **34**
Changes in 3.41 34
Changes in 3.40 34
Changes in 3.39 34
Changes in 3.29 35
Changes in 3.28 35
Changes in 3.25 35
Changes in 3.24, 2.24 and 1.24 35
Changes in 3.20, 2.20 and 1.20 36
Changes in 3.16, 2.16 and 1.16 36
Changes in 2.15 and 1.15 36
Changes in 2.14 and 1.14 37
Changes in 2.9 37
Changes in 2.7 37
Changes in 2.2 38
Changes in 2.1 38
Changes in 1.3 38
Changes in 1.2 38
Corrected Problems **39**
Corrections in 3.42 39
Corrections in 3.41a 39
Corrections in 3.40 40
Corrections in 3.39 40
Corrections in 3.38 41
Corrections in 3.37 42
Corrections in 3.35 42
Corrections in 3.31 42
Corrections in 3.30 44
Corrections in 3.29 44
Corrections in 3.28 45
Corrections in 3.27 45
Corrections in 3.26 45
Corrections in 3.25 46
Corrections in 3.24, 2.24 and 1.24 46
Corrections in 3.23 and 2.23 47
Corrections in 3.23, 2.23 and 1.23 47
Corrections in 3.22, 2.22 and 1.22 47
Corrections in 3.21, 2.21 and 1.21 47
Corrections in 3.20, 2.20 and 1.20 47
Correction in 3.19, 2.19 and 1.19 48
Corrections in 3.18, 2.18 and 1.18 48
Corrections in 3.16, 2.16 and 1.16 48
Corrections in 2.14 49
Corrections in 2.14 and 1.14 49
Corrections in 2.13 and 1.13 50
Corrections in 2.12 and 1.12 50
Corrections in 2.11 and 1.11 50
Corrections in 2.10 and 1.10 51
Corrections in 2.9 51
Corrections in 2.7 51
Corrections in 2.6 52
Corrections in 2.2 52

Corrections in 1.9 52
Corrections in 1.7 52
Known Restrictions 53
Known Problems 55
Index 57

Chapter 1

Introduction

Mimer JDBC Drivers provide access to Mimer SQL databases from Java applications and applets. The drivers are type 4 drivers, which means that they are written entirely in Java. As they are written in Java, they can be downloaded in applets.

Mimer JDBC Drivers can also be used on all platforms that support Java Virtual Machine (JVM) and so provide a very high degree of portability.

About this Guide

The guide is intended for Java application developers working with Mimer SQL. It covers all available Mimer JDBC drivers.

The guide describes the usage of SQL in Java applications, and provides, together with the *Mimer SQL Reference Manual*, the complete reference material for Mimer SQL.

To read more about JDBC and JVM, visit https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html.

The JDBC API specification implemented by this driver (packages `java.sql` and `javax.sql`) is found at https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html and https://docs.oracle.com/javase/8/docs/api/javax/sql/package-summary.html.

Definitions, Terms and Trademarks

API	Application Programming Interface
EJB	Enterprise Java Beans
JCP	Java Community Process
JDBC	The Java database API
JDK	Java Development Kit
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
OCC	Optimistic Concurrency Control

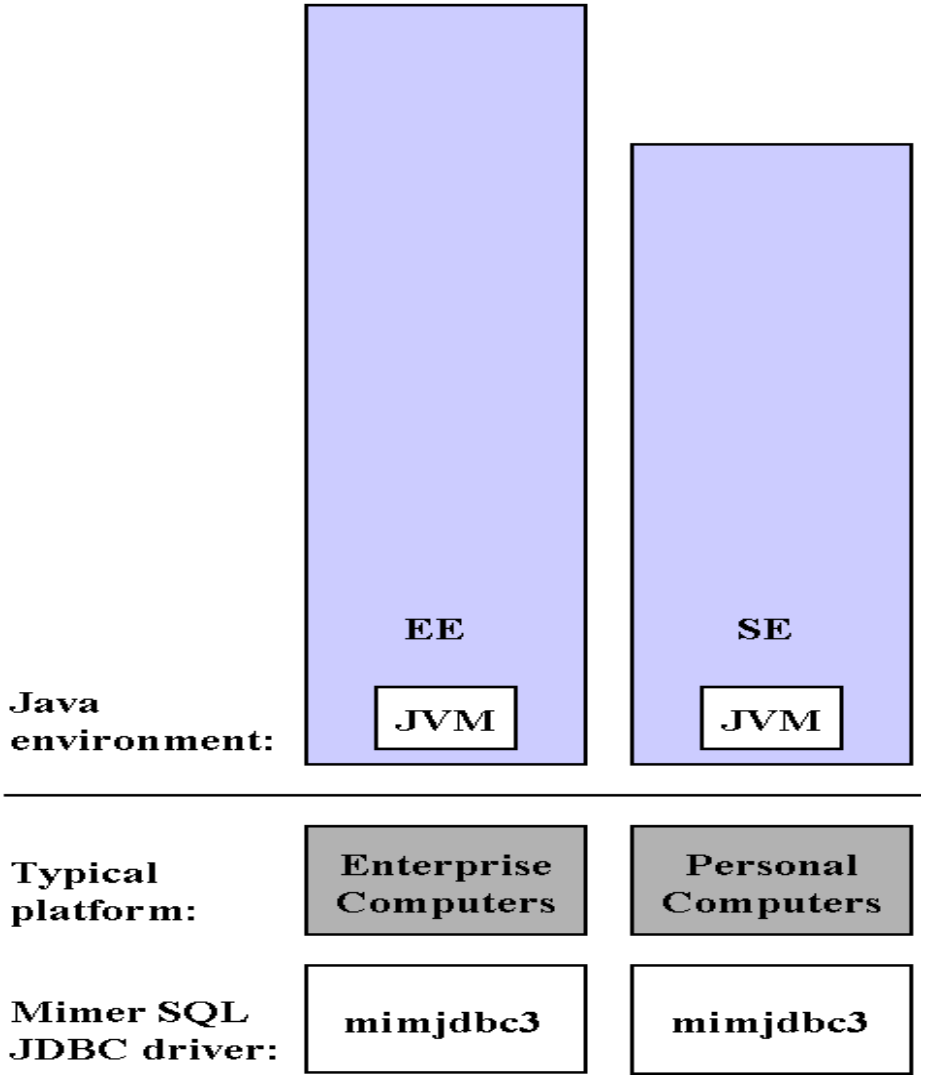
PSM	Persistent Stored Modules, the term used by ISO/ANSI for stored procedures
SQL	Structured Query Language
URL	Uniform Resource Locator

Requirements

- Mimer SQL server version 9.2 or later
- JRE 1.4 or later

Environment

Mimer JDBC has a complete range of functionality and support for the smallest devices to the high-end systems and application servers. In the picture below the various Java environments are described and coupled with computer environments and Mimer JDBC drivers:



Logging

To keep the driver size small and to optimize performance the Mimer JDBC drivers do not perform any logging. For logging, we provide a separate driver, Mimer JDBC Trace driver.

Mimer JDBC Trace driver is a full JDBC Driver that covers all of JDBC by calling the matching routines of the logged JDBC Driver.

It produces a log of every JDBC call an application makes, and also measures the elapsed time for each call. The log can be written to a file, or can be displayed directly in a window.

For more information, see https://developer.mimer.com/features/database-apis/jdbc/.

Chapter 2

Using the Mimer JDBC Driver

This chapter explains how to load the Mimer JDBC driver and how to connect to a Mimer SQL database. It also contains JDBC application examples and discusses driver characteristics.

Loading a Driver

To use the Mimer JDBC driver, it must be loaded into the Java environment. The Java environment locates a driver by a search along the class path, defined in the `CLASSPATH` environment variable.

The `CLASSPATH` environment variable informs the Java environment where to find Java class files, such as the Mimer JDBC drivers.

The Mimer JDBC driver jar file, including the directory specification, should be added to the Java class path, as can be seen in the following examples:

UNIX:
```
# echo $CLASSPATH
CLASSPATH=.:/usr/lib/mimjdbc3.jar
```

Win:
```
% set CLASSPATH=.;D:\MIMJDBC3.JAR
```

Besides defining the `CLASSPATH` environment variable explicitly, it can also be defined for a specific session when executing the application. For example:

```
java -classpath /usr/lib/mimjdbc3.jar JdbcApplication
```

Connecting the Traditional Way

The connection provides the link between the application and the Mimer SQL database server. To make a connection using the `DriverManager` class requires two operations, i.e. loading the driver and making the connection.

The class name of the Mimer JDBC Driver is:

```
com.mimer.jdbc.Driver
```

The class name of the Mimer JDBC Trace Driver is:

```
com.mimer.jtrace.Driver
```

The jar file referenced in the `CLASSPATH` determines which driver is loaded.

A driver can be explicitly loaded using the standard `Class.forName` method:

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

…

try {
        Class.forName("com.mimer.jdbc.Driver");

    } catch (java.lang.ClassNotFoundException cnf) {
        System.err.println("JDBC driver not found");
        return;
    }
```

Alternatively, `DriverManager`, when it initializes, looks for a `jdbc.drivers` property in the system properties. The `jdbc.drivers` property is a colon-separated list of drivers.

The `DriverManager` attempts to load each of the named drivers in this list of drivers. The `jdbc.drivers` property can be set like any other Java property, by using the `-D` option:

```
java -Djdbc.drivers=com.mimer.jdbc.Driver class
```

The property can also be set from within the Java application or applet:

```
Properties prp = System.getProperties();
prp.put("jdbc.drivers",
          "com.mimer.jdbc.Driver:com.mimer.jtrace.Driver");
System.setProperties(prp);
```

Note: Neither of the mechanisms used to load the driver specify that the application will actually use the driver. The driver is merely loaded and registered with the DriverManager.

Connecting With URL

To make the actual database connection, a URL string is passed to the `DriverManager.getConnection` method in the JDBC management layer.

The URL defines the data source to connect to. The JDBC management layer locates a registered driver that can connect to the database represented by the URL.

URL Syntax

The Mimer JDBC drivers support the following URL syntax:

```
jdbc:mimer:[protocol:][URL-field-list][property-list]
```

`URL-field-list` options can be combined with `property-list` options.

Protocol

If a protocol is specified, the driver will load the mimcomm JNI library and use native routines to connect to the database. If the protocol is not specified (or is an empty string), no JNI library will be loaded and a TCP/IP connection will be made using standard Java network packages in you Java runtime.

Supported protocols include:

protocol	Explanation
`decnet`	Use Decnet to connect to a remote server (VMS only).
`local`	Use shared memory communication to a server that runs on your local machine. This protocol is often much faster than TCP/IP-based communication.
`rapi`	Use the RAPI protocol to connect to mobile devices (Windows only).
`native tcp`	Connect to the server using TCP/IP, but through the mimcomm JNI library.
`tcp`	Connect to the server using the Java TCP/IP stack.
`single`	Open a database in SINGLE mode.

URL-field-list

All fields in the `URL-field-list` are optional.

The database server host computer, with or without a user specification, is introduced by `//` and the database name is introduced by `/`, like:

```
[//[user[:password]@]serverName[:portNumber]] [/databaseName]
```

A Connection object is returned from the `getConnection` method, for example:

```
String url1 = "jdbc:mimer://MIMER_ADM:admin@localhost/ExampleDB”;
String url2 = ”jdbc:mimer:local://MIMER_ADM:admin@/ExampleDB”;
Connection con1 = DriverManager.getConnection(url1);
Connection con2 = DriverManager.getConnection(url2);
```

Alternatively, the `getConnection` method allows the user name and password to be passed as parameters:

```
url = "jdbc:mimer://localhost/ExampleDB";
con = DriverManager.getConnection(url, "MIMER_ADM", "admin");
```

Property-list

The `property-list` for the Mimer JDBC Driver is optional. The list is introduced by a leading question mark `?` and where there are several properties defined they are separated by ampersands `&`, like:

```
?property=value[&property=value[&property=value]]
```

The following properties are supported:

Property	Explanation
`databaseName`	Name of database server to access
`user`	User name used to log in to the database
`password`	Password used for the login
`serverName`	Computer on which the database server is running, the default is `localhost`
`protocol`	The protocol to use when connecting. If set, load the mimcomm JNI library. If empty, use standard Java TCP/IP support.
`portNumber`	Port number to use on the database server host, the default is `1360`
`program`	Program name used to log in to the database
`programPwd`	Password used for the program

The following example demonstrates a connection using the driver properties:

```
url = "jdbc:mimer:?databaseName=ExampleDB"
               + "&user=MIMER_ADM"
               + "&password=admin"
               + "&serverName=srv2.mimer.com";
con = DriverManager.getConnection(url);
```

Alternatively a `java.util.Properties` object can be used:

```
Properties dbProp = new Properties();

dbProp.put("databaseName", "ExampleDB");
dbProp.put("user", "MIMER_ADM");
dbProp.put("password", "admin");
con = DriverManager.getConnection("jdbc:mimer:", dbProp);
```

Elements from the `URL-field-list` and the `property-list` can be combined:

```
url = "jdbc:mimer:/ExampleDB"
               + "?user=MIMER_ADM"
               + "&password=admin";
```

The `DriverPropertyInfo` class is available for programmers who need to interact with a driver to discover the properties that are required to make a connection. This enables a generic GUI tool to prompt the user for the Mimer SQL connection properties:

```
Driver drv;
DriverPropertyInfo [] drvInfo;

drv = DriverManager.getDriver("jdbc:mimer:");
drvInfo = drv.getPropertyInfo("jdbc:mimer:", null);
for (int i = 0; i < drvInfo.length; i++) {
    System.out.println(drvInfo[i].name + ": " + drvInfo[i].value);
}
```

After connecting to the database, all sorts of information about the driver and database is available through the use of the `getMetadata` method:

```
DatabaseMetaData dbmd;

dbmd = con.getMetaData();

System.out.println("Driver      " + dbmd.getDriverName());
System.out.println("    Version " + dbmd.getDriverVersion());
System.out.println("Database    " + dbmd.getDatabaseProductName());
System.out.println("    Version " + dbmd.getDatabaseProductVersion ());
con.close();
```

The `close` method tells JDBC to disconnect from the Mimer SQL database server. JDBC resources are also released.

It is usual for connections to be explicitly closed when no longer required. The normal Java garbage collection has no way of freeing external resources, such as the Mimer SQL database server.

Connecting the J2EE Way

Along with J2EE came a new way for JDBC drivers to connect to database servers. Instead of requesting connections through the `java.sql.DriverManager` class, applications should connect using the `javax.sql.DataSource`, `com.mimer.jdbc.MimerConnectionPoolDataSource` or `com.mimer.jdbc.MimerXADataSource` interfaces.

Deploying Mimer JDBC in JNDI

The Mimer DataSource class is `com.mimer.jdbc.MimerDataSource`. When applications are deployed within the J2EE environment, a properly initiated MimerDataSource object should be stored in JNDI for the application server to retrieve at runtime. Application servers may use the JavaBean interface to obtain configuration parameters for MimerDataSource objects.

These are the DataSource attributes recognized by the Mimer JDBC drivers:

DataSource Attributes	Description
`serverName`	The computer on which the database server is running, the default is `localhost`
`portNumber`	The port number to use on the server host, the default is `1360`
`description`	A textual description
`databaseName`	The name of the database on the server (required)
`user`	User name
`password`	Password
`protocol`	The protocol to use when connecting via the mimcomm JNI library

DataSource Attributes	Description
`serverName`	The computer on which the database server is running, the default is `localhost`
`service`	The service to connect to. This field plays the same role as the portNumber field, but any string can be used for protocols that don't use integer-valued port numbers (such as Decnet or named pipes). If a service value is specified, any portNumber value is ignored.

See sample programs further down for programming examples.

Deploying Mimer JDBC in a Connection Pool

Mimer JDBC may be deployed in J2EE compliant connection pools.

When deploying Mimer JDBC in a connection pool, the class `com.mimer.jdbc.MimerConnectionPoolDataSource` should be used. This class features the same attributes as described above for `com.mimer.jdbc.MimerDataSource`.

Deploying Mimer JDBC in Distributed Transaction Environments

Mimer JDBC may be used in J2EE compliant distributed transaction environments.

When deploying Mimer JDBC to be used in distributed transactions, the class `com.mimer.jdbc.MimerXADataSource` should be used. Whenever connections are created using this factory class, Mimer SQL may cooperate in transactions with any other XA compliant database server.

Read more about Mimer SQL and distributed transactions in *Mimer SQL Programmer's Manual*.

Error Handling

Error handling is taken care of by using the classes `SQLException` and `SQLWarning`.

The Mimer JDBC specific error codes are in the range -22000 to -22999. When using Java, the error message is always included in the exception that is thrown.

To get the complete and accurate list of error codes, execute the following command:

```
$ java com.mimer.jdbc.Driver -errors
```

The Class SQLException

The `SQLException` class provides information relating to database errors. Details include a textual description of the error, an SQLState string, and an error code. There may be a number of `SQLException` objects for a failure.

```
try {

…

} catch(SQLException sqe) {
    SQLException stk;

    stk = sqe;   // Save initial exception for stack trace

    System.err.println("\n*** SQLException:\n");
    while (sqe != null) {
        System.err.println("Message:     " + sqe.getMessage());
        System.err.println("SQLState:    " + sqe.getSQLState());
        System.err.println("NativeError: " + sqe.getErrorCode());
        System.err.println();

        sqe = sqe.getNextException();
    }

    stk.printStackTrace(System.err);
}
```

The Class SQLWarning

The `SQLWarning` class provides information relating to database warnings. The difference between warnings and exceptions is that warnings, unlike exceptions, are not thrown.

The `getWarnings` method of the appropriate object (`Connection`, `Statement` or `ResultSet`) is used to determine whether warnings exist.

Warning information can be retrieved using the same mechanisms as in the `SQLException` example above but with the method `getNextWarning` retrieving the next warning in the chain:

```
con = DriverManager.getConnection(url);
checkSQLWarning(con.getWarnings());

…

private static boolean checkSQLWarning( SQLWarning sqw )
throws SQLException {
    boolean rc = false;

    if (sqw != null) {
        rc = true;

        System.err.println("\n*** SQLWarning:\n");
        while (sqw != null) {
            System.err.println("Message:     " + sqw.getMessage());
            System.err.println("SQLState:    " + sqw.getSQLState());
            System.err.println("NativeError: " + sqw.getErrorCode());
            System.err.println();

            sqw = sqw.getNextWarning();
        }
    }

    return rc;
}
```

Viewing Driver Characteristics

By using the `java com.mimer.jdbc.Driver` command, you can view characteristics of a specific driver and the current environment:

```
java com.mimer.jdbc.Driver options
```

The options available are:

Option	Description
`-version`	Display driver version
`-sysprop`	Display all system properties
`-errors`	List all JDBC error codes
`-ping url`	Test the database at the specified url
`-mimcomm`	Load the mimcomm JNI library and show its version number. Displays informational messages to help fix any problems.

The following is an example that uses the `-version` option:

```
java com.mimer.jdbc.Driver -version
Mimer JDBC driver version 3.31
```

Used without any arguments, the command will display usage information.

The mimcomm JNI library

The Mimer JDBC driver can be used in a 100% native Java environment. In this case, the connection to a Mimer database server is done by the TCP/IP support included in the Java platform.

However, it is also possible to load an external library called mimcomm that includes support for all the communication protocols available on the particular platform. Please note that the mimcomm library may not be available on platforms that don't have a recent version of Mimer SQL installed.

The name of the mimcomm library varies between platforms. It is called mimcomm.dll on Windows, libmimcomm.so on Unix and MIMCOMM.EXE on VMS.

When you install a Mimer SQL distribution, the mimcomm library will normally be installed in a place where the Java environment can find it. You can test this by using the `-mimcomm` switch as a command line argument to the JDBC driver:

```
unix$ java -cp mimjdbc3.jar com.mimer.jdbc.Driver -mimcomm
System.getProperty("java.library.path"):
/opt/java/64/jdk1.6.0_31/jre/lib/amd64/server:/opt/java/64/jdk1.6.0_31/jre/
lib/amd64:/opt/java/64/jdk1.6.0_31/jre/../lib/amd64:/usr/java/packages/lib/
amd64:/usr/lib64:/lib64:/lib:/usr/lib

System.loadLibrary("mimcomm"):

mimcomm library Version: V1011A
JNI parameter method:    JNI_COPY
```

When the JDBC driver loads the mimcomm library, it looks for the library in the path specified by the java.library.path system property. If the JDBC driver cannot find the library in the path listed, you should either move the mimcomm library to a directory listed in the path or consult your Java manual for instructions on how to change the `java.library.path` system property.

Java Program Examples

Below are a collection of small basic Java programs for different environments, showing a database connection and a simple database operation with some error handling.

JDBC Application Example

The example Java program below creates a result set containing all rows of the data dictionary view `INFORMATION_SCHEMA.TABLES`, then each row is fetched and displayed on standard output.

In this example, the user name and password are given separately using the `DriverManager.getConnection` method, i.e. not given in the URL specification.

The below example will work with the mimjdbc drivers.

```
import java.sql.*;

class Example
{
    public static void main(String[] args)
    {
        try {
            Class.forName("com.mimer.jdbc.Driver");
            String url = "jdbc:mimer://my_node.mimer.se/customers";
            Connection con = DriverManager.getConnection(url,
                                                         "SYSADM","SYSPW");
            Statement stmt = con.createStatement();
            String sql = "select TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
                          from INFORMATION_SCHEMA.TABLES";
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                String schema = rs.getString(1);
                String name = rs.getString(2);
                String type = rs.getString(3);
                System.out.println(schema + "  " + name + "  " + type);
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (SQLException e) {
            System.out.println("SQLException!");
            while (e != null) {
                System.out.println("SQLState  : " + e.getSQLState());
                System.out.println("Message   : " + e.getMessage());
                System.out.println("ErrorCode : " + e.getErrorCode());
                e = e.getNextException();
                System.out.println("");
            }
        } catch (Exception e) {
            System.out.println("Other Exception");
            e.printStackTrace();
        }
    }
}
```

Another way to provide connection properties is to supply a `java.util.Properties` object to the `DriverManager.getConnection` method.

JDBC Application Example for J2EE

This example Java program deploys a `com.mimer.jdbc.MimerDataSource` in a file system JNDI repository. Note that the file system JNDI repository has to be downloaded. It is available for download at https://www.oracle.com/technetwork/java/index.html. At this site, several other service providers may be downloaded as well.

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

public class RegisterJNDI
{
    public static void main(String argv[])
    {
        try {
            com.mimer.jdbc.MimerDataSource ds =
                new com.mimer.jdbc.MimerDataSource();

            ds.setDescription("Our Mimer data source");
            ds.setServerName("my_node.mimer.se");
            ds.setDatabaseName("customers");
            ds.setPortNumber("1360");
            ds.setUser("SYSADM");
            ds.setPassword("SYSPW");

            // Set up environment for creating initial context
            Hashtable env = new Hashtable();

            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.fscontext.RefFSContextFactory");
            env.put(Context.PROVIDER_URL, "file:.");
            Context ctx = new InitialContext(env);

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

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

Once the data source is deployed, applications may connect using the deployed `DataSource` object. For instance like the below code snippet:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:.");
Context ctx = new InitialContext(env);
DataSource ds = (DataSource)ctx.lookup("jdbc/customers");
return ds.getConnection();
```

Using the Driver from Applets

The example Java applet below creates a result set containing all rows of the data dictionary view `INFORMATION_SCHEMA.TABLES`, then each row is fetched and displayed on standard output.

In this example, the user name and password are given separately using the `DriverManager.getConnection` method, i.e. not given in the URL specification.

The example will work with the mimjdbc drivers.

```
import java.sql.*;
import java.applet.*;
import java.awt.*;

public class ExampleApplet extends java.applet.Applet {
    public void init() {
        resize(1200, 600);
    }

    public void paint(Graphics g) {
        int row = 1;
        g.drawString("Listing tables:", 20, 10 * row++);
        try {
            Class.forName("com.mimer.jdbc.Driver");
            String url = "jdbc:mimer://my_node.mimer.se/customers";
            Connection con = DriverManager.getConnection(url, "SYSADM",
                                                         "SYSPW");
            Statement stmt = con.createStatement();
            String sql = "select TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
                          from INFORMATION_SCHEMA.TABLES";
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                String schema = rs.getString(1);
                String name = rs.getString(2);
                String type = rs.getString(3);
                g.drawString(schema + "  " + name + "  " + type, 50,
                             10 * row++);
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (SQLException e) {
            g.drawString("SQLException!", 20, 10 * row++);
            while (e != null) {
                g.drawString("SQLState  : " + e.getSQLState(), 20,
                             10 * row++);
                g.drawString("Message   : " + e.getMessage(), 20,
                             10 * row++);
                g.drawString("ErrorCode : " + e.getErrorCode(), 20,
                             10 * row++);
                e = e.getNextException();
                g.drawString("", 20, 10*row++);
            }
        } catch (Exception e) {
            g.drawString("Other Exception!", 20, 10 * row++);
            g.drawString(e.toString(), 20, 10 * row++);
        }
    }
}
```

Executing the Java Applet Example

To use a Mimer JDBC Driver in a Java applet, copy the driver jar file to the directory containing the applet's Java classes.

This directory must be accessible to the Web server. The driver jar file name should be given as the applet tag's `ARCHIVE` parameter in the HTML file. For example:

```
<html>
 <head>
  <title> The Example Applet
 </head>
 <body>
 Example Applet:
  <applet archive="mimjdbc3.jar"
          code="ExampleApplet.class"
          width=800
          height=600>
  </applet>
 </body>
</html>
```

You execute the applet by accessing the HTML file from a browser, for example:

```
http://my_node/ExampleApplet.html
```

Note: There is a security restriction for Java applets, which states that a network connection can only be opened to the host from which the applet itself was downloaded. This means that both the Web server distributing the applet code and the database server must reside on the same host computer.

Chapter 3

Programming With JDBC

This chapter describes some programming aspects when using the Mimer JDBC Driver.

We recommend you to read *JDBCBench, a Java Database Case Study* available on our developer web site https://developer.mimer.com/article/jdbcbench-a-java-database-case-study/.

Examples in this Chapter

The examples are based on the sample schema that is provided as part of the Mimer SQL distribution. They assume that the example database environment has been created.

Transaction Processing

Mimer SQL uses a method for transaction management called Optimistic Concurrency Control. OCC does not involve any locking of rows as such, and therefore cannot cause a deadlock.

JDBC Transactions

JDBC transactions are controlled through the `Connection` object. There are two modes for managing transactions within JDBC:

- auto-commit
- manual-commit.

The `setAutoCommit` method is used to switch between the two modes.

Auto-commit Mode

Auto-commit mode is the default transaction mode for JDBC. When a connection is made, it is in auto-commit mode until `setAutoCommit` is used to disable auto-commit.

In auto-commit mode each individual statement is automatically committed when it completes successfully, no explicit transaction management is necessary. However, the return code must still be checked, as it is possible for the implicit transaction to fail.

Manual-commit Mode

When auto-commit is disabled, i.e. manual-commit is set, all executed statements are included in the same transaction until it is explicitly completed.

When an application turns auto-commit off, the next statement against the database starts a transaction. The transaction continues either the `commit` or the `rollback` method is called. The next command sent to the database after that starts a new transaction.

Calling the `commit` method ends the transaction. At that stage, Mimer SQL checks whether the transaction is valid and raises an exception if a conflict is identified.

If a conflict is encountered, the application determines how to continue, for example whether to automatically retry the transaction or inform the user of the failure. The application is notified about the conflict by an exception that must be caught and evaluated.

A request to rollback a transaction causes Mimer SQL to discard any changes made since the start of the transaction and to end the transaction.

Use the `commit` or `rollback` methods, rather than using the `SQL COMMIT` or `ROLLBACK` statements to complete transactions, for example:

```
Statement stmt;
int transactionAttempts;

final int MAX_ATTEMPTS = 5; // Maximum transaction attempts

// Open a connection
url = "jdbc:mimer:/ExampleDB";
con = DriverManager.getConnection(url, "MIMER_ADM", "admin");

con.setAutoCommit(false); // Explicit transaction handling

stmt = con.createStatement();

// Loop until transaction successful (or max attempts exceeded)
for (transactionAttempts = 1; ; transactionAttempts++) {
    // Perform an operation under transaction control
    stmt.executeUpdate("UPDATE mimer_store.currencies"
                     + "   SET exchange_rate = exchange_rate * 1.05"
                     + "   WHERE code = 'USD'");

    try {
        con.commit(); // Commit transaction

        System.out.println("Transaction successful");
        break;

    } catch(SQLException sqe) {
        // Check commit error - allow serialization failure
        if (sqe.getSQLState().equals("40001")) {
            // Check number of times the transaction has been attempted
            if (transactionAttempts >= MAX_ATTEMPTS) {
                // Raise exception with application defined SQL state
                throw new SQLException("Transaction failure", "UET01");
            }
        }
        else {
            // Raise all other exceptions to outer handler
            throw sqe;
        }
    } finally {
        con.close();
    }
}
```

Setting the Transaction Isolation Level

The `setTransactionIsolation` method sets the transaction isolation level. The default isolation level for Mimer SQL is `TRANSACTION_REPEATABLE_READ`.

Note: With Enterprise Java Beans, the EJB environment provides the transaction management and therefore explicit transaction management is not required.

Executing an SQL Statement

The `Connection` object supports three types of Statement objects that can be used to execute an SQL statement or stored procedure:

- a Statement object is used to send SQL statements to the database
- the PreparedStatement interface inherits from Statement
- the CallableStatement object inherits both Statement and PreparedStatement methods.

Using a Statement Object

The Connection method `createStatement` is used to create a Statement object that can be used to execute SQL statements without parameters.

The `executeUpdate` method of the Statement object is used to execute an SQL DELETE, INSERT, or UPDATE statement, i.e. a statement that does not return a result set, it returns an `int` indicating the number of rows affected by the statement, for example:

```
int rowCount;

stmt = con.createStatement();

rowCount = stmt.executeUpdate(
                  "UPDATE mimer_store.currencies"
                + "   SET exchange_rate = exchange_rate * 1.05"
                + "   WHERE code = 'USD'");

System.out.println(rowCount + " rows have been updated");
```

Using a PreparedStatement Object

Where an SQL statement is being repeatedly executed, a `PreparedStatement` object is more efficient than repeated use of the `executeUpdate` method against a Statement object.

In this case the values for the parameters in the SQL statement (indicated by `?`) are supplied with the `set`*`XXX`* method, where *`XXX`* is the appropriate type for the parameter.

For example:

```
PreparedStatement pstmt;
int rowCount;

pstmt = con.prepareStatement(
                  "UPDATE mimer_store.currencies"
                + "   SET exchange_rate = exchange_rate * ?"
                + "   WHERE code = ?");

pstmt.setFloat(1, 1.05f);
pstmt.setString(2, "USD");
rowCount = pstmt.executeUpdate();

pstmt.setFloat(1, 1.08f);
pstmt.setString(2, "GBP");
rowCount = pstmt.executeUpdate();
```

Using a CallableStatement Object

Similarly, when using stored procedures, a `CallableStatement` object allows parameter values to be supplied, for example:

```
CallableStatement cstmt;

cstmt = con.prepareCall("CALL mimer_store.order_item( ?, ?, ? )");

cstmt.setInt(1, 700001);
cstmt.setInt(2, 60158);
cstmt.setInt(3, 2);
cstmt.executeUpdate();
```

The `setNull` method allows a JDBC null value to be specified as an `IN` parameter. Alternatively, use a Java null value with a `set`*`XXX`* method.

For example:

```
pstmt.setString(4, null);
```

A more complicated example illustrates how to handle an output parameter:

```
CallableStatement cstmt;

cstmt = con.prepareCall("CALL mimer_store.age_of_adult( ?, ? )");

cstmt.setString(1, "US");
cstmt.registerOutParameter(2, Types.CHAR);

cstmt.executeUpdate();
System.out.println(cstmt.getString(2) + " years");
```

Batch Update Operations

JDBC provides support for batch update operations. The `BatchUpdateException` class provides information about errors that occur during a batch update using the Statement method `executeBatch`.

The class inherits all the method from the class `SQLException` and also the method `getUpdateCounts` which returns an array of update counts for those commands in the batch that were executed successfully before the error was encountered.

For example:

```
try {

…

} catch(BatchUpdateException bue) {
    System.err.println("\n*** BatchUpdateException:\n");

    int [] affectedCount = bue.getUpdateCounts();
    for (int i = 0; i < affectedCount.length; i++) {
        System.err.print(affectedCount[i] + "  ");
    }
    System.err.println();

    System.err.println("Message:     " + bue.getMessage());
    System.err.println("SQLState:    " + bue.getSQLState());
    System.err.println("NativeError: " + bue.getErrorCode());
    System.err.println();

    SQLException sqe = bue.getNextException();
    while (sqe != null) {
        System.err.println("Message:     " + sqe.getMessage());
        System.err.println("SQLState:    " + sqe.getSQLState());
        System.err.println("NativeError: " + sqe.getErrorCode());
        System.err.println();

        sqe = sqe.getNextException();
    }
}
```

Note: The `BatchUpdateException` object points to a chain of `SQLException` objects.

Enhancing Performance

The batch update functionality allows the statement objects to support the submission of a number of update commands as a single batch.

The ability to batch a number of commands together can have significant performance benefits. The methods `addBatch`, `clearBatch` and `executeBatch` are used in processing batch updates.

The `PreparedStatement` example above could be simply rewritten to batch the commands.

For example:

```
PreparedStatement pstmt;
int [] affectedCount;

pstmt = con.prepareStatement(
                  "UPDATE mimer_store.currencies"
                + "   SET exchange_rate = exchange_rate * ?"
                + "   WHERE code = ?");

pstmt.setFloat(1, 1.05f);
pstmt.setString(2, "USD");
pstmt.addBatch();

pstmt.setFloat(1, 1.08f);
pstmt.setString(2, "GBP");
pstmt.addBatch();

affectedCount = pstmt.executeBatch();
```

The Mimer SQL database server executes each command in the order it was added to the batch and returns an update count for each completed command.

If an error is encountered while a command in the batch is being processed then a `BatchUpdateException` is thrown (see *Error Handling* on page *10*) and the unprocessed commands in the batch are ignored.

In general it may be advisable to treat all the commands in the batch as a single transaction, allowing the application to have control over whether those commands that succeeded are committed or not.

Set the Connection object's auto-commit mode to off to group the statements together in a single transaction. The application can then commit or rollback the transaction as required.

Calling the method `clearBatch` clears a Statement object's list of commands.

Using the `Close` method to close any of the Statement objects releases the database and JDBC resources immediately. It is recommended that Statement objects be explicitly closed as soon as they are no longer required.

Result Set Processing

There are a number of ways of returning a result set. Perhaps the simplest is as the result of executing an SQL statement using the `executeQuery` method, for example:

```
Statement stmt;
ResultSet rs;

stmt = con.createStatement();

rs = stmt.executeQuery("SELECT *"
                     + "   FROM mimer_store.currencies");

while (rs.next()) {
    System.out.println(rs.getString("CURRENCY"));
```

A ResultSet can be thought of as an array of rows. The 'current row' is the row being examined and manipulated at any given time, and the location in the ResultSet is the 'current row position'.

Information about the columns in a result set can be retrieved from the metadata, for example:

```
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmd;

stmt = con.createStatement();

rs = stmt.executeQuery("SELECT *"
                     + "   FROM mimer_store.currencies");

rsmd = rs.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(" Type: " + rsmd.getColumnTypeName(i));
    System.out.println(" Size: " + rsmd.getColumnDisplaySize(i));
}
```

Scrolling in Result Sets

The previous examples used forward-only cursors (`TYPE_FORWARD_ONLY`), which means that they only support fetching rows serially from the start to the end of the cursor, this is the default cursor type.

In modern, screen-based applications, the user expects to be able to scroll backwards and forwards through the data. While it is possible to cache small result sets in memory on the client, this is not feasible when dealing with large result sets. Support for scrollable cursors provide the answer.

Scrollable cursors allow you to move forward and back as well as to a particular row within the ResultSet. With scrollable cursors it is possible to iterate through the result set many times.

The Mimer drivers' scrollable cursors are of type `TYPE_SCROLL_INSENSITIVE`, which means that the result set is scrollable but also that the result set does not show changes that have been made to the underlying database by other users, i.e. the view of the database is consistent. To allow changes to be reflected may cause logically inconsistent results.

Positioning the Cursor

There are a number of methods provided to position the cursor:

- `absolute`
- `afterLast`
- `beforeFirst`
- `first`
- `last`
- `next`
- `previous`
- `relative`

There are also methods to determine the current position of the cursor:

- `isAfterLast`
- `isBeforeFirst`
- `isFirst`
- `isLast`

The `getRow` method returns the current cursor position, starting from 1. This provides a simple means of finding the number of rows in a result set.

For example:

```
Statement stmt;
ResultSet rs;

stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_READ_ONLY);

rs = stmt.executeQuery("SELECT code, currency"
                     + "   FROM mimer_store.currencies"
                     + "   WHERE code LIKE 'A%'");

System.out.println("\nOriginal sort order");
while (rs.next()) {
    System.out.println(rs.getString(1) + "  " + rs.getString(2));
}
```

```
System.out.println("\nReverse order");
while (rs.previous()) {
    System.out.println(rs.getString(1) + "  " + rs.getString(2));
}

rs.last();
System.out.println("\nThere are " + rs.getRow() + " rows");
```

The Mimer JDBC Driver will automatically perform a pre-fetch whenever a result set is created. This means that a number of rows are transferred to the client in a single communication across the network. If only a small number of rows are actually required use `setMaxRows` to limit the number of rows returned in the result set.

Result Set Capabilities

A instance of the `ResultSet` class is created when a query is executed. The capabilities of the result set depend on the arguments used with the `createStatement` (or `prepareStatement` or `prepareCall`) method.

The first argument defines the type of the `ResultSet`, whether it is scrollable or non-scrollable, and the second argument defines the concurrency option, i.e. the update capabilities.

A `ResultSet` should only be made updatable if the functionality is going to be used, otherwise the option `CONCUR_READ_ONLY` should be used. If used, both the type and the concurrency option must be specified.

The following example creates a scrollable result set cursor that is also updatable:

```
stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);
```

Even if the options used specify that the result set will be scrollable and updatable, it is possible that the actual SQL query will return a `ResultSet` that is non-scrollable or non-updatable.

Holdable cursors

The `mimjdbc3.jar` driver supports the JDBC 3 specification. As such it provides an opportunity for application developers to create holdable cursors. The difference between a holdable cursor and a regular cursor is that regular cursors are closed at the end of the transaction. The holdable cursor can (theoretically) stay opened for an unlimited period of time. However, leaving a cursor open for a long period of time may have serious performance implications for the same reason long lasting transactions may impair server performance.

Updating Data

Applications can update data by executing the `UPDATE`, `DELETE`, and `INSERT` statements. An alternative method is to position the cursor on a particular row and then use `DELETE CURRENT`, or `UPDATE CURRENT` statements.

The following example illustrates how this can be done:

```
Statement select;
PreparedStatement update;
ResultSet rs;

select = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                             ResultSet.CONCUR_UPDATABLE);

select.setCursorName("CRN"); /* Name the cursor */

rs = select.executeQuery("SELECT currency"
                       + "   FROM mimer_store.currencies"
                       + "   WHERE code = 'ALL'"
                       + "   FOR UPDATE OF currency");

update = con.prepareStatement("UPDATE mimer_store.currencies"
                            + "   SET currency = ?"
                            + "   WHERE CURRENT OF crn");

while (rs.next()) {
    if (rs.getString("CURRENCY").startsWith("Leke")) {
        update.setString(1, "Albanian Leke");
    }
    else {
         update.setString(1, "Leke");
    }
    update.executeUpdate();
}
```

User-Defined Types

Whenever the application working with user-defined types, a type mapping is used. For DISTINCT types, the default type mapping is given by the core SQL type which makes up the DISTINCT type. A structured type is mapped by default to a predefined JDBC interface which provides a basic functionality to work with its attributes.

Applications may alter the type mapping to integrate its own type classes with the database types. The custom type mapping will allow JDBC getter and setter methods to work directly with the classes in the application.

Default Type Mapping

By default, when fetching a user defined type from the database or supplying one to the database, the generic class `java.sql.Struct` is being used to hold the type attributes. Objects of this class simply holds an array of objects, each one corresponding the attribute in question.

For example, consider the following SQL type:

```
create type NAME as (TITLE nvarchar(20),
                     GIVEN_NAME nvarchar(50),
                     FAMILY_NAME nvarchar(50));
```

When retrieving columns of this type, the method `ResultSet.getObject` is used, which returns a `java.sql.Struct` object. For example:

```
ResultSet rs = stmt.executeQuery("select EMPLOYEE_NAME from EMPLOYEES");
while (rs.next()) {
   Struct employee_name = rs.getObject(1);
   Object[] employee_name_attributes = employee_name.getAttributes();
   String title = (String)employee_name_attributes[0];
   String given_name = (String)employee_name_attributes[1];
   String family_name = (String)employee_name_attributes[2];

   /* At this point the respective attributes are available in the
      above String objects for further processing. */
}
rs.close();
```

Custom Java Classes With Type Mapping

A more involved way is to map the SQL type against a Java class which implements the `java.sql.SQLData` interface. When mapping the SQL type name, the Java class might look like this:

```
import java.sql.*;

public class Name implements SQLData {
   String title,given_name,family_name;
   String type_name;

   /* SQLData interface routines. */

   public String getSQLTypeName() {
       return type_name;
   }

   public void readSQL(java.sql.SQLInput stream,String typeName) throws
java.sql.SQLException
   {
       type_name = typeName;
       title = stream.readString();
       given_name = stream.readString();
       family_name = stream.readString();
   }

   public void writeSQL(java.sql.SQLOutput stream) throws
java.sql.SQLException
   {
       stream.writeString(title);
       stream.writeString(given_name);
       stream.writeString(family_name);
   }

   /* Here follows additional methods to define the characteristics of
      this class.
    * This might be ordinary setter/getter methods for applications to use,
      for example the following
    */

   public String getCombinedName()
   {
       if (title!=null && title.length()>0)
           return title+" "+given_name+" "+family_name;
       return given_name+" "+family_name;
   }
```

```
    public String getTitle()
    {
        return title;
    }

    public void setTitle(String title)
    {
        this.title = title;
    }

    public String toString()
    {
        return getCombinedName();
    }
}
```

The application must register its own type mapping with the connection to make the JDBC driver aware of the custom class. Whenever the custom type map is activated, the JDBC methods `getObject` and `setObject` will return and accept parameters of the specified class, for example the following will create a map between the SQL type MYSCHEMA.NAME and the Java class Name.

```
java.util.Map map = con.getTypeMap();
map.put("MYSCHEMA.NAME", Class.forName("Name"));
con.setTypeMap(map);
```

Programming Considerations

Below is a summary of issues to be considered when programming with Mimer JDBC.

Interval Data

Both the JDBC specification and the Java language lack support for `INTERVAL` data types.

You can use the `getString` and `setString` methods for values accessed by a driver from database columns containing `INTERVAL` data.

Closing Objects

Although Java has automatic garbage collection, it is essential that you close JDBC objects, such as `ResultSets`, `Statements` and `Connections`, when done with them.

Closing objects gives your application better control over resources.

If you don't close objects, resources are kept allocated in the database server until garbage collection is triggered, this can exhaust server resources.

Increasing Performance

- **Use Stored Procedures**

  One of the main issues when trying to increase performance is to reduce network traffic. For example, you can increase performance by using the database server to return more accurate data instead of returning a large amount of unqualified data which your application must handle. You can achieve this by using more sophisticated SQL statements or by using stored procedures (PSM).

- **Use More Than One Connection**

  Mimer JDBC drivers are thread-safe and use one lock per connection. So, to achieve higher concurrency, an application should use several connections.

- **Prefetching Data**

  The drivers are implemented to perform automatic `prefetch`, i.e. whenever a `resultSet` is created, a buffer is filled with successive rows. This is an optimization for throughput, allowing more data to be fetched to the client in the single communication made.

  The flip side of the coin is that response time, i.e. the time it takes to receive the first record, may be increased (see *Use setMaxRows* below.)

- **Use setMaxRows**

  If you know that only a small number of records are to be fetched, then you can use the `setMaxRows` method to optimize the response time, i.e. to avoid an array fetch.

- **Use PreparedStatements**

  Another way of increasing performance is to avoid recompiling SQL statements unnecessarily. Whenever you use `Statement.execute`*`XXX`* methods, statements are compiled. Instead, use parameterized precompiled statements, i.e. `PreparedStatement`, whenever possible.

- **Use Batched Statements**

  Using the Mimer JDBC Driver version 2 or later, you can reduce the number of network requests by using batched statements.

  If, for example, you replace 20 calls to `Statement.execute()` with 20 calls to `Statement.addBatch()` followed by a `Statement.executeBatch()` call, 20 server network requests are replaced by a single network request.

  If response time is an issue, this simple change may give a twenty-fold performance improvement!

  Note that batched statements for `PreparedStatement` and `CallableStatement` differ from the implementation for the `Statement` class. When using `PreparedStatement` or `CallableStatement`, only a single SQL statement can be used in a batch. The `addBatch()` call (without argument) adds a set of parameters to the batch. When you use batches, the same SQL statement is called repeatedly with different parameters in a single network request.

  In versions 2 and later, you can use the `setFetchSize` method to control the amount of data fetched.

Appendix A

Change History

The following sections document changes in the drivers.

New Functions

This section describes the main new functions of each Mimer JDBC version.

New Functions in 3.42

- **LOAD statements**

  Support for `START LOAD`, `COMMIT LOAD` and `ROLLBACK LOAD` has been added. Executing these statements with an older JDBC driver will return the error -22053 "Statement type not recognized", when any of these statements is attempted.

New Functions in 3.41

- **Encrypted Network Communication**

  128-bit AES-GCM encryption may now be used for network communication between the server and the clients.

  For further information, check the Mimer SQL System Management Handbook, the section on Network Encryption under the chapter Managing a Database Server.

New Functions in 3.39

- **Support for the BUILTIN.UUID server data type**

  Support for the type `BUILTIN.UUID` is added in the form of being able to set the UUID using `PreparedStatement.setObject`, `PreparedStatement.setString`, `PreparedStatement.setBytes` and `PreparedStatement.setBinaryStream`.

  Conversely, result set columns (or output parameters) may be accessed using `ResultSet.getObject`, `ResultSet.getString`, `ResultSet.getBytes` or `ResultSet.getBinaryStream`.

New Functions in 3.38

- **Connection authentication using the Secure Remote Password protocol**

  Connection authentication using the Secure Remote Password protocol is now used whenever available on the server. This authentication method is available only on Mimer version 11 servers and later.

New Functions in 3.35

- **Connection.isValid**

  The JDBC 4 feature `Connection.isValid` is now implemented and supported. The function confirms the validity of the connection by issuing a server request. If there is a communication error or the response is not received within the specified timeout period, the connection is deemed invalid and false is returned.

New Functions in 3.31

- **Running JDBC Applications in SINGLE Mode**

  From version 3.31 it is now possible to run JDBC in SINGLE mode. To run SINGLE the mimcomm shared library is required and is used when the protocol "single" is being used.

VMS: On OpenVMS, mimcomm libraries older than version 10.1 will not work.

New Functions in 3.30

- **Support for SQL statements START TRANSACTION, COMMIT and ROLLBACK**

  The JDBC driver now interprets the SQL statements START TRANSACTION, COMMIT and ROLLBACK properly. Previously, the driver threw an exception referring to the transaction control methods available through the java.sql.Connection class.

  If a START TRANSACTION statement is executed, a transaction is explicitly started. If auto commit is on, the auto commit mode is temporarily turned off until the transaction is committed. If a transaction is already running, the START TRANSACTION statement is ignored.

  If a COMMIT statement is executed the transaction is committed, very much like calling java.sql.Connection.commit.

New Functions in 3.28

- **Error Position and Length Returned in SQLException and SQLWarning Objects**

  If an error is related to a specific part of the supplied SQL code, such as a grammatical error, the character position if such an error is now available through a call to the Mimer specific method `getErrorPosition()`. The length of the error is available through a call to the method `getErrorLength()`. To call these methods, the application need either to cast the `SQLException` to a `com.mimer.jdbc.SQLException` object, or use the reflection API to get hold of the method in question.

  For example, the following code example would allow the application to retrieve the character position within the SQL code concerned with the error:

```
} catch (java.sql.SQLException se) {
  try {
    java.lang.reflect.Method method =
      se.getClass().getMethod("getErrorLength",new Class[0]);
    output.println("-- errorlength = "+method.invoke(se,new Object[0]));
  } catch (NoSuchMethodException nsme) {
    output.println("-- errorlength = unknown");
  }
}
```

- **Support For Very Many Columns**

  Previous versions of this JDBC driver did not support SQL queries with more than 1017 columns. The application would experience an "output descriptor overflow" error (-19053) when calling `Statement.execute`, `Statement.executeQuery`, `Connection.prepareStatement` or `Connection.prepareCall`. This limit has now been removed.

- **New Connection Properties**

  Two new connection properties are introduced for entering a program ident during connect. The properties are `program` and `programPwd`, where `program` is program ident name, and `programPwd` is the password for the program ident.

- **setString on BOOLEAN**

  `setString` on BOOLEAN parameters now accepts the strings "1" and "0" in addition to "true" and "false".

New Functions in 3.26

- **User-Defined Types**

  The driver now offers full support for user-defined types. User-defined types may be fetched into generalized java.`sql.Struct` objects as well as custom made Java objects which interacts with the driver using the `java.sql.SQLData`, `java.sql.SQLInput` and `java.sql.SQLOutput` interfaces. The driver also provides the type mapping feature needed to map SQL types to the created custom Java classes.

  The metadata views needed to inspect existing types and their attributes (`DatabaseMetaData.getUDTs` and `DatabaseMetaData.getAttributes`) are also now provided.

  See *User-Defined Types* on page *25* for information and examples.

New Functions in 3.25

- **Full Support for Mimer 10.1 Server Unicode identifiers**

  The JDBC driver offers full support for the full Unicode identifiers that will be introduced with the Mimer SQL version 10.1 servers.

New Functions in 3.24 and 2.24

- **Support For .setNetworkProtocol And .getNetworkProtocol**

  The setter and getter routines `javax.sql.DataSource.setNetworkProtocol` and `javax.sql.DataSource.getNetworkProtocol` has been added to allow applications to alter protocol type. These routines are synonyms to the already existing `.setProtocol` and `.getProtocol`.

New Functions in 3.18, 2.18 and 1.18

The JDBC version 18 drivers may now connect to Mimer SQL Micro servers. Note however, that many features you normally expect in a Mimer SQL Engine are not available in the Mimer SQL Micro Edition server.

An application may detect the Mimer SQL product type by calling `DatabaseMetaData.getDatabaseProductName()`. This will return “Mimer SQL Micro”, “Mimer SQL Mobile”, or ”Mimer SQL Engine” - depending on the server type.

New Functions in 3.17, 2.17 and 1.17

Support for the `BOOLEAN` SQL data type that was introduced in Mimer SQL 9.3 servers.

New Functions in 3.16, 2.16 and 1.16

- The driver can load and use the mimcomm JNI library which allows the JDBC driver to use all communication methods supported by Mimer on the platform.
- The classes `MimerDataSource`, `MimerConnectionPoolDataSource` and `MimerXADataSource` have two additional properties: protocol and service. These are needed when using the mimcomm JNI library. The new properties are explained further in *Deploying Mimer JDBC in JNDI* on page *9*.

New Functions in 3.15

- The first release of a JDBC 3 compliant driver.
- Holdable cursors.

New Functions in 2.9

Server data type `NATIONAL CHARACTER LARGE OBJECT` (`NCLOB`) is now supported.

New Functions in 2.8

The method `PreparedStatement.setBytes` is now supported on `LONGVARBINARY` and `PreparedStatement.setString` on `LONGVARCHAR`. In the case of Mimer, `LONGVARBINARY` is the same as a `BLOB`, and `LONGVARCHAR` is the same as a `CLOB`.

New Functions in 2.7

The object returned when calling `.getBinaryStream`, `.getAsciiStream` and `.getCharacterStream` on `BLOB` and `CLOB` objects now implements the `.mark()`, `.reset()` and `.skip()` methods.

New Functions in 2.5

Support for large objects; `BINARY LARGE OBJECT` (`BLOB`) and `CHARACTER LARGE OBJECT` (`CLOB`). `BLOB`'s store any sequence of bytes, `CLOB`'s store Latin-1 character data.

New Functions in 2.4

Support for server `NCHAR` and `NCHAR VARYING` data types. They are used to store Unicode data. By using these data types, any Java String object can now be stored in the database. This is not the case when using `CHARACTER` or `CHARACTER VARYING` data types since these can only store Latin-1 characters.

New Functions in 2.3

- Support for `javax.sql.DataSource`.
- Support for connection pooling using `javax.sql.ConnectionPool` DataSource
- Support for distributed transactions (XA).

New Functions in 2.0

- Scrollable cursors are now fully supported.
- All date, time and timestamp methods now support the `java.util.Calendar` class for handling time zones. Mimer SQL 8.2 servers do not currently support time zones and this feature enables you to use time zones.
- Batches of statements are supported.
- Batches of prepared statements are supported. Batches of prepared statements are really useful for increasing performance when executing several `INSERT`, `UPDATE` or `DELETE` statements.
- Batches of callable statements are supported.
- There are now setter and getter methods for `CharacterStreams`.
- Several new `DatabaseMetaData` methods.
- Support for the Mimer SQL statements `ENTER` and `LEAVE`.

New Functions in 1.9

Server data type `NATIONAL CHARACTER LARGE OBJECT` (`NCLOB`) is now supported.

New Functions in 1.7

- When working with a Mimer SQL version 9 server, the JDBC 1 driver now supports the new version 9 data types (`NCHAR`, `NCHAR VARYING`, `BINARY LARGE OBJECTS`, and `CHARACTER LARGE OBJECT`).
- The SQL statements `ENTER` and `LEAVE` are now supported.

New Functions in 1.2

Support for query timeout and cancel. (Connection timeout is not supported.)

Changed Functions

This section describes the main changed functions of each Mimer JDBC version.

Changes in 3.41

- **Trailing blanks are now ignored during connect**

  The JDBC driver now ignores trailing blanks in ident names and passwords during connects. For example, this means connecting to the ident `"NOBLANKS"` using the name `"NOBLANKS "` will succeed. Previously, this failed with a Login failure.

Changes in 3.40

- **Reduced memory consumption**

  Substantial work has been undertaken to reduce the amount of memory allocated by the driver, specifically during common operations such as preparing and executing statements, creating results sets, and working with common data types.

Changes in 3.39

- **Corrected behavior of ResultSet.isBeforeFirst for empty result sets**

  The previous behavior of `ResultSet.isBeforeFirst` was to always return true when at the beginning of the result set, no matter if it contained any rows or not. This has been incorrect and not compatible with the JDBC specification which mandates that false should be returned for empty result sets.

  The behavior is now changed to make the routine return false for empty result sets.

- **Corrected behavior of current row number on scrollable empty result sets**

  Previously `ResultSet.getRow()` returned 1 on empty result sets if `ResultSet.next()`, or any other cursor positioning call was made. This was incorrect. Now, the cursor position is always 0 on empty result sets.

- **Corrected behavior of current row number when at the end of the result set**

  Previously `ResultSet.getRow()` returned the number of rows in the result set plus one if the cursor was at the end. This was incorrect. If the cursor is not on a row, 0 should be returned. This is now corrected.

- **Statement.setEscapeProcessing is ignored by version 11 servers**

  The escape clauses are from version 11 servers onward part of the SQL compiler grammar and always considered. The method `java.sql.Statement.setEscapeProcessing` to enable or disable escape clause processing, is therefore ignored when connected to those servers.

- **JDBC now requires 9.2 servers or later**

  Previously JDBC clients has supported version 8.2 servers (or later.) Now, version 9.2 is required.

Changes in 3.29

- **Specific Login Failure Error Code**

  When both the `program` and `programPwd` connection properties are supplied, and the connection attempt fails, a specific error code is now returned to catch these errors. From this version, an SQLException with the native error code -2110, message "Program login failure" is thrown.

Changes in 3.28

- **Object Finalizers Now Never Make Server Calls**

  Object finalizers, such as the finalizers for Connection, ResultSet, and Statement-objects, used to make server calls to close the corresponding server objects. They now never do that, but rather note that the object should be freed on the server side, and the actually close call to the server takes place at later point in time. This is because finalizers are called by garbage collector and on some platforms the garbage collector is sensitive to things that might take a long time, such as a remote network request.

  Programmers are advice to explicitly close their database objects at any times to make the server release unused resources as soon as possible.

Changes in 3.25

When using the protocol local to connect to the database server the shared library mimcomm.dll is used. When using 64-bit Java the system has been changed to use the mimcomm64.dll. This library is present in 64-bit installations of Mimer SQL from version 10.0.6.

Changes in 3.24, 2.24 and 1.24

- **Strict Sort Order in DatabaseMetaData.getTypeInfo**

  The sort order in which rows are returned when calling `DatabaseMetaData.getTypeInfo` is now more strict. Previously, rows were only sorted on DATA_TYPE (the integer type code). Now, the rows are sorted on DATA_TYPE, core data types first then domains, and TYPE_NAME.

Changes in 3.20, 2.20 and 1.20

- **Improved absolute positioning on scrollable result sets**

  Previously the driver made two server calls when positioning relative to the end of result set when the result set size was not known. This includes positioning to the last record. Now, the driver will under all circumstances make at most one server call to position the cursor.

Changes in 3.16, 2.16 and 1.16

- **Changed type mapping for FLOAT(n)**

  Mimer SQL supports the data type FLOAT(n) which can store a floating point number with n digits of mantissa and an exponent ranging from -999 to 999.

  This data type was previously mapped to the Java type double (which only supports exponents ranging from -308 to 308). This was problematic since some routines (in particular getObject()) would fail for very large (or small) values in the database.

  The FLOAT(n) data type is now mapped to java.math.BigDecimal. While not a perfect match, this data type can accurately represent all values that can be stored in FLOAT(n) columns in the database.

  Note that it is still possible to use the methods getDouble() and getFloat() on FLOAT(n) data, but those methods will fail when the data is out of range for a Java double (or float).

  To store Java double and float values, consider using the Mimer data type DOUBLE PRECISION for Java double and the Mimer data type REAL for Java float.

  Note that the Mimer data type FLOAT (without a precision) is synonymous to DOUBLE PRECISION and is a bad match for the Java float type which is single precision.

- **Changed string representation for floating point data**

  The JDBC driver supports the getString() method on all Mimer floating point columns. Previously this method padded the returned value with zeros to its declared precision (a FLOAT(15) could return “1.00000000000000”). This version will not add those zeros (getString() on the same value will return “1”).

Changes in 2.15 and 1.15

- **Login failure now returns SQLSTATE 08004.**

  Previously login failure threw an `SQLException` with the SQLState `28000`. According to the SQL-2003 standard, this is incorrect, and has been corrected to return `08004`. The 08-class of SQLStates relates to error conditions during the connect phase.

- **Several error messages have been clarified**

  Error texts returned when a cast from a character column to something else now more clearly state the failed cast. Note that this particular improvement applies to client side casts only. For instance, this includes casts where an SQL parameter type is `INTEGER` and its value is set using the `PreparedStatement.setString` method.

The driver now displays an accurate error text when a connection attempt fails because the application hasn't specified the database name, or it has specified an empty database name.

When the application refers to a column name that does not exist in the result set, the error text now includes existing columns names. To keep error texts reasonably short, if this error occurs on a result set with many columns, only a selection of column names. This situation is indicated with three consecutive periods in the error text.

Errors returned from the Mimer TCP server (listening on port 1360 on behalf of Mimer SQL servers) now include a descriptive text, previously only the error code was displayed to the caller.

Changes in 2.14 and 1.14

- **Extended Server Information**

  JDBC clients now present more detailed information to the servers about who it is, which version it is and in which environment it is executing in. Future servers will provide tools to monitor this information.

- **Changing autocommit mode always commits open transactions**

  Earlier on, the Mimer JDBC driver mimiced the ODBC behavior when autocommit mode is changed. The ODBC spec says that open transactions should be automatically committed when the autocommit mode goes from off to on. The JDBC specification requires drivers to commit open transactions on all changes in autocommit state. From version 14 onwards, the Mimer JDBC driver implements this behavior.

  An observant reader might question why this has any significance at all? After all, when autocommit mode is on, we expect all statements to be committed automatically anyway? The difference lies in how open result sets are treated. As you may know, result sets are by default closed when transactions are committed. In practice, running in autocommit mode means that transactions are committed _as_soon_as_possible_. For instance, a statement returning a result set will typically be committed when the application explicitly closes the result set, or if the result set is forward-only when the entire set has been read. Changing the autocommit mode during the life of the result set will now always trigger a commit which will close the result set.

Changes in 2.9

The driver now returns the correct object type when doing `CallableStatement.getObject`. According to the JDBC specification, `getObject` should return a Java object whose type corresponds to what type the output parameter was registered to with the `CallableStatement.registerOutParameter` method call. Earlier drivers always returned the default Java object type.

Changes in 2.7

- All `.getUnicodeStream` on `NCHAR` columns no longer throw `IndexOutOfBoundsException`.

- All `.getCharacterStream` returned incorrect results for `NCHAR` and `NCHAR VARYING` columns. This problem is corrected.
- All `.getAsciiStream`, `.getBinaryStream`, and `.getCharacterStream` on `CHAR`, `CHAR VARYING`, `NCHAR`, `NCHAR VARYING`, `BINARY` and `BINARY VARYING` columns have been reworked to reduce memory footprint, and also to provide more efficient `.mark()`, `.reset()`, and `.skip()` implementations.

Changes in 2.2

- Column names and labels are now regarded as equal. From an SQL standard point of view, the column name should be hidden when a correlation name is specified.
- Both `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` return the correlation name when one is specified.
- Default network buffers have been reduced in size to increase server scalability.

Changes in 2.1

A Mimer SQL beta license key is no longer required on the server.

Changes in 1.3

`Statement.executeQuery` no longer accepts non-query statements and `Statement.executeUpdate` no longer accepts statements other than updates, inserts or deletes.

Changes in 1.2

The name of the Mimer driver class is changed to `com.mimer.jdbc.Driver` (earlier `com.mimer.jdbc1.Driver`).

Corrected Problems

This section describes the main corrected functions of each Mimer JDBC version.

Corrections in 3.42

- **ResultSet.getDate, .getTime and .getTimestamp used the incorrect timezone when java.util.Calendar was specified**

  Previous versions of the JDBC driver used the incorrect timezone when a `java.util.Calendar` was supplied when setting an SQL DATE, TIME or TIMESTAMP (which does not have a time zone) parameter value using a `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp` object. The correct behavior is that the Calendar object contains a timezone which specifies the timezone to be used for the stored value while the time in the java object contains the timezone which is was created in (nearly always the current time zone of the Java runtime). Essentially, the Calendar object should in most cases specify the timezone of the database server. This means that if the Java runtime runs in the GMT+4 timezone, setting the time 12:00:00 using a Calendar object specified to be in timezone GMT+1 the time used in the parameter value should be 09:00:00.

  The previous, incorrect, behavior was that the timezone of the Calendar object was used to determine the timezone of the value to be set, and where the actual data in the database was assumed to be GMT+0. This meant, in the above example, that when the time 12:00:00 was to be set with the Calendar with timezone GMT+1, the value used in the parameter was 11:00:00.

- **Register pair 1/2 contains wrong type.**

  When calling the `java.sql.SQLInput.readDouble` or `java.sql.SQLInput.readLong` methods when accessing User Defined Types in version 10.1 servers, the following errors was triggered in JDBC version 3.41.a. The problem does not appear when working with version 11 servers. There is no known workaround, other than using other methods.

  readLong signature: ()J) Register pair 1/2 contains wrong type

  readDouble signature: ()D) Register pair 1/2 contains wrong type

Corrections in 3.41a

- **Incomplete Scrollable Result set**

  Scrollable result sets created with a statement using at least one parameter, when connected to a server of version 10.1 or older, were erroneously detecting end of table at a certain point in the result set, causing the end of the result set to not being returned. The point where the end was detected was when about 64 kb had been returned. Smaller result sets were unaffected by the error.

Corrections in 3.40

- **Invalid internal statement identifier after Statement.getResultSet**

  Corrected issue introduced in version 3.39 where the error -19039 "Invalid internal statement identifier" occurs if execute is made using `Statement.execute`, the statement was a query and the ResultSet object was picked up using `Statement.getResultSet`. There must also be an already existing statement executing before this execute, and that there are fetches on that result set prior to the call to `ResultSet.next` which triggers the error. The result set must also have a certain size, large enough so that it does not fit in the first response from the server. This size is by default 60 kb, but may vary depending on fetch size.

- **NullPointerException after Statement or ResultSet object has been finalized prior to being returned to connection pool**

  When Statement or ResultSet objects has not been explicitly closed by the application, they are eventually garbage collected by the JVM. If this occurred just before the connection was returned to the connection pool, a NullPointerException occurred. This problem was introduced in JDBC 3.26.

- **Truncation of TIME and TIMESTAMP decimals produced an exception, rather than a warning.**

  When TIME and TIMESTAMP decimals was truncated when setting a parameter, an exception was previously thrown with the error code -22061 and text "Datetime field overflow". This has now been changed to a warning (accessible by calling `PreparedStatement.getWarnings`). An example situation were this will occur is when a TIMESTAMP(2) is set with the value '2019-11-10 12:13:14.156'.

Corrections in 3.39

- **NullPointerException calling ResultSet.isLast**

  Connecting to older servers could in version 3.38 give an internal -19078 error. The error appears if the ident name supplied during login was 3+4n characters long (for any n>=0).

- **-19078 errors when connecting to older servers**

  The releases 38.a and 38.b contained an error causing ResultSet.isLast to throw a NullPointerException if they were called on a forward only result set which had reached the end, or in other words was position at after last row.

- **When connecting to 10.1 servers and older an -19041 internal error could appear**

  Unusual errors during the connect phase would trigger an auxiliary error -19041 with the following text "Error -19041 while retrieving error description for ...". The situation occurred when the server returned an error which wasn't already known by the JDBC client, which in itself in rather unusual.

- **ResultSet.isLast problem**

  The releases 38.a and 38.b contained an error causing ResultSet.isLast to throw a NullPointerException if they were called on a forward only result set which had reached the end, or in other words was position at after last row.

- **Corrected ArrayIndexOutOfBoundsException when positioning from before-First**

  Corrected ArrayIndexOutOfBoundsException or eternal loop occurring when a scrollable cursor has been started (at least one row fetched) and positioned in the result set, then positioned on before first (such as using ResultSet.absolute(0), and then positioned near the end of a result set whose size must also exceed the size of one server response package.

  This error was introduced with version 2.1.

- **Corrected positioning for large negative values for ResultSet.absolute**

  Previously, when starting the cursor (the first positioning call of the result set), with a call to ResultSet.absolute(-n) where n is equal to the size of the result set plus one, ResultSet.isBeforeFirst was incorrectly returning false. Also, ResultSet.getRow() did not return 0 as expected.

  This error was introduced with version 2.1.

Corrections in 3.38

- **FLOAT(p) parameters could be partially corrupted when p is even**

  When FLOAT(p) parameters were set, the value could sometimes be corrupt for even values of p, when the value was set using a call to `PreparedStatement.setInt` or `PreparedStatement.setLong`, and that the number of digits in the integer exceeded the precision.

  That is, FLOAT(2)-parameters was vulnerable for values 100 and up, FLOAT(4) was vulnerable for values 10000 and up, etc.

  Note: No corrupted values were ever stored in tables.

- **Incorrect character values in streams returned by ResultSet.getAsciiStream and CallableStatement.getAsciiStream**

  Character codes in the range 128-255 was not returned properly when calling `InputStream.read()` on streams returned by `ResultSet.getAsciiStream` and `CallableStatement.getAsciiStream`. Instead a negative character value was returned.

  Characters were returned correct by `InputStream.read(byte[])` and `InputStream.read(byte[],int,int)`.

- **Parameters were required to be set before statement being executed**

  Older drivers required all parameters of a PreparedStatement to always be set after the statement was executed. This is no longer the case. Parameters now retain their values through successive executions of the statement. For example, while the below sequence previously inserted only the tuple (1,2), it now inserts the tuples (1,2) and (1,3).

```
PreparedStatement ps = con.prepareStatement("insert into A values (?,?)");
ps.setInt(1,1);
ps.setInt(2,2);
ps.execute();
ps.setInt(2,3);
ps.execute();
```

The older behavior was not compatible with the JDBC specification which in section 13.2.2 specifically states that "the values set for the parameter markers of a PreparedStatement object are not reset when it is executed".

- **Error handling when adding parameter sets to batches**

  Previously, when an error occurred during the last `PreparedStatement.addBatch`, the batch was left in an error state making it impossible to execute the batch until the error was corrected, for example by adding another parameter set. The error seen during the call to `PreparedStatement.executeBatch` was -22065, SQLSTATE HY010, with the text "Must call addBatch() before executing the batch".

Corrections in 3.37

- **Internal error on timeouts when using local shared memory communication**

  In previous versions there was a chance there was an internal error -22046 with the text "An internal error occurred in readFromServer ([...])", if a garbage collection occurred at the same time a server request timed out if local shared memory communication was used.

  (If the timeout was not set with `Statement.setQueryTimeout`, or any other communication protocol than local shared memory was used (JDBC URL starting with anything other than `jdbc:mimer:local:`), the error never occurred.)

- **Corrected problem, NullPointerException if PreparedStatement.setString was called after PreparedStatement.getMoreResults**

  In previous versions, a NullPointerException would occur if `PreparedStatement.setString` (or any other setter method) was called after a call to `PreparedStatement.getMoreResults`. This is no longer the case.

Corrections in 3.35

- **Casting numerical values to Java boolean variables**

  When calling `ResultSet.getBoolean` and `CallableStatement.getBoolean` on a numerical column or output parameter (such as INTEGER, BIGINT, DECIMAL, REAL or DOUBLE PRECISION), `true` is now returned for all non-zero values. Previously, `true` was only returned for the value 1, while an exception was thrown for all other non-zero values.

Corrections in 3.31

- **Connection Hangs When Using a Stream From a Large Object as Input to Another Large Object on the same connection**

  When obtaining a stream from a large object column (such as by calling `Clob.getCharacterStream`, `ResultSet.getCharacterStream`, `Blob.getBinaryStream` or `ResultSet.getBinaryStream`) and using this stream as a parameter on the same connection, the JDBC driver will hang, and the process has to be aborted.

- **Cannot Use the String "0e0" to Set a Floating Point Value**

  Previously, attempting to use the value "`0e0`" to set a parameter value of any numeric data type would result in a -22038 numeric conversion error. It now succeeds.

- **Output Descriptor Overflow Errors (-19053) When Compiling Unusually Large SQL Statements**

  When compiling large SQL statements, specifically with many parameters, an output descriptor overflow error (-19053) could occur. The number of parameters for the error to occur was about 350.

- **Finalized Connections May Stall the Garbage Collector**

  In versions prior to 3.31, connections being finalized issued a server request to end the connection. This could take some time, thus stalling the garbage collector. Finalized connections no longer issue this request, but rather close the network connection.

- **Extra Zero Characters Appended to Character Columns With UTF-16 Surrogate Characters**

  When setting a NCHAR or NVARCHAR column with a value containing UTF-16 surrogate characters, one or more zero characters was inserted at the end of the character string. For example, inserting a string containing the character *MUSICAL SYMBOL G CLEF* (`0x1d11e`) which in a Java String is represented by the char pair 0xd834 and 0xdd1e, the resulting characters in the database would be `0x1d11e` and `0x00000`.

  The problem was introduced in 3.30.

- **Warning For TYPE_SCROLL_SENSITIVE Cursor**

  The cursor modes which are supported in Mimer are `TYPE_FORWARD_ONLY` and `TYPE_SCROLL_INSENSITIVE`. This means, in Mimer a cursor is always either forward only or scrollable, and scrollable cursors are always insensitive to changes made by other statements and other transactions. (This is in database terms called "cursor stability" and is a basic characteristic of Mimer.)

  The third JDBC cursor mode `TYPE_SCROLL_SENSITIVE` is supported as an option, but the underlying database will never show any changes in the result set - the cursor is always stable. In this case the JDBC driver issues a warning saying that an `TYPE_SCROLL_INSENSITIVE` result set was used instead.

  The problem was that this warning was not shown until the statement was executed. This is now changed so that the warning is shown directly when the statement is created.

- **PreparedStatement.setObject Failed with Byte and Short Data Types**

  Previously any attempt to set a parameter using `PreparedStatement.setObject` (or `CallableStatement.setObject`) with a Byte or Short data type would throw an SQLException with the error code -22006. This is now corrected.

Corrections in 3.30

- **Statement Leak When Executing Statements In a Program Ident Without Ever Leaving the Ident**

  Previous versions of the driver retained a strong reference to all created Statement (or PreparedStatement or CallableStatement) objects until the application leaves the program ident. If the application never leaves the program ident, memory resources may eventually become scarce. The reference was kept even though the statement object was closed properly.

  From this version onwards, the statement is delisted once being closed, thus causing properly closed statements to be released properly.

- **PreparedStatement.setString May Corrupt Parameters Or Cause an ArrayIndexOutOfBounds Exception**

  In version 3.29, applications which set parameters of the type CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER or NATIONAL CHARACTER VARYING using a call to PreparedStatement.setString or CallableStatement.setString may experience ArrayIndexOutOfBounds exceptions during the setString method call. If parameters are set in reverse order (i.e. setting parameter n+1 before n) the exception may have been avoided, but parameter contents may have been corrupted.

  Problem was introduced in version 3.29 and corrected in 3.30.

- **Update Counts Not Reported Properly After Call To PreparedStatement.executeUpdate and CallableStatement.executeUpdate**

  The update count, the number of rows affected by an INSERT, UPDATE or DELETE statement executed through a PreparedStatement or CallableStatement object was reported improperly. The correct behavior is to return the number of rows affected, or -1 if unknown. In the named cases, 1 was incorrectly returned for all cases, regardless of the number of rows affected.

  Problem was introduced in 3.28 and corrected in 3.30.

Corrections in 3.29

- **UTF-16 Surrogate Characters**

  The JDBC driver now handles UTF-16 surrogate characters properly on all uses, except when storing data into columns of the type NATIONAL CHARACTER LARGE OBJECT (see *Surrogate Characters in NATIONAL CHARACTER LARGE OBJECT Data* on page *55*).

  Earlier versions of the JDBC driver did not handle UTF-16 surrogate characters in a uniform way which led to various errors including cast errors, truncation errors or that character data was rejected by the server because valid strings was not supplied by the driver.

  An UTF-16 surrogate character is a legal Unicode character which cannot be represented using a 16-bit word. These characters are, according to the Unicode standard, represented using two consecutive 16-bit words which together make up the complete character.

Corrections in 3.28

- **Meta Data Lookup Functions In Multithreaded Applications**

  Older versions of the JDBC driver experienced threading difficulties when JDBC meta data lookup functions were being called.

  Multithreaded applications calling ResultSetMetData.getCatalogName(), ResultSetMetData.getSchemaName(), ResultSetMetData.getTableName(), or ResultSetMetData.getColumnTypeName() simultaneously may experience Connection closed problems. A common situation in which this occurred is when using DbVisualizer and reconnecting to a database.

- **Meta Data Lookup Functions In Resultsets With Many Columns**

  Versions prior to version 28 of the JDBC driver may not be able to lookup catalog, schema, table and column type name through the meta data lookup functions ResultSetMetData.getCatalogName(), ResultSetMetData.getSchemaName(), ResultSetMetData.getTableName(), or ResultSetMetData.getColumnTypeName() if the underlying result set contained more than about 360 columns. There is now no such limit.

  Note however that there is a known problem with older Mimer servers (prior to 10.1) which in combination with earlier versions of the JDBC driver made the upper limit 120 columns.

Corrections in 3.27

- **ResultSet.wasNull Reporting Incorrect Null Status For BOOLEAN Columns**

  The ResultSet.wasNull method is supposed to return whether the most recently retrieved column or output parameter held the SQL NULL value or not. Prior to version 27 this was not working properly in the Mimer JDBC Driver on SQL BOOLEAN data since the value returned by ResultSet.wasNull returned the null value status of the column or output parameter retrieved prior to the last retrieved column or output parameter. If the BOOLEAN column or output parameter was the first one to be retrieved, ResultSet.wasNull always returned false no matter the actual value of the column or output parameter in question.

- **Update counts not returned properly for WHERE CURRENT operations**

  Older JDBC drivers did not return update counts for UPDATE WHERE CURRENT and DELETE WHERE CURRENT. 0 was always returned, even though a row was updated or deleted.

  The problem affected the return values from the methods Statement.executeUpdate and Statement.getUpdateCount.

Corrections in 3.26

- **Driver fails to recognize escape character when doing dictionary lookups**

  Prior to this release, the driver did not recognize SQL LIKE-clause escape characters when doing dictionary lookups through the java.sql.DatabaseMetaData interface.

The proper behavior for an application is that it should ask the driver which character is being used as an escape character in meta data lookups. This is done through the `DatabaseetaData.getSearchStringEscape` method. In the case of Mimer this will always return an exclamation mark (`!`). The application must then precede those underscores (_) and percent marks (`%`) which aren't wildcards with an escape character. Escape characters must also be preceded by an escape character.

The problem corrected in version 3.26 was that the Mimer JDBC driver still treated escape wildcard characters as wildcard characters, thus possibly returning more results than expected.

Corrections in 3.25

- **Mimer JDBC URL's did not recognize numerical IPv6 addresses**

  Earlier versions of the Mimer JDBC driver did not recognize the method to specify numerical IPv6 addresses in URL's using square brackets which is defined in RFC 2732. For example, the following two URL's are now recognized properly:

  ```
  jdbc:mimer://[::1]/odbc_net
  jdbc:mimer://[::1]:1360/odbc_net
  ```

  It was, however, possible to use a numerical IPv6 address as an URL attribute, or when using the javax.sql.DataSource interface to make database connections. For example, the following two URL's were, and are still, working properly:

  ```
  jdbc:mimer:/odbc_net?serverName=::1
  jdbc:mimer:/odbc_net?serverName=::1&portNumber=1360
  ```

- **Enter a program ident with long ident name**

  Attempts to enter program idents using the ENTER statements and supplying an ident name longer than 18 characters, always failed with a -14006 "Login failure" error. Problem is corrected in Mimer JDBC 3.25.

Corrections in 3.24, 2.24 and 1.24

- **Invalid internal statement identifier when using a java.sql.Statement object that has been retained**

  When doing the below, the application received the -19039 "Invalid internal statement identifier" error.

  - Entering a program ident using the Mimer SQL statement ENTER.
  - Creating a new `java.sql.Statement`, `java.sql.PreparedStatement` or `java.sql.CallableStatement`
  - Leaving the program ident using the Mimer SQL statement LEAVE.
  - Attempting to execute statements created above. These are only available when the program ident entered in step 1 is active.

  This error condition is now properly managed, and the user will receive the -22071 error along with a descriptive error message.

  For more information about ENTER and LEAVE, please refer to the *Mimer SQL Reference Manual*.

- **BOOLEAN columns was shown twice in views**

  BOOLEAN columns was in earlier versions of their driver shown twice in the views returned by `DatabaseMetaData.getTypeInfo` and `DatabaseMetaData.getColumns`. This is now corrected.

Corrections in 3.23 and 2.23

Scrollable cursors created with a SELECT statement with parameter markers did not always perform well before release 23. The symptoms where either that the parameter appeared to be ignored, or the server returns a -10303 “negative overflow occurred in arithmetic operation CHP.”

Corrections in 3.23, 2.23 and 1.23

The `DatabaseMetaData.getColumns` view could previously return the -10312 error when connected to Mimer SQL 9.1 servers or older. See known problem *System Views Raising Error -10312* on page *56* for more information, as the very same problem still applies to `DatabaseMetaData.getTypeInfo()`.

`DatabaseMetaData.getColumns` in release 23 clients and newer will avoid the problem.

Corrections in 3.22, 2.22 and 1.22

When using the driver with IBM WebSphere 6, connection attempts may end with a “java.sql.SQLException: Connection is closed” message. This is because the driver did not allow pooled connections (`javax.sql.PooledConnection`) to be closed after the associated Connection has already been closed. The error was not seen when using Websphere 5 or earlier.

Corrections in 3.21, 2.21 and 1.21

Fetching a scrollable cursor by positioning the cursor on a row relative to the end of the result set (e.g. specifying a negative row number to the `ResultSet.absolute` method) after the last row of the result set has already been visited produced incorrect results. The problem was introduced in version 20 of the JDBC driver.

Corrections in 3.20, 2.20 and 1.20

- **Improved error handling**

  If the connection with the server was lost (or if the server is shut down), earlier JDBC drivers could produce a null pointer exception in some circumstances. The new JDBC driver will produce an appropriate SQLException.

  Also, the method `Connection.isClosed()` will now return true on any connection that has received an SQLException indicating that the connection with the server was lost.

- **Uninformative error message when connecting to Mimer 8.1 servers and older.**

  When connecting via TCP/IP to Mimer SQL 8.1 servers and older, which aren't accepting connections from JDBC drivers, applications may receive the uninformative internal error -22046. This problem is related to certain server versions, but JDBC drivers from version 20 can recover from this condition and return a proper error message.

Correction in 3.19, 2.19 and 1.19

Version numbers for servers older than 9.3 was not returned properly by the Mimer JDBC *n*.18 drivers. This problem was seen in `DatabaseMetaData.getProductVersion`, `DatabasMetaData.getDatabaseMajorVersion` and `DatabaseMetaData.getDatabaseMinorVersion`.

Corrections in 3.18, 2.18 and 1.18

- **PreparedStatement.setString threw no exception on BOOLEAN data types**

  Version 17 JDBC drivers did never throw an exception if the application called `PreparedStatement.setString` with an illegal string. That is, a string that is not '`true`' or '`false`'. This is corrected in version 18.

- **LITERAL_SUFFIX and LITERAL_PREFIX for DATE, TIME, TIMESTAMP and INTERVAL data types**

  Result sets returned by `DatabaseMetaData.getTypeInfo` did not contain any data in the columns `LITERAL_SUFFIX` and `LITERAL_PREFIX` for data types DATE, TIME, TIMESTAMP and all INTERVAL data types. From version 18, these columns have a relevant value.

  This correction applies for all server versions. Using an older JDBC driver against a v9.3.5 Mimer SQL server or later will also return correct values for these columns.

- **Changed behavior for protocol type tcp**

  Specifying the protocol TCP in the Mimer JDBC URL, would instruct the driver to connect using the native TCP/IP-stack. From version 18, specifying the TCP protocol makes the driver connect using the Java TCP/IP-stack.

  The behavior when the protocol is unspecified has not been changed, that is, the Java TCP/IP-stack is used.

Corrections in 3.16, 2.16 and 1.16

- **DatabaseMetaData.getColumns returns too many columns**

  Older versions of the Mimer JDBC driver returned, when calling `DatabaseMetaData.getColumns`, duplicate rows for BINARY LARGE OBJECT, CHARACTER LARGE OBJECT and NATIONAL CHARACTER LARGE OBJECT columns. For example, querying a table with one CHARACTER column and one BINARY LARGE OBJECT column returned a result set of three rows. One row for the CHARACTER column, and two for the BINARY LARGE OBJECT column. This is now corrected.

- **DATE/TIME comparison problems**

  Previous drivers did not recognize TIMESTAMP's representing a value prior to the timestamp 1000-01-01 00:00:00, DATE values prior to the date 1000-01-01 and TIME values prior to 10:00:00 correctly. Although the driver would retrieve and display those values correctly, comparison operations may fail for identical values, leading to potentially duplicate primary keys, or that query conditions may fail for no obvious reason. These problems are now corrected.

Corrections in 2.14

- **PreparedStatement batches whose size exactly matched the network buffer size failed**

  Batches of PreparedStatements failed if data in the entire batch exactly matched the amount of space available in the network buffer.

  This could mean, for instance, that a batch of 19 rows would fail, while batches of 18 and 20 rows would succeed.

Corrections in 2.14 and 1.14

- **Non-public constructor in the Driver-class made applications fail loading the Mimer JDBC driver**

  Applications that don't rely on `DriverManager.getConnection`, or the `javax.sql.DataSource` class, to create a Connection to Mimer, but instead are creating a connection by using the Driver class couldn't load the `com.mimer.jdbc.Driver` class of the 1.13 and 2.13 Mimer JDBC drivers. More specifically, the following didn't work:

  ```
  Class dc = Class.forName("com.mimer.jdbc.Driver");
  Driver d = (Driver)dc.getInstance();
  ```

  Example of products using this (or similar) techniques, and thus avoiding the `DriverManager` object, are Sun Java Studio Creator and the Squirrel SQL database viewer.

- **Reading a BLOB stream might hang the connection**

  Reading Binary Large Objects through an `InputStream` (obtained through `ResultSet.getBinaryStream`) would place the network connection in an inconsistent state, in practice the session would hang, if the size of the object is larger than the size of the default network packet size, and the application tries to read the entire object in one call (`InputStream.read(b,off,len)` where len is larger than the size of the object). This is no longer the case.

- **Clarified error texts when streams are closed**

  The error texts saying that streams have been closed now explain why the stream was closed. This could be of several reasons, the server connection went dead, the transaction was committed or rolled back, the statement in which the result set containing the stream was closed, and so forth. This is now explained in the error message.

Corrections in 2.13 and 1.13

- **Fetching data might throw an SQLException with vendor code 1**

  Fetching data (`ResultSet.next`) could erroneously throw SQLExceptions with vendor code 1. This was wrong and is now corrected.

- **Large BLOB problem**

  Whenever BLOB's was read in several passes from the server, and the application specified a length longer than the actual BLOB, the driver hanged. Many platforms deliver TCP/IP packets in chunks of about 64 kb so this problem would occur when reading BLOB's larger than that.

Corrections in 2.12 and 1.12

- The midjdbc2 driver now fully supports SQL `DATE`, `TIME,` and `TIMESTAMP` data types.
- By mistake the Beta release (1.10/2.10) lacked support of the SQL constructs for manipulating session and transaction characteristics, such as `SET TRANSACTION READ ONLY`. These SQL statements are now supported, see the *Mimer SQL Reference Manual* for more information.
- Scrollable cursors now take the value set by `Statement.setFetchDirection` into account when selecting fetch strategy on scrollable result sets. This does not apply to 1.12.
- A problem which caused a premature end of table when a scrollable cursor has seen the end of the result set, is fetching backwards (using `ResultSet.previous`) and the fetch size has been set to a value less than the size of the result set. This problem did not apply to the JDBC 1 driver.
- When earlier versions did a `ResultSet.afterLast` or `ResultSet.last` after setting `Statement.maxRows` to a value that actually limits the result set size, the cursor was positioned on the wrong row, beyond the end of the result set. This is now corrected.

Corrections in 2.11 and 1.11

- Previously an `InputStream.skip(n)` on a stream derived from a `BLOB` column, or a `CharacterStream.skip(n)` on a stream derived from a `CLOB` or `NCLOB` column may leave the network state out of sync. This was seen with the error -22046 'An internal error occurred in ReadFromServer'. This problem is now corrected.

  Note: For the JDBC 1 driver, this problem applies to streams derived from the `getUnicodeStream` method call on `CLOB` and `NCLOB` columns.

- A problem with large SQL statements has been fixed. SQL statements larger than about 20 000 characters were unable to compile because of an `ArrayIndexOutOfBoundsException`.

Corrections in 2.10 and 1.10

- Previous versions of the driver did not return the correct data type on `CallableStatement.getObject` calls. The specification states that the object type returned should match whatever type was specified when the output parameter was registered through the `CallableStatement.registerOutParameter` call. Previously, the object type returned matched the data type on the server.
- When calling `ResultSet.findColumn`, `ResultSet.getString(String)` and similar column name related methods, the Mimer driver previously did a case sensitive search. This was incorrect. The search should be case insensitive, which it now is.
- 2.8 and 1.8 versions of the Mimer JDBC driver introduced a problem setting `CHARACTER` and `CHARACTER VARYING` columns via a `CharacterStream` object. The end result lost characters without throwing errors. This is now corrected.
- A JDBC driver using a database server with many indexes could have performance problems with the `DatabaseMetaData.getSchemas` call. This is now corrected for 9.2-servers and later. Unfortunately, since the problem is server related, older servers cannot easily be corrected.
- JDBC drivers connecting to Mimer SQL 8.2 servers unexpectedly threw `SQLException` exceptions when using `DatabaseMetaData.getCatalogs` or `DatabaseMetaData.getUDTs`. This is now corrected. Note that neither of these queries should return any rows with Mimer SQL 8.2 servers.
- `java.sql.Blob` and `java.sql.Clob` objects returned from calls to `ResultSet.getBlob` and `ResultSet.getClob` now stay alive throughout the entire transaction. Once the transaction in which the object is created is ended, all calls to the objects will throw a 'transaction has ended' exception. Previously, these objects could not be used once the resultset was closed.

Corrections in 2.9

- Scrollable result sets returned an error when calling `setFetchDirection`. This is no longer the case.
- `ResultSet.getString` did not return correct characters for `å`, `ä`, `ö` and similar Latin-1 but non-ASCII characters when other default character encoding than ISO 8859-1 was used. This included for instance Macintosh computers. This is now corrected.

Corrections in 2.7

- Earlier versions incorrectly returned `SQLSTATE 22001` for numeric value out of range. The correct `22003` is now returned.
- Procedure calls with large output parameters (typically `CHAR(100)`, `VARCHAR(100)` or larger) could end with the following exception message:

  ```
  An internal error occurred in MimConnection.readFromServer (packlen=148,
  bufLen=100, maxReceive=0).
  ```

  This problem is now corrected.
- Batches of statements were not cleared when being executed. This forced the programmer to call `Statement.clearBatch()` before building another batch. From now on, batches are automatically cleared after being executed.

Corrections in 2.6

Server resources was not released even when the application was properly closing `Statement`, `PreparedStatement` and `CallableStatement` objects. This could sometimes cause the following error when attempting to drop a table:

```
Error code: -16002, msg: Table locked by another cursor, state: S1000
```

This problem is now corrected.

Corrections in 2.2

- Correction of `DatabaseMetaData.supportsTransactionIsolationLevel(0)` which erroneously returned true.
- `ResultSet.getConcurrency()` and `ResultSet.getType()` returned wrong values for scrollable cursors.
- `DatabaseMetaData.getSystemFunctions()` returned the nonexisting `USERNAME` function.
- A `ResultSet.fetchSize` with a large number no longer throws an `ArrayIndexException`.
- `ResultSets` created from a `PreparedStatement` or `CallableStatement` no longer fails on the second `.next` call.

Corrections in 1.9

- `ResultSet.getString` did not return correct characters for `å`, `ä`, `ö` and similar Latin-1 but non-ASCII characters when other default character encoding than ISO 8859-1 was used. This included for instance Macintosh computers. This is now corrected.
- The driver now returns the correct object type when doing `CallableStatement.getObject`. According to the JDBC specification, `getObject` should return a Java object whose type corresponds to what type the output parameter was registered to with the `CallableStatement.registerOutParameter` method call. Earlier drivers always returned the default Java object type.

Corrections in 1.7

Earlier versions incorrectly returned `SQLSTATE 22001` for numeric value out of range. The correct 22003 is now returned.

Known Restrictions

The following sections document known restrictions.

Mimer SQL Experience v10.1 Limited Support for Statement.getMaxFieldSize and Statement.setMaxFieldSize

Mimer SQL Experience

There are no support for `Statement.setMaxFieldSize` and `Statement.getMaxFieldSize` when connected to a version 10.1 Mimer SQL Experience server.

Mimer SQL Experience v10.1 Native SQL Escape Clause Support

Mimer SQL Experience

The version 10.1 Mimer SQL Experience server does not support the following native SQL escape clauses:

- ASCII, for example {fn ASCII(x)}
- CHAR, for example {fn CHAR(x)}
- BIT_LENGTH, for example {fn BIT_LENGTH(x)}
- POSITION of the form {fn POSITION(x in y)}.
  (The form {fn POSITION(x,y)} is still supported.)
- SUBSTRING of the form {fn SUBSTRING(x from y for z)}.
  (The form {fn SUBSTRING(x,y,z)} is still supported.)
- DATABASE, {fn DATABASE()}
- DAYNAME, {fn DAYNAME(t)}
- MONTHNAME, {fn MONTHNAME(t)}
- QUARTER, {fn QUARTER(t)
- TIMESTAMPADD,
  for example {fn TIMESTAMPADD(SQL_TSI_SECOND,t,n)}
- TIMESTAMPDIFF,
  for example {fn TIMESTAMPDIFF(SQL_TSI_SECOND,t,t2)}
- CONVERT, for example {fn CONVERT(x,SQL_CHAR)}
- DIFFERENCE, such as {fn DIFFERENCE(x,y)}
- ACOS, {fn ACOS(n)}
- ASIN, {fn ASIN(n)}
- ATAN, {fn ATAN(n)}
- ATAN2, {fn ATAN2(n,m)}
- COS, {fn COS(n)}
- COT, {fn COT(n)}
- PI, {fn PI()}
- DEGREES, {fn DEGREES(n)}
- EXP, {fn EXP(n)}
- LOG, {fn LOG(n)}
- LOG10, {fn LOG10(n)}
- POWER, {fn POWER(n,m)}
- RADIANS, {fn RADIANS(n)}
- SIN, {fn SIN(n)}
- SQRT, {fn SQRT(n)}
- TAN, {fn TAN(n)}
- NOW, {fn NOW()}

Optional JDBC 2 features not supported:

- Connection timeout
- Updatable result sets
- Searching for data in large objects using the `Blob.position` or `Clob.position` methods
- java.sql.Array java objects (dependent on the SQL ARRAY data type)
- java.sql.Ref java object (dependent on the SQL REF data type)
- Type inheritance
- Typed tables and typed table inheritance
- Referring to procedure and function parameters by name
- The JAVA_OBJECT user defined type

Optional JDBC 3 features not supported:

- Updating parts of an existing large object using `Blob.setBytes` or `Clob.setBytes`
- Truncating an existing large object using `Blob.truncate` or `Clob.truncate`
- Transaction savepoints
- Time zones
- Support for the SQL DATALINK type and setting and retrieving java.net.URL objects
- Retrieving values of auto generated keys

Known Problems

This section describes the known problems with Mimer JDBC.

Surrogate Characters in NATIONAL CHARACTER LARGE OBJECT Data

When storing large object data, current Mimer SQL servers require the client and/or application to supply the size and length of the large object in advance prior to actually storing the data. This poses a problem if the data contains UTF-16 surrogate characters, since the application will know the length of the data by the number of UTF-16 code points, while the server regards the length as being the number of Unicode characters. The number of Unicode characters may therefore be less than the number of UTF-16 code points in the input data.

In these situations, database objects of the type NATIONAL CHARACTER LARGE OBJECT will be padded with zero characters up to the length originally specified. When at a later date the object is retrieved, it may appear to have grown in length. The actual number of characters grown equals the number of surrogate characters in the input data.

System Views Raising Error -10312

The system metadata returned by `DatabaseMetaData.getTypeInfo`, `DatabaseMetaData.getProcedureColumns`, and `DatabaseMetaData.getBestRowIdentifier` from Mimer SQL 9.1 servers or older may return -10312 “numeric value out of range”.

The error will not appear if no user created domains are present and returned by the views, or if less than 32768 system objects have been created in the database.

Statements Never Executed By java.sql.Statement.executeUpdate()

When issuing the statements ENTER, LEAVE, LEAVE RETAIN, SET DATABASE, SET DATABANK, and SET SHADOW are reported as having completed successfully, but they are actually never executed on the server. UPDATE STATISTICS and DELETE STATISTICS always return with an “Invalid internal DDU identifier” error.

All these statements are executed properly when being executed either using a `java.sql.PreparedStatement` or a `java.sql.CallableStatement`, but also using the `java.sql.Statement.execute` method.

The problem appeared in Mimer SQL servers in version 9.3.1 and was corrected in version 9.3.7G.

Update Counts on Errors in Batched Statements

Whenever an error occurs in a batched Statement, the driver is unable to return the correct information about the number of executed rows. The correct behavior is to return an integer array within a thrown `BatchUpdateException` object whose length corresponds to the number of batch statements. The Mimer driver is now returning an integer array with one entry per statement, with all entries set to 0.

Index

A

applet 16

B

batch operations 20
BOOLEAN 32

C

CLASSPATH 5
commit mode 17
connection 7
connection pools 10

D

DataSource 9
distinct type 25
distributed transactions 10
DriverManager 6, 9, 13, 15

E

encryption 29
error handling 10

G

garbage collection 27

H

holdable cursors 24

I

INTERVAL 27

J

J2EE 9, 10
Java applet 16
Java Virtual Machine 1
JDBC
- batch updates 17
- callableStatement objects 20
- connecting 6
- cursors
  - positioning 23
- error handling 10
- executing 19
- JDBC 2 17
- loading 5
- performance 21
- preparedStatement objects 19
- result sets 22
  - capabilities 24
  - scrolling 23
- statement objects 19
- transactions 17
  - auto-commit 17
  - manual-commit 18
- updating data 24

JNDI 9
JNDI repository 14
JVM 1

L

locking 17
logging 3

M

Mimer SQL
- connecting to 9

N

native SQL escape clause 54
NCLOB 32

P

performance 21, 28
prefetch 28
PreparedStatements 28

PSM 27

R

Result 22

S

scrollable cursors 23
scrolling 23
Secure Remote Password 30
security restriction 16
setMaxRows 28
stored procedures 27
structured type 25

T

thread-safe 28
trace driver 3
transaction 17
type 4 drivers 1
TYPE_FORWARD_ONLY 23
TYPE_SCROLL_INSENSITIVE 23
TYPE_SCROLL_SENSITIVE 43

U

URL 7, 13, 15
user-defined types 25
UUID 29

X

XA 10