MySQL Connector/J

MySQL Connector/J

Abstract

This manual describes MySQL Connector/J, the JDBC implementation for communicating with MySQL servers.

Document generated on: 2012-04-30 (revision: 30280)

Preface and Legal Notice

This manual describes MySQL Connector/J, the JDBC implementation for communicating with MySQL servers.

Legal Notices

For more information on the terms of this license, or for details on how the MySQL documentation is built and produced, please visit MySQL Contact & Questions.

For additional licensing information, including licenses for third-party libraries used by MySQL products, see Preface and Legal Notice.

For help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists where you can discuss your issues with

other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Chapter 1. MySQL Connector/J

This section explains how to configure and develop Java applications with MySQL Connector/J, the JDBC driver integrated with MySQL.

Chapter 2. Connector/J Versions

There are currently four versions of MySQL Connector/J available:

- Connector/J 5.1 is the Type 4 pure Java JDBC driver, which conforms to the JDBC 3.0 and JDBC 4.0 specifications. It provides compatibility with all the functionality of MySQL, including 4.1, 5.0, 5.1, 5.4 and 5.5. Connector/J 5.1 provides ease of development features, including auto-registration with the Driver Manager, standardized validity checks, categorized SQLExceptions, support for the JDBC-4.0 XML processing, per connection client information, NCHAR, NVARCHAR and NCLOB types. This release also includes all bug fixes up to and including Connector/J 5.0.6.

- Connector/J 5.0 provides support for all the functionality offered by Connector/J 3.1 and includes distributed transaction (XA) support.

- Connector/J 3.1 was designed for connectivity to MySQL 4.1 and MySQL 5.0 servers and provides support for all the functionality in MySQL 5.0 except distributed transaction (XA) support.

- Connector/J 3.0 provides core functionality and was designed with connectivity to MySQL 3.x or MySQL 4.1 servers, although it will provide basic compatibility with later versions of MySQL. Connector/J 3.0 does not support server-side prepared statements, and does not support any of the features in versions of MySQL later than 4.1.

The following table summarizes the Connector/J versions available:

Connector/J version	Driver Type	JDBC version	MySQL Server version	Status
5.1	4	3.0, 4.0	4.1, 5.0, 5.1, 5.4, 5.5	Recommended version
5.0	4	3.0	4.1, 5.0	Released version
3.1	4	3.0	4.1, 5.0	Obsolete
3.0	4	3.0	3.x, 4.1	Obsolete

The current recommended version for Connector/J is 5.1. This guide covers all four connector versions, with specific notes given where a setting applies to a specific option.

2.1. Java Versions Supported

The following table summarizes Connector/J Java dependencies:

Connector/J version	Java RTE required	JDK required (to build source code)
5.1	1.5.x, 1.6.x	1.6.x and 1.5.x
5.0	1.3.x, 1.4.x, 1.5.x, 1.6.x	1.4.2, 1.5.x, 1.6.x
3.1	1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x	1.4.2, 1.5.x, 1.6.x
3.0	1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x	1.4.2, 1.5.x, 1.6.x

If you are building Connector/J from source code using the source distribution (see Section 3.4, "Installing from the Development Source Tree"), you must use JDK 1.4.2 or newer to compile the Connector package. For Connector/J 5.1, you must have both JDK-1.6.x and JDK-1.5.x installed to be able to build the source code.

Because of the implementation of `java.sql.Savepoint`, Connector/J 3.1.0 and newer will not run on a Java runtime older than 1.4 unless the class verifier is turned off (by setting the `-Xverify:none` option to the Java runtime). This is because the class verifier will try to load the class definition for `java.sql.Savepoint` even though it is not accessed by the driver unless you actually use savepoint functionality.

Caching functionality provided by Connector/J 3.1.0 or newer is also not available on JVMs older than 1.4.x, as it relies on `java.util.LinkedHashMap` which was first available in JDK-1.4.0.

MySQL Connector/J does not support JDK-1.1.x or JDK-1.0.x.

Chapter 3. Connector/J Installation

You can install the Connector/J package using either the binary or source distribution. The binary distribution provides the easiest method for installation; the source distribution lets you customize your installation further. With either solution, you manually add the Connector/J location to your Java `CLASSPATH`.

If you are upgrading from a previous version, read the upgrade information in Section 3.3, "Upgrading from an Older Version" before continuing.

Connector/J is also available as part of the Maven project. For more information, and to download the Connector/J JAR files, see the Maven repository.

3.1. Installing Connector/J from a Binary Distribution

For the easiest method of installation, use the binary distribution of the Connector/J package. The binary distribution is available either as a tar/gzip or zip file. Extract it to a suitable location, then optionally make the information about the package available by changing your `CLASSPATH` (see Section 3.2, "Installing the Driver and Configuring the `CLASSPATH`").

MySQL Connector/J is distributed as a `.zip` or `.tar.gz` archive containing the sources, the class files, and the JAR archive named `mysql-connector-java-version-bin.jar`.

Starting with Connector/J 3.1.9, the `.class` files that constitute the JAR files are only included as part of the driver JAR file.

Starting with Connector/J 3.1.8, the archive also includes a debug build of the driver in a file named `mysql-connector-java-version-bin-g.jar`. Do not use the debug build of the driver unless instructed to do so when reporting a problem or a bug, as it is not designed to be run in production environments, and will have adverse performance impact when used. The debug binary also depends on the Aspect/J runtime library, which is located in the `src/lib/aspectjrt.jar` file that comes with the Connector/J distribution.

Use the appropriate graphical or command-line utility to extract the distribution (for example, WinZip for the .zip archive, and `tar` for the .tar.gz archive). Because there are potentially long file names in the distribution, we use the GNU tar archive format. Use GNU tar (or an application that understands the GNU tar archive format) to unpack the .tar.gz variant of the distribution.

3.2. Installing the Driver and Configuring the `CLASSPATH`

Once you have extracted the distribution archive, you can install the driver by placing `mysql-connector-java-version-bin.jar` in your classpath, either by adding the full path to it to your `CLASSPATH` environment variable, or by directly specifying it with the command line switch `-cp` when starting the JVM.

To use the driver with the JDBC `DriverManager`, use `com.mysql.jdbc.Driver` as the class that implements `java.sql.Driver`.

You can set the `CLASSPATH` environment variable under UNIX, Linux or Mac OS X either locally for a user within their `.profile`, `.login` or other login file. You can also set it globally by editing the global `/etc/profile` file.

For example, add the Connector/J driver to your `CLASSPATH` using one of the following forms, depending on your command shell:

```
# Bourne-compatible shell (sh, ksh, bash, zsh):
shell> export CLASSPATH=/path/mysql-connector-java-ver-bin.jar:$CLASSPATH
# C shell (csh, tcsh):
shell> setenv CLASSPATH /path/mysql-connector-java-ver-bin.jar:$CLASSPATH
```

Within Windows 2000, Windows XP, Windows Server 2003 and Windows Vista, you set the environment variable through the System Control Panel.

To use MySQL Connector/J with an application server such as GlassFish, Tomcat or JBoss, read your vendor's documentation for more information on how to configure third-party class libraries, as most application servers ignore the `CLASSPATH` environment variable. For configuration examples for some J2EE application servers, see Chapter 7, *Developing J2EE Applications with Connector/J*. However, the authoritative source for JDBC connection pool configuration information for your particular application server is the documentation for that application server.

If you are developing servlets or JSPs, and your application server is J2EE-compliant, you can put the driver's `.jar` file in the `WEB-INF/lib` subdirectory of your webapp, as this is a standard location for third party class libraries in J2EE web applications.

You can also use the `MysqlDataSource` or `MysqlConnectionPoolDataSource` classes in the `com.mysql.jdbc.jdbc2.optional` package, if your J2EE application server supports or requires them. Starting with Connector/J 5.0.0, the `javax.sql.XADataSource` interface is implemented using the `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource` class, which supports XA distributed transactions when used in combination with MySQL server version 5.0.

The various `MysqlDataSource` classes support the following parameters (through standard set mutators):

- `user`

- `password`

- `serverName` (see the previous section about fail-over hosts)

- `databaseName`

- `port`

3.3. Upgrading from an Older Version

This section has information for users who are upgrading from one version of Connector/J to another, or to a new version of the MySQL server that supports a more recent level of JDBC. A newer version of Connector/J might include changes to support new features, improve existing functionality, or comply with new standards.

3.3.1. Upgrading to MySQL Connector/J 5.1.x

- In Connector/J 5.0.x and earlier, the alias for a table in a `SELECT` statement is returned when accessing the result set metadata using `ResultSetMetaData.getColumnName()`. This behavior however is not JDBC compliant, and in Connector/J 5.1 this behavior was changed so that the original table name, rather than the alias, is returned.

  The JDBC-compliant behavior is designed to let API users reconstruct the DML statement based on the metadata within `ResultSet` and `ResultSetMetaData`.

  You can get the alias for a column in a result set by calling `ResultSetMetaData.getColumnLabel()`. To use the old non-compliant behavior with `ResultSetMetaData.getColumnName()`, use the `useOldAliasMetadataBehavior` option and set the value to `true`.

  In Connector/J 5.0.x, the default value of `useOldAliasMetadataBehavior` was `true`, but in Connector/J 5.1 this was changed to a default value of `false`.

3.3.2. JDBC-Specific Issues When Upgrading to MySQL Server 4.1 or Newer

- *Using the UTF-8 Character Encoding* - Prior to MySQL server version 4.1, the UTF-8 character encoding was not supported by the server, however the JDBC driver could use it, allowing storage of multiple character sets in `latin1` tables on the server.

  Starting with MySQL-4.1, this functionality is deprecated. If you have applications that rely on this functionality, and can not upgrade them to use the official Unicode character support in MySQL server version 4.1 or newer, add the following property to your connection URL:

  `useOldUTF8Behavior=true`

- *Server-side Prepared Statements* - Connector/J 3.1 will automatically detect and use server-side prepared statements when they are available (MySQL server version 4.1.0 and newer). If your application encounters issues with server-side prepared statements, you can revert to the older client-side emulated prepared statement code that is still presently used for MySQL servers older than 4.1.0 with the following connection property:

  `useServerPrepStmts=false`

3.3.3. Upgrading from MySQL Connector/J 3.0 to 3.1

Connector/J 3.1 is designed to be backward-compatible with Connector/J 3.0 as much as possible. Major changes are isolated to new functionality exposed in MySQL-4.1 and newer, which includes Unicode character sets, server-side prepared statements, `SQLState` codes returned in error messages by the server and various performance enhancements that can be enabled or disabled using configuration properties.

- **Unicode Character Sets**: See the next section, as well as Character Set Support, for information on this MySQL feature. If you have something misconfigured, it will usually show up as an error with a message similar to `Illegal mix of collations`.

- **Server-side Prepared Statements**: Connector/J 3.1 will automatically detect and use server-side prepared statements when they are available (MySQL server version 4.1.0 and newer).

  Starting with version 3.1.7, the driver scans SQL you are preparing using all variants of `Connection.prepareStatement()` to determine if it is a supported type of statement to prepare on the server side, and if it is not supported by the server, it instead prepares it as a client-side emulated prepared statement. You can disable this feature by passing `emulateUnsupportedPstmts=false` in your JDBC URL.

  If your application encounters issues with server-side prepared statements, you can revert to the older client-side emulated prepared statement code that is still presently used for MySQL servers older than 4.1.0 with the connection property `useServerPrepStmts=false`.

- **Datetimes** with all-zero components (`0000-00-00 ...`): These values cannot be represented reliably in Java. Connector/J 3.0.x always converted them to `NULL` when being read from a ResultSet.

  Connector/J 3.1 throws an exception by default when these values are encountered, as this is the most correct behavior according to the JDBC and SQL standards. This behavior can be modified using the `zeroDateTimeBehavior` configuration property. The permissible values are:

  - `exception` (the default), which throws an SQLException with an SQLState of `S1009`.

  - `convertToNull`, which returns `NULL` instead of the date.

  - `round`, which rounds the date to the nearest closest value which is `0001-01-01`.

  Starting with Connector/J 3.1.7, `ResultSet.getString()` can be decoupled from this behavior using `noDatetimeStringSync=true` (the default value is `false`) so that you can retrieve the unaltered all-zero value as a String. Note that this also precludes using any time zone conversions, therefore the driver will not allow you to enable `noDatetimeStringSync` and `useTimezone` at the same time.

- **New SQLState Codes**: Connector/J 3.1 uses SQL:1999 SQLState codes returned by the MySQL server (if supported), which are different from the legacy X/Open state codes that Connector/J 3.0 uses. If connected to a MySQL server older than MySQL-4.1.0 (the oldest version to return SQLStates as part of the error code), the driver will use a built-in mapping. You can revert to the old mapping by using the configuration property `useSqlStateCodes=false`.

- **`ResultSet.getString()`**: Calling `ResultSet.getString()` on a `BLOB` column will now return the address of the `byte[]` array that represents it, instead of a `String` representation of the `BLOB`. `BLOB` values have no character set, so they cannot be converted to `java.lang.String`s without data loss or corruption.

  To store strings in MySQL with LOB behavior, use one of the `TEXT` types, which the driver will treat as a `java.sql.Clob`.

- **Debug builds**: Starting with Connector/J 3.1.8 a debug build of the driver in a file named `mysql-connector-java-version-bin-g.jar` is shipped alongside the normal binary jar file that is named `mysql-connector-java-version-bin.jar`.

  Starting with Connector/J 3.1.9, we do not ship the `.class` files unbundled, they are only available in the JAR archives that ship with the driver.

  Do not use the debug build of the driver unless instructed to do so when reporting a problem or bug, as it is not designed to be run in production environments, and will have adverse performance impact when used. The debug binary also depends on the Aspect/J runtime library, which is located in the `src/lib/aspectjrt.jar` file that comes with the Connector/J distribution.

3.4. Installing from the Development Source Tree

> **Caution**
>
> Read this section only if you are interested in helping us test our new code. To just get MySQL Connector/J up and running on your system, use a standard binary release distribution.

To install MySQL Connector/J from the development source tree, make sure that you have the following prerequisites:

- A Bazaar client, to check out the sources from our Launchpad repository (available from http://bazaar-vcs.org/).

- Apache Ant version 1.7 or newer (available from http://ant.apache.org/).

- JDK 1.4.2 or later. Although MySQL Connector/J can be be used with older JDKs, compiling it from source requires at least JDK 1.4.2. To build Connector/J 5.1 requires JDK 1.6.x *and* an older JDK such as JDK 1.5.x; point your `JAVA_HOME` environment variable at the older installation.

To check out and compile a specific branch of MySQL Connector/J, follow these steps:

1. Check out the latest code from the branch that you want with one of the following commands.

   The source code repository for MySQL Connector/J is located on Launchpad at https://code.launchpad.net/connectorj. To check out the latest development branch, use:

   ```
   shell> bzr branch lp:connectorji
   ```

   This creates a `connectorj` subdirectory in the current directory that contains the latest sources for the requested branch.

   To check out the latest 5.1 code, use:

   ```
   shell> bzr branch lp:connectorj/5.1
   ```

   This creates a `5.1` subdirectory in the current directory containing the latest 5.1 code.

2. To build Connector/J 5.1, make sure that you have both JDK 1.6.x installed and an older JDK such as JDK 1.5.x. This is because Connector/J supports both JDBC 3.0 (which was prior to JDK 1.6.x) and JDBC 4.0. Set your `JAVA_HOME` environment variable to the path of the older JDK installation.

3. Change your current working directory to either the `connectorj` or `5.1` directory, depending on which branch you intend to build.

4. To build Connector/J 5.1, edit the `build.xml` to reflect the location of your JDK 1.6.x installation. The lines to change are:

   ```
           <property name="com.mysql.jdbc.java6.javac" value="C:\jvms\jdk1.6.0\bin\javac.exe" />
           <property name="com.mysql.jdbc.java6.rtjar" value="C:\jvms\jdk1.6.0\jre\lib\rt.jar" />
   ```

   Alternatively, you can set the value of these property names through the Ant `-D` option.

5. Issue the following command to compile the driver and create a `.jar` file suitable for installation:

   ```
   shell> ant dist
   ```

   This creates a `build` directory in the current directory, where all build output will go. A directory is created in the `build` directory that includes the version number of the sources you are building from. This directory contains the sources, compiled `.class` files, and a `.jar` file suitable for deployment. For other possible targets, including ones that will create a fully packaged distribution, issue the following command:

   ```
   shell> ant -projecthelp
   ```

6. A newly created `.jar` file containing the JDBC driver will be placed in the directory `build/mysql-connector-java-version`.

   Install the newly created JDBC driver as you would a binary `.jar` file that you download from MySQL, by following the instruc-

tions in Section 3.2, "Installing the Driver and Configuring the CLASSPATH".

A package containing both the binary and source code for Connector/J 5.1 can also be found at the following location: Connector/J 5.1 Download

Chapter 4. Connector/J Examples

Examples of using Connector/J are located throughout this document. This section provides a summary and links to these examples.

Chapter 5. Connector/J (JDBC) Reference

This section of the manual contains reference material for MySQL Connector/J.

5.1. Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J

The name of the class that implements `java.sql.Driver` in MySQL Connector/J is `com.mysql.jdbc.Driver`. The `org.gjt.mm.mysql.Driver` class name is also usable for backward compatibility with MM.MySQL, the predecessor of Connector/J. Use this class name when registering the driver, or when otherwise configuring software to use MySQL Connector/J.

JDBC URL Format

The JDBC URL format for MySQL Connector/J is as follows, with items in square brackets ([,]) being optional:

```
jdbc:mysql://[host][,failoverhost...][:port]/[database] »
[?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
```

If the host name is not specified, it defaults to `127.0.0.1`. If the port is not specified, it defaults to `3306`, the default port number for MySQL servers.

```
jdbc:mysql://[host:port],[host:port].../[database] »
[?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
```

Here is a sample connection URL:

```
jdbc:mysql://localhost:3306/sakila?profileSQL=true
```

IPv6 Connections

For IPv6 connections, use this alternative syntax to specify hosts in the URL, `address=(key=value)`. Supported keys are:

- `(protocol=tcp)`, or `(protocol=pipe)` for named pipes on Windows.

- `(path=path_to_pipe)` for named pipes.

- `(host=hostname)` for TCP connections.

- `(port=port_number)` for TCP connections.

For example:

```
jdbc:mysql://address=(protocol=tcp)(host=localhost)(port=3306)(user=test)/db
```

Any other parameters are treated as host-specific properties that follow the conventions of the JDBC URL properties. This now allows per-host overrides of any configuration property for multi-host connections (that is, when using failover, load balancing, or replication). Limit the overrides to user, password, network timeouts and statement and metadata cache sizes; the results of other per-host overrides are not defined.

Initial Database for Connection

If the database is not specified, the connection is made with no default database. In this case, either call the `setCatalog()` method on the Connection instance, or fully specify table names using the database name (that is, `SELECT dbname.tablename.colname FROM dbname.tablename...`) in your SQL. Opening a connection without specifying the database to use is generally only useful when building tools that work with multiple databases, such as GUI database managers.

> **Note**

> Always use the `Connection.setCatalog()` method to specify the desired database in JDBC applications, rather than the `USE database` statement.

Failover Support

MySQL Connector/J has failover support. This enables the driver to fail over to any number of slave hosts and still perform read-only queries. Failover only happens when the connection is in an `autoCommit(true)` state, because failover cannot happen reliably when a transaction is in progress. Most application servers and connection pools set `autoCommit` to `true` at the end of every transaction/connection use.

The failover functionality has the following behavior:

- If the URL property `autoReconnect` is `false`: Failover only happens at connection initialization, and failback occurs when the driver determines that the first host has become available again.

- If the URL property `autoReconnect` is `true`: Failover happens when the driver determines that the connection has failed (checked before *every* query), and falls back to the first host when it determines that the host has become available again (after `queriesBeforeRetryMaster` queries have been issued).

In either case, whenever you are connected to a "failed-over" server, the connection is set to read-only state, so queries that attempt to modify data will throw exceptions (the query will **never** be processed by the MySQL server).

Setting Configuration Properties

Configuration properties define how Connector/J will make a connection to a MySQL server. Unless otherwise noted, properties can be set for a `DataSource` object or for a `Connection` object.

Configuration properties can be set in one of the following ways:

- Using the `set*()` methods on MySQL implementations of `java.sql.DataSource` (which is the preferred method when using implementations of `java.sql.DataSource`):

  - `com.mysql.jdbc.jdbc2.optional.MysqlDataSource`

  - `com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource`

- As a key/value pair in the `java.util.Properties` instance passed to `DriverManager.getConnection()` or `Driver.connect()`

- As a JDBC URL parameter in the URL given to `java.sql.DriverManager.getConnection()`, `java.sql.Driver.connect()` or the MySQL implementations of the `javax.sql.DataSource setURL()` method.

> **Note**
>
> If the mechanism you use to configure a JDBC URL is XML-based, use the XML character literal & to separate configuration parameters, as the ampersand is a reserved character for XML.

The properties are listed in the following tables.

Connection/Authentication.

Property Name	Definition	Default Value	Since Version
user	The user to connect as		all versions
password	The password to use when connecting		all versions
socketFactory	The name of the class that the driver should use for creating socket connections to the server. This class must implement the interface 'com.mysql.jdbc.SocketFactory' and have public no-args constructor.	com.mysql.jdbc.Standard-Socket-	3.0.3

		Factory	
`connectTimeout`	Timeout for socket connect (in milliseconds), with 0 being no timeout. Only works on JDK-1.4 or newer. Defaults to '0'.	`0`	3.0.1
`socketTimeout`	Timeout on network socket operations (0, the default means no timeout).	`0`	3.0.1
`connectionLifecycleInter-ceptors`	A comma-delimited list of classes that implement "com.mysql.jdbc.ConnectionLifecycleInterceptor" that should notified of connection lifecycle events (creation, destruction, commit, rollback, setCatalog and setAutoCommit) and potentially alter the execution of these commands. ConnectionLifecycleInterceptors are "stackable", more than one interceptor may be specified via the configuration property as a comma-delimited list, with the interceptors executed in order from left to right.		5.1.4
`useConfigs`	Load the comma-delimited list of configuration properties before parsing the URL or applying user-specified properties. These configurations are explained in the 'Configurations' of the documentation.		3.1.5
`authenticationPlugins`	Comma-delimited list of classes that implement com.mysql.jdbc.AuthenticationPlugin and which will be used for authentication unless disabled by "disabledAuthenticationPlugins" property.		5.1.19
`defaultAuthenticationPlu-gin`	Name of a class implementing com.mysql.jdbc.AuthenticationPlugin which will be used as the default authentication plugin (see below). It is an error to use a class which is not listed in "authenticationPlugins" nor it is one of the built-in plugins. It is an error to set as default a plugin which was disabled with "disabledAuthenticationPlugins" property. It is an error to set this value to null or the empty string (i.e. there must be at least a valid default authentication plugin specified for the connection, meeting all constraints listed above).	`com.mysql.jdbc.a uthentic ation.My sqlNat-ivePass-wordPlu-gin`	5.1.19
`disabledAuthenticationPlu-gins`	Comma-delimited list of classes implementing com.mysql.jdbc.AuthenticationPlugin or mechanisms, i.e. "mysql_native_password". The authentication plugins or mechanisms listed will not be used for authentication which will fail if it requires one of them. It is an error to disable the default authentication plugin (either the one named by "defaultAuthenticationPlugin" property or the hard-coded one if "defaultAuthenticationPlugin" propery is not set).		5.1.19
`interactiveClient`	Set the CLIENT_INTERACTIVE flag, which tells MySQL to timeout connections based on INTERACTIVE_TIMEOUT instead of WAIT_TIMEOUT	`false`	3.1.0
`localSocketAddress`	Hostname or IP address given to explicitly configure the interface that the driver will bind the client side of the TCP/IP connection to when connecting.		5.0.5
`propertiesTransform`	An implementation of com.mysql.jdbc.ConnectionPropertiesTransform that the driver will use to modify URL properties passed to the driver before attempting a connection		3.1.4
`useCompression`	Use zlib compression when communicating with the server (true/false)? Defaults to 'false'.	`false`	3.0.17

Networking.

Property Name	Definition	Default Value	Since Version
`maxAllowedPacket`	Maximum allowed packet size to send to server. If not set, the value of system variable 'max_allowed_packet' will be used to initialize this upon connecting. This value will not take effect if set	`-1`	5.1.8

	larger than the value of 'max_allowed_packet'.		
tcpKeepAlive	If connecting using TCP/IP, should the driver set SO_KEEPALIVE?	true	5.0.7
tcpNoDelay	If connecting using TCP/IP, should the driver set SO_TCP_NODELAY (disabling the Nagle Algorithm)?	true	5.0.7
tcpRcvBuf	If connecting using TCP/IP, should the driver set SO_RCV_BUF to the given value? The default value of '0', means use the platform default value for this property)	0	5.0.7
tcpSndBuf	If connecting using TCP/IP, should the driver set SO_SND_BUF to the given value? The default value of '0', means use the platform default value for this property)	0	5.0.7
tcpTrafficClass	If connecting using TCP/IP, should the driver set traffic class or type-of-service fields ?See the documentation for java.net.Socket.setTrafficClass() for more information.	0	5.0.7

High Availability and Clustering.

Property Name	Definition	Default Value	Since Version
autoReconnect	Should the driver try to re-establish stale and/or dead connections? If enabled the driver will throw an exception for a queries issued on a stale or dead connection, which belong to the current transaction, but will attempt reconnect before the next query issued on the connection in a new transaction. The use of this feature is not recommended, because it has side effects related to session state and data consistency when applications don't handle SQLExceptions properly, and is only designed to be used when you are unable to configure your application to handle SQLExceptions resulting from dead and stale connections properly. Alternatively, as a last option, investigate setting the MySQL server variable "wait_timeout" to a high value, rather than the default of 8 hours.	false	1.1
autoReconnectForPools	Use a reconnection strategy appropriate for connection pools (defaults to 'false')	false	3.1.3
failOverReadOnly	When failing over in autoReconnect mode, should the connection be set to 'read-only'?	true	3.0.12
maxReconnects	Maximum number of reconnects to attempt if autoReconnect is true, default is '3'.	3	1.1
reconnectAtTxEnd	If autoReconnect is set to true, should the driver attempt reconnections at the end of every transaction?	false	3.0.10
retriesAllDown	When using loadbalancing, the number of times the driver should cycle through available hosts, attempting to connect. Between cycles, the driver will pause for 250ms if no servers are available.	120	5.1.6
initialTimeout	If autoReconnect is enabled, the initial time to wait between reconnect attempts (in seconds, defaults to '2').	2	1.1
roundRobinLoadBalance	When autoReconnect is enabled, and failoverReadonly is false, should we pick hosts to connect to on a round-robin basis?	false	3.1.2
queriesBeforeRetryMaster	Number of queries to issue before falling back to master when failed over (when using multi-host failover). Whichever condition is met first, 'queriesBeforeRetryMaster' or 'secondsBeforeRetryMaster' will cause an attempt to be made to reconnect to the master. Defaults to 50.	50	3.0.2
secondsBeforeRetryMaster	How long should the driver wait, when failed over, before attempting	30	3.0.2
selfDestructOnPingMaxOperations	=If set to a non-zero value, the driver will report close the connection and report failure when Connection.ping() or Connection.isValid(int) is called if the connnection's count of commands	0	5.1.6

	sent to the server exceeds this value.		
`selfDestructOnPingSecond-sLifetime`	If set to a non-zero value, the driver will report close the connection and report failure when Connection.ping() or Connection.isValid(int) is called if the connnection's lifetime exceeds this value.	`0`	5.1.6
`resourceId`	A globally unique name that identifies the resource that this datasource or connection is connected to, used for XAResource.isSameRM() when the driver can't determine this value based on hostnames used in the URL		5.0.1

Security.

Property Name	Definition	Default Value	Since Version
`allowMultiQueries`	Allow the use of ';' to delimit multiple queries during one statement (true/false), defaults to 'false', and does not affect the addBatch() and executeBatch() methods, which instead rely on rewriteBatchStatements.	`false`	3.1.1
`useSSL`	Use SSL when communicating with the server (true/false), defaults to 'false'	`false`	3.0.2
`requireSSL`	Require SSL connection if useSSL=true? (defaults to 'false').	`false`	3.1.0
`verifyServerCertificate`	If "useSSL" is set to "true", should the driver verify the server's certificate? When using this feature, the keystore parameters should be specified by the "clientCertificateKeyStore*" properties, rather than system properties.	`true`	5.1.6
`clientCertificateKey-StoreUrl`	URL to the client certificate KeyStore (if not specified, use defaults)		5.1.0
`clientCertificateKeyStore-Type`	KeyStore type for client certificates (NULL or empty means use the default, which is "JKS". Standard keystore types supported by the JVM are "JKS" and "PKCS12", your environment may have more available depending on what security products are installed and available to the JVM.	`JKS`	5.1.0
`clientCertificateKeyStore-Password`	Password for the client certificates KeyStore		5.1.0
`trustCertificateKey-StoreUrl`	URL to the trusted root certificate KeyStore (if not specified, use defaults)		5.1.0
`trustCertificateKeyStore-Type`	KeyStore type for trusted root certificates (NULL or empty means use the default, which is "JKS". Standard keystore types supported by the JVM are "JKS" and "PKCS12", your environment may have more available depending on what security products are installed and available to the JVM.	`JKS`	5.1.0
`trustCertificateKeyStore-Password`	Password for the trusted root certificates KeyStore		5.1.0
`allowLoadLocalInfile`	Should the driver allow use of 'LOAD DATA LOCAL INFILE...' (defaults to 'true').	`true`	3.0.3
`allowUrlInLocalInfile`	Should the driver allow URLs in 'LOAD DATA LOCAL INFILE' statements?	`false`	3.1.4
`paranoid`	Take measures to prevent exposure sensitive information in error messages and clear data structures holding sensitive data when possible? (defaults to 'false')	`false`	3.0.1
`passwordCharacterEncoding`	What character encoding is used for passwords? Leaving this set to the default value (null), uses the platform character set, which works for ISO8859_1 (i.e. "latin1") passwords. For passwords in other character encodings, the encoding will have to be specified with this property, as it's not possible for the driver to auto-detect this.		5.1.7

Performance Extensions.

Property Name	Definition	Default Value	Since Version
callableStmtCacheSize	If 'cacheCallableStmts' is enabled, how many callable statements should be cached?	100	3.1.2
metadataCacheSize	The number of queries to cache ResultSetMetadata for if cacheResultSetMetaData is set to 'true' (default 50)	50	3.1.1
useLocalSessionState	Should the driver refer to the internal values of autocommit and transaction isolation that are set by Connection.setAutoCommit() and Connection.setTransactionIsolation() and transaction state as maintained by the protocol, rather than querying the database or blindly sending commands to the database for commit() or rollback() method calls?	false	3.1.7
useLocalTransactionState	Should the driver use the in-transaction state provided by the MySQL protocol to determine if a commit() or rollback() should actually be sent to the database?	false	5.1.7
prepStmtCacheSize	If prepared statement caching is enabled, how many prepared statements should be cached?	25	3.0.10
prepStmtCacheSqlLimit	If prepared statement caching is enabled, what's the largest SQL the driver will cache the parsing for?	256	3.0.10
alwaysSendSetIsolation	Should the driver always communicate with the database when Connection.setTransactionIsolation() is called? If set to false, the driver will only communicate with the database when the requested transaction isolation is different than the whichever is newer, the last value that was set via Connection.setTransactionIsolation(), or the value that was read from the server when the connection was established. Note that useLocalSessionState=true will force the same behavior as alwaysSendSetIsolation=false, regardless of how alwaysSendSetIsolation is set.	true	3.1.7
maintainTimeStats	Should the driver maintain various internal timers to enable idle time calculations as well as more verbose error messages when the connection to the server fails? Setting this property to false removes at least two calls to System.getCurrentTimeMillis() per query.	true	3.1.9
useCursorFetch	If connected to MySQL > 5.0.2, and setFetchSize() > 0 on a statement, should that statement use cursor-based fetching to retrieve rows?	false	5.0.0
blobSendChunkSize	Chunk to use when sending BLOB/CLOBs via ServerPreparedStatements	1048576	3.1.9
cacheCallableStmts	Should the driver cache the parsing stage of CallableStatements	false	3.1.2
cachePrepStmts	Should the driver cache the parsing stage of PreparedStatements of client-side prepared statements, the "check" for suitability of server-side prepared and server-side prepared statements themselves?	false	3.0.10
cacheResultSetMetadata	Should the driver cache ResultSetMetaData for Statements and PreparedStatements? (Req. JDK-1.4+, true/false, default 'false')	false	3.1.1
cacheServerConfiguration	Should the driver cache the results of 'SHOW VARIABLES' and 'SHOW COLLATION' on a per-URL basis?	false	3.1.5
defaultFetchSize	The driver will call setFetchSize(n) with this value on all newly-created Statements	0	3.1.9
dontTrackOpenResources	The JDBC specification requires the driver to automatically track and close resources, however if your application doesn't do a good job of explicitly calling close() on statements or result sets, this can cause memory leakage. Setting this property to true relaxes this constraint, and can be more memory efficient for some applications.	false	3.1.7

`dynamicCalendars`	Should the driver retrieve the default calendar when required, or cache it per connection/session?	`false`	3.1.5
`elideSetAutoCommits`	If using MySQL-4.1 or newer, should the driver only issue 'set autocommit=n' queries when the server's state doesn't match the requested state by Connection.setAutoCommit(boolean)?	`false`	3.1.3
`enableQueryTimeouts`	When enabled, query timeouts set via Statement.setQueryTimeout() use a shared java.util.Timer instance for scheduling. Even if the timeout doesn't expire before the query is processed, there will be memory used by the TimerTask for the given timeout which won't be reclaimed until the time the timeout would have expired if it hadn't been cancelled by the driver. High-load environments might want to consider disabling this functionality.	`true`	5.0.6
`holdResultsOpenOverState-` `mentClose`	Should the driver close result sets on Statement.close() as required by the JDBC specification?	`false`	3.1.7
`largeRowSizeThreshold`	What size result set row should the JDBC driver consider "large", and thus use a more memory-efficient way of representing the row internally?	`2048`	5.1.1
`loadBalanceStrategy`	If using a load-balanced connection to connect to SQL nodes in a MySQL Cluster/NDB configuration (by using the URL prefix "jdbc:mysql:loadbalance://"), which load balancing algorithm should the driver use: (1) "random" - the driver will pick a random host for each request. This tends to work better than round-robin, as the randomness will somewhat account for spreading loads where requests vary in response time, while round-robin can sometimes lead to overloaded nodes if there are variations in response times across the workload. (2) "bestResponseTime" - the driver will route the request to the host that had the best response time for the previous transaction.	`random`	5.0.6
`locatorFetchBufferSize`	If 'emulateLocators' is configured to 'true', what size buffer should be used when fetching BLOB data for getBinaryInputStream?	`1048576`	3.2.1
`rewriteBatchedStatements`	Should the driver use multiqueries (irregardless of the setting of "allowMultiQueries") as well as rewriting of prepared statements for INSERT into multi-value inserts when executeBatch() is called? Notice that this has the potential for SQL injection if using plain java.sql.Statements and your code doesn't sanitize input correctly. Notice that for prepared statements, server-side prepared statements can not currently take advantage of this rewrite option, and that if you don't specify stream lengths when using PreparedStatement.set*Stream(), the driver won't be able to determine the optimum number of parameters per batch and you might receive an error from the driver that the resultant packet is too large. Statement.getGeneratedKeys() for these rewritten statements only works when the entire batch includes INSERT statements.	`false`	3.1.13
`useDirectRowUnpack`	Use newer result set row unpacking code that skips a copy from network buffers to a MySQL packet instance and instead reads directly into the result set row data buffers.	`true`	5.1.1
`useDynamicCharsetInfo`	Should the driver use a per-connection cache of character set information queried from the server when necessary, or use a built-in static mapping that is more efficient, but isn't aware of custom character sets or character sets implemented after the release of the JDBC driver?	`true`	5.0.6
`useFastDateParsing`	Use internal String->Date/Time/Timestamp conversion routines to avoid excessive object creation?	`true`	5.0.5
`useFastIntParsing`	Use internal String->Integer conversion routines to avoid excessive object creation?	`true`	3.1.4
`useJvmCharsetConverters`	Always use the character encoding routines built into the JVM, rather than using lookup tables for single-byte character sets?	`false`	5.0.1

`useReadAheadInput`	Use newer, optimized non-blocking, buffered input stream when reading from the server?	`true`	3.1.5

Debugging/Profiling.

Property Name	Definition	Default Value	Since Version
`logger`	The name of a class that implements "com.mysql.jdbc.log.Log" that will be used to log messages to. (default is "com.mysql.jdbc.log.StandardLogger", which logs to STDERR)	`com.mysql.jdbc.log.StandardLogger`	3.1.1
`gatherPerfMetrics`	Should the driver gather performance metrics, and report them via the configured logger every 'reportMetricsIntervalMillis' milliseconds?	`false`	3.1.2
`profileSQL`	Trace queries and their execution/fetch times to the configured logger (true/false) defaults to 'false'	`false`	3.1.0
`profileSql`	Deprecated, use 'profileSQL' instead. Trace queries and their execution/fetch times on STDERR (true/false) defaults to 'false'		2.0.14
`reportMetricsIntervalMillis`	If 'gatherPerfMetrics' is enabled, how often should they be logged (in ms)?	`30000`	3.1.2
`maxQuerySizeToLog`	Controls the maximum length/size of a query that will get logged when profiling or tracing	`2048`	3.1.3
`packetDebugBufferSize`	The maximum number of packets to retain when 'enablePacketDebug' is true	`20`	3.1.3
`slowQueryThresholdMillis`	If 'logSlowQueries' is enabled, how long should a query (in ms) before it is logged as 'slow'?	`2000`	3.1.2
`slowQueryThresholdNanos`	If 'useNanosForElapsedTime' is set to true, and this property is set to a non-zero value, the driver will use this threshold (in nanosecond units) to determine if a query was slow.	`0`	5.0.7
`useUsageAdvisor`	Should the driver issue 'usage' warnings advising proper and efficient usage of JDBC and MySQL Connector/J to the log (true/false, defaults to 'false')?	`false`	3.1.1
`autoGenerateTestcaseScript`	Should the driver dump the SQL it is executing, including server-side prepared statements to STDERR?	`false`	3.1.9
`autoSlowLog`	Instead of using slowQueryThreshold* to determine if a query is slow enough to be logged, maintain statistics that allow the driver to determine queries that are outside the 99th percentile?	`true`	5.1.4
`clientInfoProvider`	The name of a class that implements the com.mysql.jdbc.JDBC4ClientInfoProvider interface in order to support JDBC-4.0's Connection.get/setClientInfo() methods	`com.mysql.jdbc.JDBC4CommentClientInfoProvider`	5.1.0
`dumpMetadataOnColumnNotFound`	Should the driver dump the field-level metadata of a result set into the exception message when ResultSet.findColumn() fails?	`false`	3.1.13
`dumpQueriesOnException`	Should the driver dump the contents of the query sent to the server in the message for SQLExceptions?	`false`	3.1.3
`enablePacketDebug`	When enabled, a ring-buffer of 'packetDebugBufferSize' packets will be kept, and dumped when exceptions are thrown in key areas in the driver's code	`false`	3.1.3
`explainSlowQueries`	If 'logSlowQueries' is enabled, should the driver automatically issue an 'EXPLAIN' on the server and send the results to the configured log at a WARN level?	`false`	3.1.2

includeInnodbStatusInDead-lockExceptions	Include the output of "SHOW ENGINE INNODB STATUS" in exception messages when deadlock exceptions are detected?	false	5.0.7
includeThreadDumpInDead-lockExceptions	Include a current Java thread dump in exception messages when deadlock exceptions are detected?	false	5.1.15
includeThreadNamesAsState-mentComment	Include the name of the current thread as a comment visible in "SHOW PROCESSLIST", or in Innodb deadlock dumps, useful in correlation with "includeInnodbStatusInDeadlockExceptions=true" and "includeThreadDumpInDeadlockExceptions=true".	false	5.1.15
logSlowQueries	Should queries that take longer than 'slowQueryThresholdMillis' be logged?	false	3.1.2
logXaCommands	Should the driver log XA commands sent by MysqlXaConnection to the server, at the DEBUG level of logging?	false	5.0.5
profilerEventHandler	Name of a class that implements the interface com.mysql.jdbc.profiler.ProfilerEventHandler that will be used to handle profiling/tracing events.	com.mysql.jdbc.pro-filer.Logging-Pro-filerEventHand-ler	5.1.6
resultSetSizeThreshold	If the usage advisor is enabled, how many rows should a result set contain before the driver warns that it is suspiciously large?	100	5.0.5
traceProtocol	Should trace-level network protocol be logged?	false	3.1.2
useNanosForElapsedTime	For profiling/debugging functionality that measures elapsed time, should the driver try to use nanoseconds resolution if available (JDK >= 1.5)?	false	5.0.7

Miscellaneous.

Property Name	Definition	Default Value	Since Version
useUnicode	Should the driver use Unicode character encodings when handling strings? Should only be used when the driver can't determine the character set mapping, or you are trying to 'force' the driver to use a character set that MySQL either doesn't natively support (such as UTF-8), true/false, defaults to 'true'	true	1.1g
characterEncoding	If 'useUnicode' is set to true, what character encoding should the driver use when dealing with strings? (defaults is to 'autodetect')		1.1g
characterSetResults	Character set to tell the server to return results as.		3.0.13
connectionCollation	If set, tells the server to use this collation via 'set colla-tion_connection'		3.0.13
useBlobToSt-oreUTF8OutsideBMP	Tells the driver to treat [MEDIUM/LONG]BLOB columns as [LONG]VARCHAR columns holding text encoded in UTF-8 that has characters outside the BMP (4-byte encodings), which MySQL server can't handle natively.	false	5.1.3
utf8OutsideBmpExcludedColumnNamePattern	When "useBlobToStoreUTF8OutsideBMP" is set to "true", column names matching the given regex will still be treated as BLOBs unless they match the regex specified for "utf8OutsideBmpIncludedColumnNamePattern". The regex must follow the patterns used for the java.util.regex package.		5.1.3
utf8OutsideBmpIncludedColumnNamePattern	Used to specify exclusion rules to "utf8OutsideBmpExcludedColumnNamePattern". The regex must follow the patterns used for the java.util.regex package.		5.1.3
loadBalanceEnableJMX	Enables JMX-based management of load-balanced connection	false	5.1.13

	groups, including live addition/removal of hosts from load-balancing pool.		
sessionVariables	A comma-separated list of name/value pairs to be sent as SET SESSION ... to the server when the driver connects.		3.1.8
useColumnNamesInFindColumn	Prior to JDBC-4.0, the JDBC specification had a bug related to what could be given as a "column name" to ResultSet methods like findColumn(), or getters that took a String property. JDBC-4.0 clarified "column name" to mean the label, as given in an "AS" clause and returned by ResultSetMetaData.getColumnLabel(), and if no AS clause, the column name. Setting this property to "true" will give behavior that is congruent to JDBC-3.0 and earlier versions of the JDBC specification, but which because of the specification bug could give unexpected results. This property is preferred over "useOldAliasMetadataBehavior" unless you need the specific behavior that it provides with respect to ResultSetMetadata.	false	5.1.7
allowNanAndInf	Should the driver allow NaN or +/- INF values in PreparedStatement.setDouble()?	false	3.1.5
autoClosePStmtStreams	Should the driver automatically call .close() on streams/readers passed as arguments via set*() methods?	false	3.1.12
autoDeserialize	Should the driver automatically detect and de-serialize objects stored in BLOB fields?	false	3.1.5
blobsAreStrings	Should the driver always treat BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?	false	5.0.8
capitalizeTypeNames	Capitalize type names in DatabaseMetaData? (usually only useful when using WebObjects, true/false, defaults to 'false')	true	2.0.7
clobCharacterEncoding	The character encoding to use for sending and retrieving TEXT, MEDIUMTEXT and LONGTEXT values instead of the configured connection characterEncoding		5.0.0
clobberStreamingResults	This will cause a 'streaming' ResultSet to be automatically closed, and any outstanding data still streaming from the server to be discarded if another query is executed before all the data has been read from the server.	false	3.0.9
compensateOnDuplicateKey-UpdateCounts	Should the driver compensate for the update counts of "ON DUPLICATE KEY" INSERT statements (2 = 1, 0 = 1) when using prepared statements?	false	5.1.7
continueBatchOnError	Should the driver continue processing batch commands if one statement fails. The JDBC spec allows either way (defaults to 'true').	true	3.0.3
createDatabaseIfNotExist	Creates the database given in the URL if it doesn't yet exist. Assumes the configured user has permissions to create databases.	false	3.1.9
emptyStringsConvertToZero	Should the driver allow conversions from empty string fields to numeric values of '0'?	true	3.1.8
emulateLocators	Should the driver emulate java.sql.Blobs with locators? With this feature enabled, the driver will delay loading the actual Blob data until the one of the retrieval methods (getInputStream(), getBytes(), and so forth) on the blob data stream has been accessed. For this to work, you must use a column alias with the value of the column to the actual name of the Blob. The feature also has the following restrictions: The SELECT that created the result set must reference only one table, the table must have a primary key; the SELECT must alias the original blob column name, specified as a string, to an alternate name; the SELECT must cover all columns that make up the primary key.	false	3.1.0
emulateUnsupportedPstmts	Should the driver detect prepared statements that are not supported by the server, and replace them with client-side emulated versions?	true	3.1.7

`exceptionInterceptors`	Comma-delimited list of classes that implement com.mysql.jdbc.ExceptionInterceptor. These classes will be instantiated one per Connection instance, and all SQLExceptions thrown by the driver will be allowed to be intercepted by these interceptors, in a chained fashion, with the first class listed as the head of the chain.		5.1.8
`functionsNeverReturnBlobs`	Should the driver always treat data from functions returning BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?	`false`	5.0.8
`generateSimpleParameter-Metadata`	Should the driver generate simplified parameter metadata for PreparedStatements when no metadata is available either because the server couldn't support preparing the statement, or server-side prepared statements are disabled?	`false`	5.0.5
`ignoreNonTxTables`	Ignore non-transactional table warning for rollback? (defaults to 'false').	`false`	3.0.9
`jdbcCompliantTruncation`	Should the driver throw java.sql.DataTruncation exceptions when data is truncated as is required by the JDBC specification when connected to a server that supports warnings (MySQL 4.1.0 and newer)? This property has no effect if the server sql-mode includes STRICT_TRANS_TABLES.	`true`	3.1.2
`loadBalanceAutoCommit-StatementRegex`	When load-balancing is enabled for auto-commit statements (via loadBalanceAutoCommitStatementThreshold), the statement counter will only increment when the SQL matches the regular expression. By default, every statement issued matches.		5.1.15
`loadBalanceAutoCommit-StatementThreshold`	When auto-commit is enabled, the number of statements which should be executed before triggering load-balancing to rebalance. Default value of 0 causes load-balanced connections to only rebalance when exceptions are encountered, or auto-commit is disabled and transactions are explicitly committed or rolled back.	`0`	5.1.15
`loadBalanceBlacklist-Timeout`	Time in milliseconds between checks of servers which are unavailable, by controlling how long a server lives in the global blacklist.	`0`	5.1.0
`loadBalanceConnectionGroup`	Logical group of load-balanced connections within a classloader, used to manage different groups independently. If not specified, live management of load-balanced connections is disabled.		5.1.13
`loadBalanceExceptionCheck-er`	Fully-qualified class name of custom exception checker. The class must implement com.mysql.jdbc.LoadBalanceExceptionChecker interface, and is used to inspect SQLExceptions and determine whether they should trigger fail-over to another host in a load-balanced deployment.	`com.mysql.jdbc.StandardLoadBalanceExceptionChecker`	5.1.13
`loadBalancePingTimeout`	Time in milliseconds to wait for ping response from each of load-balanced physical connections when using load-balanced Connection.	`0`	5.1.13
`loadBalanceSQLException-SubclassFailover`	Comma-delimited list of classes/interfaces used by default load-balanced exception checker to determine whether a given SQLException should trigger failover. The comparison is done using Class.isInstance(SQLException) using the thrown SQLException.		5.1.13
`loadBalanceSQLStateFail-over`	Comma-delimited list of SQLState codes used by default load-balanced exception checker to determine whether a given SQLException should trigger failover. The SQLState of a given SQLException is evaluated to determine whether it begins with any value in the comma-delimited list.		5.1.13
`loadBalanceValidateConnec-tionOnSwapServer`	Should the load-balanced Connection explicitly check whether the connection is live when swapping to a new physical connection at commit/rollback?	`false`	5.1.13
`maxRows`	The maximum number of rows to return (0, the default means re-	`-1`	all versions

	turn all rows).		
`netTimeoutForStreamingRes-ults`	What value should the driver automatically set the server setting 'net_write_timeout' to when the streaming result sets feature is in use? (value has unit of seconds, the value '0' means the driver will not try and adjust this value)	`600`	5.1.0
`noAccessToProcedureBodies`	When determining procedure parameter types for CallableStatements, and the connected user can't access procedure bodies through "SHOW CREATE PROCEDURE" or select on mysql.proc should the driver instead create basic metadata (all parameters reported as IN VARCHARs, but allowing registerOutParameter() to be called on them anyway) instead of throwing an exception?	`false`	5.0.3
`noDatetimeStringSync`	Don't ensure that ResultSet.getDatetimeType().toString().equals(ResultSet.getString())	`false`	3.1.7
`noTimezoneConversionForTimeType`	Don't convert TIME values using the server timezone if 'useTimezone'='true'	`false`	5.0.0
`nullCatalogMeansCurrent`	When DatabaseMetadataMethods ask for a 'catalog' parameter, does the value null mean use the current catalog? (this is not JDBC-compliant, but follows legacy behavior from earlier versions of the driver)	`true`	3.1.8
`nullNamePatternMatchesAll`	Should DatabaseMetaData methods that accept *pattern parameters treat null the same as '%' (this is not JDBC-compliant, however older versions of the driver accepted this departure from the specification)	`true`	3.1.8
`overrideSupportsIntegrity-EnhancementFacility`	Should the driver return "true" for DatabaseMetaData.supportsIntegrityEnhancementFacility() even if the database doesn't support it to workaround applications that require this method to return "true" to signal support of foreign keys, even though the SQL specification states that this facility contains much more than just foreign key support (one such application being OpenOffice)?	`false`	3.1.12
`padCharsWithSpace`	If a result set column has the CHAR type and the value does not fill the amount of characters specified in the DDL for the column, should the driver pad the remaining characters with space (for ANSI compliance)?	`false`	5.0.6
`pedantic`	Follow the JDBC spec to the letter.	`false`	3.0.0
`pinGlobalTxToPhysicalCon-nection`	When using XAConnections, should the driver ensure that operations on a given XID are always routed to the same physical connection? This allows the XAConnection to support "XA START ... JOIN" after "XA END" has been called	`false`	5.0.1
`populateInsertRowWithDe-faultValues`	When using ResultSets that are CONCUR_UPDATABLE, should the driver pre-populate the "insert" row with default values from the DDL for the table used in the query so those values are immediately available for ResultSet accessors? This functionality requires a call to the database for metadata each time a result set of this type is created. If disabled (the default), the default values will be populated by the an internal call to refreshRow() which pulls back default values and/or values changed by triggers.	`false`	5.0.5
`processEscapeCodesForPrep-Stmts`	Should the driver process escape codes in queries that are prepared?	`true`	3.1.12
`queryTimeoutKillsConnec-tion`	If the timeout given in Statement.setQueryTimeout() expires, should the driver forcibly abort the Connection instead of attempting to abort the query?	`false`	5.1.9
`relaxAutoCommit`	If the version of MySQL the driver connects to does not support transactions, still allow calls to commit(), rollback() and setAutoCommit() (true/false, defaults to 'false')?	`false`	2.0.13
`retainStatementAfterRes-ultSetClose`	Should the driver retain the Statement reference in a ResultSet after ResultSet.close() has been called. This is not JDBC-com-	`false`	3.1.11

	pliant after JDBC-4.0.		
`rollbackOnPooledClose`	Should the driver issue a rollback() when the logical connection in a pool is closed?	`true`	3.0.15
`runningCTS13`	Enables workarounds for bugs in Sun's JDBC compliance testsuite version 1.3	`false`	3.1.7
`serverTimezone`	Override detection/mapping of timezone. Used when timezone from server doesn't map to Java timezone		3.0.2
`statementInterceptors`	A comma-delimited list of classes that implement "com.mysql.jdbc.StatementInterceptor" that should be placed "in between" query execution to influence the results. StatementInterceptors are "chainable", the results returned by the "current" interceptor will be passed on to the next in in the chain, from left-to-right order, as specified in this property.		5.1.1
`strictFloatingPoint`	Used only in older versions of compliance test	`false`	3.0.0
`strictUpdates`	Should the driver do strict checking (all primary keys selected) of updatable result sets (true, false, defaults to 'true')?	`true`	3.0.4
`tinyInt1isBit`	Should the driver treat the datatype TINYINT(1) as the BIT type (because the server silently converts BIT -> TINYINT(1) when creating tables)?	`true`	3.0.16
`transformedBitIsBoolean`	If the driver converts TINYINT(1) to a different type, should it use BOOLEAN instead of BIT for future compatibility with MySQL-5.0, as MySQL-5.0 has a BIT type?	`false`	3.1.9
`treatUtilDateAsTimestamp`	Should the driver treat java.util.Date as a TIMESTAMP for the purposes of PreparedStatement.setObject()?	`true`	5.0.5
`ultraDevHack`	Create PreparedStatements for prepareCall() when required, because UltraDev is broken and issues a prepareCall() for _all_ statements? (true/false, defaults to 'false')	`false`	2.0.3
`useAffectedRows`	Don't set the CLIENT_FOUND_ROWS flag when connecting to the server (not JDBC-compliant, will break most applications that rely on "found" rows vs. "affected rows" for DML statements), but does cause "correct" update counts from "INSERT ... ON DUPLICATE KEY UPDATE" statements to be returned by the server.	`false`	5.1.7
`useGmtMillisForDatetimes`	Convert between session timezone and GMT before creating Date and Timestamp instances (value of "false" is legacy behavior, "true" leads to more JDBC-compliant behavior.	`false`	3.1.12
`useHostsInPrivileges`	Add '@hostname' to users in Database-MetaData.getColumn/TablePrivileges() (true/false), defaults to 'true'.	`true`	3.0.2
`useInformationSchema`	When connected to MySQL-5.0.7 or newer, should the driver use the INFORMATION_SCHEMA to derive information used by DatabaseMetaData?	`false`	5.0.0
`useJDBCCompliantTimezone-Shift`	Should the driver use JDBC-compliant rules when converting TIME/TIMESTAMP/DATETIME values' timezone information for those JDBC arguments which take a java.util.Calendar argument? (Notice that this option is exclusive of the "useTimezone=true" configuration option.)	`false`	5.0.0
`useLegacyDatetimeCode`	Use code for DATE/TIME/DATETIME/TIMESTAMP handling in result sets and statements that consistently handles timezone conversions from client to server and back again, or use the legacy code for these datatypes that has been in the driver for backwards-compatibility?	`true`	5.1.6
`useOldAliasMetadataBehavi-or`	Should the driver use the legacy behavior for "AS" clauses on columns and tables, and only return aliases (if any) for ResultSet-MetaData.getColumnName() or ResultSet-MetaData.getTableName() rather than the original column/table name? In 5.0.x, the default value was true.	`false`	5.0.4

`useOldUTF8Behavior`	Use the UTF-8 behavior the driver did when communicating with 4.0 and older servers	`false`	3.1.6
`useOnlyServerErrorMessages`	Don't prepend 'standard' SQLState error messages to error messages returned by the server.	`true`	3.0.15
`useSSPSCompatibleTimezone-Shift`	If migrating from an environment that was using server-side prepared statements, and the configuration property "useJDBCCompliantTimeZoneShift" set to "true", use compatible behavior when not using server-side prepared statements when sending TIMESTAMP values to the MySQL server.	`false`	5.0.5
`useServerPrepStmts`	Use server-side prepared statements if the server supports them?	`false`	3.1.0
`useSqlStateCodes`	Use SQL Standard state codes instead of 'legacy' X/Open/SQL state codes (true/false), default is 'true'	`true`	3.1.3
`useStreamLengthsInPrepSt-mts`	Honor stream length parameter in PreparedStatement/ResultSet.setXXXStream() method calls (true/false, defaults to 'true')?	`true`	3.0.2
`useTimezone`	Convert time/date types between client and server timezones (true/false, defaults to 'false')?	`false`	3.0.2
`useUnbufferedInput`	Don't use BufferedInputStream for reading data from the server	`true`	3.0.11
`yearIsDateType`	Should the JDBC driver treat the MySQL type "YEAR" as a java.sql.Date, or as a SHORT?	`true`	3.1.9
`zeroDateTimeBehavior`	What should happen when the driver encounters DATETIME values that are composed entirely of zeros (used by MySQL to represent invalid dates)? Valid values are "exception", "round" and "convertToNull".	`exception`	3.1.4

Connector/J also supports access to MySQL using named pipes on Windows NT, Windows 2000, or Windows XP using the `NamedPipeSocketFactory` as a plugin-socket factory using the `socketFactory` property. If you do not use a `namedPipePath` property, the default of `'\\.\pipe\MySQL'` is used. If you use the `NamedPipeSocketFactory`, the host name and port number values in the JDBC url are ignored. To enable this feature, use:

```
socketFactory=com.mysql.jdbc.NamedPipeSocketFactory
```

Named pipes only work when connecting to a MySQL server on the same physical machine where the JDBC driver is running. In simple performance tests, named pipe access is between 30%-50% faster than the standard TCP/IP access. However, this varies per system, and named pipes are slower than TCP/IP in many Windows configurations.

To create your own socket factories, follow the example code in `com.mysql.jdbc.NamedPipeSocketFactory`, or `com.mysql.jdbc.StandardSocketFactory`.

5.1.1. Properties Files for the `useConfigs` Option

The `useConfigs` connection option is a convenient shorthand for specifying combinations of options for particular scenarios. The argument values you can use with this option correspond to the names of `.properties` files within the Connector/J `mysql-connector-java-version-bin.jar` JAR file. For example, the Connector/J 5.1.9 driver includes the following configuration properties files:

```
$ unzip mysql-connector-java-5.1.19-bin.jar '*/configs/*'
Archive:  mysql-connector-java-5.1.19-bin.jar
   creating: com/mysql/jdbc/configs/
  inflating: com/mysql/jdbc/configs/3-0-Compat.properties
  inflating: com/mysql/jdbc/configs/5-0-Compat.properties
  inflating: com/mysql/jdbc/configs/clusterBase.properties
  inflating: com/mysql/jdbc/configs/coldFusion.properties
  inflating: com/mysql/jdbc/configs/fullDebug.properties
  inflating: com/mysql/jdbc/configs/maxPerformance.properties
  inflating: com/mysql/jdbc/configs/solarisMaxPerformance.properties
```

To specify one of these combinations of options, specify `useConfigs=3-0-Compat`, `useConfigs=maxPerformance`, and so on. The following sections show the options that are part of each `useConfigs` setting. For the details of why each one is included, see the comments in the `.properties` files.

3-0-Compat

```
emptyStringsConvertToZero=true
jdbcCompliantTruncation=false
noDatetimeStringSync=true
nullCatalogMeansCurrent=true
nullNamePatternMatchesAll=true
transformedBitIsBoolean=false
dontTrackOpenResources=true
zeroDateTimeBehavior=convertToNull
useServerPrepStmts=false
autoClosePStmtStreams=true
processEscapeCodesForPrepStmts=false
useFastDateParsing=false
populateInsertRowWithDefaultValues=false
useDirectRowUnpack=false
```

5-0-Compat

```
useDirectRowUnpack=false
```

clusterBase

```
autoReconnect=true
failOverReadOnly=false
roundRobinLoadBalance=true
```

coldFusion

```
useDynamicCharsetInfo=false
alwaysSendSetIsolation=false
useLocalSessionState=true
autoReconnect=true
```

fullDebug

```
profileSQL=true
gatherPerMetrics=true
useUsageAdvisor=true
logSlowQueries=true
explainSlowQueries=true
```

maxPerformance

```
cachePrepStmts=true
cacheCallableStmts=true
cacheServerConfiguration=true
useLocalSessionState=true
elideSetAutoCommits=true
alwaysSendSetIsolation=false
enableQueryTimeouts=false
```

solarisMaxPerformance

```
useUnbufferedInput=false
useReadAheadInput=false
maintainTimeStats=false
```

5.2. JDBC API Implementation Notes

MySQL Connector/J passes all of the tests in the publicly available version of Sun's JDBC compliance test suite. This section gives details on a interface-by-interface level about implementation decisions that might affect how you code applications with MySQL Connector/J. The JDBC specification is vague about how certain functionality should be implemented, or the specification enables leeway in implementation.

- **BLOB**

Starting with Connector/J version 3.1.0, you can emulate BLOBs with locators by adding the property `emulateLocators=true` to your JDBC URL. Using this method, the driver will delay loading the actual BLOB data until you retrieve the other data and then use retrieval methods (`getInputStream()`, `getBytes()`, and so forth) on the BLOB data stream.

You must use a column alias with the value of the column to the actual name of the BLOB, for example:

```
SELECT id, 'data' as blob_data from blobtable
```

You must also follow these rules:

- The `SELECT` must reference only one table. The table must have a primary key.

- The `SELECT` must alias the original BLOB column name, specified as a string, to an alternate name.

- The `SELECT` must cover all columns that make up the primary key.

The BLOB implementation does not allow in-place modification (they are copies, as reported by the `DatabaseMetaData.locatorsUpdateCopies()` method). Because of this, use the corresponding `PreparedStatement.setBlob()` or `ResultSet.updateBlob()` (in the case of updatable result sets) methods to save changes back to the database.

- **CallableStatement**

  Starting with Connector/J 3.1.1, stored procedures are supported when connecting to MySQL version 5.0 or newer using the `CallableStatement` interface. Currently, the `getParameterMetaData()` method of `CallableStatement` is not supported.

- **CLOB**

  The CLOB implementation does not allow in-place modification (they are copies, as reported by the `DatabaseMetaData.locatorsUpdateCopies()` method). Because of this, use the `PreparedStatement.setClob()` method to save changes back to the database. The JDBC API does not have a `ResultSet.updateClob()` method.

- **Connection**

  Unlike the pre-Connector/J JDBC driver (`MM.MySQL`), the `isClosed()` method does not ping the server to determine if it is available. In accordance with the JDBC specification, it only returns true if `closed()` has been called on the connection. If you need to determine if the connection is still valid, issue a simple query, such as `SELECT 1`. The driver will throw an exception if the connection is no longer valid.

- **DatabaseMetaData**

  Foreign key information (`getImportedKeys()`/`getExportedKeys()` and `getCrossReference()`) is only available from `InnoDB` tables. The driver uses `SHOW CREATE TABLE` to retrieve this information, so if any other storage engines add support for foreign keys, the driver would transparently support them as well.

- **PreparedStatement**

  PreparedStatements are implemented by the driver, as MySQL does not have a prepared statement feature. Because of this, the driver does not implement `getParameterMetaData()` or `getMetaData()` as it would require the driver to have a complete SQL parser in the client.

  Starting with version 3.1.0 MySQL Connector/J, server-side prepared statements and binary-encoded result sets are used when the server supports them.

  Take care when using a server-side prepared statement with **large** parameters that are set using `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()`, `setBlob()`, or `setClob()`. To re-execute the statement with any large parameter changed to a nonlarge parameter, call `clearParameters()` and set all parameters again. The reason for this is as follows:

  - During both server-side prepared statements and client-side emulation, large data is exchanged only when `PreparedStatement.execute()` is called.

  - Once that has been done, the stream used to read the data on the client side is closed (as per the JDBC spec), and cannot be read from again.

- If a parameter changes from large to nonlarge, the driver must reset the server-side state of the prepared statement to allow the parameter that is being changed to take the place of the prior large value. This removes all of the large data that has already been sent to the server, thus requiring the data to be re-sent, using the `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()`, `setBlob()` or `setClob()` method.

Consequently, to change the type of a parameter to a nonlarge one, you must call `clearParameters()` and set all parameters of the prepared statement again before it can be re-executed.

- **ResultSet**

  By default, ResultSets are completely retrieved and stored in memory. In most cases this is the most efficient way to operate, and due to the design of the MySQL network protocol is easier to implement. If you are working with ResultSets that have a large number of rows or large values, and cannot allocate heap space in your JVM for the memory required, you can tell the driver to stream the results back one row at a time.

  To enable this functionality, create a `Statement` instance in the following manner:

  ```
  stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
              java.sql.ResultSet.CONCUR_READ_ONLY);
  stmt.setFetchSize(Integer.MIN_VALUE);
  ```

  The combination of a forward-only, read-only result set, with a fetch size of `Integer.MIN_VALUE` serves as a signal to the driver to stream result sets row-by-row. After this, any result sets created with the statement will be retrieved row-by-row.

  There are some caveats with this approach. You must read all of the rows in the result set (or close it) before you can issue any other queries on the connection, or an exception will be thrown.

  The earliest the locks these statements hold can be released (whether they be `MyISAM` table-level locks or row-level locks in some other storage engine such as `InnoDB`) is when the statement completes.

  If the statement is within scope of a transaction, then locks are released when the transaction completes (which implies that the statement needs to complete first). As with most other databases, statements are not complete until all the results pending on the statement are read or the active result set for the statement is closed.

  Therefore, if using streaming results, process them as quickly as possible if you want to maintain concurrent access to the tables referenced by the statement producing the result set.

- **ResultSetMetaData**

  The `isAutoIncrement()` method only works when using MySQL servers 4.0 and newer.

- **Statement**

  When using versions of the JDBC driver earlier than 3.2.1, and connected to server versions earlier than 5.0.3, the `setFetchSize()` method has no effect, other than to toggle result set streaming as described above.

  Connector/J 5.0.0 and later include support for both `Statement.cancel()` and `Statement.setQueryTimeout()`. Both require MySQL 5.0.0 or newer server, and require a separate connection to issue the `KILL QUERY` statement. In the case of `setQueryTimeout()`, the implementation creates an additional thread to handle the timeout functionality.

  > **Note**
  >
  > Failures to cancel the statement for `setQueryTimeout()` may manifest themselves as `RuntimeException` rather than failing silently, as there is currently no way to unblock the thread that is executing the query being cancelled due to timeout expiration and have it throw the exception instead.

  > **Note**
  >
  > The MySQL statement `KILL QUERY` (which is what the driver uses to implement `Statement.cancel()`) is non-deterministic; thus, avoid the use of `Statement.cancel()` if possible. If no query is in process, the next query issued will be killed by the server. This race condition is guarded against as of Connector/J 5.1.18.

  MySQL does not support SQL cursors, and the JDBC driver doesn't emulate them, so `setCursorName()` has no effect.

  Connector/J 5.1.3 and later include two additional methods:

- `setLocalInfileInputStream()` sets an `InputStream` instance that will be used to send data to the MySQL server for a `LOAD DATA LOCAL INFILE` statement rather than a `FileInputStream` or `URLInputStream` that represents the path given as an argument to the statement.

  This stream will be read to completion upon execution of a `LOAD DATA LOCAL INFILE` statement, and will automatically be closed by the driver, so it needs to be reset before each call to `execute*()` that would cause the MySQL server to request data to fulfill the request for `LOAD DATA LOCAL INFILE`.

  If this value is set to `NULL`, the driver will revert to using a `FileInputStream` or `URLInputStream` as required.

- `getLocalInfileInputStream()` returns the `InputStream` instance that will be used to send data in response to a `LOAD DATA LOCAL INFILE` statement.

  This method returns `NULL` if no such stream has been set using `setLocalInfileInputStream()`.

5.3. Java, JDBC and MySQL Types

MySQL Connector/J is flexible in the way it handles conversions between MySQL data types and Java data types.

In general, any MySQL data type can be converted to a `java.lang.String`, and any numeric type can be converted to any of the Java numeric types, although round-off, overflow, or loss of precision may occur.

> **Note**
>
> All `TEXT` types return `Types.LONGVARCHAR` with different `getPrecision()` values (65535, 255, 16777215, and 2147483647 respectively) with `getColumnType()` returning `-1`. This behavior is intentional even though `TINYTEXT` does not fall, regarding to its size, within the `LONGVARCHAR` category. This is to avoid different handling inside the same base type. And `getColumnType()` returns `-1` because the internal server handling is of type `TEXT`, which is similar to `BLOB`.
>
> Also note that `getColumnTypeName()` will return `VARCHAR` even though `getColumnType()` returns `Types.LONGVARCHAR`, because `VARCHAR` is the designated column database-specific name for this type.

Starting with Connector/J 3.1.0, the JDBC driver issues warnings or throws `DataTruncation` exceptions as is required by the JDBC specification unless the connection was configured not to do so by using the property `jdbcCompliantTruncation` and setting it to `false`.

The conversions that are always guaranteed to work are listed in the following table:

Connection Properties - Miscellaneous.

These MySQL Data Types	Can always be converted to these Java types
`CHAR, VARCHAR, BLOB, TEXT, ENUM, and SET`	`java.lang.String, java.io.InputStream, java.io.Reader, java.sql.Blob, java.sql.Clob`
`FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, IN-TEGER, BIGINT`	`java.lang.String, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Double, java.math.BigDecimal`
`DATE, TIME, DATETIME, TIMESTAMP`	`java.lang.String, java.sql.Date, java.sql.Timestamp`

> **Note**
>
> Round-off, overflow or loss of precision may occur if you choose a Java numeric data type that has less precision or capacity than the MySQL data type you are converting to/from.

The `ResultSet.getObject()` method uses the type conversions between MySQL and Java types, following the JDBC specification where appropriate. The value returned by `ResultSetMetaData.GetColumnClassName()` is also shown below. For more information on the `java.sql.Types` classes see Java 2 Platform Types.

MySQL Types to Java Types for ResultSet.getObject().

MySQL Type Name	Return value of `GetColumn-ClassName`	Returned as Java Class
`BIT(1)` (new in MySQL-5.0)	`BIT`	`java.lang.Boolean`
`BIT( > 1)` (new in MySQL-5.0)	`BIT`	`byte[]`
`TINYINT`	`TINYINT`	`java.lang.Boolean` if the configuration property `tiny-Int1isBit` is set to `true` (the default) and the storage size is 1, or `java.lang.Integer` if not.
`BOOL`, `BOOLEAN`	`TINYINT`	See `TINYINT`, above as these are aliases for `TINYINT(1)`, currently.
`SMALLINT[(M)] [UNSIGNED]`	`SMALLINT [UNSIGNED]`	`java.lang.Integer` (regardless if `UNSIGNED` or not)
`MEDIUMINT[(M)] [UNSIGNED]`	`MEDIUMINT [UNSIGNED]`	`java.lang.Integer`, if `UNSIGNED` `java.lang.Long` (C/J 3.1 and earlier), or `java.lang.Integer` for C/J 5.0 and later
`INT,INTEGER[(M)] [UNSIGNED]`	`INTEGER [UNSIGNED]`	`java.lang.Integer`, if `UNSIGNED` `java.lang.Long`
`BIGINT[(M)] [UNSIGNED]`	`BIGINT [UNSIGNED]`	`java.lang.Long`, if UNSIGNED `java.math.BigInteger`
`FLOAT[(M,D)]`	`FLOAT`	`java.lang.Float`
`DOUBLE[(M,B)]`	`DOUBLE`	`java.lang.Double`
`DECIMAL[(M[,D])]`	`DECIMAL`	`java.math.BigDecimal`
`DATE`	`DATE`	`java.sql.Date`
`DATETIME`	`DATETIME`	`java.sql.Timestamp`
`TIMESTAMP[(M)]`	`TIMESTAMP`	`java.sql.Timestamp`
`TIME`	`TIME`	`java.sql.Time`
`YEAR[(2|4)]`	`YEAR`	If `yearIsDateType` configuration property is set to `false`, then the returned object type is `java.sql.Short`. If set to `true` (the default), then the returned object is of type `java.sql.Date` with the date set to January 1st, at midnight.
`CHAR(M)`	`CHAR`	`java.lang.String` (unless the character set for the column is `BINARY`, then `byte[]` is returned.
`VARCHAR(M) [BINARY]`	`VARCHAR`	`java.lang.String` (unless the character set for the column is `BINARY`, then `byte[]` is returned.
`BINARY(M)`	`BINARY`	`byte[]`
`VARBINARY(M)`	`VARBINARY`	`byte[]`
`TINYBLOB`	`TINYBLOB`	`byte[]`
`TINYTEXT`	`VARCHAR`	`java.lang.String`
`BLOB`	`BLOB`	`byte[]`
`TEXT`	`VARCHAR`	`java.lang.String`
`MEDIUMBLOB`	`MEDIUMBLOB`	`byte[]`
`MEDIUMTEXT`	`VARCHAR`	`java.lang.String`
`LONGBLOB`	`LONGBLOB`	`byte[]`
`LONGTEXT`	`VARCHAR`	`java.lang.String`
`ENUM('value1','value2',...)`	`CHAR`	`java.lang.String`
`SET('value1','value2',...)`	`CHAR`	`java.lang.String`

5.4. Using Character Sets and Unicode

All strings sent from the JDBC driver to the server are converted automatically from native Java Unicode form to the client character encoding, including all queries sent using `Statement.execute()`, `Statement.executeUpdate()`, `Statement.executeQuery()` as well as all `PreparedStatement` and `CallableStatement` parameters with the exclusion of parameters set using `setBytes()`, `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()` and `setBlob()`.

Number of Encodings Per Connection

In MySQL Server 4.1 and higher, Connector/J supports a single character encoding between client and server, and any number of character encodings for data returned by the server to the client in `ResultSets`.

Prior to MySQL Server 4.1, Connector/J supported a single character encoding per connection, which could either be automatically detected from the server configuration, or could be configured by the user through the *useUnicode* and *characterEncoding* properties.

Setting the Character Encoding

The character encoding between client and server is automatically detected upon connection. You specify the encoding on the server using the `character_set_server` for server versions 4.1.0 and newer, and `character_set` system variable for server versions older than 4.1.0. The driver automatically uses the encoding specified by the server. For more information, see Server Character Set and Collation.

For example, to use 4-byte UTF-8 character sets with Connector/J, configure the MySQL server with `character_set_server=utf8mb4`, and leave `characterEncoding` out of the Connector/J connection string. Connector/J will then autodetect the UTF-8 setting.

To override the automatically detected encoding on the client side, use the *characterEncoding* property in the URL used to connect to the server.

To allow multiple character sets to be sent from the client, use the UTF-8 encoding, either by configuring `utf8` as the default server character set, or by configuring the JDBC driver to use UTF-8 through the `characterEncoding` property.

When specifying character encodings on the client side, use Java-style names. The following table lists Java-style names for MySQL character sets:

MySQL to Java Encoding Name Translations.

MySQL Character Set Name	Java-Style Character Encoding Name
ascii	US-ASCII
big5	Big5
gbk	GBK
sjis	SJIS (or Cp932 or MS932 for MySQL Server < 4.1.11)
cp932	Cp932 or MS932 (MySQL Server > 4.1.11)
gb2312	EUC_CN
ujis	EUC_JP
euckr	EUC_KR
latin1	Cp1252
latin2	ISO8859_2
greek	ISO8859_7
hebrew	ISO8859_8
cp866	Cp866
tis620	TIS620
cp1250	Cp1250
cp1251	Cp1251
cp1257	Cp1257

MySQL Character Set Name	Java-Style Character Encoding Name
macroman	MacRoman
macce	MacCentralEurope
utf8	UTF-8
ucs2	UnicodeBig

> **Warning**
>
> Do not issue the query `set names` with Connector/J, as the driver will not detect that the character set has changed, and will continue to use the character set detected during the initial connection setup.

5.5. Connecting Securely Using SSL

SSL in MySQL Connector/J encrypts all data (other than the initial handshake) between the JDBC driver and the server. The performance penalty for enabling SSL is an increase in query processing time between 35% and 50%, depending on the size of the query, and the amount of data it returns.

For SSL Support to work, you must have the following:

- A JDK that includes JSSE (Java Secure Sockets Extension), like JDK-1.4.1 or newer. SSL does not currently work with a JDK that you can add JSSE to, like JDK-1.2.x or JDK-1.3.x due to the following JSSE bug: http://developer.java.sun.com/developer/bugParade/bugs/4273544.html

- A MySQL server that supports SSL and has been compiled and configured to do so, which is MySQL 4.0.4 or later, see Using SSL for Secure Connections for more information.

- A client certificate (covered later in this section)

The system works through two Java truststore files, one file contains the certificate information for the server (`truststore` in the examples below). The other file contains the certificate for the client (`keystore` in the examples below). All Java truststore files are password protected by supplying a suitable password to the `keytool` when you create the files. You need the file names and associated passwords to create an SSL connection.

You will first need to import the MySQL server CA Certificate into a Java truststore. A sample MySQL server CA Certificate is located in the `SSL` subdirectory of the MySQL source distribution. This is what SSL will use to determine if you are communicating with a secure MySQL server. Alternatively, use the CA Certificate that you have generated or been provided with by your SSL provider.

To use Java's `keytool` to create a truststore in the current directory , and import the server's CA certificate (`cacert.pem`), you can do the following (assuming that `keytool` is in your path. The `keytool` is typically located in the `bin` subdirectory of your JDK or JRE):

```
shell> keytool -import -alias mysqlServerCACert \
                                    -file cacert.pem -keystore truststore
```

Enter the password when prompted for the keystore file. Interaction with `keytool` looks like this:

```
Enter keystore password:  *********
Owner: EMAILADDRESS=walrus@example.com, CN=Walrus,
       O=MySQL AB, L=Orenburg, ST=Some-State, C=RU
Issuer: EMAILADDRESS=walrus@example.com, CN=Walrus,
       O=MySQL AB, L=Orenburg, ST=Some-State, C=RU
Serial number: 0
Valid from:
   Fri Aug 02 16:55:53 CDT 2002 until: Sat Aug 02 16:55:53 CDT 2003
Certificate fingerprints:
   MD5:  61:91:A0:F2:03:07:61:7A:81:38:66:DA:19:C4:8D:AB
    SHA1: 25:77:41:05:D5:AD:99:8C:14:8C:CA:68:9C:2F:B8:89:C3:34:4D:6C
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

You then have two options: either import the client certificate that matches the CA certificate you just imported, or create a new client certificate.

Importing an existing certificate requires the certificate to be in DER format. You can use `openssl` to convert an existing certificate into the new format. For example:

```
shell> openssl x509 -outform DER -in client-cert.pem -out client.cert
```

Now import the converted certificate into your keystore using `keytool`:

```
shell> keytool -import -file client.cert -keystore keystore -alias mysqlClientCertificate
```

To generate your own client certificate, use `keytool` to create a suitable certificate and add it to the `keystore` file:

```
shell> keytool -genkey -keyalg rsa \
     -alias mysqlClientCertificate -keystore keystore
```

Keytool will prompt you for the following information, and create a keystore named `keystore` in the current directory.

Respond with information that is appropriate for your situation:

```
Enter keystore password:  *********
What is your first and last name?
  [Unknown]:  Matthews
What is the name of your organizational unit?
  [Unknown]:  Software Development
What is the name of your organization?
  [Unknown]:  MySQL AB
What is the name of your City or Locality?
  [Unknown]:  Flossmoor
What is the name of your State or Province?
  [Unknown]:  IL
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Matthews, OU=Software Development, O=MySQL AB,
 L=Flossmoor, ST=IL, C=US> correct?
  [no]:  y
Enter key password for <mysqlClientCertificate>
        (RETURN if same as keystore password):
```

Finally, to get JSSE to use the keystore and truststore that you have generated, you need to set the following system properties when you start your JVM, replacing *path_to_keystore_file* with the full path to the keystore file you created, *path_to_truststore_file* with the path to the truststore file you created, and using the appropriate password values for each property. You can do this either on the command line:

```
-Djavax.net.ssl.keyStore=path_to_keystore_file
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=path_to_truststore_file
-Djavax.net.ssl.trustStorePassword=password
```

Or you can set the values directly within the application:

```
System.setProperty("javax.net.ssl.keyStore","path_to_keystore_file");
System.setProperty("javax.net.ssl.keyStorePassword","password");
System.setProperty("javax.net.ssl.trustStore","path_to_truststore_file");
System.setProperty("javax.net.ssl.trustStorePassword","password");
```

You will also need to set `useSSL` to `true` in your connection parameters for MySQL Connector/J, either by adding `useSSL=true` to your URL, or by setting the property `useSSL` to `true` in the `java.util.Properties` instance you pass to `DriverManager.getConnection()`.

You can test that SSL is working by turning on JSSE debugging (as detailed below), and look for the following key events:

```
...
*** ClientHello, v3.1
RandomCookie:  GMT: 1018531834 bytes = { 199, 148, 180, 215, 74, 12, »
  54, 244, 0, 168, 55, 103, 215, 64, 16, 138, 225, 190, 132, 153, 2, »
  217, 219, 239, 202, 19, 121, 78 }
Session ID:  {}
Cipher Suites:  { 0, 5, 0, 4, 0, 9, 0, 10, 0, 18, 0, 19, 0, 3, 0, 17 }
Compression Methods:  { 0 }
***
[write] MD5 and SHA1 hashes:  len = 59
0000: 01 00 00 37 03 01 3D B6 90 FA C7 94 B4 D7 4A 0C  ...7..=.......J.
0010: 36 F4 00 A8 37 67 D7 40 10 8A E1 BE 84 99 02 D9  6...7g.@........
0020: DB EF CA 13 79 4E 00 00 10 00 05 00 04 00 09 00  ....yN..........
```

```
0030: 0A 00 12 00 13 00 03 00 11 01 00                    ..........
main, WRITE:  SSL v3.1 Handshake, length = 59
main, READ:  SSL v3.1 Handshake, length = 74
*** ServerHello, v3.1
RandomCookie:  GMT: 1018577560 bytes = { 116, 50, 4, 103, 25, 100, 58, »
   202, 79, 185, 178, 100, 215, 66, 254, 21, 83, 187, 190, 42, 170, 3, »
   132, 110, 82, 148, 160, 92 }
Session ID:  {163, 227, 84, 53, 81, 127, 252, 254, 178, 179, 68, 63, »
   182, 158, 30, 11, 150, 79, 170, 76, 255, 92, 15, 226, 24, 17, 177, »
   219, 158, 177, 187, 143}
Cipher Suite:  { 0, 5 }
Compression Method: 0
***
%% Created:  [Session-1, SSL_RSA_WITH_RC4_128_SHA]
** SSL_RSA_WITH_RC4_128_SHA
[read] MD5 and SHA1 hashes:  len = 74
0000: 02 00 00 46 03 01 3D B6 43 98 74 32 04 67 19 64  ...F..=.C.t2.g.d
0010: 3A CA 4F B9 B2 64 D7 42 FE 15 53 BB BE 2A AA 03  :.O..d.B..S..*..
0020: 84 6E 52 94 A0 5C 20 A3 E3 54 35 51 7F FC FE B2  .nR..\ ..T5Q....
0030: B3 44 3F B6 9E 1E 0B 96 4F AA 4C FF 5C 0F E2 18  .D?.....O.L.\...
0040: 11 B1 DB 9E B1 BB 8F 00 05 00                    ..........
main, READ:  SSL v3.1 Handshake, length = 1712
...
```

JSSE provides debugging (to `stdout`) when you set the following system property: `-Djavax.net.debug=all` This will tell you what keystores and truststores are being used, as well as what is going on during the SSL handshake and certificate exchange. It will be helpful when trying to determine what is not working when trying to get an SSL connection to happen.

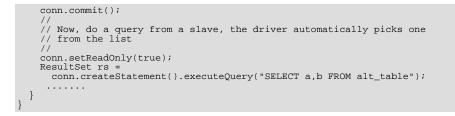# 5.6. Using Master/Slave Replication with ReplicationConnection

Connector/J 3.1.7 and higher includes a variant of the driver that will automatically send queries to a read/write master, or a failover or round-robin loadbalanced set of slaves based on the state of `Connection.getReadOnly()`.

An application signals that it wants a transaction to be read-only by calling `Connection.setReadOnly(true)`, this replication-aware connection will use one of the slave connections, which are load-balanced per-vm using a round-robin scheme (a given connection is sticky to a slave unless that slave is removed from service). If you have a write transaction, or if you have a read that is time-sensitive (remember, replication in MySQL is asynchronous), set the connection to be not read-only, by calling `Connection.setReadOnly(false)` and the driver will ensure that further calls are sent to the master MySQL server. The driver takes care of propagating the current state of autocommit, isolation level, and catalog between all of the connections that it uses to accomplish this load balancing functionality.

To enable this functionality, use the `com.mysql.jdbc.ReplicationDriver` class when configuring your application server's connection pool or when creating an instance of a JDBC driver for your standalone application. Because it accepts the same URL format as the standard MySQL JDBC driver, `ReplicationDriver` does not currently work with `java.sql.DriverManager`-based connection creation unless it is the only MySQL JDBC driver registered with the `DriverManager` .

Here is a short example of how `ReplicationDriver` might be used in a standalone application:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.util.Properties;
import com.mysql.jdbc.ReplicationDriver;
public class ReplicationDriverDemo {
  public static void main(String[] args) throws Exception {
    ReplicationDriver driver = new ReplicationDriver();
    Properties props = new Properties();
    // We want this for failover on the slaves
    props.put("autoReconnect", "true");
    // We want to load balance between the slaves
    props.put("roundRobinLoadBalance", "true");
    props.put("user", "foo");
    props.put("password", "bar");
    //
    // Looks like a normal MySQL JDBC url, with a
    // comma-separated list of hosts, the first
    // being the 'master', the rest being any number
    // of slaves that the driver will load balance against
    //
    Connection conn =
        driver.connect("jdbc:mysql:replication://master,slave1,slave2,slave3/test",
            props);
    //
    // Perform read/write work on the master
    // by setting the read-only flag to "false"
    //
    conn.setReadOnly(false);
    conn.setAutoCommit(false);
    conn.createStatement().executeUpdate("UPDATE some_table ....");
```

```
    conn.commit();
    //
    // Now, do a query from a slave, the driver automatically picks one
    // from the list
    //
    conn.setReadOnly(true);
    ResultSet rs =
      conn.createStatement().executeQuery("SELECT a,b FROM alt_table");
     .......
  }
}
```

Consider investigating the Load Balancing JDBC Pool (`lbpool`) tool, which provides a wrapper around the standard JDBC driver and enables you to use DB connection pools that includes checks for system failures and uneven load distribution. For more information, see Load Balancing JDBC Pool (lbpool).

5.7. Mapping MySQL Error Numbers to JDBC SQLState Codes

The table below provides a mapping of the MySQL error numbers to JDBC `SQLState` values.

Table 5.1. Mapping of MySQL Error Numbers to SQLStates

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQL-State	SQL Standard SQL-State
1022	ER_DUP_KEY	S1000	23000
1037	ER_OUTOFMEMORY	S1001	HY001
1038	ER_OUT_OF_SORTMEMORY	S1001	HY001
1040	ER_CON_COUNT_ERROR	08004	08004
1042	ER_BAD_HOST_ERROR	08004	08S01
1043	ER_HANDSHAKE_ERROR	08004	08S01
1044	ER_DBACCESS_DENIED_ERROR	S1000	42000
1045	ER_ACCESS_DENIED_ERROR	28000	28000
1047	ER_UNKNOWN_COM_ERROR	08S01	HY000
1050	ER_TABLE_EXISTS_ERROR	S1000	42S01
1051	ER_BAD_TABLE_ERROR	42S02	42S02
1052	ER_NON_UNIQ_ERROR	S1000	23000
1053	ER_SERVER_SHUTDOWN	S1000	08S01
1054	ER_BAD_FIELD_ERROR	S0022	42S22
1055	ER_WRONG_FIELD_WITH_GROUP	S1009	42000
1056	ER_WRONG_GROUP_FIELD	S1009	42000
1057	ER_WRONG_SUM_SELECT	S1009	42000
1058	ER_WRONG_VALUE_COUNT	21S01	21S01
1059	ER_TOO_LONG_IDENT	S1009	42000
1060	ER_DUP_FIELDNAME	S1009	42S21
1061	ER_DUP_KEYNAME	S1009	42000
1062	ER_DUP_ENTRY	S1009	23000
1063	ER_WRONG_FIELD_SPEC	S1009	42000
1064	ER_PARSE_ERROR	42000	42000
1065	ER_EMPTY_QUERY	42000	42000
1066	ER_NONUNIQ_TABLE	S1009	42000
1067	ER_INVALID_DEFAULT	S1009	42000
1068	ER_MULTIPLE_PRI_KEY	S1009	42000
1069	ER_TOO_MANY_KEYS	S1009	42000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQL-State	SQL Standard SQL-State
1070	ER_TOO_MANY_KEY_PARTS	S1009	42000
1071	ER_TOO_LONG_KEY	S1009	42000
1072	ER_KEY_COLUMN_DOES_NOT_EXITS	S1009	42000
1073	ER_BLOB_USED_AS_KEY	S1009	42000
1074	ER_TOO_BIG_FIELDLENGTH	S1009	42000
1075	ER_WRONG_AUTO_KEY	S1009	42000
1080	ER_FORCING_CLOSE	S1000	08S01
1081	ER_IPSOCK_ERROR	08S01	08S01
1082	ER_NO_SUCH_INDEX	S1009	42S12
1083	ER_WRONG_FIELD_TERMINATORS	S1009	42000
1084	ER_BLOBS_AND_NO_TERMINATED	S1009	42000
1090	ER_CANT_REMOVE_ALL_FIELDS	S1000	42000
1091	ER_CANT_DROP_FIELD_OR_KEY	S1000	42000
1101	ER_BLOB_CANT_HAVE_DEFAULT	S1000	42000
1102	ER_WRONG_DB_NAME	S1000	42000
1103	ER_WRONG_TABLE_NAME	S1000	42000
1104	ER_TOO_BIG_SELECT	S1000	42000
1106	ER_UNKNOWN_PROCEDURE	S1000	42000
1107	ER_WRONG_PARAMCOUNT_TO_PROCEDURE	S1000	42000
1109	ER_UNKNOWN_TABLE	S1000	42S02
1110	ER_FIELD_SPECIFIED_TWICE	S1000	42000
1112	ER_UNSUPPORTED_EXTENSION	S1000	42000
1113	ER_TABLE_MUST_HAVE_COLUMNS	S1000	42000
1115	ER_UNKNOWN_CHARACTER_SET	S1000	42000
1118	ER_TOO_BIG_ROWSIZE	S1000	42000
1120	ER_WRONG_OUTER_JOIN	S1000	42000
1121	ER_NULL_COLUMN_IN_INDEX	S1000	42000
1129	ER_HOST_IS_BLOCKED	08004	HY000
1130	ER_HOST_NOT_PRIVILEGED	08004	HY000
1131	ER_PASSWORD_ANONYMOUS_USER	S1000	42000
1132	ER_PASSWORD_NOT_ALLOWED	S1000	42000
1133	ER_PASSWORD_NO_MATCH	S1000	42000
1136	ER_WRONG_VALUE_COUNT_ON_ROW	S1000	21S01
1138	ER_INVALID_USE_OF_NULL	S1000	42000
1139	ER_REGEXP_ERROR	S1000	42000
1140	ER_MIX_OF_GROUP_FUNC_AND_FIELDS	S1000	42000
1141	ER_NONEXISTING_GRANT	S1000	42000
1142	ER_TABLEACCESS_DENIED_ERROR	S1000	42000
1143	ER_COLUMNACCESS_DENIED_ERROR	S1000	42000
1144	ER_ILLEGAL_GRANT_FOR_TABLE	S1000	42000
1145	ER_GRANT_WRONG_HOST_OR_USER	S1000	42000
1146	ER_NO_SUCH_TABLE	S1000	42S02
1147	ER_NONEXISTING_TABLE_GRANT	S1000	42000
1148	ER_NOT_ALLOWED_COMMAND	S1000	42000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQL-State	SQL Standard SQL-State
1149	ER_SYNTAX_ERROR	S1000	42000
1152	ER_ABORTING_CONNECTION	S1000	08S01
1153	ER_NET_PACKET_TOO_LARGE	S1000	08S01
1154	ER_NET_READ_ERROR_FROM_PIPE	S1000	08S01
1155	ER_NET_FCNTL_ERROR	S1000	08S01
1156	ER_NET_PACKETS_OUT_OF_ORDER	S1000	08S01
1157	ER_NET_UNCOMPRESS_ERROR	S1000	08S01
1158	ER_NET_READ_ERROR	S1000	08S01
1159	ER_NET_READ_INTERRUPTED	S1000	08S01
1160	ER_NET_ERROR_ON_WRITE	S1000	08S01
1161	ER_NET_WRITE_INTERRUPTED	S1000	08S01
1162	ER_TOO_LONG_STRING	S1000	42000
1163	ER_TABLE_CANT_HANDLE_BLOB	S1000	42000
1164	ER_TABLE_CANT_HANDLE_AUTO_INCREMENT	S1000	42000
1166	ER_WRONG_COLUMN_NAME	S1000	42000
1167	ER_WRONG_KEY_COLUMN	S1000	42000
1169	ER_DUP_UNIQUE	S1000	23000
1170	ER_BLOB_KEY_WITHOUT_LENGTH	S1000	42000
1171	ER_PRIMARY_CANT_HAVE_NULL	S1000	42000
1172	ER_TOO_MANY_ROWS	S1000	42000
1173	ER_REQUIRES_PRIMARY_KEY	S1000	42000
1177	ER_CHECK_NO_SUCH_TABLE	S1000	42000
1178	ER_CHECK_NOT_IMPLEMENTED	S1000	42000
1179	ER_CANT_DO_THIS_DURING_AN_TRANSACTION	S1000	25000
1184	ER_NEW_ABORTING_CONNECTION	S1000	08S01
1189	ER_MASTER_NET_READ	S1000	08S01
1190	ER_MASTER_NET_WRITE	S1000	08S01
1203	ER_TOO_MANY_USER_CONNECTIONS	S1000	42000
1205	ER_LOCK_WAIT_TIMEOUT	41000	41000
1207	ER_READ_ONLY_TRANSACTION	S1000	25000
1211	ER_NO_PERMISSION_TO_CREATE_USER	S1000	42000
1213	ER_LOCK_DEADLOCK	41000	40001
1216	ER_NO_REFERENCED_ROW	S1000	23000
1217	ER_ROW_IS_REFERENCED	S1000	23000
1218	ER_CONNECT_TO_MASTER	S1000	08S01
1222	ER_WRONG_NUMBER_OF_COLUMNS_IN_SELECT	S1000	21000
1226	ER_USER_LIMIT_REACHED	S1000	42000
1230	ER_NO_DEFAULT	S1000	42000
1231	ER_WRONG_VALUE_FOR_VAR	S1000	42000
1232	ER_WRONG_TYPE_FOR_VAR	S1000	42000
1234	ER_CANT_USE_OPTION_HERE	S1000	42000
1235	ER_NOT_SUPPORTED_YET	S1000	42000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQL-State	SQL Standard SQL-State
1239	ER_WRONG_FK_DEF	S1000	42000
1241	ER_OPERAND_COLUMNS	S1000	21000
1242	ER_SUBQUERY_NO_1_ROW	S1000	21000
1247	ER_ILLEGAL_REFERENCE	S1000	42S22
1248	ER_DERIVED_MUST_HAVE_ALIAS	S1000	42000
1249	ER_SELECT_REDUCED	S1000	01000
1250	ER_TABLENAME_NOT_ALLOWED_HERE	S1000	42000
1251	ER_NOT_SUPPORTED_AUTH_MODE	S1000	08004
1252	ER_SPATIAL_CANT_HAVE_NULL	S1000	42000
1253	ER_COLLATION_CHARSET_MISMATCH	S1000	42000
1261	ER_WARN_TOO_FEW_RECORDS	S1000	01000
1262	ER_WARN_TOO_MANY_RECORDS	S1000	01000
1263	ER_WARN_NULL_TO_NOTNULL	S1000	01000
1264	ER_WARN_DATA_OUT_OF_RANGE	S1000	01000
1265	ER_WARN_DATA_TRUNCATED	S1000	01000
1280	ER_WRONG_NAME_FOR_INDEX	S1000	42000
1281	ER_WRONG_NAME_FOR_CATALOG	S1000	42000
1286	ER_UNKNOWN_STORAGE_ENGINE	S1000	42000

Chapter 6. JDBC Concepts

This section provides some general JDBC background.

6.1. Connecting to MySQL Using the `DriverManager` Interface

When you are using JDBC outside of an application server, the `DriverManager` class manages the establishment of Connections.

Specify to the `DriverManager` which JDBC drivers to try to make Connections with. The easiest way to do this is to use `Class.forName()` on the class that implements the `java.sql.Driver` interface. With MySQL Connector/J, the name of this class is `com.mysql.jdbc.Driver`. With this method, you could use an external configuration file to supply the driver class name and driver parameters to use when connecting to a database.

The following section of Java code shows how you might register MySQL Connector/J from the `main()` method of your application. If testing this code, first read the installation section at Chapter 3, *Connector/J Installation*, to make sure you have connector installed correctly and the `CLASSPATH` set up. Also, ensure that MySQL is configured to accept external TCP/IP connections.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
// Notice, do not import com.mysql.jdbc.*
// or you will have problems!
public class LoadDriver {
    public static void main(String[] args) {
        try {
            // The newInstance() call is a work around for some
            // broken Java implementations
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception ex) {
            // handle the error
        }
    }
}
```

After the driver has been registered with the `DriverManager`, you can obtain a `Connection` instance that is connected to a particular database by calling `DriverManager.getConnection()`:

Example 6.1. Connector/J: Obtaining a connection from the `DriverManager`

If you have not already done so, please review the section Section 6.1, "Connecting to MySQL Using the `DriverManager` Interface" before working with these examples.

This example shows how you can obtain a `Connection` instance from the `DriverManager`. There are a few different signatures for the `getConnection()` method. Consult the API documentation that comes with your JDK for more specific information on how to use them.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
Connection conn = null;
...
try {
    conn =
        DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                                    "user=monty&password=greatsqldb");
    // Do something with the Connection
    ...
} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
```

Once a `Connection` is established, it can be used to create `Statement` and `PreparedStatement` objects, as well as retrieve metadata about the database. This is explained in the following sections.

6.2. Using `Statement` Objects to Execute SQL

`Statement` objects allow you to execute basic SQL queries and retrieve the results through the `ResultSet` class, which is described later.

To create a `Statement` instance, you call the `createStatement()` method on the `Connection` object you have retrieved using one of the `DriverManager.getConnection()` or `DataSource.getConnection()` methods described earlier.

Once you have a `Statement` instance, you can execute a `SELECT` query by calling the `executeQuery(String)` method with the SQL you want to use.

To update data in the database, use the `executeUpdate(String SQL)` method. This method returns the number of rows matched by the update statement, not the number of rows that were modified.

If you do not know ahead of time whether the SQL statement will be a `SELECT` or an `UPDATE`/`INSERT`, then you can use the `execute(String SQL)` method. This method will return true if the SQL query was a `SELECT`, or false if it was an `UPDATE`, `INSERT`, or `DELETE` statement. If the statement was a `SELECT` query, you can retrieve the results by calling the `getResultSet()` method. If the statement was an `UPDATE`, `INSERT`, or `DELETE` statement, you can retrieve the affected rows count by calling `getUpdateCount()` on the `Statement` instance.

Example 6.2. Connector/J: Using java.sql.Statement to execute a `SELECT` query

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
// assume that conn is an already created JDBC connection (see previous examples)
Statement stmt = null;
ResultSet rs = null;
try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT foo FROM bar");
    // or alternatively, if you don't know ahead of time that
    // the query will be a SELECT...
    if (stmt.execute("SELECT foo FROM bar")) {
        rs = stmt.getResultSet();
    }
    // Now do something with the ResultSet ....
}
catch (SQLException ex){
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
    // it is a good idea to release
    // resources in a finally{} block
    // in reverse-order of their creation
    // if they are no-longer needed
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) { } // ignore
        rs = null;
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) { } // ignore
        stmt = null;
    }
}
```

6.3. Using `CallableStatements` to Execute Stored Procedures

Starting with MySQL server version 5.0 when used with Connector/J 3.1.1 or newer, the `java.sql.CallableStatement` interface is fully implemented with the exception of the `getParameterMetaData()` method.

For more information on MySQL stored procedures, please refer to http://dev.mysql.com/doc/mysql/en/stored-routines.html.

Connector/J exposes stored procedure functionality through JDBC's `CallableStatement` interface.

▋ **Note**

Current versions of MySQL server do not return enough information for the JDBC driver to provide result set metadata for callable statements. This means that when using `CallableStatement`, `ResultSetMetaData` may return `NULL`.

The following example shows a stored procedure that returns the value of `inOutParam` incremented by 1, and the string passed in using `inputParam` as a `ResultSet`:

Example 6.3. Connector/J: Calling Stored Procedures

```
CREATE PROCEDURE demoSp(IN inputParam VARCHAR(255), \
                        INOUT inOutParam INT)
BEGIN
    DECLARE z INT;
    SET z = inOutParam + 1;
    SET inOutParam = z;
    SELECT inputParam;
    SELECT CONCAT('zyxw', inputParam);
END
```

To use the `demoSp` procedure with Connector/J, follow these steps:

1. Prepare the callable statement by using `Connection.prepareCall()`.

   Notice that you have to use JDBC escape syntax, and that the parentheses surrounding the parameter placeholders are not optional:

   ## Example 6.4. Connector/J: Using `Connection.prepareCall()`

   ```
   import java.sql.CallableStatement;
   ...
       //
       // Prepare a call to the stored procedure 'demoSp'
       // with two parameters
       //
       // Notice the use of JDBC-escape syntax ({call ...})
       //
       CallableStatement cStmt = conn.prepareCall("{call demoSp(?, ?)}");
       cStmt.setString(1, "abcdefg");
   ```

   Note

   `Connection.prepareCall()` is an expensive method, due to the metadata retrieval that the driver performs to support output parameters. For performance reasons, minimize unnecessary calls to `Connection.prepareCall()` by reusing `CallableStatement` instances in your code.

2. Register the output parameters (if any exist)

   To retrieve the values of output parameters (parameters specified as `OUT` or `INOUT` when you created the stored procedure), JDBC requires that they be specified before statement execution using the various `registerOutputParameter()` methods in the `CallableStatement` interface:

   ## Example 6.5. Connector/J: Registering output parameters

   ```
   import java.sql.Types;
   ...
   //
   // Connector/J supports both named and indexed
   // output parameters. You can register output
   // parameters using either method, as well
   // as retrieve output parameters using either
   // method, regardless of what method was
   // used to register them.
   //
   // The following examples show how to use
   // the various methods of registering
   // output parameters (you should of course
   // use only one registration per parameter).
   //
   //
   ```

```
// Registers the second parameter as output, and
// uses the type 'INTEGER' for values returned from
// getObject()
//
cStmt.registerOutParameter(2, Types.INTEGER);
//
// Registers the named parameter 'inOutParam', and
// uses the type 'INTEGER' for values returned from
// getObject()
//
cStmt.registerOutParameter("inOutParam", Types.INTEGER);
...
```

3. Set the input parameters (if any exist)

Input and in/out parameters are set as for `PreparedStatement` objects. However, `CallableStatement` also supports setting parameters by name:

Example 6.6. Connector/J: Setting `CallableStatement` input parameters

```
...
    //
    // Set a parameter by index
    //
    cStmt.setString(1, "abcdefg");
    //
    // Alternatively, set a parameter using
    // the parameter name
    //
    cStmt.setString("inputParameter", "abcdefg");
    //
    // Set the 'in/out' parameter using an index
    //
    cStmt.setInt(2, 1);
    //
    // Alternatively, set the 'in/out' parameter
    // by name
    //
    cStmt.setInt("inOutParam", 1);
...
```

4. Execute the `CallableStatement`, and retrieve any result sets or output parameters.

Although `CallableStatement` supports calling any of the `Statement` execute methods (`executeUpdate()`, `executeQuery()` or `execute()`), the most flexible method to call is `execute()`, as you do not need to know ahead of time if the stored procedure returns result sets:

Example 6.7. Connector/J: Retrieving results and output parameter values

```
...
    boolean hadResults = cStmt.execute();
    //
    // Process all returned result sets
    //
    while (hadResults) {
        ResultSet rs = cStmt.getResultSet();
        // process result set
        ...
        hadResults = cStmt.getMoreResults();
    }
    //
    // Retrieve output parameters
    //
    // Connector/J supports both index-based and
    // name-based retrieval
    //
    int outputValue = cStmt.getInt(2); // index-based
    outputValue = cStmt.getInt("inOutParam"); // name-based
...
```

6.4. Retrieving `AUTO_INCREMENT` Column Values

Before version 3.0 of the JDBC API, there was no standard way of retrieving key values from databases that supported auto increment or identity columns. With older JDBC drivers for MySQL, you could always use a MySQL-specific method on the `Statement` inter-face, or issue the query `SELECT LAST_INSERT_ID()` after issuing an `INSERT` to a table that had an `AUTO_INCREMENT` key. Us-ing the MySQL-specific method call isn't portable, and issuing a `SELECT` to get the `AUTO_INCREMENT` key's value requires another round-trip to the database, which isn't as efficient as possible. The following code snippets demonstrate the three different ways to re-trieve `AUTO_INCREMENT` values. First, we demonstrate the use of the new JDBC 3.0 method `getGeneratedKeys()` which is now the preferred method to use if you need to retrieve `AUTO_INCREMENT` keys and have access to JDBC 3.0. The second example shows how you can retrieve the same value using a standard `SELECT LAST_INSERT_ID()` query. The final example shows how updatable result sets can retrieve the `AUTO_INCREMENT` value when using the `insertRow()` method.

Example 6.8. Connector/J: Retrieving `AUTO_INCREMENT` column values using `Statement.getGeneratedKeys()`

```
  Statement stmt = null;
  ResultSet rs = null;
  try {
   //
   // Create a Statement instance that we can use for
   // 'normal' result sets assuming you have a
   // Connection 'conn' to a MySQL database already
   // available
   stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                               java.sql.ResultSet.CONCUR_UPDATABLE);
   //
   // Issue the DDL queries for the table for this example
   //
   stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
   stmt.executeUpdate(
           "CREATE TABLE autoIncTutorial ("
           + "priKey INT NOT NULL AUTO_INCREMENT, "
           + "dataField VARCHAR(64), PRIMARY KEY (priKey))");
   //
   // Insert one row that will generate an AUTO INCREMENT
   // key in the 'priKey' field
   //
   stmt.executeUpdate(
           "INSERT INTO autoIncTutorial (dataField) "
           + "values ('Can I Get the Auto Increment Field?')",
           Statement.RETURN_GENERATED_KEYS);
   //
   // Example of using Statement.getGeneratedKeys()
   // to retrieve the value of an auto-increment
   // value
   //
   int autoIncKeyFromApi = -1;
   rs = stmt.getGeneratedKeys();
   if (rs.next()) {
       autoIncKeyFromApi = rs.getInt(1);
   } else {
       // throw an exception from here
   }
   rs.close();
   rs = null;
   System.out.println("Key returned from getGeneratedKeys():"
       + autoIncKeyFromApi);
} finally {
   if (rs != null) {
       try {
           rs.close();
       } catch (SQLException ex) {
           // ignore
       }
   }
   if (stmt != null) {
       try {
           stmt.close();
       } catch (SQLException ex) {
           // ignore
       }
   }
}
```

Example 6.9. Connector/J: Retrieving `AUTO_INCREMENT` column values using `SELECT LAST_INSERT_ID()`

```
    Statement stmt = null;
    ResultSet rs = null;
    try {
     //
     // Create a Statement instance that we can use for
     // 'normal' result sets.
     stmt = conn.createStatement();
     //
     // Issue the DDL queries for the table for this example
     //
     stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
     stmt.executeUpdate(
            "CREATE TABLE autoIncTutorial ("
            + "priKey INT NOT NULL AUTO_INCREMENT, "
            + "dataField VARCHAR(64), PRIMARY KEY (priKey))");
     //
     // Insert one row that will generate an AUTO INCREMENT
     // key in the 'priKey' field
     //
     stmt.executeUpdate(
            "INSERT INTO autoIncTutorial (dataField) "
            + "values ('Can I Get the Auto Increment Field?')");
     //
     // Use the MySQL LAST_INSERT_ID()
     // function to do the same thing as getGeneratedKeys()
     //
     int autoIncKeyFromFunc = -1;
     rs = stmt.executeQuery("SELECT LAST_INSERT_ID()");
     if (rs.next()) {
         autoIncKeyFromFunc = rs.getInt(1);
     } else {
         // throw an exception from here
     }
     rs.close();
     System.out.println("Key returned from " +
                        "'SELECT LAST_INSERT_ID()': " +
                        autoIncKeyFromFunc);
    } finally {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException ex) {
                // ignore
            }
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException ex) {
                // ignore
            }
        }
    }
}
```

Example 6.10. Connector/J: Retrieving `AUTO_INCREMENT` column values in `Updatable ResultSets`

```
    Statement stmt = null;
    ResultSet rs = null;
    try {
     //
     // Create a Statement instance that we can use for
     // 'normal' result sets as well as an 'updatable'
     // one, assuming you have a Connection 'conn' to
     // a MySQL database already available
     //
     stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                java.sql.ResultSet.CONCUR_UPDATABLE);
     //
     // Issue the DDL queries for the table for this example
     //
     stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
     stmt.executeUpdate(
            "CREATE TABLE autoIncTutorial ("
            + "priKey INT NOT NULL AUTO_INCREMENT, "
            + "dataField VARCHAR(64), PRIMARY KEY (priKey))");
     //
     // Example of retrieving an AUTO INCREMENT key
     // from an updatable result set
     //
     rs = stmt.executeQuery("SELECT priKey, dataField "
        + "FROM autoIncTutorial");
     rs.moveToInsertRow();
     rs.updateString("dataField", "AUTO INCREMENT here?");
     rs.insertRow();
```

```
    //
    // the driver adds rows at the end
    //
    rs.last();
    //
    // We should now be on the row we just inserted
    //
    int autoIncKeyFromRS = rs.getInt("priKey");
    rs.close();
    rs = null;
    System.out.println("Key returned for inserted row: "
        + autoIncKeyFromRS);
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
}
```

Running the preceding example code should produce the following output:

```
Key returned from getGeneratedKeys(): 1
Key returned from SELECT LAST_INSERT_ID(): 1
Key returned for inserted row: 2
```

At times, it can be tricky to use the SELECT LAST_INSERT_ID() query, as that function's value is scoped to a connection. So, if some other query happens on the same connection, the value is overwritten. On the other hand, the getGeneratedKeys() method is scoped by the Statement instance, so it can be used even if other queries happen on the same connection, but not on the same Statement instance.

Chapter 7. Developing J2EE Applications with Connector/J

This section provides general background on J2EE concepts that pertain to use of Connector/J.

7.1. Understanding J2EE Connection Pooling

Connection pooling is a technique of creating and managing a pool of connections that are ready for use by any thread that needs them. Connection pooling can greatly increase the performance of your Java application, while reducing overall resource usage.

How Connection Pooling Works

Most applications only need a thread to have access to a JDBC connection when they are actively processing a transaction, which often takes only milliseconds to complete. When not processing a transaction, the connection sits idle. Connection pooling enables the idle connection to be used by some other thread to do useful work.

In practice, when a thread needs to do work against a MySQL or other database with JDBC, it requests a connection from the pool. When the thread is finished using the connection, it returns it to the pool, so that it can be used by any other threads.

When the connection is loaned out from the pool, it is used exclusively by the thread that requested it. From a programming point of view, it is the same as if your thread called `DriverManager.getConnection()` every time it needed a JDBC connection. With connection pooling, your thread may end up using either a new connection or an already-existing connection.

Benefits of Connection Pooling

The main benefits to connection pooling are:

- Reduced connection creation time.

  Although this is not usually an issue with the quick connection setup that MySQL offers compared to other databases, creating new JDBC connections still incurs networking and JDBC driver overhead that will be avoided if connections are recycled.

- Simplified programming model.

  When using connection pooling, each individual thread can act as though it has created its own JDBC connection, allowing you to use straightforward JDBC programming techniques.

- Controlled resource usage.

  If you create a new connection every time a thread needs one, rather than using connection pooling, your application's resource usage can be wasteful and lead to unpredictable behavior under load.

Using Connection Pooling with Connector/J

Sun has standardized the concept of connection pooling in JDBC through the JDBC 2.0 Optional interfaces, and all major application servers have implementations of these APIs that work with MySQL Connector/J.

Generally, you configure a connection pool in your application server configuration files, and access it through the Java Naming and Directory Interface (JNDI). The following code shows how you might use a connection pool from an application deployed in a J2EE application server:

Example 7.1. Connector/J: Using a connection pool with a J2EE application server

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import javax.naming.InitialContext;
import javax.sql.DataSource;
public class MyServletJspOrEjb {
    public void doSomething() throws Exception {
        /*
         * Create a JNDI Initial context to be able to
         *  lookup  the DataSource
```

```
       *
       * In production-level code, this should be cached as
       * an instance or static variable, as it can
       * be quite expensive to create a JNDI context.
       *
       * Note: This code only works when you are using servlets
       * or EJBs in a J2EE application server. If you are
       * using connection pooling in standalone Java code, you
       * will have to create/configure datasources using whatever
       * mechanisms your particular connection pooling library
       * provides.
       */
      InitialContext ctx = new InitialContext();
       /*
        * Lookup the DataSource, which will be backed by a pool
        * that the application server provides. DataSource instances
        * are also a good candidate for caching as an instance
        * variable, as JNDI lookups can be expensive as well.
        */
      DataSource ds =
        (DataSource)ctx.lookup("java:comp/env/jdbc/MySQLDB");
       /*
        * The following code is what would actually be in your
        * Servlet, JSP or EJB 'service' method...where you need
        * to work with a JDBC connection.
        */
      Connection conn = null;
      Statement stmt = null;
      try {
          conn = ds.getConnection();
          /*
           * Now, use normal JDBC programming to work with
           * MySQL, making sure to close each resource when you're
           * finished with it, which permits the connection pool
           * resources to be recovered as quickly as possible
           */
          stmt = conn.createStatement();
          stmt.execute("SOME SQL QUERY");
          stmt.close();
          stmt = null;
          conn.close();
          conn = null;
      } finally {
          /*
           * close any jdbc instances here that weren't
           * explicitly closed during normal code path, so
           * that we don't 'leak' resources...
           */
          if (stmt != null) {
              try {
                  stmt.close();
              } catch (sqlexception sqlex) {
                  // ignore, as we can't do anything about it here
              }
              stmt = null;
          }
          if (conn != null) {
              try {
                  conn.close();
              } catch (sqlexception sqlex) {
                  // ignore, as we can't do anything about it here
              }
              conn = null;
          }
      }
   }
}
```

As shown in the example above, after obtaining the JNDI `InitialContext`, and looking up the `DataSource`, the rest of the code follows familiar JDBC conventions.

When using connection pooling, always make sure that connections, and anything created by them (such as statements or result sets) are closed. This rule applies no matter what happens in your code (exceptions, flow-of-control, and so forth). When these objects are closed, they can be re-used; otherwise, they will be stranded, which means that the MySQL server resources they represent (such as buffers, locks, or sockets) are tied up for some time, or in the worst case can be tied up forever.

Sizing the Connection Pool

Each connection to MySQL has overhead (memory, CPU, context switches, and so forth) on both the client and server side. Every connection limits how many resources there are available to your application as well as the MySQL server. Many of these resources will be used whether or not the connection is actually doing any useful work! Connection pools can be tuned to maximize performance, while keeping resource utilization below the point where your application will start to fail rather than just run slower.

The optimal size for the connection pool depends on anticipated load and average database transaction time. In practice, the optimal connection pool size can be smaller than you might expect. If you take Sun's Java Petstore blueprint application for example, a connection pool of 15-20 connections can serve a relatively moderate load (600 concurrent users) using MySQL and Tomcat with acceptable response times.

To correctly size a connection pool for your application, create load test scripts with tools such as Apache JMeter or The Grinder, and load test your application.

An easy way to determine a starting point is to configure your connection pool's maximum number of connections to be unbounded, run a load test, and measure the largest amount of concurrently used connections. You can then work backward from there to determine what values of minimum and maximum pooled connections give the best performance for your particular application.

Validating Connections

MySQL Connector/J can validate the connection by executing a lightweight ping against a server. In the case of load-balanced connections, this is performed against all active pooled internal connections that are retained. This is beneficial to Java applications using connection pools, as the pool can use this feature to validate connections. Depending on your connection pool and configuration, this validation can be carried out at different times:

1. Before the pool returns a connection to the application.

2. When the application returns a connection to the pool.

3. During periodic checks of idle connections.

To use this feature, specify a validation query in your connection pool that starts with `/* ping */`. Note that the syntax must be exactly as specified. This will cause the driver send a ping to the server and return a dummy lightweight result set. When using a `ReplicationConnection` or `LoadBalancedConnection`, the ping will be sent across all active connections.

It is critical that the syntax be specified correctly. The syntax needs to be exact for reasons of efficiency, as this test is done for every statement that is executed:

```
protected static final String PING_MARKER = "/* ping */";
...
if (sql.charAt(0) == '/') {
if (sql.startsWith(PING_MARKER)) {
doPingInstead();
...
```

None of the following snippets will work, because the ping syntax is sensitive to whitespace, capitalization, and placement:

```
sql = "/* PING */ SELECT 1";
sql = "SELECT 1 /* ping*/";
sql = "/*ping*/ SELECT 1";
sql = " /* ping */ SELECT 1";
sql = "/*to ping or not to ping*/ SELECT 1";
```

All of the previous statements will issue a normal `SELECT` statement and will **not** be transformed into the lightweight ping. Further, for load-balanced connections, the statement will be executed against one connection in the internal pool, rather than validating each underlying physical connection. This results in the non-active physical connections assuming a stale state, and they may die. If Connector/J then re-balances, it might select a dead connection, resulting in an exception being passed to the application. To help prevent this, you can use `loadBalanceValidateConnectionOnSwapServer` to validate the connection before use.

If your Connector/J deployment uses a connection pool that allows you to specify a validation query, take advantage of it, but ensure that the query starts *exactly* with `/* ping */`. This is particularly important if you are using the load-balancing or replication-aware features of Connector/J, as it will help keep alive connections which otherwise will go stale and die, causing problems later.

7.2. Managing Load-Balanced Connections

Connector/J has long provided an effective means to distribute read/write load across multiple MySQL server instances for Cluster or master-master replication deployments. Starting with Connector/J 5.1.3, you can now dynamically configure load-balanced connections, with no service outage. In-process transactions are not lost, and no application exceptions are generated if any application is trying to use that particular server instance.

There are two connection string options associated with this functionality:

- `loadBalanceConnectionGroup` – This provides the ability to group connections from different sources. This allows you to manage these JDBC sources within a single class loader in any combination you choose. If they use the same configuration, and you want to manage them as a logical single group, give them the same name. This is the key property for management: if you do not define a name (string) for `loadBalanceConnectionGroup`, you cannot manage the connections. All load-balanced connections sharing the same `loadBalanceConnectionGroup` value, regardless of how the application creates them, will be managed together.

- `loadBalanceEnableJMX` – The ability to manage the connections is exposed when you define a `loadBalanceConnectionGroup`, but if you want to manage this externally, enable JMX by setting this property to `true`. This enables a JMX implementation, which exposes the management and monitoring operations of a connection group. Further, start your application with the `-Dcom.sun.management.jmxremote` JVM flag. You can then perform connect and perform operations using a JMX client such as `jconsole`.

Once a connection has been made using the correct connection string options, a number of monitoring properties are available:

- Current active host count.

- Current active physical connection count.

- Current active logical connection count.

- Total logical connections created.

- Total transaction count.

The following management operations can also be performed:

- Add host.

- Remove host.

The JMX interface, `com.mysql.jdbc.jmx.LoadBalanceConnectionGroupManagerMBean`, has the following methods:

- `int getActiveHostCount(String group);`

- `int getTotalHostCount(String group);`

- `long getTotalLogicalConnectionCount(String group);`

- `long getActiveLogicalConnectionCount(String group);`

- `long getActivePhysicalConnectionCount(String group);`

- `long getTotalPhysicalConnectionCount(String group);`

- `long getTotalTransactionCount(String group);`

- `void removeHost(String group, String host) throws SQLException;`

- `void stopNewConnectionsToHost(String group, String host) throws SQLException;`

- `void addHost(String group, String host, boolean forExisting);`

- `String getActiveHostsList(String group);`

- `String getRegisteredConnectionGroups();`

The `getRegisteredConnectionGroups()` method returns the names of all connection groups defined in that class loader.

You can test this setup with the following code:

```
public class Test {
    private static String URL = "jdbc:mysql:loadbalance://" +
        "localhost:3306,localhost:3310/test?" +
        "loadBalanceConnectionGroup=first&loadBalanceEnableJMX=true";
    public static void main(String[] args) throws Exception {
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
    }
    static Connection getNewConnection() throws SQLException, ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        return DriverManager.getConnection(URL, "root", "");
    }
    static void executeSimpleTransaction(Connection c, int conn, int trans){
        try {
            c.setAutoCommit(false);
            Statement s = c.createStatement();
            s.executeQuery("SELECT SLEEP(1) /* Connection: " + conn + ", transaction: " + trans + " */");
            c.commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public static class Repeater implements Runnable {
        public void run() {
            for(int i=0; i < 100; i++){
                try {
                    Connection c = getNewConnection();
                    for(int j=0; j < 10; j++){
                        executeSimpleTransaction(c, i, j);
                        Thread.sleep(Math.round(100 * Math.random()));
                    }
                    c.close();
                    Thread.sleep(100);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

After compiling, the application can be started with the `-Dcom.sun.management.jmxremote` flag, to enable remote management. `jconsole` can then be started. The `Test` main class will be listed by `jconsole`. Select this and click CONNECT. You can then navigate to the `com.mysql.jdbc.jmx.LoadBalanceConnectionGroupManager` bean. At this point, you can click on various operations and examine the returned result.

If you now had an additional instance of MySQL running on port 3309, you could ensure that Connector/J starts using it by using the `addHost()`, which is exposed in `jconsole`. Note that these operations can be performed dynamically without having to stop the application running.

7.2.1. Load Balancing Failover Policies

Connector/J provides a useful load-balancing implementation for Cluster or multi-master deployments. As of Connector/J 5.1.12, this same implementation is used for balancing load between read-only slaves with `ReplicationDriver`. When trying to balance workload between multiple servers, the driver has to determine when it is safe to swap servers, doing so in the middle of a transaction, for example, could cause problems. It is important not to lose state information. For this reason, Connector/J will only try to pick a new server when one of the following happens:

1. At transaction boundaries (transactions are explicitly committed or rolled back).

2. A communication exception (SQL State starting with "08") is encountered.

3. When a `SQLException` matches conditions defined by user, using the extension points defined by the `loadBalanceSQL-StateFailover`, `loadBalanceSQLExceptionSubclassFailover` or `loadBalanceExceptionChecker` properties.

The third condition revolves around three new properties introduced with Connector/J 5.1.13. It allows you to control which `SQLException`s trigger failover.

- `loadBalanceExceptionChecker` - The `loadBalanceExceptionChecker` property is really the key. This takes a fully-qualified class name which implements the new `com.mysql.jdbc.LoadBalanceExceptionChecker` interface. This interface is very simple, and you only need to implement the following method:

```
public boolean shouldExceptionTriggerFailover(SQLException ex)
```

A `SQLException` is passed in, and a boolean returned. A value of `true` triggers a failover, `false` does not.

You can use this to implement your own custom logic. An example where this might be useful is when dealing with transient errors when using MySQL Cluster, where certain buffers may become overloaded. The following code snippet illustrates this:

```
public class NdbLoadBalanceExceptionChecker
 extends StandardLoadBalanceExceptionChecker {
 public boolean shouldExceptionTriggerFailover(SQLException ex) {
  return super.shouldExceptionTriggerFailover(ex)
     ||   checkNdbException(ex);
 }
 private boolean checkNdbException(SQLException ex){
 // Have to parse the message since most NDB errors
 // are mapped to the same DEMC.
  return (ex.getMessage().startsWith("Lock wait timeout exceeded") ||
  (ex.getMessage().startsWith("Got temporary error")
  && ex.getMessage().endsWith("from NDB")));
 }
}
```

The code above extends `com.mysql.jdbc.StandardLoadBalanceExceptionChecker`, which is the default implementation. There are a few convenient shortcuts built into this, for those who want to have some level of control using properties, without writing Java code. This default implementation uses the two remaining properties: `loadBalanceSQLStateFailover` and `loadBalanceSQLExceptionSubclassFailover`.

- `loadBalanceSQLStateFailover` - allows you to define a comma-delimited list of `SQLState` code prefixes, against which a `SQLException` is compared. If the prefix matches, failover is triggered. So, for example, the following would trigger a failover if a given `SQLException` starts with "00", or is "12345":

```
loadBalanceSQLStateFailover=00,12345
```

- `loadBalanceSQLExceptionSubclassFailover` - can be used in conjunction with `loadBalanceSQLStateFailover` or on its own. If you want certain subclasses of `SQLException` to trigger failover, simply provide a comma-delimited list of fully-qualified class or interface names to check against. For example, if you want all `SQLTransientConnectionExceptions` to trigger failover, you would specify:

```
loadBalanceSQLExceptionSubclassFailover=java.sql.SQLTransientConnectionException
```

While the three fail-over conditions enumerated earlier suit most situations, if `auto-commit` is enabled, Connector/J never re-balances, and continues using the same physical connection. This can be problematic, particularly when load-balancing is being used to distribute read-only load across multiple slaves. However, Connector/J can be configured to re-balance after a certain number of statements are executed, when `auto-commit` is enabled. This functionality is dependent upon the following properties:

- `loadBalanceAutoCommitStatementThreshold` – defines the number of matching statements which will trigger the driver to potentially swap physical server connections. The default value, 0, retains the behavior that connections with `auto-commit` enabled are never balanced.

- `loadBalanceAutoCommitStatementRegex` – the regular expression against which statements must match. The default value, blank, matches all statements. So, for example, using the following properties will cause Connector/J to re-balance after every third statement that contains the string "test":

```
loadBalanceAutoCommitStatementThreshold=3
loadBalanceAutoCommitStatementRegex=.*test.*
```

`loadBalanceAutoCommitStatementRegex` can prove useful in a number of situations. Your application may use temporary tables, server-side session state variables, or connection state, where letting the driver arbitrarily swap physical connections before processing is complete could cause data loss or other problems. This allows you to identify a trigger statement that is only executed when it is safe to swap physical connections.

Chapter 8. Using Connector/J with Tomcat

The following instructions are based on the instructions for Tomcat-5.x, available at http://tomcat.apache.org/tomcat-5.5-doc/jndi-datasource-examples-howto.html which is current at the time this document was written.

First, install the `.jar` file that comes with Connector/J in `$CATALINA_HOME/common/lib` so that it is available to all applications installed in the container.

Next, configure the JNDI DataSource by adding a declaration resource to `$CATALINA_HOME/conf/server.xml` in the context that defines your web application:

```
  <Context ....>
  ...
  <Resource name="jdbc/MySQLDB"
              auth="Container"
              type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/MySQLDB">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>10</value>
    </parameter>
    <parameter>
      <name>maxIdle</name>
      <value>5</value>
    </parameter>
    <parameter>
      <name>validationQuery</name>
      <value>SELECT 1</value>
    </parameter>
    <parameter>
      <name>testOnBorrow</name>
      <value>true</value>
    </parameter>
    <parameter>
      <name>testWhileIdle</name>
      <value>true</value>
    </parameter>
    <parameter>
      <name>timeBetweenEvictionRunsMillis</name>
      <value>10000</value>
    </parameter>
    <parameter>
      <name>minEvictableIdleTimeMillis</name>
      <value>60000</value>
    </parameter>
    <parameter>
     <name>username</name>
     <value>someuser</value>
    </parameter>
    <parameter>
     <name>password</name>
     <value>somepass</value>
    </parameter>
    <parameter>
       <name>driverClassName</name>
       <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost:3306/test</value>
    </parameter>
  </ResourceParams>
</Context>
```

Note that Connector/J 5.1.3 introduced a facility whereby, rather than use a `validationQuery` value of `SELECT 1`, it is possible to use `validationQuery` with a value set to `/* ping */`. This sends a ping to the server which then returns a fake result set. This is a lighter weight solution. It also has the advantage that if using `ReplicationConnection` or `LoadBalancedConnection` type connections, the ping will be sent across all active connections. The following XML snippet illustrates how to select this option:

```
<parameter>
 <name>validationQuery</name>
 <value>/* ping */</value>
</parameter>
```

Note that `/* ping */` has to be specified exactly.

In general, follow the installation instructions that come with your version of Tomcat, as the way you configure datasources in Tomcat changes from time to time, and if you use the wrong syntax in your XML file, you will most likely end up with an exception similar to the following:

```
Error: java.sql.SQLException: Cannot load JDBC driver class 'null ' SQL
state: null
```

Note that the auto-loading of drivers having the `META-INF/service/java.sql.Driver` class in JDBC 4.0 causes an improper undeployment of the Connector/J driver in Tomcat on Windows. Namely, the Connector/J jar remains locked. This is an initialization problem that is not related to the driver. The possible workarounds, if viable, are as follows: use "`antiResourceLocking=true`" as a Tomcat Context attribute, or remove the `META-INF/` directory.

Chapter 9. Using Connector/J with JBoss

These instructions cover JBoss-4.x. To make the JDBC driver classes available to the application server, copy the `.jar` file that comes with Connector/J to the `lib` directory for your server configuration (which is usually called `default`). Then, in the same configuration directory, in the subdirectory named deploy, create a datasource configuration file that ends with `-ds.xml`, which tells JBoss to deploy this file as a JDBC Datasource. The file should have the following contents:

```
<datasources>
    <local-tx-datasource>
        <jndi-name>MySQLDB</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/dbname</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>user</user-name>
        <password>pass</password>
        <min-pool-size>5</min-pool-size>
        <max-pool-size>20</max-pool-size>
        <idle-timeout-minutes>5</idle-timeout-minutes>
        <exception-sorter-class-name>
  com.mysql.jdbc.integration.jboss.ExtendedMysqlExceptionSorter
        </exception-sorter-class-name>
        <valid-connection-checker-class-name>
  com.mysql.jdbc.integration.jboss.MysqlValidConnectionChecker
        </valid-connection-checker-class-name>
    </local-tx-datasource>
</datasources>
```

Chapter 10. Using Connector/J with Spring

The Spring Framework is a Java-based application framework designed for assisting in application design by providing a way to configure components. The technique used by Spring is a well known design pattern called Dependency Injection (see Inversion of Control Containers and the Dependency Injection pattern). This article will focus on Java-oriented access to MySQL databases with Spring 2.0. For those wondering, there is a .NET port of Spring appropriately named Spring.NET.

Spring is not only a system for configuring components, but also includes support for aspect oriented programming (AOP). This is one of the main benefits and the foundation for Spring's resource and transaction management. Spring also provides utilities for integrating resource management with JDBC and Hibernate.

For the examples in this section the MySQL world sample database will be used. The first task is to set up a MySQL data source through Spring. Components within Spring use the "bean" terminology. For example, to configure a connection to a MySQL server supporting the world sample database, you might use:

```
<util:map id="dbProps">
    <entry key="db.driver" value="com.mysql.jdbc.Driver"/>
    <entry key="db.jdbcurl" value="jdbc:mysql://localhost/world"/>
    <entry key="db.username" value="myuser"/>
    <entry key="db.password" value="mypass"/>
</util:map>
```

In the above example, we are assigning values to properties that will be used in the configuration. For the datasource configuration:

```
<bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${db.driver}"/>
    <property name="url" value="${db.jdbcurl}"/>
    <property name="username" value="${db.username}"/>
    <property name="password" value="${db.password}"/>
</bean>
```

The placeholders are used to provide values for properties of this bean. This means that you can specify all the properties of the configuration in one place instead of entering the values for each property on each bean. We do, however, need one more bean to pull this all together. The last bean is responsible for actually replacing the placeholders with the property values.

```
<bean
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties" ref="dbProps"/>
</bean>
```

Now that we have our MySQL data source configured and ready to go, we write some Java code to access it. The example below will retrieve three random cities and their corresponding country using the data source we configured with Spring.

```
// Create a new application context. this processes the Spring config
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("ex1appContext.xml");
// Retrieve the data source from the application context
    DataSource ds = (DataSource) ctx.getBean("dataSource");
// Open a database connection using Spring's DataSourceUtils
Connection c = DataSourceUtils.getConnection(ds);
try {
    // retrieve a list of three random cities
    PreparedStatement ps = c.prepareStatement(
        "select City.Name as 'City', Country.Name as 'Country' " +
        "from City inner join Country on City.CountryCode = Country.Code " +
        "order by rand() limit 3");
    ResultSet rs = ps.executeQuery();
    while(rs.next()) {
        String city = rs.getString("City");
        String country = rs.getString("Country");
        System.out.printf("The city %s is in %s%n", city, country);
    }
} catch (SQLException ex) {
    // something has failed and we print a stack trace to analyse the error
    ex.printStackTrace();
    // ignore failure closing connection
    try { c.close(); } catch (SQLException e) { }
} finally {
    // properly release our connection
    DataSourceUtils.releaseConnection(c, ds);
}
```

This is very similar to normal JDBC access to MySQL with the main difference being that we are using DataSourceUtils instead of the DriverManager to create the connection.

While it may seem like a small difference, the implications are somewhat far reaching. Spring manages this resource in a way similar to a container managed data source in a J2EE application server. When a connection is opened, it can be subsequently accessed in other parts of the code if it is synchronized with a transaction. This makes it possible to treat different parts of your application as transactional instead of passing around a database connection.

10.1. Using `JdbcTemplate`

Spring makes extensive use of the Template method design pattern (see Template Method Pattern). Our immediate focus will be on the `JdbcTemplate` and related classes, specifically `NamedParameterJdbcTemplate`. The template classes handle obtaining and releasing a connection for data access when one is needed.

The next example shows how to use `NamedParameterJdbcTemplate` inside of a DAO (Data Access Object) class to retrieve a random city given a country code.

```
public class Ex2JdbcDao {
    /**
     * Data source reference which will be provided by Spring.
     */
    private DataSource dataSource;
    /**
     * Our query to find a random city given a country code. Notice
     * the ":country" parameter toward the end. This is called a
     * named parameter.
     */
    private String queryString = "select Name from City " +
        "where CountryCode = :country order by rand() limit 1";
    /**
     * Retrieve a random city using Spring JDBC access classes.
     */
    public String getRandomCityByCountryCode(String cntryCode) {
        // A template that permits using queries with named parameters
        NamedParameterJdbcTemplate template =
        new NamedParameterJdbcTemplate(dataSource);
        // A java.util.Map is used to provide values for the parameters
        Map params = new HashMap();
        params.put("country", cntryCode);
        // We query for an Object and specify what class we are expecting
        return (String)template.queryForObject(queryString, params, String.class);
    }
    /**
     * A JavaBean setter-style method to allow Spring to inject the data source.
     * @param dataSource
     */
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

The focus in the above code is on the `getRandomCityByCountryCode()` method. We pass a country code and use the `NamedParameterJdbcTemplate` to query for a city. The country code is placed in a Map with the key "country", which is the parameter is named in the SQL query.

To access this code, you need to configure it with Spring by providing a reference to the data source.

```
<bean id="dao" class="code.Ex2JdbcDao">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

At this point, we can just grab a reference to the DAO from Spring and call `getRandomCityByCountryCode()`.

```
    // Create the application context
    ApplicationContext ctx =
    new ClassPathXmlApplicationContext("ex2appContext.xml");
    // Obtain a reference to our DAO
    Ex2JdbcDao dao = (Ex2JdbcDao) ctx.getBean("dao");
    String countryCode = "USA";
    // Find a few random cities in the US
    for(int i = 0; i < 4; ++i)
        System.out.printf("A random city in %s is %s%n", countryCode,
            dao.getRandomCityByCountryCode(countryCode));
```

This example shows how to use Spring's JDBC classes to completely abstract away the use of traditional JDBC classes including `Connection` and `PreparedStatement`.

10.2. Transactional JDBC Access

You might be wondering how we can add transactions into our code if we do not deal directly with the JDBC classes. Spring provides a transaction management package that not only replaces JDBC transaction management, but also enables declarative transaction management (configuration instead of code).

To use transactional database access, we will need to change the storage engine of the tables in the world database. The downloaded script explicitly creates MyISAM tables which do not support transactional semantics. The InnoDB storage engine does support transactions and this is what we will be using. We can change the storage engine with the following statements.

```
ALTER TABLE City ENGINE=InnoDB;
ALTER TABLE Country ENGINE=InnoDB;
ALTER TABLE CountryLanguage ENGINE=InnoDB;
```

A good programming practice emphasized by Spring is separating interfaces and implementations. What this means is that we can create a Java interface and only use the operations on this interface without any internal knowledge of what the actual implementation is. We will let Spring manage the implementation and with this it will manage the transactions for our implementation.

First you create a simple interface:

```
public interface Ex3Dao {
    Integer createCity(String name, String countryCode,
    String district, Integer population);
}
```

This interface contains one method that will create a new city record in the database and return the id of the new record. Next you need to create an implementation of this interface.

```
public class Ex3DaoImpl implements Ex3Dao {
    protected DataSource dataSource;
    protected SqlUpdate updateQuery;
    protected SqlFunction idQuery;
    public Integer createCity(String name, String countryCode,
        String district, Integer population) {
            updateQuery.update(new Object[] { name, countryCode,
                district, population });
            return getLastId();
        }
    protected Integer getLastId() {
        return idQuery.run();
    }
}
```

You can see that we only operate on abstract query objects here and do not deal directly with the JDBC API. Also, this is the complete implementation. All of our transaction management will be dealt with in the configuration. To get the configuration started, we need to create the DAO.

```
<bean id="dao" class="code.Ex3DaoImpl">
    <property name="dataSource" ref="dataSource"/>
    <property name="updateQuery">...</property>
    <property name="idQuery">...</property>
</bean>
```

Now you need to set up the transaction configuration. The first thing you must do is create transaction manager to manage the data source and a specification of what transaction properties are required for the `dao` methods.

```
<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

The preceding code creates a transaction manager that handles transactions for the data source provided to it. The `txAdvice` uses this transaction manager and the attributes specify to create a transaction for all methods. Finally you need to apply this advice with an AOP pointcut.

```
<aop:config>
    <aop:pointcut id="daoMethods"
        expression="execution(* code.Ex3Dao.*(..))"/>
     <aop:advisor advice-ref="txAdvice" pointcut-ref="daoMethods"/>
</aop:config>
```

This basically says that all methods called on the `Ex3Dao` interface will be wrapped in a transaction. To make use of this, you only have to retrieve the `dao` from the application context and call a method on the `dao` instance.

```
Ex3Dao dao = (Ex3Dao) ctx.getBean("dao");
Integer id = dao.createCity(name,  countryCode, district, pop);
```

We can verify from this that there is no transaction management happening in our Java code and it is all configured with Spring. This is a very powerful notion and regarded as one of the most beneficial features of Spring.

10.3. Connection Pooling with Spring

In many situations, such as web applications, there will be a large number of small database transactions. When this is the case, it usually makes sense to create a pool of database connections available for web requests as needed. Although MySQL does not spawn an extra process when a connection is made, there is still a small amount of overhead to create and set up the connection. Pooling of connections also alleviates problems such as collecting large amounts of sockets in the `TIME_WAIT` state.

Setting up pooling of MySQL connections with Spring is as simple as changing the data source configuration in the application context. There are a number of configurations that we can use. The first example is based on the Jakarta Commons DBCP library. The example below replaces the source configuration that was based on `DriverManagerDataSource` with DBCP's BasicDataSource.

```
<bean id="dataSource" destroy-method="close"
  class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${db.driver}"/>
    <property name="url" value="${db.jdbcurl}"/>
    <property name="username" value="${db.username}"/>
    <property name="password" value="${db.password}"/>
    <property name="initialSize" value="3"/>
</bean>
```

The configuration of the two solutions is very similar. The difference is that DBCP will pool connections to the database instead of creating a new connection every time one is requested. We have also set a parameter here called `initialSize`. This tells DBCP that we want three connections in the pool when it is created.

Another way to configure connection pooling is to configure a data source in our J2EE application server. Using JBoss as an example, you can set up the MySQL connection pool by creating a file called `mysql-local-ds.xml` and placing it in the server/default/deploy directory in JBoss. Once we have this setup, we can use JNDI to look it up. With Spring, this lookup is very simple. The data source configuration looks like this.

```
<jee:jndi-lookup id="dataSource" jndi-name="java:MySQL_DS"/>
```

Chapter 11. Using Connector/J with GlassFish

This section explains how to use MySQL Connector/J with Glassfish ™ Server Open Source Edition 3.0.1. Glassfish can be downloaded from the Glassfish website.

Once Glassfish is installed you will need to make sure it can access MySQL Connector/J. To do this copy the MySQL Connector/J JAR file to the directory *GLASSFISH_INSTALL*/glassfish/lib. For example, copy mysql-connect-or-java-5.1.12-bin.jar to C:\glassfishv3\glassfish\lib. Restart the Glassfish Application Server.

You are now ready to create JDBC Connection Pools and JDBC Resources.

Creating a Connection Pool

1. In the Glassfish Administration Console, using the navigation tree navigate to **RESOURCES**, **JDBC**, **CONNECTION POOLS**.

2. In the **JDBC CONNECTION POOLS** frame click NEW. You will enter a two step wizard.

3. In the **NAME** field under **GENERAL SETTINGS** enter the name for the connection pool, for example enter **MySQLConnPool**.

4. In the **RESOURCE TYPE** field, select javax.sql.DataSource from the drop-down listbox.

5. In the **DATABASE VENDOR** field, select MySQL from the drop-down listbox. Click NEXT to go to the next page of the wizard.

6. You can accept the default settings for General Settings, Pool Settings and Transactions for this example. Scroll down to Additional Properties.

7. In Additional Properties you will need to ensure the following properties are set:

   - **ServerName** - The server to connect to. For local testing this will be localhost.

   - **User** - The user name with which to connect to MySQL.

   - **Password** - The corresponding password for the user.

   - **DatabaseName** - The database to connect to, for example the sample MySQL database World.

8. Click FINISH to exit the wizard. You will be taken to the **JDBC CONNECTION POOLS** page where all current connection pools, including the one you just created, will be displayed.

9. In the **JDBC CONNECTION POOLS** frame click on the connection pool you just created. Here you can review and edit information about the connection pool.

10. To test your connection pool click the PING button at the top of the frame. A message will be displayed confirming correct operation or otherwise. If an error message is received recheck the previous steps, and ensure that MySQL Connector/J has been correctly copied into the previously specified location.

Now that you have created a connection pool you will also need to create a JDBC Resource (data source) for use by your application.

Creating a JDBC Resource

Your Java application will usually reference a data source object to establish a connection with the database. This needs to be created first using the following procedure.

- Using the navigation tree in the Glassfish Administration Console, navigate to **RESOURCES**, **JDBC**, **JDBC RESOURCES**. A list of resources will be displayed in the **JDBC RESOURCES** frame.

- Click NEW. The **NEW JDBC RESOURCE** frame will be displayed.

- In the **JNDI NAME** field, enter the JNDI name that will be used to access this resource, for example enter **jdbc/MySQLDataSource**.

- In the **POOL NAME** field, select a connection pool you want this resource to use from the drop-down listbox.

- Optionally, you can enter a description into the **DESCRIPTION** field.

- Additional properties can be added if required.

- Click OK to create the new JDBC resource. The **JDBC RESOURCES** frame will list all available JDBC Resources.

11.1. A Simple JSP Application with Glassfish, Connector/J and MySQL

This section shows how to deploy a simple JSP application on Glassfish, that connects to a MySQL database.

This example assumes you have already set up a suitable Connection Pool and JDBC Resource, as explained in the preceding sections. It is also assumed you have a sample database installed, such as `world`.

The main application code, `index.jsp` is presented here:

```
<%@ page import="java.sql.*, javax.sql.*, java.io.*, javax.naming.*" %>
<html>
<head><title>Hello world from JSP</title></head>
<body>
<%
  InitialContext ctx;
  DataSource ds;
  Connection conn;
  Statement stmt;
  ResultSet rs;
  try {
    ctx = new InitialContext();
    ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
    //ds = (DataSource) ctx.lookup("jdbc/MySQLDataSource");
    conn = ds.getConnection();
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT * FROM Country");
    while(rs.next()) {
%>
    <h3>Name: <%= rs.getString("Name") %></h3>
    <h3>Population: <%= rs.getString("Population") %></h3>
<%
    }
  }
  catch (SQLException se) {
%>
    <%= se.getMessage() %>
<%
  }
  catch (NamingException ne) {
%>
    <%= ne.getMessage() %>
<%
  }
%>
</body>
</html>
```

In addition two XML files are required: `web.xml`, and `sun-web.xml`. There may be other files present, such as classes and images. These files are organized into the directory structure as follows:

```
index.jsp
WEB-INF
   |
   - web.xml
   - sun-web.xml
```

The code for `web.xml` is:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:s
  <display-name>HelloWebApp</display-name>
  <distributable/>
  <resource-ref>
    <res-ref-name>jdbc/MySQLDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
```

```
    </resource-ref>
</web-app>
```

The code for `sun-web.xml` is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 8.1 Servlet 2.4//EN" "http://www.sun.com/s
<sun-web-app>
  <context-root>HelloWebApp</context-root>
  <resource-ref>
    <res-ref-name>jdbc/MySQLDataSource</res-ref-name>
    <jndi-name>jdbc/MySQLDataSource</jndi-name>
  </resource-ref>
</sun-web-app>
```

These XML files illustrate a very important aspect of running JDBC applications on Glassfish. On Glassfish it is important to map the string specified for a JDBC resource to its JNDI name, as set up in the Glassfish administration console. In this example, the JNDI name for the JDBC resource, as specified in the Glassfish Administration console when creating the JDBC Resource, was `jdbc/MySQLDataSource`. This must be mapped to the name given in the application. In this example the name specified in the application, `jdbc/MySQLDataSource`, and the JNDI name, happen to be the same, but this does not necessarily have to be the case. Note that the XML element <res-ref-name> is used to specify the name as used in the application source code, and this is mapped to the JNDI name specified using the <jndi-name> element, in the file `sun-web.xml`. The resource also has to be created in the `web.xml` file, although the mapping of the resource to a JNDI name takes place in the `sun-web.xml` file.

If you do not have this mapping set up correctly in the XML files you will not be able to lookup the data source using a JNDI lookup string such as:

```
ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
```

You will still be able to access the data source directly using:

```
ds = (DataSource) ctx.lookup("jdbc/MySQLDataSource");
```

With the source files in place, in the correct directory structure, you are ready to deploy the application:

1. In the navigation tree, navigate to **APPLICATIONS** - the **APPLICATIONS** frame will be displayed. Click DEPLOY.

2. You can now deploy an application packaged into a single WAR file from a remote client, or you can choose a packaged file or directory that is locally accessible to the server. If you are simply testing an application locally you can simply point Glassfish at the directory that contains your application, without needing to package the application into a WAR file.

3. Now select the application type from the **TYPE** drop-down listbox, which in this example is `Web application`.

4. Click OK.

Now, when you navigate to the **APPLICATIONS** frame, you will have the option to **LAUNCH**, **REDEPLOY**, or **RESTART** your application. You can test your application by clicking **LAUNCH**. The application will connection to the MySQL database and display the Name and Population of countries in the `Country` table.

11.2. A Simple Servlet with Glassfish, Connector/J and MySQL

This section describes a simple servlet that can be used in the Glassfish environment to access a MySQL database. As with the previous section, this example assumes the sample database `world` is installed.

The project is set up with the following directory structure:

```
index.html
WEB-INF
   |
   - web.xml
   - sun-web.xml
   - classes
        |
        - HelloWebServlet.java
        - HelloWebServlet.class
```

The code for the servlet, located in `HelloWebServlet.java`, is as follows:

```java
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
public class HelloWebServlet extends HttpServlet {
  InitialContext ctx = null;
  DataSource ds = null;
  Connection conn = null;
  PreparedStatement ps = null;
  ResultSet rs = null;
  String sql = "SELECT Name, Population FROM Country WHERE Name=?";
  public void init () throws ServletException {
    try {
      ctx = new InitialContext();
      ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
      conn = ds.getConnection();
      ps = conn.prepareStatement(sql);
    }
    catch (SQLException se) {
      System.out.println("SQLException: "+se.getMessage());
    }
    catch (NamingException ne) {
      System.out.println("NamingException: "+ne.getMessage());
    }
  }
  public void destroy () {
    try {
      if (rs != null)
        rs.close();
      if (ps != null)
        ps.close();
      if (conn != null)
        conn.close();
      if (ctx != null)
        ctx.close();
    }
    catch (SQLException se) {
      System.out.println("SQLException: "+se.getMessage());
    }
    catch (NamingException ne) {
      System.out.println("NamingException: "+ne.getMessage());
    }
  }
  public void doPost(HttpServletRequest req, HttpServletResponse resp){
    try {
      String country_name = req.getParameter("country_name");
      resp.setContentType("text/html");
      PrintWriter writer = resp.getWriter();
      writer.println("<html><body>");
      writer.println("<p>Country: "+country_name+"</p>");
      ps.setString(1, country_name);
      rs = ps.executeQuery();
      if (!rs.next()){
        writer.println("<p>Country does not exist!</p>");
      }
      else {
        rs.beforeFirst();
        while(rs.next()) {
          writer.println("<p>Name: "+rs.getString("Name")+"</p>");
          writer.println("<p>Population: "+rs.getString("Population")+"</p>");
        }
      }
      writer.println("</body></html>");
      writer.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
  public void doGet(HttpServletRequest req, HttpServletResponse resp){
    try {
      resp.setContentType("text/html");
      PrintWriter writer = resp.getWriter();
      writer.println("<html><body>");
      writer.println("<p>Hello from servlet doGet()</p>");
      writer.println("</body></html>");
      writer.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

In the preceding code a basic `doGet()` method is implemented, but is not used in the example. The code to establish the connection with the database is as shown in the previous example, Section 11.1, "A Simple JSP Application with Glassfish, Connector/J and MySQL", and is most conveniently located in the servlet `init()` method. The corresponding freeing of resources is located in the destroy method. The main functionality of the servlet is located in the `doPost()` method. If the user enters nto the input form a country name that can be located in the database, the population of the country is returned. The code is invoked using a POST action associated with the input form. The form is defined in the file `index.html`:

```
<html>
  <head><title>HelloWebServlet</title></head>

  <body>
    <h1>HelloWebServlet</h1>

    <p>Please enter country name:</p>

    <form action="HelloWebServlet" method="POST">
      <input type="text" name="country_name" length="50" />
      <input type="submit" value="Submit" />
    </form>

  </body>
</html>
```

The XML files `web.xml` and `sun-web.xml` are as for the example in the preceding section, Section 11.1, "A Simple JSP Application with Glassfish, Connector/J and MySQL", no additional changes are required.

Whe compiling the Java source code, you will need to specify the path to the file `javaee.jar`. On Windows, this can be done as follows:

```
shell> javac -classpath c:\glassfishv3\glassfish\lib\javaee.jar HelloWebServlet.java
```

Once the code is correctly located within its directory structure, and compiled, the application can be deployed in Glassfish. This is done in exactly the same way as described in the preceding section, Section 11.1, "A Simple JSP Application with Glassfish, Connector/J and MySQL".

Once deployed the application can be launched from within the Glassfish Administration Console. Enter a country name such as "England", and the application will return "Country does not exist!". Enter "France", and the application will return a population of 59225700.

Chapter 12. Troubleshooting Connector/J Applications

This section explains the symptoms and resolutions for the most commonly encountered issues with applications using MySQL Connector/J.

Questions

- 12.1: When I try to connect to the database with MySQL Connector/J, I get the following exception:

```
SQLException: Server configuration denies access to data source
SQLState: 08001
VendorError: 0
```

What is going on? I can connect just fine with the MySQL command-line client.

- 12.2: My application throws an SQLException 'No Suitable Driver'. Why is this happening?

- 12.3: I'm trying to use MySQL Connector/J in an applet or application and I get an exception similar to:

```
SQLException: Cannot connect to MySQL server on host:3306.
Is there a MySQL server running on the machine/port you
are trying to connect to?
(java.security.AccessControlException)
SQLState: 08S01
VendorError: 0
```

- 12.4: I have a servlet/application that works fine for a day, and then stops working overnight

- 12.5: I'm trying to use JDBC 2.0 updatable result sets, and I get an exception saying my result set is not updatable.

- 12.6: I cannot connect to the MySQL server using Connector/J, and I'm sure the connection parameters are correct.

- 12.7: I am trying to connect to my MySQL server within my application, but I get the following error and stack trace:

```
java.net.SocketException
MESSAGE: Software caused connection abort: recv failed
STACKTRACE:
java.net.SocketException: Software caused connection abort: recv failed
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(Unknown Source)
at com.mysql.jdbc.MysqlIO.readFully(MysqlIO.java:1392)
at com.mysql.jdbc.MysqlIO.readPacket(MysqlIO.java:1414)
at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:625)
at com.mysql.jdbc.Connection.createNewIO(Connection.java:1926)
at com.mysql.jdbc.Connection.<init>(Connection.java:452)
at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:411)
```

- 12.8: My application is deployed through JBoss and I am using transactions to handle the statements on the MySQL database. Under heavy loads I am getting a error and stack trace, but these only occur after a fixed period of heavy activity.

- 12.9: When using `gcj` an `java.io.CharConversionException` is raised when working with certain character sequences.

- 12.10: Updating a table that contains a primary key that is either `FLOAT` or compound primary key that uses `FLOAT` fails to update the table and raises an exception.

- 12.11: You get an `ER_NET_PACKET_TOO_LARGE` exception, even though the binary blob size you want to insert using JDBC is safely below the `max_allowed_packet` size.

- 12.12: What should you do if you receive error messages similar to the following: "Communications link failure – Last packet sent to the server was X ms ago"?

- 12.13: Why does Connector/J not reconnect to MySQL and re-issue the statement after a communication failure, instead of throwing an Exception, even though I use the `autoReconnect` connection string option?

- 12.14: How can I use 3-byte UTF8 with Connector/J?

- 12.15: How can I use 4-byte UTF8, `utf8mb4` with Connector/J?

- 12.16: Using `useServerPrepStmts=false` and certain character encodings can lead to corruption when inserting BLOBs.

How can this be avoided?

Questions and Answers

12.1: When I try to connect to the database with MySQL Connector/J, I get the following exception:

```
SQLException: Server configuration denies access to data source
SQLState: 08001
VendorError: 0
```

What is going on? I can connect just fine with the MySQL command-line client.

MySQL Connector/J must use TCP/IP sockets to connect to MySQL, as Java does not support Unix Domain Sockets. Therefore, when MySQL Connector/J connects to MySQL, the security manager in MySQL server will use its grant tables to determine whether the connection is permitted.

You must add the necessary security credentials to the MySQL server for this to happen, using the GRANT statement to your MySQL Server. See <literal>GRANT</literal> Syntax, for more information.

> **Note**
>
> Testing your connectivity with the mysql command-line client will not work unless you add the "host" flag, and use something other than localhost for the host. The mysql command-line client will use Unix domain sockets if you use the special host name localhost. If you are testing connectivity to localhost, use 127.0.0.1 as the host name instead.

> **Warning**
>
> Changing privileges and permissions improperly in MySQL can potentially cause your server installation to not have optimal security properties.

12.2: My application throws an SQLException 'No Suitable Driver'. Why is this happening?

There are three possible causes for this error:

- The Connector/J driver is not in your CLASSPATH, see Chapter 3, *Connector/J Installation*.

- The format of your connection URL is incorrect, or you are referencing the wrong JDBC driver.

- When using DriverManager, the jdbc.drivers system property has not been populated with the location of the Connector/J driver.

12.3: I'm trying to use MySQL Connector/J in an applet or application and I get an exception similar to:

```
SQLException: Cannot connect to MySQL server on host:3306.
Is there a MySQL server running on the machine/port you
are trying to connect to?
(java.security.AccessControlException)
SQLState: 08S01
VendorError: 0
```

Either you're running an Applet, your MySQL server has been installed with the "skip-networking" option set, or your MySQL server has a firewall sitting in front of it.

Applets can only make network connections back to the machine that runs the web server that served the .class files for the applet. This means that MySQL must run on the same machine (or you must have some sort of port re-direction) for this to work. This also means that you will not be able to test applets from your local file system, you must always deploy them to a web server.

MySQL Connector/J can only communicate with MySQL using TCP/IP, as Java does not support Unix domain sockets. TCP/IP communication with MySQL might be affected if MySQL was started with the "skip-networking" flag, or if it is firewalled.

If MySQL has been started with the "skip-networking" option set (the Debian Linux package of MySQL server does this for example), you need to comment it out in the file /etc/mysql/my.cnf or /etc/my.cnf. Of course your my.cnf file might also exist in the data directory of your MySQL server, or anywhere else (depending on how MySQL was compiled for your system). Binaries created by us always look in /etc/my.cnf and [datadir]/my.cnf. If your MySQL server has been firewalled, you will need to have the firewall configured to al-

low TCP/IP connections from the host where your Java code is running to the MySQL server on the port that MySQL is listening to (by default, 3306).

12.4: I have a servlet/application that works fine for a day, and then stops working overnight

MySQL closes connections after 8 hours of inactivity. You either need to use a connection pool that handles stale connections or use the "autoReconnect" parameter (see Section 5.1, "Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J").

Also, catch SQLExceptions in your application and deal with them, rather than propagating them all the way until your application exits. This is just good programming practice. MySQL Connector/J will set the SQLState (see `java.sql.SQLException.getSQLState()` in your API docs) to "08S01" when it encounters network-connectivity issues during the processing of a query. Attempt to reconnect to MySQL at this point.

The following (simplistic) example shows what code that can handle these exceptions might look like:

Example 12.1. Connector/J: Example of transaction with retry logic

```
public void doBusinessOp() throws SQLException {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    //
    // How many times do you want to retry the transaction
    // (or at least _getting_ a connection)?
    //
    int retryCount = 5;
    boolean transactionCompleted = false;
    do {
        try {
            conn = getConnection(); // assume getting this from a
                                    // javax.sql.DataSource, or the
                                    // java.sql.DriverManager
            conn.setAutoCommit(false);
            //
            // Okay, at this point, the 'retry-ability' of the
            // transaction really depends on your application logic,
            // whether or not you're using autocommit (in this case
            // not), and whether you're using transactional storage
            // engines
            //
            // For this example, we'll assume that it's _not_ safe
            // to retry the entire transaction, so we set retry
            // count to 0 at this point
            //
            // If you were using exclusively transaction-safe tables,
            // or your application could recover from a connection going
            // bad in the middle of an operation, then you would not
            // touch 'retryCount' here, and just let the loop repeat
            // until retryCount == 0.
            //
            retryCount = 0;
            stmt = conn.createStatement();
            String query = "SELECT foo FROM bar ORDER BY baz";
            rs = stmt.executeQuery(query);
            while (rs.next()) {
            }
            rs.close();
            rs = null;
            stmt.close();
            stmt = null;
            conn.commit();
            conn.close();
            conn = null;
            transactionCompleted = true;
        } catch (SQLException sqlEx) {
            //
            // The two SQL states that are 'retry-able' are 08S01
            // for a communications error, and 40001 for deadlock.
            //
            // Only retry if the error was due to a stale connection,
            // communications problem or deadlock
            //
            String sqlState = sqlEx.getSQLState();
            if ("08S01".equals(sqlState) || "40001".equals(sqlState)) {
                retryCount -= 1;
            } else {
                retryCount = 0;
            }
        } finally {
            if (rs != null) {
```

```
            try {
                rs.close();
            } catch (SQLException sqlEx) {
                // You'd probably want to log this...
            }
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException sqlEx) {
                // You'd probably want to log this as well...
            }
        }
        if (conn != null) {
            try {
                //
                // If we got here, and conn is not null, the
                // transaction should be rolled back, as not
                // all work has been done
                try {
                    conn.rollback();
                } finally {
                    conn.close();
                }
            } catch (SQLException sqlEx) {
                //
                // If we got an exception here, something
                // pretty serious is going on, so we better
                // pass it up the stack, rather than just
                // logging it...
                throw sqlEx;
            }
        }
    }
} while (!transactionCompleted && (retryCount > 0));
}
```

> **Note**
>
> Use of the `autoReconnect` option is not recommended because there is no safe method of reconnecting to the MySQL
> server without risking some corruption of the connection state or database state information. Instead, use a connection
> pool, which will enable your application to connect to the MySQL server using an available connection from the pool. The
> `autoReconnect` facility is deprecated, and may be removed in a future release.

12.5: I'm trying to use JDBC 2.0 updatable result sets, and I get an exception saying my result set is not updatable.

Because MySQL does not have row identifiers, MySQL Connector/J can only update result sets that have come from queries on tables
that have at least one primary key, the query must select every primary key and the query can only span one table (that is, no joins).
This is outlined in the JDBC specification.

Note that this issue only occurs when using updatable result sets, and is caused because Connector/J is unable to guarantee that it can
identify the correct rows within the result set to be updated without having a unique reference to each row. There is no requirement to
have a unique field on a table if you are using `UPDATE` or `DELETE` statements on a table where you can individually specify the criteria
to be matched using a `WHERE` clause.

12.6: I cannot connect to the MySQL server using Connector/J, and I'm sure the connection parameters are correct.

Make sure that the `skip-networking` option has not been enabled on your server. Connector/J must be able to communicate with
your server over TCP/IP, named sockets are not supported. Also ensure that you are not filtering connections through a Firewall or other
network security system. For more information, see <literal>Can't connect to [local] MySQL server</literal>.

12.7: I am trying to connect to my MySQL server within my application, but I get the following error and stack trace:

```
java.net.SocketException
MESSAGE: Software caused connection abort: recv failed
STACKTRACE:
java.net.SocketException: Software caused connection abort: recv failed
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(Unknown Source)
at com.mysql.jdbc.MysqlIO.readFully(MysqlIO.java:1392)
at com.mysql.jdbc.MysqlIO.readPacket(MysqlIO.java:1414)
at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:625)
at com.mysql.jdbc.Connection.createNewIO(Connection.java:1926)
at com.mysql.jdbc.Connection.<init>(Connection.java:452)
at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:411)
```

The error probably indicates that you are using a older version of the Connector/J JDBC driver (2.0.14 or 3.0.x) and you are trying to connect to a MySQL server with version 4.1x or newer. The older drivers are not compatible with 4.1 or newer of MySQL as they do not support the newer authentication mechanisms.

It is likely that the older version of the Connector/J driver exists within your application directory or your `CLASSPATH` includes the older Connector/J package.

12.8: My application is deployed through JBoss and I am using transactions to handle the statements on the MySQL database. Under heavy loads I am getting a error and stack trace, but these only occur after a fixed period of heavy activity.

This is a JBoss, not Connector/J, issue and is connected to the use of transactions. Under heavy loads the time taken for transactions to complete can increase, and the error is caused because you have exceeded the predefined timeout.

You can increase the timeout value by setting the `TransactionTimeout` attribute to the `TransactionManagerService` within the `/conf/jboss-service.xml` file (pre-4.0.3) or `/deploy/jta-service.xml` for JBoss 4.0.3 or later. See Transaction-Timeoute within the JBoss wiki for more information.

12.9: When using `gcj` an `java.io.CharConversionException` is raised when working with certain character sequences.

This is a known issue with `gcj` which raises an exception when it reaches an unknown character or one it cannot convert. Add `use-JvmCharsetConverters=true` to your connection string to force character conversion outside of the `gcj` libraries, or try a different JDK.

12.10: Updating a table that contains a primary key that is either `FLOAT` or compound primary key that uses `FLOAT` fails to update the table and raises an exception.

Connector/J adds conditions to the `WHERE` clause during an `UPDATE` to check the old values of the primary key. If there is no match then Connector/J considers this a failure condition and raises an exception.

The problem is that rounding differences between supplied values and the values stored in the database may mean that the values never match, and hence the update fails. The issue will affect all queries, not just those from Connector/J.

To prevent this issue, use a primary key that does not use `FLOAT`. If you have to use a floating point column in your primary key use `DOUBLE` or `DECIMAL` types in place of `FLOAT`.

12.11: You get an `ER_NET_PACKET_TOO_LARGE` exception, even though the binary blob size you want to insert using JDBC is safely below the `max_allowed_packet` size.

This is because the `hexEscapeBlock()` method in `com.mysql.jdbc.PreparedStatement.streamToBytes()` may almost double the size of your data.

12.12: What should you do if you receive error messages similar to the following: "Communications link failure – Last packet sent to the server was X ms ago"?

Generally speaking, this error suggests that the network connection has been closed. There can be several root causes:

- Firewalls or routers may clamp down on idle connections (the MySQL client/server protocol does not ping).

- The MySQL Server may be closing idle connections which exceed the `wait_timeout` or `interactive_timeout` threshold.

To help troubleshoot these issues, the following tips can be used. If a recent (5.1.13+) version of Connector/J is used, you will see an improved level of information compared to earlier versions. Older versions simply display the last time a packet was sent to the server, which is frequently 0 ms ago. This is of limited use, as it may be that a packet was just sent, while a packet from the server has not been received for several hours. Knowing the period of time since Connector/J last received a packet from the server is useful information, so if this is not displayed in your exception message, it is recommended that you update Connector/J.

Further, if the time a packet was last sent/received exceeds the `wait_timeout` or `interactive_timeout` threshold, this is noted in the exception message.

Although network connections can be volatile, the following can be helpful in avoiding problems:

- Ensure connections are valid when used from the connection pool. Use a query that starts with `/* ping */` to execute a lightweight ping instead of full query. Note, the syntax of the ping needs to be exactly as specified here.

- Minimize the duration a connection object is left idle while other application logic is executed.

- Explicitly validate the connection before using it if the connection has been left idle for an extended period of time.

- Ensure that `wait_timeout` and `interactive_timeout` are set sufficiently high.

- Ensure that `tcpKeepalive` is enabled.

- Ensure that any configurable firewall or router timeout settings allow for the maximum expected connection idle time.

> **Note**
>
> Do not expect to be able to reuse a connection without problems, if it has being lying idle for a period. If a connection is to be reused after being idle for any length of time, ensure that you explicitly test it before reusing it.

12.13: Why does Connector/J not reconnect to MySQL and re-issue the statement after a communication failure, instead of throwing an Exception, even though I use the `autoReconnect` connection string option?

There are several reasons for this. The first is transactional integrity. The MySQL Reference Manual states that "there is no safe method of reconnecting to the MySQL server without risking some corruption of the connection state or database state information". Consider the following series of statements for example:

```
conn.createStatement().execute(
  "UPDATE checking_account SET balance = balance - 1000.00 WHERE customer='Smith'");
conn.createStatement().execute(
  "UPDATE savings_account SET balance = balance + 1000.00 WHERE customer='Smith'");
conn.commit();
```

Consider the case where the connection to the server fails after the `UPDATE` to `checking_account`. If no exception is thrown, and the application never learns about the problem, it will continue executing. However, the server did not commit the first transaction in this case, so that will get rolled back. But execution continues with the next transaction, and increases the `savings_account` balance by 1000. The application did not receive an exception, so it continued regardless, eventually committing the second transaction, as the commit only applies to the changes made in the new connection. Rather than a transfer taking place, a deposit was made in this example.

Note that running with `auto-commit` enabled does not solve this problem. When Connector/J encounters a communication problem, there is no means to determine whether the server processed the currently executing statement or not. The following theoretical states are equally possible:

- The server never received the statement, and therefore no related processing occurred on the server.

- The server received the statement, executed it in full, but the response was not received by the client.

If you are running with `auto-commit` enabled, it is not possible to guarantee the state of data on the server when a communication exception is encountered. The statement may have reached the server, or it may not. All you know is that communication failed at some point, before the client received confirmation (or data) from the server. This does not only affect `auto-commit` statements though. If the communication problem occurred during `Connection.commit()`, the question arises of whether the transaction was committed on the server before the communication failed, or whether the server received the commit request at all.

The second reason for the generation of exceptions is that transaction-scoped contextual data may be vulnerable, for example:

- Temporary tables.

- User-defined variables.

- Server-side prepared statements.

These items are lost when a connection fails, and if the connection silently reconnects without generating an exception, this could be detrimental to the correct execution of your application.

In summary, communication errors generate conditions that may well be unsafe for Connector/J to simply ignore by silently reconnect-

ing. It is necessary for the application to be notified. It is then for the application developer to decide how to proceed in the event of connection errors and failures.

12.14: How can I use 3-byte UTF8 with Connector/J?

To use 3-byte UTF8 with Connector/J set `characterEncoding=utf8` and set `useUnicode=true` in the connection string.

12.15: How can I use 4-byte UTF8, `utf8mb4` with Connector/J?

To use 4-byte UTF8 with Connector/J configure the MySQL server with `character_set_server=utf8mb4`. Connector/J will then use that setting as long as `characterEncoding` has not been set in the connection string. This is equivalent to autodetection of the character set.

12.16: Using `useServerPrepStmts=false` and certain character encodings can lead to corruption when inserting BLOBs. How can this be avoided?

When using certain character encodings, such as SJIS, CP932, and BIG5, it is possible that BLOB data contains characters that can be interpreted as control characters, for example, backslash, '\'. This can lead to corrupted data when inserting BLOBs into the database. There are two things that need to be done to avoid this:

1. Set the connection string option `useServerPrepStmts` to `true`.

2. Set `SQL_MODE` to `NO_BACKSLASH_ESCAPES`.

Chapter 13. Connector/J Support

13.1. Connector/J Community Support

Oracle provides assistance to the user community by means of its mailing lists. For Connector/J related issues, you can get help from experienced users by using the MySQL and Java mailing list. Archives and subscription information is available online at ht-tp://lists.mysql.com/java.

For information about subscribing to MySQL mailing lists or to browse list archives, visit http://lists.mysql.com/. See MySQL Mailing Lists.

Community support from experienced users is also available through the JDBC Forum. You may also find help from other users in the other MySQL Forums, located at http://forums.mysql.com. See MySQL Community Support at the MySQL Forums.

13.2. How to Report Connector/J Bugs or Problems

The normal place to report bugs is http://bugs.mysql.com/, which is the address for our bugs database. This database is public, and can be browsed and searched by anyone. If you log in to the system, you will also be able to enter new reports.

If you have found a sensitive security bug in MySQL, you can send email to `<security@mysql.com>`.

Writing a good bug report takes patience, but doing it right the first time saves time both for us and for yourself. A good bug report, containing a full test case for the bug, makes it very likely that we will fix the bug in the next release.

This section will help you write your report correctly so that you do not waste your time doing things that may not help us much or at all.

If you have a repeatable bug report, please report it to the bugs database at http://bugs.mysql.com/. Any bug that we are able to repeat has a high chance of being fixed in the next MySQL release.

To report other problems, you can use one of the MySQL mailing lists.

Remember that it is possible for us to respond to a message containing too much information, but not to one containing too little. People often omit facts because they think they know the cause of a problem and assume that some details do not matter.

A good principle is this: If you are in doubt about stating something, state it. It is faster and less troublesome to write a couple more lines in your report than to wait longer for the answer if we must ask you to provide information that was missing from the initial report.

The most common errors made in bug reports are (a) not including the version number of Connector/J or MySQL used, and (b) not fully describing the platform on which Connector/J is installed (including the JVM version, and the platform type and version number that MySQL itself is installed on).

This is highly relevant information, and in 99 cases out of 100, the bug report is useless without it. Very often we get questions like, "Why doesn't this work for me?" Then we find that the feature requested wasn't implemented in that MySQL version, or that a bug described in a report has already been fixed in newer MySQL versions.

Sometimes the error is platform-dependent; in such cases, it is next to impossible for us to fix anything without knowing the operating system and the version number of the platform.

If at all possible, create a repeatable, standalone testcase that doesn't involve any third-party classes.

To streamline this process, we ship a base class for testcases with Connector/J, named `'com.mysql.jdbc.util.BaseBugReport'`. To create a testcase for Connector/J using this class, create your own class that in-herits from `com.mysql.jdbc.util.BaseBugReport` and override the methods `setUp()`, `tearDown()` and `runTest()`.

In the `setUp()` method, create code that creates your tables, and populates them with any data needed to demonstrate the bug.

In the `runTest()` method, create code that demonstrates the bug using the tables and data you created in the `setUp` method.

In the `tearDown()` method, drop any tables you created in the `setUp()` method.

In any of the above three methods, use one of the variants of the `getConnection()` method to create a JDBC connection to MySQL:

- `getConnection()` - Provides a connection to the JDBC URL specified in `getUrl()`. If a connection already exists, that connection is returned, otherwise a new connection is created.

- `getNewConnection()` - Use this if you need to get a new connection for your bug report (that is, there is more than one connection involved).

- `getConnection(String url)` - Returns a connection using the given URL.

- `getConnection(String url, Properties props)` - Returns a connection using the given URL and properties.

If you need to use a JDBC URL that is different from 'jdbc:mysql:///test', override the method `getUrl()` as well.

Use the `assertTrue(boolean expression)` and `assertTrue(String failureMessage, boolean expression)` methods to create conditions that must be met in your testcase demonstrating the behavior you are expecting (vs. the behavior you are observing, which is why you are most likely filing a bug report).

Finally, create a `main()` method that creates a new instance of your testcase, and calls the `run` method:

```
public static void main(String[] args) throws Exception {
      new MyBugReport().run();
}
```

Once you have finished your testcase, and have verified that it demonstrates the bug you are reporting, upload it with your bug report to http://bugs.mysql.com/.

13.3. Connector/J Change History

The Connector/J Change History (Changelog) is located with the main Changelog for MySQL. See Appendix A, *MySQL Connector/J Change History*.

Appendix A. MySQL Connector/J Change History

A.1. Changes in MySQL Connector/J 5.1.x

A.1.1. Changes in MySQL Connector/J 5.1.20 (01 May 2012)

Bugs Fixed

- **Important Change**: This fix corrects an issue introduced in Connector/J 5.1.19 that caused connection errors with MySQL 4.1 and earlier servers. A `java.lang.ClassCastException` exception occurred during connection initialization when `com.mysql.jdbc.ConnectionImpl.buildCollationMapping()` interpreted the output of the `SHOW COLLATION` statement. (Bug #13958793)

- A `java.lang.StringIndexOutOfBoundsException` exception could occur when manipulating date/time values with fractional seconds. (Bug #13960556)

- A `MySQLSyntaxErrorException` could occur when calling certain methods while connected to a MySQL 5.6.5 or higher server. Affected methods included `StatementImpl.execute()` and `PreparedStatement.execute()`. (Bug #13955027)

- The savepoint identifier generated by the `java.sql.Connection#.setSavepoint()` function could be misinterpreted as a floating-point number, for example values such as `123e10` or `123e10foo`. Such values could cause replication errors on slave servers because the values are not quoted in the binary log. The fix ensures that the savepoint identifiers do not begin with digits. (Bug #11763271, Bug #55962)

- If the string `limit` was used in a column name, prepared statements incorrectly treated the statement as if it used a `LIMIT` clause. For example, a prepared statement with `maxrows` set to 0 could incorrectly reuse the value from a previous call to `set-MaxRows()`. This issue applied to both quoted and unquoted column names, and server-side and client-side prepared statements. (Bug #11748492, Bug #36478)

A.1.2. Changes in MySQL Connector/J 5.1.19 (April 2012)

Fixes bugs found since release 5.1.18.

Functionality Added or Changed

- For a UTF-8 table using a collation other than the default (`utf8_general_ci`), the precision of the `ResultSetMetaData` could be different from the precision specified in the `CREATE TABLE` statement. The fix corrects the return value from `getMax-BytesPerChar()`.

  This fix changes the behavior of some connection string parameters. `useDynamicCharsetInfo` no longer has any effect. With the setting `cacheServerConfiguration=true`, the cached settings also include the results of the `SHOW CHARACTER SET` statement. (Bug #13495590, Bug #63456)

- Added support for pluggable authentication. via the `com.mysql.jdbc.AuthenticationPlugin` interface (which extends the standard "extension" interface). Examples are in `com/mysql/jdbc/authentication` and in `test-suite.regression.ConnectionRegressionTest.` This feature introduces three new connection properties:

  - `authenticationPlugins` defines a comma-delimited list of classes that implement `com.mysql.jdbc.AuthenticationPlugin` and are used for authentication unless disabled by the `disabledAu-thenticationPlugins` property.

  - `disabledAuthenticationPlugins` defines a comma-delimited list of classes implementing `com.mysql.jdbc.AuthenticationPlugin` or mechanisms, i.e. `mysql_native_password`. The authentication plugins or mechanisms cannot be used for authentication. Authentication will fail if it requires one of these classes. It is an error to disable the default authentication plugin, either the one named by `defaultAuthenticationPlugin` property or the hardcoded one if `defaultAuthenticationPlugin` propery is not set.

  - `defaultAuthenticationPlugin` defines the name of a class implementing `com.mysql.jdbc.AuthenticationPlugin`, which is used as the default authentication plugin. It is an error to use a

class that is not listed in `authenticationPlugins` and is not one of the built-in plugins. It is an error to set as default a plugin that is disabled by being listed in the `disabledAuthenticationPlugins` property. It is an error to set this value to null or the empty string; there must be at least one valid default authentication plugin specified for the connection, meeting all the constraints listed above.

Bugs Fixed

- **Performance**: An unnecessary call to `bind()` during socket operations could limit scalability on some platforms. (Bug #13875070, Bug #63811)

- `setMaxRows` was not correctly processed during metadata collection for client-side prepared statements, causing the entire result set to be fetched and possibly leading to an out-of-memory error. (Bug #13839395, Bug #64621)

- Underprivileged execution of stored procedures fixed. (Bug #13508993, Bug #61203)

- A combination of failover connections, proxied or prepared statements, and database connection pool could cause a memory leak due to improper implementation of `equals()`. (Bug #13441718, Bug #63284)

- Reduced the memory overhead for server-side prepared statements. Each prepared statement allocated a 4K buffer for converting streams. Now this allocation is skipped when no `set*Stream()` methods have been used.

- The `Connection.changeUser()` method did not check for closed connections, leading to `NullPointerException` errors when this method was called on a closed connection.

A.1.3. Changes in MySQL Connector/J 5.1.18 (04 October 2011)

Fixes bugs found since release 5.1.17.

Functionality Added or Changed

- Added the function `MYSQL_INDEX_TO_MYSQL_CHARSET` to retrieve the server charset name, using an index instead of parsing variables to `CharsetMapping.java`

Bugs Fixed

- The `LRUCache` implementation removed the eldest entry, rather than the least-recently accessed. (Bug #13036537)

- Changed `cacheCallableStatements` to `cacheCallableStmts` in `maxPerformance.properties`, to allow proper caching. (Bug #13036309)

- Added a new `ant` flag, `com.mysql.jdbc.junit.fork`, which controls whether `JUnit` will fork new processes.

  The value `on`: The default, and legacy behavior. Or `off`): Required for Windows, as otherwise `process fork failure` errors will result while running the test suite via `ant` on Windows. (Bug #12784170)

- Not putting a space between `VALUES()` and `ON DUPLICATE KEY UPDATE` causes connector/J to both (A) Rewrite the query, although it includes an `ON UPDATE` statement and (B) To generate the wrong query with multiple `ON DUPLICATE KEY` statements. (Bug #12565726)

- The `loadBalanceBlacklistTimeout` option was not functioning properly. Working connections were not being removed from the blacklist. (Bug #63135)

- Connector/J now guards against the condition where a call to `KILL QUERY` will kill the next query issued by the server, if no query is in process. (Bug #61501)

- The "old" warnings were returned when `Statement.getWarnings()` was called after `Statement.clearWarnings()`. (Bug #61866, Bug #12791594)

- Calling `Statement.cancel()` on a statement that isn't currently executing, will cause a later-executed query on the same con-

nection to be unexpectedly canceled. The driver now guards against this condition, but it is an underlying server issue. The MySQL statement `KILL QUERY` (which is what the driver uses to implement `Statement.cancel()`) is rather non-deterministic, and thus the use of `Statement.cancel()` should be avoided if possible. (Bug #61501)

- A connection could not be established when the URL contained both `sessionVariables` and `characterEncoding`. (Bug #61201, Bug #12649557)

- Reverting changes made to `ConnectionImpl.java`, the `private boolean characterSetNamesMatches` function.

A.1.4. Changes in MySQL Connector/J 5.1.17 (07 July 2011)

Fixes bugs found since release 5.1.16.

Bugs Fixed

- `LIKE` was not optimized in then server when run against `INFORMATION_SCHEMA` tables and no wildcards were used. Databases/tables with `'_'` or `'%'` in their names (escaped or not) are handled by this code path, although slower, since it is rare to find these characters in table names in SQL. If there is a `'_'` or `'%'` in the string, `LIKE` takes care of that; otherwise, `'='` is now used instead. The only exception is the `information_schema` database, which is handled separately. The patch covers both `get-Tables()` and `getColumns()`. (Bug #61332)

- The first call to a stored procedure failed with "No Database Selected". The workaround introduced in `Database-MetaData.getCallStmtParameterTypes` to fix the server bug where `SHOW CREATE PROCEDURE` was not respecting lowercase table names was misbehaving when the connection was not attached to a database and on case-insensitive operating systems. (Bug #61150)

- There was a concurrency bottleneck in Java's character set encoding/decoding when converting bytes to/from `String` values.

  > **Important**
  >
  > No longer use `String.getBytes(...)`, or `new String(byte[]...)`. Use the `StringUtils` method instead.

  (Bug #61105)

A.1.5. Changes in MySQL Connector/J 5.1.16 (Not released)

Fixes bugs found since release 5.1.15.

Functionality Added or Changed

- The `Connection.isServerLocal()` method can now determine if a connection is against a server on the same host.

Bugs Fixed

- When auto-reconnect was used with `cacheServerConfiguration`, errors could occur when the host changed (in an HA setup, for example). (Bug #12325877)

- `DBMD.getTables` and `getColumns` fail with table names that contain a dot (like "`junk_[Sp:e,c/ C-h+a=.r]`"). The workaround is to use `useInformationSchema=True`. (Bug #11782297)

- `ResultSetRow.getTimestampFast` and `getTime` had invalid offsets. (Bug #60313)

- Fixed a bug where Connector/J would kill the matching `ConnectionID`, but on the wrong server. (Bug #54135)

- `wasNull` was not set for a `DATE` field with the value of `0000-00-00` in `getDate()`, although `zeroDateTimeBehavior` is defined as `convertToNull`. (Bug #57808)

- A bypassing of the server protocol bug, where the DB should be null-terminated whether it exists or not. This affects `COM_CHANGE_USER`. (Bug #54425)

- Fixed a timestamp offset bug in `com.mysql.jdbc.ResultSetRow.getTimestampFast()`. (Bug #60313, Bug #11890729)

A.1.6. Changes in MySQL Connector/J 5.1.15 (09 February 2011)

Fixes bugs found since release 5.1.14.

Bugs Fixed

- Optional logging class `com.mysql.jdbc.log.Log4JLogger` was not included in the source/binary package for 5.1.14.

  5.1.15 will ship with an SLF4J logger (which can then be plugged into Log4J). Unfortunately, it is not possible to ship a direct Log4J integration because the GPL is not compatible with Log4J's license. (Bug #59511, Bug #11766408)

- The hard-coded list of reserved words in Connector/J was not updated to reflect the list of reserved words in MySQL Server 5.5. (Bug #59224)

- MySqlProcedure accepted null arguments as parameters, however the JDBC meta data did not indicate that. (Bug #38367, Bug #11749186)

- Using Connector/J to connect from a z/OS machine to a MySQL Server failed when the database name to connect to was included in the connection URL. This was because the name was sent in z/OS default platform encoding, but the MySQL Server expected Latin1.

  It should be noted that when connecting from systems that do not use Latin1 as the default platform encoding, the following connection string options can be useful: `passwordCharacterEncoding=ASCII` and `characterEncoding=ASCII`. (Bug #18086, Bug #11745647)

A.1.7. Changes in MySQL Connector/J 5.1.14 (6th December 2010)

Fixes bugs found since release 5.1.13.

Functionality Added or Changed

- Connector/J's load-balancing functionality only allowed the following events to trigger failover:

  - Transaction commit/rollback

  - CommunicationExceptions

  - Matches to user-defined Exceptions using the loadBalanceSQLStateFailover, loadBalanceSQLExceptionSubclassFailover or loadBalanceExceptionChecker property.

  This meant that connections where auto-commit was enabled were not balanced, except for Exceptions, and this was problematic in the case of distribution of read-only work across slaves in a replication deployment.

  The ability to load-balance while auto-commit is enabled has now been added to Connector/J. This introduces two new properties:

  1. loadBalanceAutoCommitStatementThreshold - defines the number of matching statements which will trigger the driver to (potentially) swap physical server connections. The default value (0) retains the previously-established behavior that connections with auto-commit enabled are never balanced.

  2. loadBalanceAutoCommitStatementRegex - the regular expression against which statements must match. The default value (blank) matches all statements.

  Load-balancing will be done after the statement is executed, before control is returned to the application. If rebalancing fails, the driver will silently swallow the resulting Exception (as the statement itself completed successfully). (Bug #55723)

Bugs Fixed

- Added diagnostic information to `SQLException` messages that are caused by usage of a closed load-balanced connection, to help clarify the root cause of a connection closure. (Bug #56200)

- Connection failover left slave/secondary in read-only mode. Failover attempts between two read-write masters did not properly set `this.currentConn.setReadOnly(false).` (Bug #58706)

- Connector/J mapped both 3-byte and 4-byte UTF8 encodings to the same Java UTF8 encoding.

  To use 3-byte UTF8 with Connector/J set `characterEncoding=utf8` and set `useUnicode=true` in the connection string.

  To use 4-byte UTF8 with Connector/J configure the MySQL server with `character_set_server=utf8mb4`. Connector/J will then use that setting as long as `characterEncoding` has not been set in the connection string. This is equivalent to autodetection of the character set. (Bug #58232)

- The `CallableStatementRegression` test suite failed with a Null Pointer Exception because the `OUT` parameter in the `I___S.PARAMETERS` table had no name, that is `COLUMN_NAME` had the value `NULL`. (Bug #58232)

- Attempting to use JDBC4 functions on `Connection` objects resulted in errors being generated:

```
Exception in thread "main" java.lang.AbstractMethodError:
com.mysql.jdbc.LoadBalancedMySQLConnection.createBlob()Ljava/sql/Blob;
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at
com.mysql.jdbc.LoadBalancingConnectionProxy.invoke(LoadBalancingConnectionProxy.java:476)
    at $Proxy0.createBlob(Unknown Source)
```

  (Bug #56099)

- `DatabaseMetaData.supportsMultipleResultSets()` was hard-coded to return `false`, even though Connector/J supports multiple result sets. (Bug #57380)

- Invoking a stored procedure containing output parameters by its full name, where the procedure was located in another database, generated the following exception:

```
Parameter index of 1 is out of range (1, 0)
```

  (Bug #57022)

- Using the `useOldUTF8Behavior` parameter failed to set the connection character set to `latin1` as required.

  In versions prior to 5.1.3, the handshake was done using `latin1`, and while there was logic in place to explicitly set the character set after the handshake was complete, this was bypassed when `useOldUTF8Behavior` was true. This was not a problem until 5.1.3, when the handshake was modified to use `utf8`, but the logic continued to allow the character set configured during that handshake process to be retained for later use. As a result, `useOldUTF8Behavior` effectively failed. (Bug #57262)

- Leaving Trust/ClientCertStoreType properties unset caused an exception to be thrown when connecting with `useSSL=true`, as no default was used. (Bug #56955)

- When a JDBC client disconnected from a remote server using `Connection.close()`, the TCP connection remained in the `TIME_WAIT` state on the server side, rather than on the client side. (Bug #56979)

- When load-balanced connections swap servers, certain session state was copied from the previously active connection to the newly-selected connection. State synchronized included:

  - Auto-commit state

  - Transaction isolation state

  - Current schema/catalog

  However, the read-only state was not synchronized, which caused problems if a write was attempted on a read-only connection. (Bug #56706)

- When using Connector/J configured for failover (jdbc:mysql://host1,host2,... URLs), the non-primary servers re-balanced when the

transactions on the master were committed or rolled-back. (Bug #56429)

- An unhandled Null Pointer Exception (NPE) was generated in `DatabaseMetaData.java` when calling an incorrectly cased function name where no permission to access `mysql.proc` was available.

  In addition to catching potential NPEs, a guard against calling JDBC functions with `db_name.proc_name` notation was also added. (Bug #56305)

A.1.8. Changes in MySQL Connector/J 5.1.13 (24 June 2010)

Fixes bugs found since release 5.1.12.

Functionality Added or Changed

- Connector/J did not support `utf8mb4` for servers 5.5.2 and newer.

  Connector/J now auto-detects servers configured with `character_set_server=utf8mb4` or treats the Java encoding `utf-8` passed using `characterEncoding=...` as `utf8mb4` in the `SET NAMES=` calls it makes when establishing the connection. (Bug #54175)

Bugs Fixed

- When the `allowMultiQueries` connection string option was set to `true`, a call to `Statement.executeBatch()` scanned the query for escape codes, even though `setEscapeProcessing(false)` had been called previously. (Bug #51704)

- The method `unSafeStatementInterceptors()` contained an erroneous line of code, which resulted in the interceptor being called, but the result being thrown away. (Bug #53041)

- There was a performance regression of roughly 25% between r906 and r907, which appeared to be caused by pushing the Proxy down to the I/O layer. (Bug #52534)

- In the file `DatabaseMetadata.java`, the function `private void getCallStmtParameterTypes` failed if the parameter was defined over more than one line by using the '\n' character. (Bug #52167)

- The catalog parameter, `PARAM_CAT`, was not correctly processed when calling for metadata with `getMetaData()` on stored procedures. This was because `PARAM_CAT` was hardcoded in the code to `NULL`. In the case where `nullcatalogmeanscurrent` was `true`, which is its default value, a crash did not occur, but the metadata returned was for the stored procedures from the catalog currently attached to. If, however, `nullcatalogmeanscurrent` was set to `false` then a crash resulted.

  Connector/J has been changed so that when `NULL` is passed as `PARAM_CAT` it will not crash when `nullcatalogmeanscurrent` is `false`, but rather iterate all catalogs in search of stored procedures. This means that `PARAM_CAT` is no longer hardcoded to `NULL` (see Bug #51904). (Bug #51912)

- Logic in implementations of `LoadBalancingConnectionProxy` and `LoadBalanceStrategy` behaved differently as to which `SQLException`s trigger failover to a new host. The former looked at the first two characters of the SQLState:

```
if (sqlState.startsWith("08"))
...
```

  The latter used a different test:

```
if (sqlEx instanceof CommunicationsException
  || "08S01".equals(sqlEx.getSQLState())) {
...
```

  This meant it was possible for a new `Connection` object to throw an `Exception` when the first selected host was unavailable. This happened because `MySqlIO.createNewIO()` could throw an `SQLException` with a `SQLState` of "08001", which did not trigger the "try another host" logic in the `LoadBalanceStrategy` implementations, so an `Exception` was thrown after having only attempted connecting to a single host. (Bug #52231)

- A load balanced `Connection` object with multiple open underlying physical connections rebalanced on `commit()`, `roll-`

back(), or on a communication exception, without validating the existing connection. This caused a problem when there was no pinging of the physical connections, using queries starting with "/* ping */", to ensure they remained active. This meant that calls to Connection.commit() could throw a SQLException. This did not occur when the transaction was actually committed; it occurred when the new connection was chosen and the driver attempted to set the auto-commit or transaction isolation state on the newly chosen physical connection. (Bug #51783)

- The rollback() method could fail to rethrow a SQLException if the server became unavailable during a rollback. The errant code only rethrew when ignoreNonTxTables was true and the exception did not have the error code 1196, SQLError.ER_WARNING_NOT_COMPLETE_ROLLBACK. (Bug #51776)

- Objects created by ConnectionImpl, such as prepared statements, hold a reference to the ConnectionImpl that created them. However, when the load balancer picked a new connection, it did not update the reference contained in, for example, the PreparedStatement. This resulted in inserts and updates being directed to invalid connections, while commits were directed to the new connection. This resulted in silent data loss. (Bug #51643)

- When a StatementInterceptor was used and an alternate ResultSet was returned from preProcess(), the original statement was still executed. (Bug #51666)

- jdbc:mysql:loadbalance:// would connect to the same host, even though loadBalanceStrategy was set to a value of random, and multiple hosts were specified. (Bug #51266)

- An unexpected exception when trying to register OUT parameters in CallableStatement.

  Sometimes Connector/J was not able to register OUT parameters for CallableStatements. (Bug #43576)

A.1.9. Changes in MySQL Connector/J 5.1.12 (18 February 2010)

Fixes bugs found since release 5.1.11.

Bugs Fixed

- The catalog parameter was ignored in the DatabaseMetaData.getProcedure() method. It returned all procedures in all databases. (Bug #51022)

- A call to DatabaseMetaData.getDriverVersion() returned the revision as mysql-connector-java-5.1.11 (Revision: ${svn.Revision}). The variable ${svn.Revision} was not replaced by the SVN revision number. (Bug #50288)

A.1.10. Changes in MySQL Connector/J 5.1.11 (21 January 2010)

Fixes bugs found since release 5.1.10.

Functionality Added or Changed

- Replication connections, those with URLs that start with jdbc:mysql:replication, now use a jdbc:mysql:loadbalance connection for the slave pool. This means that it is possible to set load balancing properties such as loadBalanceBlacklistTimeout and loadBalanceStrategy to choose a mechanism for balancing the load, and failover or fault tolerance strategy for the slave pool. (Bug #49537)

Bugs Fixed

- NullPointerException sometimes occurred in invalidateCurrentConnection() for load-balanced connections. (Bug #50288)

- For pooled connections, Connector/J did not process the session variable time_zone when set using the URL, resulting in incorrect timestamp values being stored. (Bug #49700)

- The deleteRow method caused a full table scan, when using an updatable cursor and a multibyte character set. (Bug #49745)

- The `ExceptionInterceptor` class did not provide a `Connection` context. (Bug #49607)

- Ping left closed connections in the liveConnections map, causing subsequent Exceptions when that connection was used. (Bug #48605)

- Using `MysqlConnectionPoolDataSource` with a load-balanced URL generated exceptions of type `ClassCastException`:

```
ClassCastException in MysqlConnectionPoolDataSource
Caused by: java.lang.ClassCastException: $Proxy0
    at
com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource.getPooledConnection(MysqlConne
ctionPoolDataSource.java:80)
```

```
java.lang.ClassCastException: $Proxy2
 at com.mysql.jdbc.jdbc2.optional.StatementWrapper.executeQuery(StatementWrapper.java:744)
```

  (Bug #48486)

- The implementation for load-balanced `Connection` used a proxy, which delegated method calls, including `equals()` and `hashCode()`, to underlying `Connection` objects. This meant that successive calls to `hashCode()` on the same object potentially returned different values, if the proxy state had changed such that it was utilizing a different underlying connection. (Bug #48442)

- The batch rewrite functionality attempted to identify the start of the `VALUES` list by looking for "VALUES " (with trailing space). However, valid MySQL syntax permits `VALUES` to be followed by whitespace or an opening parenthesis:

```
INSERT INTO tbl VALUES
(1);

INSERT INTO tbl VALUES(1);
```

  Queries written with the above formats did not therefore gain the performance benefits of the batch rewrite. (Bug #48172)

- A PermGen memory leaked was caused by the Connector/J statement cancellation timer (`java.util.Timer`). When the application was unloaded the cancellation timer did not terminate, preventing the ClassLoader from being garbage collected. (Bug #36565)

- With the connection string option `noDatetimeStringSync` set to `true`, and server-side prepared statements enabled, the following exception was generated if an attempt was made to obtain, using `ResultSet.getString()`, a datetime value containing all zero components:

```
java.sql.SQLException: Value '0000-00-00' can not be represented as java.sql.Date
```

  (Bug #32525)

A.1.11. Changes in MySQL Connector/J 5.1.10 (23 September 2009)

Fixes bugs found since release 5.1.9.

Bugs Fixed

- The `DriverManager.getConnection()` method ignored a non-standard port if it was specified in the JDBC connection string. Connector/J always used the standard port 3306 for connection creation. For example, if the string was `jdbc:mysql://localhost:6777`, Connector/J would attempt to connect to port 3306, rather than 6777. (Bug #47494)

A.1.12. Changes in MySQL Connector/J 5.1.9 (21 September 2009)

Bugs Fixed

- When Connector/J encountered an error condition that caused it to create a `CommunicationsException`, it tried to build a friendly error message that helped diagnose what was wrong. However, if there had been no network packets received from the

server, the error message contained the following incorrect text:

```
The last packet successfully received from the server was 1,249,932,468,916 milliseconds
ago.  The last packet sent successfully to the server was 0 milliseconds ago.
```

(Bug #46637)

- In the class `com.mysql.jdbc.jdbc2.optional.SuspendableXAConnection`, which is used when `pinGlobalTx-ToPhysicalConnection=true`, there is a static map (XIDS_TO_PHYSICAL_CONNECTIONS) that tracks the Xid with the XAConnection, however this map was not populated. The effect was that the `SuspendableXAConnection` was never pinned to the real XA connection. Instead it created new connections on calls to `start`, `end`, `resume`, and `prepare`. (Bug #46925)

- When using the ON DUPLICATE KEY UPDATE functionality together with the rewriteBatchedStatements option set to true, an exception was generated when trying to execute the prepared statement:

```
INSERT INTO config_table (modified,id_) VALUES (?,?) ON DUPLICATE KEY UPDATE modified=?
```

The exception generated was:

```
java.sql.SQLException: Parameter index out of range (3 > number of parameters, which is
2).
  at com.sag.etl.job.processors.JdbcInsertProcessor.flush(JdbcInsertProcessor.java:135)
......
Caused by: java.sql.SQLException: Parameter index out of range (3 > number of parameters,
which is 2).
  at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1055)
  at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)
  at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:926)
  at com.mysql.jdbc.PreparedStatement.checkBounds(PreparedStatement.java:3657)
  at com.mysql.jdbc.PreparedStatement.setInternal(PreparedStatement.java:3641)
  at
com.mysql.jdbc.PreparedStatement.setBytesNoEscapeNoQuotes(PreparedStatement.java:3391)
  at
com.mysql.jdbc.PreparedStatement.setOneBatchedParameterSet(PreparedStatement.java:4203)
  at com.mysql.jdbc.PreparedStatement.executeBatchedInserts(PreparedStatement.java:1759)
  at com.mysql.jdbc.PreparedStatement.executeBatch(PreparedStatement.java:1441)
  at com.sag.etl.job.processors.JdbcInsertProcessor.flush(JdbcInsertProcessor.java:131)
  ... 16 more
```

(Bug #46788)

- Accessing result set columns by name after the result set had been closed resulted in a NullPointerException instead of a SQLException. (Bug #41484)

- The `getSuperTypes` method returned a result set with incorrect names for the first two columns. The name of the first column in the result set was expected to be `TYPE_CAT` and that of the second column `TYPE_SCHEM`. The method however returned the names as `TABLE_CAT` and `TABLE_SCHEM` for first and second column respectively. (Bug #44508)

- Calling `ResultSet.deleteRow()` on a table with a primary key of type `BINARY(8)` silently failed to delete the row, but only in some repeatable cases. The generated `DELETE` statement generated corrupted part of the primary key data. Specifically, one of the bytes was changed from 0x90 to 0x9D, although the corruption appeared to be different depending on whether the application was run on Windows or Linux. (Bug #43759)

- SQLException for data truncation error gave the error code as 0 instead of 1265. (Bug #44324)

- `QueryTimeout` did not work for batch statements waiting on a locked table.

  When a batch statement was issued to the server and was forced to wait because of a locked table, Connector/J only terminated the first statement in the batch when the timeout was exceeded, leaving the rest hanging. (Bug #34555)

- The `parseURL` method in class `com.mysql.jdbc.Driver` did not work as expected. When given a URL such as "jdbc:mysql://www.mysql.com:12345/my_database" to parse, the property `PORT_PROPERTY_KEY` was found to be `null` and the `HOST_PROPERTY_KEY` property was found to be "www.mysql.com:12345".

  > **Note**
  >
  > Connector/J has been fixed so that it will now always fill in the `PORT` property (using 3306 if not specified), and the `HOST` property (using `localhost` if not specified) when `parseURL()` is called. The driver also parses a list of hosts into `HOST.n` and `PORT.n` properties as well as adding a property `NUM_HOSTS` for the number of hosts it has found. If a list

of hosts is passed to the driver, HOST and PORT will be set to the values given by HOST.1 and PORT.1 respectively. This change has centralized and cleaned up a large section of code used to generate lists of hosts, both for load-balanced and fault tolerant connections and their tests.

(Bug #32216)

• Attempting to delete rows using ResultSet.deleteRow() did not delete rows correctly. (Bug #27431)

• The setDate method silently ignored the Calendar parameter. The code was implemented as follows:

```
public void setDate(int parameterIndex, java.sql.Date x, Calendar cal) throws SQLException {
    setDate(parameterIndex, x);
}
```

From reviewing the code it was apparent that the Calendar parameter cal was ignored. (Bug #23584)

A.1.13. Changes in MySQL Connector/J 5.1.8 (16 July 2009)

Bugs Fixed

• Calling Connection.serverPreparedStatement() variants that do not take result set type or concurrency arguments returned statements that produced result sets with incorrect defaults, namely TYPE_SCROLL_SENSITIVE. (Bug #45171)

• The reported milliseconds since the last server packets were received/sent was incorrect by a factor of 1000. For example, the following method call:

```
SQLError.createLinkFailureMessageBasedOnHeuristics(
(ConnectionImpl) this.conn,
System.currentTimeMillis() - 1000,
System.currentTimeMillis() - 2000,
e,
false);
```

returned the following string:

```
The last packet successfully received from the server
was 2 milliseconds ago. The last packet sent successfully to the
server was 1 milliseconds ago.
```

(Bug #45419)

• The method Statement.getGeneratedKeys() did not return values for UNSIGNED BIGINTS with values greater than Long.MAX_VALUE.

Unfortunately, because the server does not tell clients what TYPE the auto increment value is, the driver cannot consistently return BigIntegers for the result set returned from getGeneratedKeys(), it will only return them if the value is greater than Long.MAX_VALUE. If your application needs this consistency, it will need to check the class of the return value from .getObject() on the ResultSet returned by Statement.getGeneratedKeys() and if it is not a BigInteger, create one based on the java.lang.Long that is returned. (Bug #43196)

• A statement interceptor received the incorrect parameters when used with a batched statement. (Bug #39426)

• When using Connector/J 5.1.7 to connect to MySQL Server 4.1.18 the following error message was generated:

```
Thu Dec 11 17:38:21 PST 2008 WARN: Invalid value {1} for server variable named {0},
falling back to sane default of {2}
```

This occurred with MySQL Server version that did not support auto_increment_increment. The error message should not have been generated. (Bug #41416)

• The RETURN_GENERATED_KEYS flag was being ignored. For example, in the following code the RETURN_GENERATED_KEYS flag was ignored:

```
PreparedStatement ps = connection.prepareStatement("INSERT INTO table
```

```
  values(?,?)",PreparedStatement.RETURN_GENERATED_KEYS);
```

(Bug #41448)

- Using Connector/J 5.1.6 the method `ResultSet.getObject` returned a `BYTE[]` for following:

```
SELECT TRIM(rowid) FROM tbl
```

Where `rowid` had a type of `INT(11) PRIMARY KEY AUTO_INCREMENT`.

The expected return type was one of `CHAR`, `VARCHAR`, `CLOB`, however, a `BYTE[]` was returned.

Further, adding `functionsNeverReturnBlobs=true` to the connection string did not have any effect on the return type. (Bug #38387)

- Connector/J generated an unhandled `StringIndexOutOfBoundsException`:

```
java.lang.StringIndexOutOfBoundsException: String index out of range: -1
at java.lang.String.substring(String.java:1938)
at com.mysql.jdbc.EscapeProcessor.processTimeToken(EscapeProcessor.java:353)
at com.mysql.jdbc.EscapeProcessor.escapeSQL(EscapeProcessor.java:257)
at com.mysql.jdbc.StatementImpl.executeUpdate(StatementImpl.java:1546)
at com.mysql.jdbc.StatementImpl.executeUpdate(StatementImpl.java:1524)
```

(Bug #42253)

- When using `rewriteBatchedStatements=true` with:

```
INSERT INTO table_name_values (...) VALUES (...)
```

Query rewriting failed because "values" at the end of the table name was mistaken for the reserved keyword. The error generated was as follows:

```
testBug40439(testsuite.simple.TestBug40439)java.sql.BatchUpdateException: You have an
error in your SQL syntax; check the manual that corresponds to your MySQL server version
for the right syntax to use near 'values (2,'toto',2),(id,data, ordr) values
(3,'toto',3),(id,data, ordr) values (' at line 1
at com.mysql.jdbc.PreparedStatement.executeBatchedInserts(PreparedStatement.java:1495)
at com.mysql.jdbc.PreparedStatement.executeBatch(PreparedStatement.java:1097)
at testsuite.simple.TestBug40439.testBug40439(TestBug40439.java:42)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at testsuite.simple.TestBug40439.main(TestBug40439.java:57)
```

(Bug #40439)

- Error message strings contained variable values that were not expanded. For example:

```
Mon Nov 17 11:43:18 JST 2008 WARN: Invalid value {1} for server variable named {0},
falling back to sane default of {2}
```

(Bug #40772)

- When accessing a result set column by name using `ResultSetImpl.findColumn()` an exception was generated:

```
java.lang.NullPointerException
at com.mysql.jdbc.ResultSetImpl.findColumn(ResultSetImpl.java:1103)
at com.mysql.jdbc.ResultSetImpl.getShort(ResultSetImpl.java:5415)
at org.apache.commons.dbcp.DelegatingResultSet.getShort(DelegatingResultSet.java:219)
at com.zimbra.cs.db.DbVolume.constructVolume(DbVolume.java:297)
at com.zimbra.cs.db.DbVolume.get(DbVolume.java:197)
at com.zimbra.cs.db.DbVolume.create(DbVolume.java:95)
at com.zimbra.cs.store.Volume.create(Volume.java:227)
at com.zimbra.cs.store.Volume.create(Volume.java:189)
at com.zimbra.cs.service.admin.CreateVolume.handle(CreateVolume.java:48)
at com.zimbra.soap.SoapEngine.dispatchRequest(SoapEngine.java:428)
at com.zimbra.soap.SoapEngine.dispatch(SoapEngine.java:285)
```

(Bug #41484)

- If there was an apostrophe in a comment in a statement that was being sent through Connector/J, the apostrophe was still recognized as a quote and put the state machine in `EscapeTokenizer` into the `inQuotes` state. This led to further parse errors.

  For example, consider the following statement:

```
String sql = "-- Customer's zip code will be fixed\n" +
  "update address set zip_code = 99999\n" +
  "where not regexp '^[0-9]{5}([[.-.]])?([0-9]{4})?$'";
```

  When passed through Connector/J, the `EscapeTokenizer` did not recognize that the first apostrophe was in a comment and thus set `inQuotes` to true. When that happened, the quote count was incorrect and thus the regular expression did not appear to be in quotation marks. With the parser not detecting that the regular expression was in quotation marks, the curly braces were recognized as escape sequences and were removed from the regular expression, breaking it. The server thus received SQL such as:

```
-- Customer's zip code will be fixed
update address set zip_code = '99999'
where not regexp '^[0-9]([[.-.]])?([0-9])?$'
```

  (Bug #41566)

- Using `useInformationSchema` with `DatabaseMetaData.getExportedKeys()` generated the following exception:

```
com.mysql.jdbc.exceptions.MySQLIntegrityConstraintViolationException: Column
'REFERENCED_TABLE_NAME' in where clause is ambiguous
...
at com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java:1772)
at com.mysql.jdbc.PreparedStatement.executeQuery(PreparedStatement.java:1923)
at
com.mysql.jdbc.DatabaseMetaDataUsingInfoSchema.executeMetadataQuery(
DatabaseMetaDataUsingInfoSchema.java:50)
at
com.mysql.jdbc.DatabaseMetaDataUsingInfoSchema.getExportedKeys(
DatabaseMetaDataUsingInfoSchema.java:603)
```

  (Bug #43714)

- The `SQLError.createLinkFailureMessageBasedOnHeuristics()` method created a message text for communication link failures. When certain conditions were met, this message included both "last packet sent" and "last packet received" information, but when those conditions were not met, only "last packet sent" information was provided.

  Information about when the last packet was successfully received should be provided in all cases. (Bug #44587)

- When `DatabaseMetaData.getProcedureColumns()` was called, the value for `LENGTH` was always returned as 65535, regardless of the column type (fixed or variable) or the actual length of the column.

  However, if you obtained the `PRECISION` value, this was correct for both fixed and variable length columns. (Bug #41269)

- The `DEFERRABILITY` column in database metadata result sets was expected to be of type `SHORT`. However, Connector/J returned it as `INTEGER`.

  This affected the following methods: `getImportedKeys()`, `getExportedKeys()`, `getCrossReference()`. (Bug #44867)

- The result set returned by `getIndexInfo()` did not have the format defined in the JDBC API specifications. The fourth column, `DATA_TYPE`, of the result set should be of type `BOOLEAN`. Connector/J however returns `CHAR`. (Bug #44869)

- The result set returned by `getTypeInfo()` did not have the format defined in the JDBC API specifications. The second column, `DATA_TYPE`, of the result set should be of type `INTEGER`. Connector/J however returns `SMALLINT`. (Bug #44868)

- The result set returned by `getColumns()` did not have the format defined in the JDBC API specifications. The fifth column, `DATA_TYPE`, of the result set should be of type `INTEGER`. Connector/J however returns `SMALLINT`. (Bug #44865)

- The result set returned by `getVersionColumns()` did not have the format defined in the JDBC API specifications. The third column, `DATA_TYPE`, of the result set should be of type `INTEGER`. Connector/J however returns `SMALLINT`. (Bug #44863)

- The result set returned by `getBestRowIdentifier()` did not have the format defined in the JDBC API specifications. The third column, `DATA_TYPE`, of the result set should be of type `INTEGER`. Connector/J however returns `SMALLINT`. (Bug #44862)

- Connector/J contains logic to generate a message text specifically for streaming result sets when there are `CommunicationsException` exceptions generated. However, this code was never reached.

  In the `CommunicationsException` code:

```
private boolean streamingResultSetInPlay = false;

public CommunicationsException(ConnectionImpl conn, long lastPacketSentTimeMs,
long lastPacketReceivedTimeMs, Exception underlyingException) {

this.exceptionMessage = SQLError.createLinkFailureMessageBasedOnHeuristics(conn,
    lastPacketSentTimeMs, lastPacketReceivedTimeMs, underlyingException,
    this.streamingResultSetInPlay);
```

  `streamingResultSetInPlay` was always false, which in the following code in `SQLError.createLinkFailureMessageBasedOnHeuristics()` never being executed:

```
if (streamingResultSetInPlay) {
    exceptionMessageBuf.append(
    Messages.getString("CommunicationsException.ClientWasStreaming")); //$NON-NLS-1$
} else {
...
```

  (Bug #44588)

- SQL injection was possible when using a string containing U+00A5 in a client-side prepared statement, and the character set being used was SJIS/Windows-31J. (Bug #41730)

- `Statement.getGeneratedKeys()` retained result set instances until the statement was closed. This caused memory leaks for long-lived statements, or statements used in tight loops. (Bug #44056)

- MySQL Connector/J 5.1.7 was slower than previous versions when the `rewriteBatchedStatements` option was set to `true`.

  > **Note**
  >
  > The performance regression in `indexOfIgnoreCaseRespectMarker()` has been fixed. It has also been made possible for the driver to rewrite `INSERT` statements with `ON DUPLICATE KEY UPDATE` clauses in them, as long as the `UPDATE` clause contains no reference to `LAST_INSERT_ID()`, as that would cause the driver to return bogus values for `getGeneratedKeys()` invocations. This has resulted in improved performance over version 5.1.7.

  (Bug #41532)

- `LoadBalancingConnectionProxy.doPing()` did not have blacklist awareness.

  `LoadBalancingConnectionProxy` implemented `doPing()` to ping all underlying connections, but it threw any exceptions it encountered during this process.

  With the global blacklist enabled, it catches these exceptions, adds the host to the global blacklist, and only throws an exception if all hosts are down. (Bug #43421)

- When connecting with `traceProtocol=true`, no trace data was generated for the server greeting or login request. (Bug #43070)

- When the MySQL Server was upgraded from 4.0 to 5.0, the Connector/J application then failed to connect to the server. This was because authentication failed when the application ran from EBCDIC platforms such as z/OS. (Bug #43071)

- A `ConcurrentModificationException` was generated in `LoadBalancingConnectionProxy`:

```
java.util.ConcurrentModificationException
 at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
 at java.util.HashMap$KeyIterator.next(Unknown Source)
 at
com.mysql.jdbc.LoadBalancingConnectionProxy.getGlobalBlacklist(LoadBalancingConnectionProxy.java:520)
 at com.mysql.jdbc.RandomBalanceStrategy.pickConnection(RandomBalanceStrategy.java:55)
 at
com.mysql.jdbc.LoadBalancingConnectionProxy.pickNewConnection(LoadBalancingConnectionProxy.java:414)
 at
com.mysql.jdbc.LoadBalancingConnectionProxy.invoke(LoadBalancingConnectionProxy.java:390)
```

(Bug #42055)

- `PreparedStatement.addBatch()` did not check for all parameters being set, which led to inconsistent behavior in `executeBatch()`, especially when rewriting batched statements into multi-value `INSERT`s. (Bug #41161)

A.1.14. Changes in MySQL Connector/J 5.1.7 (21 October 2008)

Functionality Added or Changed

- When statements include `ON DUPLICATE UPDATE`, and `rewriteBatchedStatements` is set to true, batched statements are not rewritten into the form `INSERT INTO table VALUES (), (), ()`, instead the statements are executed sequentially.

Bugs Fixed

- When using `trustCertificateKeyStoreUrl` or `clientCertificateKeyStoreUrl`, an `IllegalStateException` was caused by an uninitialized `TrustManagerFactoryImpl` object. (Bug #11748637, Bug #36948, Bug #38192)

- `Statement.getGeneratedKeys()` returned two keys when using `ON DUPLICATE KEY UPDATE` and the row was updated, not inserted. (Bug #42309)

- When configuring the Java Replication Driver the last slave specified was never used. (Bug #39611)

- When using the replication driver with `autoReconnect=true`, Connector/J checks in `PreparedStatement.execute` (also called by `CallableStatement.execute`) to determine if the first character of the statement is an "S", in an attempt to block all statements that are not read-only-safe, for example non-`SELECT` statements. However, this also blocked `CALL`s to stored procedures, even if the stored procedures were defined as `SQL READ DATA` or `NO SQL`. (Bug #40031)

- When the `LoadBalancingConnectionProxy` handles a `SQLException` with SQL state starting with "08", it calls `invalidateCurrentConnection`, which in turn removes that `Connection` from `liveConnections` and the `connectionsToHostsMap`, but it did not add the host to the new global blacklist, if the global blacklist was enabled.

  There was also the possibility of a `NullPointerException` when trying to update stats, where `connectionsToHostsMap.get(this.currentConn)` was called:

  ```
  int hostIndex = ((Integer) this.hostsToListIndexMap.get(this.connectionsToHostsMap.get(this.currentConn))).intValue();
  ```

  This could happen if a client tried to issue a rollback after catching a `SQLException` caused by a connection failure. (Bug #39784)

- When using the random load balancing strategy and starting with two servers that were both unavailable, an `IndexOutOfBoundsException` was generated when removing a server from the `whiteList`. (Bug #38782)

- With large result sets `ResultSet.findColumn` became a performance bottleneck. (Bug #39962)

- Connector/J ignored the value of the MySQL Server variable `auto_increment_increment`. (Bug #39956)

- When an `INSERT ON DUPLICATE KEY UPDATE` was performed, and the key already existed, the `affected-rows` value was returned as 1 instead of 0. (Bug #39352)

- Connector/J failed to parse `TIMESTAMP` strings for nanos correctly. (Bug #39911)

- Connector/J threw the following exception when using a read-only connection:

  ```
  java.sql.SQLException: Connection is read-only. Queries leading to data
  modification are not allowed.
  ```

  (Bug #38747)

- Connector/J was unable to connect when using a non-`latin1` password. (Bug #37570)

- The `useOldAliasMetadataBehavior` connection property was ignored. (Bug #35753)

- When `getGeneratedKeys()` was called on a statement that had not been created with `RETURN_GENERATED_KEYS`, no exception was thrown, and batched executions then returned erroneous values. (Bug #34185)

- The `loadBalance bestResponseTime` blacklists did not have a global state. (Bug #33861)

- Incorrect result is returned from `isAfterLast()` in streaming `ResultSet` when using `setFetchS-ize(Integer.MIN_VALUE)`. (Bug #35170)

A.1.15. Changes in MySQL Connector/J 5.1.6 (07 March 2008)

Functionality Added or Changed

- Multiple result sets were not supported when using streaming mode to return data. Both normal statements and the result sets from stored procedures now return multiple results sets, with the exception of result sets using registered `OUTPUT` parameters. (Bug #33678)

- Add the `verifyServerCertificate` property. If set to "false" the driver will not verify the server's certificate when `useSSL` is set to "true"

  When using this feature, the keystore parameters should be specified by the `clientCertificateKeyStore*` properties, rather than system properties, as the JSSE doesn't it straightforward to have a nonverifying trust store and the "default" key store.

- The profiler event handling has been made extensible using the `profilerEventHandler` connection property.

- XAConnections and datasources have been updated to the JDBC-4.0 standard.

Bugs Fixed

- Prepared statements from pooled connections caused a `NullPointerException` when `closed()` under JDBC-4.0. (Bug #35489)

- When `useServerPrepStmts=true` and slow query logging is enabled, the connector throws a `NullPointerException` when it encounters a slow query. (Bug #35666)

- The JDBC driver uses a different method for evaluating column names in `resultsetmetadata.getColumnName()` and when looking for a column in `resultset.getObject(columnName)`. This causes Hibernate to fail in queries where the two methods yield different results, for example in queries that use alias names:

```
SELECT column AS aliasName from table
```

  (Bug #35150)

- `DatabaseMetaData.getColumns()` returns incorrect `COLUMN_SIZE` value for `SET` column. (Bug #36830)

- When trying to read `Time` values like "00:00:00" with `ResultSet.getTime(int)` an exception is thrown. (Bug #36051)

- When using the keyword "loadbalance" in the connection string and trying to perform load balancing between two databases, the driver appears to hang. (Bug #35660)

- JDBC data type getter method was changed to accept only column name, whereas previously it accepted column label. (Bug #35610)

- In calling a stored function returning a `bigint`, an exception is encountered beginning:

```
java.sql.SQLException: java.lang.NumberFormatException: For input string:
```

  followed by the text of the stored function starting after the argument list. (Bug #35199)

- JDBC connection URL parameters is ignored when using `MysqlConnectionPoolDataSource`. (Bug #35810)

- Retrieving the server version information for an active connection could return invalid information if the default character encoding on the host was not ASCII compatible. (Bug #31192)

- `MysqlConnectionPoolDataSource` does not support `ReplicationConnection`. Notice that we implemented `com.mysql.jdbc.Connection` for `ReplicationConnection`, however, only accessors from ConnectionProperties are implemented (not the mutators), and they return values from the currently active connection. All other methods from `com.mysql.jdbc.Connection` are implemented, and operate on the currently active connection, with the exception of `re-setServerState()` and `changeUser()`. (Bug #34937)

- When calling `isValid()` on an active connection, if the timeout is nonzero then the `Connection` is invalidated even if the `Connection` is valid. (Bug #34703)

- When retrieving the column type name of a geometry field, the driver would return `UNKNOWN` instead of `GEOMETRY`. (Bug #34194)

- The internal class `ResultSetInternalMethods` referenced the nonpublic class `com.mysql.jdbc.CachedResultSetMetaData`. (Bug #33823)

- `ResultSet.getTimestamp()` would throw a `NullPointerException` instead of a `SQLException` when called on an empty `ResultSet`. (Bug #33162)

- `ResultSet.getTimestamp()` returns incorrect values for month/day of `TIMESTAMP`s when using server-side prepared statements (not enabled by default). (Bug #34913)

- `RowDataStatic` doesn't always set the metadata in `ResultSetRow`, which can lead to failures when unpacking `DATE`, `TIME`, `DATETIME` and `TIMESTAMP` types when using absolute, relative, and previous result set navigation methods. (Bug #34762)

- It was not possible to truncate a `BLOB` using `Blog.truncate()` when using 0 as an argument. (Bug #34677)

- Statements with batched values do not return correct values for `getGeneratedKeys()` when `rewriteBatchedState-ments` is set to `true`, and the statement has an `ON DUPLICATE KEY UPDATE` clause. (Bug #34093)

- A `NullPointerException` could be raised when using client-side prepared statements and enabled the prepared statement cache using the `cachePrepStmts`. (Bug #33734)

- When using a cursor fetch for a statement, the internal prepared statement could cause a memory leak until the connection was closed. The internal prepared statement is now deleted when the corresponding result set is closed. (Bug #34518)

- Using server side cursors and cursor fetch, the table metadata information would return the data type name instead of the column name. (Bug #33594)

- `ResultSet` returned by `Statement.getGeneratedKeys()` is not closed automatically when statement that created it is closed. (Bug #30508)

- Load balancing connection using best response time would incorrectly "stick" to hosts that were down when the connection was first created.

  We solve this problem with a black list that is used during the picking of new hosts. If the black list ends up including all configured hosts, the driver will retry for a configurable number of times (the `retriesAllDown` configuration property, with a default of 120 times), sleeping 250ms between attempts to pick a new connection.

  We've also went ahead and made the balancing strategy extensible. To create a new strategy, implement the interface `com.mysql.jdbc.BalanceStrategy` (which also includes our standard "extension" interface), and tell the driver to use it by passing in the class name using the `loadBalanceStrategy` configuration property. (Bug #32877)

- Using `CallableStatement.setNull()` on a stored function would throw an `ArrayIndexOutOfBounds` exception when setting the last parameter to null. (Bug #31823)

- When using a connection from `ConnectionPoolDataSource`, some `Connection.prepareStatement()` methods would return null instead of the prepared statement. (Bug #32101)

- `MysqlValidConnectionChecker` doesn't properly handle connections created using `ReplicationConnection`. (Bug #31790)

- During a Daylight Savings Time (DST) switchover, there was no way to store two timestamp/datetime values , as the hours end up being the same when sent as the literal that MySQL requires.

Note that to get this scenario to work with MySQL (since it doesn't support per-value timezones), you need to configure your server (or session) to be in UTC, and tell the driver not to use the legacy date/time code by setting `useLegacyDatetimeCode` to "false". This will cause the driver to always convert to/from the server and client timezone consistently.

This bug fix also fixes Bug #15604, by adding entirely new date/time handling code that can be switched on by `useLegacyDatetimeCode` being set to "false" as a JDBC configuration property. For Connector/J 5.1.x, the default is "true", in trunk and beyond it will be "false" (that is, the old date/time handling code will be deprecated) (Bug #32577, Bug #15604)

• When unpacking rows directly, we don't hand off error message packets to the internal method which decodes them correctly, so no exception is raised, and the driver than hangs trying to read rows that aren't there. This tends to happen when calling stored procedures, as normal SELECTs won't have an error in this spot in the protocol unless an I/O error occurs. (Bug #32246)

• Further fixes have been made to this bug in the event that a node is nonresponsive. Connector/J will now try a different random node instead of waiting for the node to recover before continuing. (Bug #31053)

• `DatabaseMetadata.getColumns()` doesn't return the correct column names if the connection character isn't UTF-8. A bug in MySQL server compounded the issue, but was fixed within the MySQL 5.0 release cycle. The fix includes changes to all the sections of the code that access the server metadata. (Bug #20491)

• Fixed `ResultSetMetadata.getColumnName()` for result sets returned from `Statement.getGeneratedKeys()` - it was returning null instead of "GENERATED_KEY" as in 5.0.x.

A.1.16. Changes in MySQL Connector/J 5.1.5 (09 October 2007)

New Features, Compared to the 5.0 Series of Connector/J

• JDBC-4.0 standardized unwrapping to interfaces that include vendor extensions.

• Support for JDBC-4.0 XML processing using JAXP interfaces to DOM, SAX and StAX.

• JDBC-4.0 support for setting per-connection client information (which can be viewed in the comments section of a query using `SHOW PROCESSLIST` on a MySQL server, or can be extended to support custom persistence of the information using a public interface).

• Support for JDBC-4.0 `NCHAR`, `NVARCHAR` and `NCLOB` types.

Functionality Added or Changed

• Added `autoSlowLog` configuration property, overrides `slowQueryThreshold*` properties, driver determines slow queries by those that are slower than 5 * stddev of the mean query time (outside the 96% percentile).

Bugs Fixed

• When a connection is in read-only mode, queries that are wrapped in parentheses were incorrectly identified DML statements. (Bug #28256)

• When calling `setTimestamp` on a prepared statement, the timezone information stored in the calendar object was ignored. This resulted in the incorrect `DATETIME` information being stored. The following example illustrates this:

```
Timestamp t = new Timestamp( cal.getTimeInMillis() );
ps.setTimestamp( N, t, cal );
```

(Bug #15604)

A.1.17. Changes in MySQL Connector/J 5.1.4 (Not Released)

Only released internally.

Version 5.1.4 has no changelog entries.

A.1.18. Changes in MySQL Connector/J 5.1.3 (10 September 2007)

New Features, Compared to the 5.0 Series of Connector/J

- JDBC-4.0 standardized unwrapping to interfaces that include vendor extensions.

- Support for JDBC-4.0 XML processing using JAXP interfaces to DOM, SAX and StAX.

- JDBC-4.0 support for setting per-connection client information (which can be viewed in the comments section of a query using `SHOW PROCESSLIST` on a MySQL server, or can be extended to support custom persistence of the information using a public interface).

- Support for JDBC-4.0 `NCHAR`, `NVARCHAR` and `NCLOB` types.

Functionality Added or Changed

- Connector/J now connects using an initial character set of `utf-8` solely for the purpose of authentication to permit user names or database names in any character set to be used in the JDBC connection URL. (Bug #29853)

- Setting `useBlobToStoreUTF8OutsideBMP` to `true` tells the driver to treat `[MEDIUM/LONG]BLOB` columns as `[LONG]VARCHAR` columns holding text encoded in UTF-8 that has characters outside the BMP (4-byte encodings), which MySQL server can't handle natively.

  Set `utf8OutsideBmpExcludedColumnNamePattern` to a regex so that column names matching the given regex will still be treated as `BLOB`s The regex must follow the patterns used for the `java.util.regex`package. The default is to exclude no columns, and include all columns.

  Set `utf8OutsideBmpIncludedColumnNamePattern` to specify exclusion rules to utf8OutsideBmpExcludedColumnNamePattern". The regex must follow the patterns used for the `java.util.regex` package.

- New methods on com.mysql.jdbc.Statement: `setLocalInfileInputStream()` and `getLocalInfileInputStream()`:

  - `setLocalInfileInputStream()` sets an `InputStream` instance that will be used to send data to the MySQL server for a `LOAD DATA LOCAL INFILE` statement rather than a `FileInputStream` or `URLInputStream` that represents the path given as an argument to the statement.

    This stream will be read to completion upon execution of a `LOAD DATA LOCAL INFILE` statement, and will automatically be closed by the driver, so it needs to be reset before each call to `execute*()` that would cause the MySQL server to request data to fulfill the request for `LOAD DATA LOCAL INFILE`.

    If this value is set to `NULL`, the driver will revert to using a `FileInputStream` or `URLInputStream` as required.

  - `getLocalInfileInputStream()` returns the `InputStream` instance that will be used to send data in response to a `LOAD DATA LOCAL INFILE` statement.

    This method returns `NULL` if no such stream has been set using `setLocalInfileInputStream()`.

- Errors encountered during `Statement`/`PreparedStatement`/`CallableStatement.executeBatch()` when `rewriteBatchStatements` has been set to `true` now return `BatchUpdateExceptions` according to the setting of `continueBatchOnError`.

  If `continueBatchOnError` is set to `true`, the update counts for the "chunk" that were sent as one unit will all be set to `EXECUTE_FAILED`, but the driver will attempt to process the remainder of the batch. You can determine which "chunk" failed by looking at the update counts returned in the `BatchUpdateException`.

  If `continueBatchOnError` is set to "false", the update counts returned will contain all updates up-to and including the failed "chunk", with all counts for the failed "chunk" set to `EXECUTE_FAILED`.

  Since MySQL doesn't return multiple error codes for multiple-statements, or for multi-value `INSERT`/`REPLACE`, it is the application's responsibility to handle determining which item(s) in the "chunk" actually failed.

- `Statement.setQueryTimeout()`s now affect the entire batch for batched statements, rather than the individual statements that make up the batch.

- JDBC-4.0 ease-of-development features including auto-registration with the `DriverManager` through the service provider mechanism, standardized Connection validity checks and categorized `SQLExceptions` based on recoverability/retry-ability and class of the underlying error.

- The driver will automatically adjust the server session variable `net_write_timeout` when it determines its been asked for a "streaming" result, and resets it to the previous value when the result set has been consumed. (The configuration property is named `netTimeoutForStreamingResults`, with a unit of seconds, the value '0' means the driver will not try and adjust this value).

- Added experimental support for statement "interceptors" through the `com.mysql.jdbc.StatementInterceptor` interface, examples are in `com/mysql/jdbc/interceptors`. Implement this interface to be placed "in between" query execution, so that it can be influenced (currently experimental).

- The data (and how it is stored) for `ResultSet` rows are now behind an interface which enables us (in some cases) to allocate less memory per row, in that for "streaming" result sets, we re-use the packet used to read rows, since only one row at a time is ever active.

- The driver now picks appropriate internal row representation (whole row in one buffer, or individual byte[]s for each column value) depending on heuristics, including whether or not the row has `BLOB` or `TEXT` types and the overall row-size. The threshold for row size that will cause the driver to use a buffer rather than individual byte[]s is configured by the configuration property `largeRowSizeThreshold`, which has a default value of 2KB.

- Setting `rewriteBatchedStatements` to `true` now causes CallableStatements with batched arguments to be re-written in the form "CALL (...); CALL (...); ..." to send the batch in as few client/server round trips as possible.

- Added two configuration parameters:

  - `blobsAreStrings`: Should the driver always treat BLOBs as Strings. Added specifically to work around dubious metadata returned by the server for `GROUP BY` clauses. Defaults to false.

  - `functionsNeverReturnBlobs`: Should the driver always treat data from functions returning `BLOBs` as Strings. Added specifically to work around dubious metadata returned by the server for `GROUP BY` clauses. Defaults to false.

Bugs Fixed

- `CallableStatement.executeBatch()` doesn't work when connection property `noAccessToProcedureBodies` has been set to `true`.

  The fix involves changing the behavior of `noAccessToProcedureBodies`,in that the driver will now report all parameters as `IN` parameters but permit callers to call registerOutParameter() on them without throwing an exception. (Bug #28689)

- NPE with null column values when `padCharsWithSpace` is set to true. (Bug #30851)

- `setObject(int, Object, int, int)` delegate in PreparedStatementWrapper delegates to wrong method. (Bug #30892)

- Closing a load-balanced connection would cause a `ClassCastException`. (Bug #29852)

- `DatabaseMetaData.getTypeInfo()` for the types `DECIMAL` and `NUMERIC` will return a precision of 254 for server versions older than 5.0.3, 64 for versions 5.0.3 to 5.0.5 and 65 for versions newer than 5.0.5. (Bug #28972)

- An `ArithmeticException` or `NullPointerException` would be raised when the batch had zero members and `rewriteBatchedStatements=true` when `addBatch()` was never called, or `executeBatch()` was called immediately after `clearBatch()`. (Bug #30550)

- Collation on `VARBINARY` column types would be misidentified. A fix has been added, but this fix only works for MySQL server versions 5.0.25 and newer, since earlier versions didn't consistently return correct metadata for functions, and thus results from subqueries and functions were indistinguishable from each other, leading to type-related bugs. (Bug #30664)

- Connection checker for JBoss didn't use same method parameters using reflection, causing connections to always seem "bad". (Bug #29106)

- Schema objects with identifiers other than the connection character aren't retrieved correctly in `ResultSetMetadata`. (Bug

#27867)

- `Connection.getServerCharacterEncoding()` doesn't work for servers with version >= 4.1. (Bug #27182)

- `DatabaseMetaData.getColumns()` doesn't contain `SCOPE_*` or `IS_AUTOINCREMENT` columns. (Bug #27915)

- The automated SVN revisions in `DBMD.getDriverVersion()`. The SVN revision of the directory is now inserted into the version information during the build. (Bug #21116)

- Specifying a "validation query" in your connection pool that starts with "/* ping */" _exactly_ will cause the driver to instead send a ping to the server and return a fake result set (much lighter weight), and when using a ReplicationConnection or a LoadBalanced-Connection, will send the ping across all active connections.

A.1.19. Changes in MySQL Connector/J 5.1.2 (29 June 2007)

This is a new Beta development release, fixing recently discovered bugs.

Functionality Added or Changed

- Setting the configuration property `rewriteBatchedStatements` to `true` will now cause the driver to rewrite batched prepared statements with more than 3 parameter sets in a batch into multi-statements (separated by ";") if they are not plain (that is, without `SELECT` or `ON DUPLICATE KEY UPDATE` clauses) `INSERT` or `REPLACE` statements.

A.1.20. Changes in MySQL Connector/J 5.1.1 (22 June 2007)

This is a new Alpha development release, adding new features and fixing recently discovered bugs.

Functionality Added or Changed

- **Incompatible Change**: Pulled vendor-extension methods of `Connection` implementation out into an interface to support `java.sql.Wrapper` functionality from `ConnectionPoolDataSource`. The vendor extensions are javadoc'd in the `com.mysql.jdbc.Connection` interface.

  For those looking further into the driver implementation, it is not an API that is used for pluggability of implementations inside our driver (which is why there are still references to `ConnectionImpl` throughout the code).

  We've also added server and client `prepareStatement()` methods that cover all of the variants in the JDBC API.

  `Connection.serverPrepare(String)` has been re-named to `Connection.serverPrepareStatement()` for consistency with `Connection.clientPrepareStatement()`.

- Setting `rewriteBatchedStatements` to `true` now causes `CallableStatements` with batched arguments to be rewritten in the form `CALL (...); CALL (...); ...` to send the batch in as few client/server round trips as possible.

- See the sources (fully javadoc'd) for `com.mysql.jdbc.StatementInterceptor` for more details until we iron out the API and get it documented in the manual.

- Added experimental support for statement "interceptors" through the `com.mysql.jdbc.StatementInterceptor` interface, examples are in `com/mysql/jdbc/interceptors`.

  Implement this interface to be placed "in between" query execution, so that you can influence it. (currently experimental).

  `StatementInterceptors` are "chainable" when configured by the user, the results returned by the "current" interceptor will be passed on to the next on in the chain, from left-to-right order, as specified by the user in the JDBC configuration property `statementInterceptors`.

- Driver now picks appropriate internal row representation (whole row in one buffer, or individual byte[]s for each column value) depending on heuristics, including whether or not the row has `BLOB` or `TEXT` types and the overall row-size. The threshold for row size that will cause the driver to use a buffer rather than individual byte[]s is configured by the configuration property `largeRowSizeThreshold`, which has a default value of 2KB.

- Similar to `Connection`, we pulled out vendor extensions to `Statement` into an interface named `com.mysql.Statement`, and moved the `Statement` class into `com.mysql.StatementImpl`. The two methods (javadoc'd in `com.mysql.Statement` are `enableStreamingResults()`, which already existed, and `disableStreamingResults()` which sets the statement instance back to the fetch size and result set type it had before `enableStreamingResults()` was called.

- The data (and how it is stored) for `ResultSet` rows are now behind an interface which enables us (in some cases) to allocate less memory per row, in that for "streaming" result sets, we re-use the packet used to read rows, since only one row at a time is ever active.

- Externalized the descriptions of connection properties.

- Made it possible to retrieve prepared statement parameter bindings (to be used in `StatementInterceptors`, primarily).

- Row navigation now causes any streams/readers open on the result set to be closed, as in some cases we're reading directly from a shared network packet and it will be overwritten by the "next" row.

A.1.21. Changes in MySQL Connector/J 5.1.0 (11 April 2007)

This is the first public alpha release of the current Connector/J 5.1 development branch, providing an insight to upcoming features. Although some of these are still under development, this release includes the following new features and changes (in comparison to the current Connector/J 5.0 production release):

Important change: Due to a number of issues with the use of server-side prepared statements, Connector/J 5.0.5 has disabled their use by default. The disabling of server-side prepared statements does not affect the operation of the connector in any way.

To enable server-side prepared statements you must add the following configuration property to your connector string:

```
useServerPrepStmts=true
```

The default value of this property is `false` (that is, Connector/J does not use server-side prepared statements).

> **Note**
>
> The disabling of server-side prepared statements does not affect the operation of the connector. However, if you use the `useTimezone=true` connection option and use client-side prepared statements (instead of server-side prepared statements) you should also set `useSSPSCompatibleTimezoneShift=true`.

Functionality Added or Changed

- Added support for JDBC-4.0's `Wrapper` interface.

- `com.mysql.jdbc.java6.rtjar`: Full path to your Java-6 `rt.jar` file

- Added support for JDBC-4.0's client information. The backend storage of information provided using `Connection.setClientInfo()` and retrieved by `Connection.getClientInfo()` is pluggable by any class that implements the `com.mysql.jdbc.JDBC4ClientInfoProvider` interface and has a no-args constructor.

  The implementation used by the driver is configured using the `clientInfoProvider` configuration property (with a default of value of `com.mysql.jdbc.JDBC4CommentClientInfoProvider`, an implementation which lists the client information as a comment prepended to every query sent to the server).

  This functionality is only available when using Java-6 or newer.

- New feature—driver will automatically adjust session variable `net_write_timeout` when it determines it has been asked for a "streaming" result, and resets it to the previous value when the result set has been consumed. (configuration property is named `netTimeoutForStreamingResults` value and has a unit of seconds, the value `0` means the driver will not try and adjust this value).

- Re-worked Ant buildfile to build JDBC-4.0 classes separately, as well as support building under Eclipse (since Eclipse can't mix/match JDKs).

  To build, you must set `JAVA_HOME` to J2SDK-1.4.2 or Java-5, and set the following properties on your Ant command line:

- • `com.mysql.jdbc.java6.javac`: Full path to your Java-6 `javac` executable

  - • `com.mysql.jdbc.java6.rtjar`: Full path to your Java-6 `rt.jar` file

- • Added support for JDBC-4.0's SQLXML interfaces.

- • `com.mysql.jdbc.java6.javac`: Full path to your Java-6 `javac` executable

- • Added support for JDBC-4.0's `NCLOB`, and `NCHAR`/`NVARCHAR` types.

- • Added support for JDBC-4.0 categorized `SQLExceptions`.

- • Refactored `CommunicationsException` into a JDBC-3.0 version, and a JDBC-4.0 version (which extends `SQLRecover-` `ableException`, now that it exists).

  > **Note**
  >
  > This change means that if you were catching `com.mysql.jdbc.CommunicationsException` in your applications instead of looking at the SQLState class of `08`, and are moving to Java 6 (or newer), you need to change your imports to that exception to be `com.mysql.jdbc.exceptions.jdbc4.CommunicationsException`, as the old class will not be instantiated for communications link-related errors under Java 6.

A.2. Changes in MySQL Connector/J 5.0.x

A.2.1. Changes in MySQL Connector/J 5.0.8 (09 October 2007)

Functionality Added or Changed

- • Driver will now fall back to sane defaults for `max_allowed_packet` and `net_buffer_length` if the server reports them incorrectly (and will log this situation at `WARN` level, since it is actually an error condition).

- • XAConnections now start in auto-commit mode (as per JDBC-4.0 specification clarification).

- • `functionsNeverReturnBlobs`: Should the driver always treat data from functions returning `BLOBs` as Strings. Added specifically to work around dubious metadata returned by the server for `GROUP BY` clauses. Defaults to false.

- • Added two configuration parameters:

  - • `blobsAreStrings`: Should the driver always treat BLOBs as Strings. Added specifically to work around dubious metadata returned by the server for `GROUP BY` clauses. Defaults to false.

  - • `functionsNeverReturnBlobs`: Should the driver always treat data from functions returning `BLOBs` as Strings. Added specifically to work around dubious metadata returned by the server for `GROUP BY` clauses. Defaults to false.

- • `blobsAreStrings`: Should the driver always treat BLOBs as Strings. Added specifically to work around dubious metadata returned by the server for `GROUP BY` clauses. Defaults to false.

Bugs Fixed

- • Connections established using URLs of the form `jdbc:mysql:loadbalance://` weren't doing failover if they tried to connect to a MySQL server that was down. The driver now attempts connections to the next "best" (depending on the load balance strategy in use) server, and continues to attempt connecting to the next "best" server every 250 milliseconds until one is found that is up and running or 5 minutes has passed.

  If the driver gives up, it will throw the last-received `SQLException`. (Bug #31053)

- • `CallableStatement.executeBatch()` doesn't work when connection property `noAccessToProcedureBodies` has been set to `true`.

  The fix involves changing the behavior of `noAccessToProcedureBodies`,in that the driver will now report all parameters as

IN parameters but permit callers to call registerOutParameter() on them without throwing an exception. (Bug #28689)

- NPE with null column values when `padCharsWithSpace` is set to true. (Bug #30851)

- `setObject(int, Object, int, int)` delegate in PreparedStatementWrapper delegates to wrong method. (Bug #30892)

- Closing a load-balanced connection would cause a `ClassCastException`. (Bug #29852)

- `DatabaseMetaData.getTypeInfo()` for the types `DECIMAL` and `NUMERIC` will return a precision of 254 for server versions older than 5.0.3, 64 for versions 5.0.3 to 5.0.5 and 65 for versions newer than 5.0.5. (Bug #28972)

- An `ArithmeticException` or `NullPointerException` would be raised when the batch had zero members and `rewriteBatchedStatements=true` when `addBatch()` was never called, or `executeBatch()` was called immediately after `clearBatch()`. (Bug #30550)

- Collation on `VARBINARY` column types would be misidentified. A fix has been added, but this fix only works for MySQL server versions 5.0.25 and newer, since earlier versions didn't consistently return correct metadata for functions, and thus results from subqueries and functions were indistinguishable from each other, leading to type-related bugs. (Bug #30664)

- Connection checker for JBoss didn't use same method parameters using reflection, causing connections to always seem "bad". (Bug #29106)

- Cached metadata with `PreparedStatement.execute()` throws `NullPointerException`. (Bug #27412)

- `UNSIGNED` types not reported using `DBMD.getTypeInfo()`, and capitalization of type names is not consistent between `DBMD.getColumns()`, `RSMD.getColumnTypeName()` and `DBMD.getTypeInfo()`.

  This fix also ensures that the precision of `UNSIGNED MEDIUMINT` and `UNSIGNED BIGINT` is reported correctly using `DBMD.getColumns()`. (Bug #27916)

- When a connection is in read-only mode, queries that are wrapped in parentheses were incorrectly identified DML statements. (Bug #28256)

- Schema objects with identifiers other than the connection character aren't retrieved correctly in `ResultSetMetadata`. (Bug #27867)

- `Connection.getServerCharacterEncoding()` doesn't work for servers with version >= 4.1. (Bug #27182)

- `DatabaseMetaData.getColumns()` doesn't contain `SCOPE_*` or `IS_AUTOINCREMENT` columns. (Bug #27915)

- The automated SVN revisions in `DBMD.getDriverVersion()`. The SVN revision of the directory is now inserted into the version information during the build. (Bug #21116)

- Specifying a "validation query" in your connection pool that starts with "/* ping */" _exactly_ will cause the driver to instead send a ping to the server and return a fake result set (much lighter weight), and when using a ReplicationConnection or a LoadBalancedConnection, will send the ping across all active connections.

A.2.2. Changes in MySQL Connector/J 5.0.7 (20 July 2007)

Functionality Added or Changed

- Added configuration property `useNanosForElapsedTime` - for profiling/debugging functionality that measures elapsed time, should the driver try to use nanoseconds resolution if available (requires JDK >= 1.5)?

  > **Note**
  >
  > If `useNanosForElapsedTime` is set to `true`, and this property is set to "0" (or left default), then elapsed times will still be measured in nanoseconds (if possible), but the slow query threshold will be converted from milliseconds to nanoseconds, and thus have an upper bound of approximately 2000 milliseconds (as that threshold is represented as an integer, not a long).

- Added new debugging functionality - Setting configuration property `includeInnodbStatusInDeadlockExceptions` to `true` will cause the driver to append the output of `SHOW ENGINE INNODB STATUS` to deadlock-related exceptions, which will

enumerate the current locks held inside InnoDB.

- `tcpTrafficClass` - Should the driver set traffic class or type-of-service fields? See the documentation for java.net.Socket.setTrafficClass() for more information.

- `tcpSndBuf` - Should the driver set SO_SND_BUF to the given value? The default value of '0', means use the platform default value for this property.

- Setting the configuration parameter `useCursorFetch` to `true` for MySQL-5.0+ enables the use of cursors that enable Connect- or/J to save memory by fetching result set rows in chunks (where the chunk size is set by calling setFetchSize() on a Statement or ResultSet) by using fully-materialized cursors on the server.

- Added configuration properties to enable tuning of TCP/IP socket parameters:

  - `tcpNoDelay` - Should the driver set SO_TCP_NODELAY (disabling the Nagle Algorithm, default `true`)?

  - `tcpKeepAlive` - Should the driver set SO_KEEPALIVE (default `true`)?

  - `tcpRcvBuf` - Should the driver set SO_RCV_BUF to the given value? The default value of '0', means use the platform default value for this property.

  - `tcpSndBuf` - Should the driver set SO_SND_BUF to the given value? The default value of '0', means use the platform default value for this property.

  - `tcpTrafficClass` - Should the driver set traffic class or type-of-service fields? See the documentation for java.net.Socket.setTrafficClass() for more information.

- Setting `useDynamicCharsetInfo` to `false` now causes driver to use static lookups for collations as well (makes ResultSet- Metadata.isCaseSensitive() much more efficient, which leads to performance increase for ColdFusion, which calls this method for every column on every table it sees, it appears).

- `tcpRcvBuf` - Should the driver set SO_RCV_BUF to the given value? The default value of '0', means use the platform default value for this property.

- Added configuration property `slowQueryThresholdNanos` - if `useNanosForElapsedTime` is set to `true`, and this prop- erty is set to a nonzero value the driver will use this threshold (in nanosecond units) to determine if a query was slow, instead of us- ing millisecond units.

- `tcpNoDelay` - Should the driver set SO_TCP_NODELAY (disabling the Nagle Algorithm, default `true`)?

- Driver detects when it is running in a ColdFusion MX server (tested with version 7), and uses the configuration bundle `coldFu- sion`, which sets `useDynamicCharsetInfo` to `false` (see previous entry), and sets `useLocalSessionState` and autoReconnect to `true`.

- Give more information in EOFExceptions thrown out of MysqlIO (how many bytes the driver expected to read, how many it actu- ally read, say that communications with the server were unexpectedly lost).

- `tcpKeepAlive` - Should the driver set SO_KEEPALIVE (default `true`)?

- The driver will now automatically set `useServerPrepStmts` to `true` when `useCursorFetch` has been set to `true`, since the feature requires server-side prepared statements to function.

Bugs Fixed

- Parser in client-side prepared statements eats character following '/' if it is not a multi-line comment. (Bug #28851)

- Parser in client-side prepared statements runs to end of statement, rather than end-of-line for '#' comments. Also added support for '- -' single-line comments. (Bug #28956)

- Don't send any file data in response to LOAD DATA LOCAL INFILE if the feature is disabled at the client side. This is to prevent a malicious server or man-in-the-middle from asking the client for data that the client is not expecting. Thanks to Jan Kneschke for discovering the exploit and Andrey "Poohie" Hristov, Konstantin Osipov and Sergei Golubchik for discussions about implications and possible fixes. (Bug #29605)

- PreparedStatement.getMetaData() for statements containing leading one-line comments is not returned correctly.

  As part of this fix, we also overhauled detection of DML for `executeQuery()` and `SELECT`s for `executeUpdate()` in plain and prepared statements to be aware of the same types of comments. (Bug #28469)

A.2.3. Changes in MySQL Connector/J 5.0.6 (15 May 2007)

Functionality Added or Changed

- More intelligent initial packet sizes for the "shared" packets are used (512 bytes, rather than 16K), and initial packets used during handshake are now sized appropriately as to not require reallocation.

- Driver will now use `INSERT INTO ... VALUES (DEFAULT)`form of statement for updatable result sets for `Result-Set.insertRow()`, rather than pre-populating the insert row with values from `DatabaseMetaData.getColumns()`(which results in a `SHOW FULL COLUMNS` on the server for every result set). If an application requires access to the default values before `insertRow()` has been called, the JDBC URL should be configured with `populateInsertRowWithDefaultValues` set to `true`.

  This fix specifically targets performance issues with ColdFusion and the fact that it seems to ask for updatable result sets no matter what the application does with them.

- Fixed issue where a failed-over connection would let an application call `setReadOnly(false)`, when that call should be ignored until the connection is reconnected to a writable master unless `failoverReadOnly` had been set to `false`.

- `com.mysql.jdbc.[NonRegistering]Driver` now understands URLs of the format `jdbc:mysql:replication://` and `jdbc:mysql:loadbalance://` which will create a ReplicationConnection (exactly like when using `[NonRegistering]ReplicationDriver`) and an experimental load-balanced connection designed for use with SQL nodes in a MySQL Cluster/NDB environment, respectively.

  In an effort to simplify things, we're working on deprecating multiple drivers, and instead specifying different core behavior based upon JDBC URL prefixes, so watch for `[NonRegistering]ReplicationDriver` to eventually disappear, to be replaced with `com.mysql.jdbc[NonRegistering]Driver` with the new URL prefix.

- `random`: The driver will pick a random host for each request. This tends to work better than round-robin, as the randomness will somewhat account for spreading loads where requests vary in response time, while round-robin can sometimes lead to overloaded nodes if there are variations in response times across the workload.

- Give better error message when "streaming" result sets, and the connection gets clobbered because of exceeding `net_write_timeout` on the server.

- New configuration property, `enableQueryTimeouts` (default `true`).

  When enabled, query timeouts set with `Statement.setQueryTimeout()` use a shared `java.util.Timer` instance for scheduling. Even if the timeout doesn't expire before the query is processed, there will be memory used by the `TimerTask` for the given timeout which won't be reclaimed until the time the timeout would have expired if it hadn't been cancelled by the driver. High-load environments might want to consider disabling this functionality. (this configuration property is part of the `maxPer-formance` configuration bundle).

- Added configuration property `useDynamicCharsetInfo`. If set to `false` (the default), the driver will use a per-connection cache of character set information queried from the server when necessary, or when set to `true`, use a built-in static mapping that is more efficient, but isn't aware of custom character sets or character sets implemented after the release of the JDBC driver.

  > **Note**
  >
  > This only affects the `padCharsWithSpace` configuration property and the `ResultSet-MetaData.getColumnDisplayWidth()` method.

- When `useLocalSessionState` is set to `true` and connected to a MySQL-5.0 or later server, the JDBC driver will now determine whether an actual `commit` or `rollback` statement needs to be sent to the database when `Connection.commit()` or `Connection.rollback()` is called.

  This is especially helpful for high-load situations with connection pools that always call `Connection.rollback()` on connection check-in/check-out because it avoids a round-trip to the server.

- Added configuration property `padCharsWithSpace` (defaults to `false`). If set to `true`, and a result set column has the `CHAR` type and the value does not fill the amount of characters specified in the DDL for the column, the driver will pad the remaining characters with space (for ANSI compliance).

- `bestResponseTime`: The driver will route the request to the host that had the best response time for the previous transaction.

- Added an experimental load-balanced connection designed for use with SQL nodes in a MySQL Cluster/NDB environment (This is not for master-slave replication. For that, we suggest you look at `ReplicationConnection` or `lbpool`).

  If the JDBC URL starts with `jdbc:mysql:loadbalance://host-1,host-2,...host-n`, the driver will create an implementation of `java.sql.Connection` that load balances requests across a series of MySQL JDBC connections to the given hosts, where the balancing takes place after transaction commit.

  Therefore, for this to work (at all), you must use transactions, even if only reading data.

  Physical connections to the given hosts will not be created until needed.

  The driver will invalidate connections that it detects have had communication errors when processing a request. A new connection to the problematic host will be attempted the next time it is selected by the load balancing algorithm.

  There are two choices for load balancing algorithms, which may be specified by the `loadBalanceStrategy` JDBC URL configuration property:

  - `random`: The driver will pick a random host for each request. This tends to work better than round-robin, as the randomness will somewhat account for spreading loads where requests vary in response time, while round-robin can sometimes lead to overloaded nodes if there are variations in response times across the workload.

  - `bestResponseTime`: The driver will route the request to the host that had the best response time for the previous transaction.

Bugs Fixed

- When the configuration property `useCursorFetch` was set to `true`, sometimes server would return new, more exact metadata during the execution of the server-side prepared statement that enables this functionality, which the driver ignored (using the original metadata returned during `prepare()`), causing corrupt reading of data due to type mismatch when the actual rows were returned. (Bug #26173)

- Whitespace surrounding storage/size specifiers in stored procedure parameters declaration causes `NumberFormatException` to be thrown when calling stored procedure on JDK-1.5 or newer, as the Number classes in JDK-1.5+ are whitespace intolerant. (Bug #25624)

- `Connection.getTransactionIsolation()` uses "`SHOW VARIABLES LIKE`" which is very inefficient on MySQL-5.0+ servers. (Bug #27655)

- `ResultSet.get*()` with a column index < 1 returns misleading error message. (Bug #27317)

- Fixed issue where calling `getGeneratedKeys()` on a prepared statement after calling `execute()` didn't always return the generated keys (`executeUpdate()` worked fine however). (Bug #27655)

- `BIT(> 1)` is returned as `java.lang.String` from `ResultSet.getObject()` rather than `byte[]`. (Bug #25328)

- `PreparedStatement` is not closed in `BlobFromLocator.getBytes()`. (Bug #26592)

- Fast date/time parsing doesn't take into account `00:00:00` as a legal value. (Bug #26789)

- Client options not sent correctly when using SSL, leading to stored procedures not being able to return results. Thanks to Don Cohen for the bug report, testcase and patch. (Bug #25545)

- More useful error messages are generated when the driver thinks a result set is not updatable. (Thanks to Ashley Martens for the patch). (Bug #28085)

- `CALL /* ... */ some_proc()` doesn't work. As a side effect of this fix, you can now use `/* */` and `#` comments when preparing statements using client-side prepared statement emulation.

  If the comments happen to contain parameter markers (`?`), they will be treated as belonging to the comment (that is, not recognized)

rather than being a parameter of the statement.

> **Note**
>
> The statement when sent to the server will contain the comments as-is, they're not stripped during the process of preparing the `PreparedStatement` or `CallableStatement`.

(Bug #27400)

- Comments in DDL of stored procedures/functions confuse procedure parser, and thus metadata about them can not be created, leading to inability to retrieve said metadata, or execute procedures that have certain comments in them. (Bug #26959)

- Using `ResultSet.get*()` with a column index less than 1 returns a misleading error message. (Bug #27317)

- `CallableStatements` with `OUT/INOUT` parameters that are "binary" (`BLOB`, `BIT`, `(VAR)BINARY`, `JAVA_OBJECT`) have extra 7 bytes. (Bug #25715)

- `Statement.setMaxRows()` is not effective on result sets materialized from cursors. (Bug #25517)

A.2.4. Changes in MySQL Connector/J 5.0.5 (02 March 2007)

Functionality Added or Changed

- We've added a new configuration option `treatUtilDateAsTimestamp`, which is `false` by default, as (1) We already had specific behavior to treat java.util.Date as a java.sql.Timestamp because it is useful to many folks, and (2) that behavior will very likely be required for drivers JDBC-post-4.0.

- Added configuration property `localSocketAddress`, which is the host name or IP address given to explicitly configure the interface that the driver will bind the client side of the TCP/IP connection to when connecting.

- Fixed logging of XA commands sent to server, it is now configurable using `logXaCommands` property (defaults to `false`).

- Usage Advisor now detects empty results sets and does not report on columns not referenced in those empty sets.

- Improved speed of `datetime` parsing for ResultSets that come from plain or nonserver-side prepared statements. You can enable old implementation with `useFastDateParsing=false` as a configuration parameter.

- **Important change:** Due to a number of issues with the use of server-side prepared statements, Connector/J 5.0.5 has disabled their use by default. The disabling of server-side prepared statements does not affect the operation of the connector in any way.

  To enable server-side prepared statements, add the following configuration property to your connector string:

  ```
  useServerPrepStmts=true
  ```

  The default value of this property is `false` (that is, Connector/J does not use server-side prepared statements).

- The `rewriteBatchedStatements` feature can now be used with server-side prepared statements.

- Usage Advisor will now issue warnings for result sets with large numbers of rows. You can configure the trigger value by using the `resultSetSizeThreshold` parameter, which has a default value of 100.

Bugs Fixed

- When using a server-side prepared statement the driver would send timestamps to the server using nanoseconds instead of milliseconds. (Bug #21438)

- Connection property `socketFactory` wasn't exposed using correctly named mutator/accessor, causing data source implementations that use JavaBean naming conventions to set properties to fail to set the property (and in the case of SJAS, fail silently when trying to set this parameter). (Bug #26326)

- `ParameterMetaData` throws `NullPointerException` when prepared SQL has a syntax error. Added `generateSim-`

pleParameterMetadata configuration property, which when set to `true` will generate metadata reflecting `VARCHAR` for every parameter (the default is `false`, which will cause an exception to be thrown if no parameter metadata for the statement is actually available). (Bug #21267)

- Connector/J now returns a better error message when server doesn't return enough information to determine stored procedure/function parameter types. (Bug #24065)

- When using the `rewriteBatchedStatements` connection option with `PreparedState.executeBatch()` an internal memory leak would occur. (Bug #25073)

- `EscapeProcessor` gets confused by multiple backslashes. We now push the responsibility of syntax errors back on to the server for most escape sequences. (Bug #25399)

- Specifying `US-ASCII` as the character set in a connection to a MySQL 4.1 or newer server does not map correctly. (Bug #24840)

- `DatabaseMetaData.getSchemas()` doesn't return a `TABLE_CATALOG` column. (Bug #23303)

- `INOUT` parameters in `CallableStatements` get doubly-escaped. (Bug #25379)

- Client-side prepared statement parser gets confused by in-line comments `/*...*/` and therefore cannot rewrite batch statements or reliably detect the type of statements when they are used. (Bug #25025)

- Results sets from `UPDATE` statements that are part of multi-statement queries would cause an `SQLException` error, "Result is from UPDATE". (Bug #25009)

- When using server-side prepared statements and timestamp columns, value would be incorrectly populated (with nanoseconds, not microseconds). (Bug #21438)

- Some exceptions thrown out of `StandardSocketFactory` were needlessly wrapped, obscuring their true cause, especially when using socket timeouts. (Bug #21480)

- `StringUtils.indexOfIgnoreCaseRespectQuotes()` isn't case-insensitive on the first character of the target. This bug also affected `rewriteBatchedStatements` functionality when prepared statements did not use uppercase for the `VALUES` clause. (Bug #25047)

- Using `DatabaseMetaData.getSQLKeywords()` does not return a all of the of the reserved keywords for the current MySQL version. Current implementation returns the list of reserved words for MySQL 5.1, and does not distinguish between versions. (Bug #24794)

- A query execution which timed out did not always throw a `MySQLTimeoutException`. (Bug #25836)

- A connection error would occur when connecting to a MySQL server with certain character sets. Some collations/character sets reported as "unknown" (specifically `cias` variants of existing character sets), and inability to override the detected server character set. (Bug #23645)

- Using `setFetchSize()` breaks prepared `SHOW` and other commands. (Bug #24360)

- Fixed issue where field-level for metadata from `DatabaseMetaData` when using `INFORMATION_SCHEMA` didn't have references to current connections, sometimes leading to Null Pointer Exceptions (NPEs) when introspecting them using `ResultSetMetaData`. (Bug #25073)

- Using `DATETIME` columns would result in time shifts when `useServerPrepStmts` was true. This occurred due to different behavior when using client-side compared to server-side prepared statements and the `useJDBCCompliantTimezoneShift` option. This is now fixed if moving from server-side prepared statements to client-side prepared statements by setting `useSSPSCompatibleTimezoneShift` to `true`, as the driver can't tell if this is a new deployment that never used server-side prepared statements, or if it is an existing deployment that is switching to client-side prepared statements from server-side prepared statements. (Bug #24344)

- When using a JDBC connection URL that is malformed, the `NonRegisteringDriver.getPropertyInfo` method will throw a Null Pointer Exception (NPE). (Bug #22628)

- Storing a `java.util.Date` object in a `BLOB` column would not be serialized correctly during `setObject`. (Bug #25787)

- Inconsistency between `getSchemas` and `INFORMATION_SCHEMA`. (Bug #23304)

- Calendars and timezones are now lazily instantiated when required. (Bug #24351)

- Timer instance used for `Statement.setQueryTimeout()` created per-connection, rather than per-VM, causing memory leak. (Bug #25514)

- Calling `Statement.cancel()` could result in a Null Pointer Exception (NPE). (Bug #24721)

- Fixed an issue where `XADataSources` couldn't be bound into JNDI, as the `DataSourceFactory` didn't know how to create instances of them.

Other Changes

- When using cached metadata, skip field-level metadata packets coming from the server, rather than reading them and discarding them without creating `com.mysql.jdbc.Field` instances.

- Use a `java.util.TreeMap` to map column names to ordinal indexes for `ResultSet.findColumn()` instead of a HashMap. This enables us to have case-insensitive lookups (required by the JDBC specification) without resorting to the many transient object instances needed to support this requirement with a normal `HashMap` with either case-adjusted keys, or case-insensitive keys. (In the worst case scenario for lookups of a 1000 column result set, TreeMaps are about half as fast wall-clock time as a HashMap, however in normal applications their use gives many orders of magnitude reduction in transient object instance creation which pays off later for CPU usage in garbage collection).

- Fixed cases where `ServerPreparedStatements` weren't using cached metadata when `cacheResultSet-Metadata=true` was used.

- Take `localSocketAddress` property into account when creating instances of `CommunicationsException` when the underlying exception is a `java.net.BindException`, so that a friendlier error message is given with a little internal diagnostics.

- Fixed some Null Pointer Exceptions (NPEs) when cached metadata was used with `UpdatableResultSets`.

- When extracting foreign key information from `SHOW CREATE TABLE` in `DatabaseMetaData`, ignore exceptions relating to tables being missing (which could happen for cross-reference or imported-key requests, as the list of tables is generated first, then iterated).

- Reverted back to internal character conversion routines for single-byte character sets, as the ones internal to the JVM are using much more CPU time than our internal implementation.

- Changed cached result set metadata (when using `cacheResultSetMetadata=true`) to be cached per-connection rather than per-statement as previously implemented.

- Throw exceptions encountered during timeout to thread calling `Statement.execute*()`, rather than `RuntimeException`.

- Re-worked stored procedure parameter parser to be more robust. Driver no longer requires `BEGIN` in stored procedure definition, but does have requirement that if a stored function begins with a label directly after the "returns" clause, that the label is not a quoted identifier.

- Performance enhancement of initial character set configuration, driver will only send commands required to configure connection character set session variables if the current values on the server do not match what is required.

- Avoid static synchronized code in JVM class libraries for dealing with default timezones.

A.2.5. Changes in MySQL Connector/J 5.0.4 (20 October 2006)

Bugs Fixed

- Driver issues truncation on write exception when it shouldn't (due to sending big decimal incorrectly to server with server-side prepared statement). (Bug #22290)

- Newlines causing whitespace to span confuse procedure parser when getting parameter metadata for stored procedures. (Bug #22024)

- Driver was using milliseconds for Statement.setQueryTimeout() when specification says argument is to be in seconds. (Bug #22359)

- Added new _ci collations to CharsetMapping - utf8_unicode_ci not working. (Bug #22456)

- Workaround for server crash when calling stored procedures using a server-side prepared statement (driver now detects pre-pare(stored procedure) and substitutes client-side prepared statement). (Bug #22297)

- When using information_schema for metadata, COLUMN_SIZE for getColumns() is not clamped to range of java.lang.Integer as is the case when not using information_schema, thus leading to a truncation exception that isn't present when not using information_schema. (Bug #21544)

- DBMD.getColumns() does not return expected COLUMN_SIZE for the SET type, now returns length of largest possible set disregarding whitespace or the "," delimiters to be consistent with the ODBC driver. (Bug #22613)

- Column names don't match metadata in cases where server doesn't return original column names (column functions) thus breaking compatibility with applications that expect 1-to-1 mappings between `findColumn()` and `rsmd.getColumnName()`, usually manifests itself as "Can't find column (")" exceptions. (Bug #21379)

- Driver now supports `{call sp}` (without "()" if procedure has no arguments).

- DatabaseMetaData correctly reports `true` for `supportsCatalog*()` methods.

- Fixed configuration property `jdbcCompliantTruncation` was not being used for reads of result set values.

- Driver now sends numeric 1 or 0 for client-prepared statement `setBoolean()` calls instead of '1' or '0'.

A.2.6. Changes in MySQL Connector/J 5.0.3 (26 July 2006, beta)

Functionality Added or Changed

- Added configuration option `noAccessToProcedureBodies` which will cause the driver to create basic parameter metadata for `CallableStatements` when the user does not have access to procedure bodies using `SHOW CREATE PROCEDURE` or selecting from `mysql.proc` instead of throwing an exception. The default value for this option is `false`

Bugs Fixed

- Fixed `Statement.cancel()` causes `NullPointerException` if underlying connection has been closed due to server failure. (Bug #20650)

- If the connection to the server has been closed due to a server failure, then the cleanup process will call `Statement.cancel()`, triggering a `NullPointerException`, even though there is no active connection. (Bug #20650)

A.2.7. Changes in MySQL Connector/J 5.0.2 (11 July 2006)

Bugs Fixed

- `MysqlXaConnection.recover(int flags)` now permits combinations of `XAResource.TMSTARTRSCAN` and `TMENDRSCAN`. To simulate the "scanning" nature of the interface, we return all prepared XIDs for `TMSTARTRSCAN`, and no new XIDs for calls with `TMNOFLAGS`, or `TMENDRSCAN` when not in combination with `TMSTARTRSCAN`. This change was made for API compliance, as well as integration with IBM WebSphere's transaction manager. (Bug #20242)

- Fixed `MysqlValidConnectionChecker` for JBoss doesn't work with `MySQLXADataSources`. (Bug #20242)

- Added connection/datasource property `pinGlobalTxToPhysicalConnection` (defaults to `false`). When set to `true`, when using `XAConnections`, the driver ensures that operations on a given XID are always routed to the same physical connection. This enables the `XAConnection` to support `XA START ... JOIN` after `XA END` has been called, and is also a workaround for transaction managers that don't maintain thread affinity for a global transaction (most either always maintain thread affinity, or have it as a configuration option). (Bug #20242)

- Fixed driver fails on non-ASCII platforms. The driver was assuming that the platform character set would be a superset of MySQL's `latin1` when doing the handshake for authentication, and when reading error messages. We now use Cp1252 for all strings sent to the server during the handshake phase, and a hard-coded mapping of the `language` system variable to the character set that is used for error messages. (Bug #18086)

- Fixed can't use `XAConnection` for local transactions when no global transaction is in progress. (Bug #17401)

- Better caching of character set converters (per-connection) to remove a bottleneck for multibyte character sets. (Bug #20242)

- Fixed `ConnectionProperties` (and thus some subclasses) are not serializable, even though some J2EE containers expect them to be. (Bug #19169)

A.2.8. Changes in MySQL Connector/J 5.0.1 (Not Released)

Not released due to a packaging error

Version 5.0.1 has no changelog entries.

A.2.9. Changes in MySQL Connector/J 5.0.0 (22 December 2005)

Bugs Fixed

- Added support for Connector/MXJ integration using url subprotocol `jdbc:mysql:mxj://...`. (Bug #14729)

- Idle timeouts cause `XAConnections` to whine about rolling themselves back. (Bug #14729)

- When fix for Bug #14562 was merged from 3.1.12, added functionality for `CallableStatement`'s parameter metadata to return correct information for `.getParameterClassName()`. (Bug #14729)

- Added service-provider entry to `META-INF/services/java.sql.Driver` for JDBC-4.0 support. (Bug #14729)

- Fuller synchronization of `Connection` to avoid deadlocks when using multithreaded frameworks that multithread a single connection (usually not recommended, but the JDBC spec permits it anyways), part of fix to Bug #14972). (Bug #14729)

- Moved all `SQLException` constructor usage to a factory in `SQLError` (ground-work for JDBC-4.0 `SQLState`-based exception classes). (Bug #14729)

- Removed Java5-specific calls to `BigDecimal` constructor (when result set value is `''`, `(int)0` was being used as an argument indirectly using method return value. This signature doesn't exist prior to Java5.) (Bug #14729)

- Implementation of `Statement.cancel()` and `Statement.setQueryTimeout()`. Both require MySQL-5.0.0 or newer server, require a separate connection to issue the `KILL QUERY` statement, and in the case of `setQueryTimeout()` creates an additional thread to handle the timeout functionality.

  Note: Failures to cancel the statement for `setQueryTimeout()` may manifest themselves as `RuntimeExceptions` rather than failing silently, as there is currently no way to unblock the thread that is executing the query being cancelled due to timeout expiration and have it throw the exception instead. (Bug #14729)

- Return "[VAR]BINARY" for `RSMD.getColumnTypeName()` when that is actually the type, and it can be distinguished (MySQL-4.1 and newer). (Bug #14729)

- Add one level of indirection of internal representation of `CallableStatement` parameter metadata to avoid class not found issues on JDK-1.3 for `ParameterMetadata` interface (which doesn't exist prior to JDBC-3.0).

- `PreparedStatement.setString()` didn't work correctly when `sql_mode` on server contained `NO_BACKSLASH_ESCAPES` and no characters that needed escaping were present in the string.

- Setting `useJDBCCompliantTimezoneShift=true` (it is not the default) causes the driver to use GMT for *all* `TIMESTAMP`/`DATETIME` time zones, and the current VM time zone for any other type that refers to time zones. This feature can not be used when `useTimezone=true` to convert between server and client time zones.

- Return original column name for `RSMD.getColumnName()` if the column was aliased, alias name for `.getColumnLabel()` (if aliased), and original table name for `.getTableName()`. Note this only works for MySQL-4.1 and newer, as older servers don't make this information available to clients.

- Moved `-bin-g.jar` file into separate `debug` subdirectory to avoid confusion.

- `XADataSource` implemented (ported from 3.2 branch which won't be released as a product). Use

`com.mysql.jdbc.jdbc2.optional.MysqlXADataSource` as your datasource class name in your application server to utilize XA transactions in MySQL-5.0.10 and newer.

- Do not permit `.setAutoCommit(true)`, or `.commit()` or `.rollback()` on an XA-managed connection as per the JDBC specification.

- If the connection `useTimezone` is set to `true`, then also respect time zone conversions in escape-processed string literals (for example, `"{ts ...}"` and `"{t ...}"`).

- Added unit tests for `XADatasource`, as well as friendlier exceptions for XA failures compared to the "stock" `XAException` (which has no messages).

- Attempt detection of the MySQL type `BINARY` (it is an alias, so this isn't always reliable), and use the `java.sql.Types.BINARY` type mapping for it.

A.3. Changes in MySQL Connector/J 3.1.x

A.3.1. Changes in MySQL Connector/J 3.1.15 (Not yet released)

Important change: Due to a number of issues with the use of server-side prepared statements, Connector/J 5.0.5 has disabled their use by default. The disabling of server-side prepared statements does not affect the operation of the connector in any way.

To enable server-side prepared statements you must add the following configuration property to your connector string:

```
useServerPrepStmts=true
```

The default value of this property is `false` (that is, Connector/J does not use server-side prepared statements).

Bugs Fixed

- Specifying `US-ASCII` as the character set in a connection to a MySQL 4.1 or newer server does not map correctly. (Bug #24840)

A.3.2. Changes in MySQL Connector/J 3.1.14 (19 October 2006)

Bugs Fixed

- Check and store value for continueBatchOnError property in constructor of Statements, rather than when executing batches, so that Connections closed out from underneath statements don't cause NullPointerExceptions when it is required to check this property. (Bug #22290)

- Fixed Updatable result set that contains a BIT column fails when server-side prepared statements are used. (Bug #20485)

- Escape of quotation marks in client-side prepared statements parsing not respected. Patch covers more than bug report, including NO_BACKSLASH_ESCAPES being set, and stacked quote characters forms of escaping (that is, " or ""). (Bug #20888)

- Driver now sends numeric 1 or 0 for client-prepared statement setBoolean() calls instead of '1' or '0'. (Bug #22290)

- Fixed bug where driver would not advance to next host if roundRobinLoadBalance=true and the last host in the list is down. (Bug #22290)

- Driver issues truncation on write exception when it shouldn't (due to sending big decimal incorrectly to server with server-side prepared statement). (Bug #22290)

- ResultSet.getSomeInteger() doesn't work for BIT(>1). (Bug #21062)

- Fixed bug when calling stored functions, where parameters weren't numbered correctly (first parameter is now the return value, subsequent parameters if specified start at index "2"). (Bug #22290)

- Removed logger autodetection altogether, must now specify logger explicitly if you want to use a logger other than one that logs to STDERR. (Bug #21207)

- Fixed can't pool server-side prepared statements, exception raised when re-using them. (Bug #20687)

- DDriver throws NPE when tracing prepared statements that have been closed (in asSQL()). (Bug #21207)

- Fixed ResultSet.getShort() for UNSIGNED TINYINT returns incorrect values when using server-side prepared statements. (Bug #20306)

- ReplicationDriver does not always round-robin load balance depending on URL used for slaves list. (Bug #19993)

- `DatabaseMetaData.getTables()` or `getColumns()` with a bad catalog parameter threw an exception rather than return an empty result set (as required by the specification). (Bug #18258)

- Fixed memory leak with profileSQL=true. (Bug #16987)

- Connection fails to localhost when using timeout and IPv6 is configured. (Bug #19726)

- Fixed updatable result set throws ClassCastException when there is row data and moveToInsertRow() is called. (Bug #20479)

- Fixed calling toString() on ResultSetMetaData for driver-generated (that is, from DatabaseMetaData method calls, or from getGeneratedKeys()) result sets would raise a NullPointerException. (Bug #19993)

- Fixed NullPointerException in MysqlDataSourceFactory due to Reference containing RefAddrs with null content. (Bug #16791)

- ResultSet.getFloatFromString() can't retrieve values near Float.MIN/MAX_VALUE. (Bug #18880)

A.3.3. Changes in MySQL Connector/J 3.1.13 (26 May 2006)

Bugs Fixed

- Added performance feature, re-writing of batched executes for `Statement.executeBatch()` (for all DML statements) and `PreparedStatement.executeBatch()` (for INSERTs with VALUE clauses only). Enable by using "rewriteBatchedStatements=true" in your JDBC URL. (Bug #18041)

- `DBMD.getColumns()` returns wrong type for `BIT`. (Bug #15854)

- `PreparedStatement.setObject()` serializes `BigInteger` as object, rather than sending as numeric value (and is thus not complementary to `.getObject()` on an `UNSIGNED LONG` type). (Bug #15383)

- Fixed aliased column names where length of name > 251 are corrupted. (Bug #18554)

- Exception thrown for new decimal type when using updatable result sets. (Bug #14609)

- Improved performance of retrieving `BigDecimal`, `Time`, `Timestamp` and `Date` values from server-side prepared statements by creating fewer short-lived instances of `Strings` when the native type is not an exact match for the requested type. (Bug #18496)

- Fixed calling `clearParameters()` on a closed prepared statement causes NPE. (Bug #17587)

- Driver now aware of fix for `BIT` type metadata that went into MySQL-5.0.21 for server not reporting length consistently . (Bug #13601)

- No "dos" character set in MySQL > 4.1.0. (Bug #15544)

- Map "latin1" on MySQL server to CP1252 for MySQL > 4.1.0. (Bug #17587)

- Fixed `CallableStatement.registerOutParameter()` not working when some parameters pre-populated. Still waiting for feedback from JDBC experts group to determine what correct parameter count from `getMetaData()` should be, however. (Bug #17898)

- Fixed `ResultSet.wasNull()` not always reset correctly for booleans when done using conversion for server-side prepared statements. (Bug #17450)

- Added support for Apache Commons logging, use "com.mysql.jdbc.log.CommonsLogger" as the value for the "logger" configuration property. (Bug #13469)

- Fixed `ResultSet.wasNull()` returns incorrect value when extracting native string from server-side prepared statement generated result set. (Bug #19282)

- Fixed updatable result set doesn't return `AUTO_INCREMENT` values for `insertRow()` when multiple column primary keys are used. (the driver was checking for the existence of single-column primary keys and an autoincrement value > 0 instead of a straightforward `isAutoIncrement()` check). (Bug #16841)

- Fixed invalid classname returned for `ResultSetMetaData.getColumnClassName()` for `BIGINT type`. (Bug #19282)

- `INOUT` parameter does not store `IN` value. (Bug #15464)

- Fixed issue with `ReplicationConnection` incorrectly copying state, doesn't transfer connection context correctly when transitioning between the same read-only states. (Bug #15570)

- Fixed case where driver wasn't reading server status correctly when fetching server-side prepared statement rows, which in some cases could cause warning counts to be off, or multiple result sets to not be read off the wire. (Bug #19282)

- `Fixed PreparedStatement.setObject(int, Object, int)` doesn't respect scale of BigDecimals. (Bug #19615)

- Fixed driver trying to call methods that don't exist on older and newer versions of Log4j. The fix is not trying to auto-detect presence of log4j, too many different incompatible versions out there in the wild to do this reliably.

  If you relied on autodetection before, you will need to add "logger=com.mysql.jdbc.log.Log4JLogger" to your JDBC URL to enable Log4J usage, or alternatively use the new "CommonsLogger" class to take care of this. (Bug #13469)

- Fixed `Statement.getGeneratedKeys()` throws `NullPointerException` when no query has been processed. (Bug #17099)

- `lib-nodist` directory missing from package breaks out-of-box build. (Bug #15676)

- Added additional accessor and mutator methods on ConnectionProperties so that DataSource users can use same naming as regular URL properties. (Bug #17587)

- LogFactory now prepends `com.mysql.jdbc.log` to the log class name if it cannot be found as specified. This enables you to use "short names" for the built-in log factories, for example, `logger=CommonsLogger` instead of `logger=com.mysql.jdbc.log.CommonsLogger`. (Bug #13469)

- Fixed data truncation and `getWarnings()` only returns last warning in set. (Bug #18740)

- Fixed issue where server-side prepared statements don't cause truncation exceptions to be thrown when truncation happens. (Bug #18041)

- Fixed issue where driver was unable to initialize character set mapping tables. Removed reliance on `.properties` files to hold this information, as it turns out to be too problematic to code around class loader hierarchies that change depending on how an application is deployed. Moved information back into the `CharsetMapping` class. (Bug #14938)

- `ResultSet.getShort()` for `UNSIGNED TINYINT` returned wrong values. (Bug #11874)

A.3.4. Changes in MySQL Connector/J 3.1.12 (30 November 2005)

Bugs Fixed

- Driver incorrectly closes streams passed as arguments to `PreparedStatements`. Reverts to legacy behavior by setting the JDBC configuration property `autoClosePStmtStreams` to `true` (also included in the 3-0-Compat configuration "bundle"). (Bug #15024)

- `storesMixedCaseIdentifiers()` returns `false` (Bug #14562)

- `storesLowerCaseIdentifiers()` returns `true` (Bug #14562)

- Deadlock while closing server-side prepared statements from multiple threads sharing one connection. (Bug #14972)

- OpenOffice expects `DBMD.supportsIntegrityEnhancementFacility()` to return `true` if foreign keys are supported by the datasource, even though this method also covers support for check constraints, which MySQL *doesn't* have. Setting the con-

figuration property `overrideSupportsIntegrityEnhancementFacility` to `true` causes the driver to return `true` for this method. (Bug #12975)

- `storesMixedCaseQuotedIdentifiers()` returns `false` (Bug #14562)

- `storesMixedCaseQuotedIdentifiers()` returns `true` (Bug #14562)

- `maxQuerySizeToLog` is not respected. Added logging of bound values for `execute()` phase of server-side prepared statements when `profileSQL=true` as well. (Bug #13048)

- Process escape tokens in `Connection.prepareStatement(...)`. You can disable this behavior by setting the JDBC URL configuration property `processEscapeCodesForPrepStmts` to `false`. (Bug #15141)

- `DatabaseMetaData.getColumns()` doesn't return `TABLE_NAME` correctly. (Bug #14815)

- If `lower_case_table_names=0` (on server):

  - `storesLowerCaseIdentifiers()` returns `false`

  - `storesLowerCaseQuotedIdentifiers()` returns `false`

  - `storesMixedCaseIdentifiers()` returns `true`

  - `storesMixedCaseQuotedIdentifiers()` returns `true`

  - `storesUpperCaseIdentifiers()` returns `false`

  - `storesUpperCaseQuotedIdentifiers()` returns `true`

  (Bug #14562)

- `storesUpperCaseIdentifiers()` returns `false` (Bug #14562)

- `storesUpperCaseQuotedIdentifiers()` returns `true` (Bug #14562)

- If `lower_case_table_names=1` (on server):

  - `storesLowerCaseIdentifiers()` returns `true`

  - `storesLowerCaseQuotedIdentifiers()` returns `true`

  - `storesMixedCaseIdentifiers()` returns `false`

  - `storesMixedCaseQuotedIdentifiers()` returns `false`

  - `storesUpperCaseIdentifiers()` returns `false`

  - `storesUpperCaseQuotedIdentifiers()` returns `true`

  (Bug #14562)

- Extraneous sleep on `autoReconnect`. (Bug #13775)

- `storesLowerCaseQuotedIdentifiers()` returns `true` (Bug #14562)

- Fixed `DatabaseMetaData.stores*Identifiers()`:

  - If `lower_case_table_names=0` (on server):

    - `storesLowerCaseIdentifiers()` returns `false`

    - `storesLowerCaseQuotedIdentifiers()` returns `false`

    - `storesMixedCaseIdentifiers()` returns `true`

    - `storesMixedCaseQuotedIdentifiers()` returns `true`

- storesUpperCaseIdentifiers() returns false

- storesUpperCaseQuotedIdentifiers() returns true

- If lower_case_table_names=1 (on server):

  - storesLowerCaseIdentifiers() returns true

  - storesLowerCaseQuotedIdentifiers() returns true

  - storesMixedCaseIdentifiers() returns false

  - storesMixedCaseQuotedIdentifiers() returns false

  - storesUpperCaseIdentifiers() returns false

  - storesUpperCaseQuotedIdentifiers() returns true

  (Bug #14562)

- Reconnect during middle of executeBatch() should not occur if autoReconnect is enabled. (Bug #13255)

- storesMixedCaseIdentifiers() returns true (Bug #14562)

- Usage advisor complains about unreferenced columns, even though they've been referenced. (Bug #15065)

- storesLowerCaseQuotedIdentifiers() returns false (Bug #14562)

- Java type conversion may be incorrect for MEDIUMINT. (Bug #14562)

- Added com.mysql.jdbc.testsuite.url.default system property to set default JDBC url for testsuite (to speed up bug resolution when I'm working in Eclipse). (Bug #12975)

- Unable to initialize character set mapping tables (due to J2EE classloader differences). (Bug #14938)

- storesLowerCaseIdentifiers() returns false (Bug #14562)

- Escape processor replaces quote character in quoted string with string delimiter. (Bug #14909)

- Added configuration property useGmtMillisForDatetimes which when set to true causes ResultSet.getDate(), .getTimestamp() to return correct millis-since GMT when .getTime() is called on the return value (currently default is false for legacy behavior). (Bug #14562)

- logSlowQueries should give better info. (Bug #12230)

- Don't increase timeout for failover/reconnect. (Bug #6577)

- Fall back to platform-encoding for URLDecoder.decode() when parsing driver URL properties if the platform doesn't have a two-argument version of this method.

- Do not permit executeBatch() for CallableStatements with registered OUT/INOUT parameters (JDBC compliance).

- Fixed client-side prepared statement bug with embedded ? characters inside quoted identifiers (it was recognized as a placeholder, when it was not).

A.3.5. Changes in MySQL Connector/J 3.1.11 (07 October 2005)

Bugs Fixed

- getExportedKeys() (Bug #12541)

- Specifying a catalog works as stated in the API docs. (Bug #12541)

- Specifying `NULL` means that catalog will not be used to filter the results (thus all databases will be searched), unless you've set `nullCatalogMeansCurrent=true` in your JDBC URL properties. (Bug #12541)

- Tokenizer for `=` in URL properties was causing `sessionVariables=....` to be parameterized incorrectly. (Bug #12753)

- `getIndexInfo()` (Bug #12541)

- `getProcedures()` (and thus indirectly `getProcedureColumns()`) (Bug #12541)

- `getImportedKeys()` (Bug #12541)

- Specifying `""` means "current" catalog, even though this isn't quite JDBC spec compliant, it is there for legacy users. (Bug #12541)

- `getCrossReference()` (Bug #12541)

- The configuration property `sessionVariables` now permits you to specify variables that start with the "`@`" sign. (Bug #13453)

- Added `Connection.isMasterConnection()` for clients to be able to determine if a multi-host master/slave connection is connected to the first host in the list. (Bug #12541)

- Workaround for Bug #13374: `ResultSet.getStatement()` on closed result set returns `NULL` (as per JDBC 4.0 spec, but not backward-compatible). Set the connection property `retainStatementAfterResultSetClose` to `true` to be able to re-trieve a `ResultSet`'s statement after the `ResultSet` has been closed using `.getStatement()` (the default is `false`, to be JDBC-compliant and to reduce the chance that code using JDBC leaks `Statement` instances). (Bug #13277)

- `java.sql.Types.OTHER` returned for `BINARY` and `VARBINARY` columns when using `Database-MetaData.getColumns()`. (Bug #12970)

- URL configuration parameters do not permit "`&`" or "`=`" in their values. The JDBC driver now parses configuration parameters as if they are encoded using the application/x-www-form-urlencoded format as specified by `java.net.URLDecoder` (ht-tp://java.sun.com/j2se/1.5.0/docs/api/java/net/URLDecoder.html).

  If the "`%`" character is present in a configuration property, it must now be represented as `%25`, which is the encoded form of "`%`" when using application/x-www-form-urlencoded encoding. (Bug #13453)

- `getColumns()` (Bug #12541)

- Handling of catalog argument in `DatabaseMetaData.getIndexInfo()`, which also means changes to the following meth-ods in `DatabaseMetaData`:

  - `getBestRowIdentifier()`

  - `getColumns()`

  - `getCrossReference()`

  - `getExportedKeys()`

  - `getImportedKeys()`

  - `getIndexInfo()`

  - `getPrimaryKeys()`

  - `getProcedures()` (and thus indirectly `getProcedureColumns()`)

  - `getTables()`

  The `catalog` argument in all of these methods now behaves in the following way:

  - Specifying `NULL` means that catalog will not be used to filter the results (thus all databases will be searched), unless you've set `nullCatalogMeansCurrent=true` in your JDBC URL properties.

  - Specifying `""` means "current" catalog, even though this isn't quite JDBC spec compliant, it is there for legacy users.

  - Specifying a catalog works as stated in the API docs.

- Made `Connection.clientPrepare()` available from "wrapped" connections in the `jdbc2.optional` package (connections built by `ConnectionPoolDataSource` instances).

(Bug #12541)

- `getBestRowIdentifier()` (Bug #12541)

- Made `Connection.clientPrepare()` available from "wrapped" connections in the `jdbc2.optional` package (connections built by `ConnectionPoolDataSource` instances). (Bug #12541)

- `ServerPreparedStatement.getBinding()` now checks if the statement is closed before attempting to reference the list of parameter bindings, to avoid throwing a `NullPointerException`. (Bug #12970)

- `ResultSetMetaData` from `Statement.getGeneratedKeys()` caused a `NullPointerException` to be thrown whenever a method that required a connection reference was called. (Bug #13277)

- `cp1251` incorrectly mapped to `win1251` for servers newer than 4.0.x. (Bug #12752)

- When `gatherPerfMetrics` is enabled for servers older than 4.1.0, a `NullPointerException` is thrown from the constructor of `ResultSet` if the query doesn't use any tables. (Bug #13043)

- Backport of `VAR[BINARY|CHAR] [BINARY]` types detection from 5.0 branch. (Bug #13277)

- `getTables()` (Bug #12541)

- Fixed `NullPointerException` when converting `catalog` parameter in many `DatabaseMetaDataMethods` to `byte[]`s (for the result set) when the parameter is `null`. (`null` is not technically permitted by the JDBC specification, but we have historically permitted it). (Bug #13277)

- Backport of `Field` class, `ResultSetMetaData.getColumnClassName()`, and `ResultSet.getObject(int)` changes from 5.0 branch to fix behavior surrounding `VARCHAR BINARY/VARBINARY` and related types. (Bug #13277)

- Read response in `MysqlIO.sendFileToServer()`, even if the local file can't be opened, otherwise next query issued will fail, because it is reading the response to the empty `LOAD DATA INFILE` packet sent to the server. (Bug #13277)

- `Connection.prepareCall()` is database name case-sensitive (on Windows systems). (Bug #12417)

- `getPrimaryKeys()` (Bug #12541)

- Geometry types not handled with server-side prepared statements. (Bug #12104)

- `explainSlowQueries` hangs with server-side prepared statements. (Bug #12229)

- `Pstmt.setObject(...., Types.BOOLEAN)` throws exception. (Bug #11798)

- Reworked `Field` class, `*Buffer`, and `MysqlIO` to be aware of field lengths > `Integer.MAX_VALUE`. (Bug #11498)

- Foreign key information that is quoted is parsed incorrectly when `DatabaseMetaData` methods use that information. (Bug #11781)

- Fixed regression caused by fix for Bug #11552 that caused driver to return incorrect values for unsigned integers when those integers where within the range of the positive signed type. (Bug #11663)

- Escape tokenizer doesn't respect stacked single quotation marks for escapes. (Bug #11797)

- Escape processor didn't honor strings demarcated with double quotation marks. (Bug #11498)

- `maxPerformance.properties` mis-spells "elideSetAutoCommits". (Bug #11976)

- Moved source code to Subversion repository. (Bug #11663)

- The `sendBlobChunkSize` property is now clamped to `max_allowed_packet` with consideration of stream buffer size and packet headers to avoid `PacketTooBigExceptions` when `max_allowed_packet` is similar in size to the default `sendBlobChunkSize` which is 1M. (Bug #11781)

- `Statement.getWarnings()` fails with NPE if statement has been closed. (Bug #10630)

- `DBMD.storesLower/Mixed/UpperIdentifiers()` reports incorrect values for servers deployed on Windows. (Bug #11575)

- `ResultSet.moveToCurrentRow()` fails to work when preceded by a call to `ResultSet.moveToInsertRow()`. (Bug #11190)

- `VARBINARY` data corrupted when using server-side prepared statements and `.setBytes()`. (Bug #11115)

- Only get `char[]` from SQL in `PreparedStatement.ParseInfo()` when needed. (Bug #10630)

- Incorrect generation of testcase scripts for server-side prepared statements. (Bug #11663)

- `GEOMETRY` type not recognized when using server-side prepared statements. (Bug #11797)

- `CallableStatement.clearParameters()` now clears resources associated with `INOUT/OUTPUT` parameters as well as `INPUT` parameters. (Bug #11781)

- `StringUtils.getBytes()` doesn't work when using multi-byte character encodings and a length in *characters* is specified. (Bug #11614)

- Fixed statements generated for testcases missing `;` for "plain" statements. (Bug #11629)

- `ReplicationConnection` won't switch to slave, throws "Catalog can't be null" exception. (Bug #11879)

- Spurious `!` on console when character encoding is `utf8`. (Bug #11629)

- Properties shared between master and slave with replication connection. (Bug #12218)

- Updated `DBMD.supportsCorrelatedQueries()` to return `true` for versions > 4.1, `supportsGroupByUnrelated()` to return `true` and `getResultSetHoldability()` to return `HOLD_CURSORS_OVER_COMMIT`. (Bug #11498)

- Lifted restriction of changing streaming parameters with server-side prepared statements. As long as `all` streaming parameters were set before execution, `.clearParameters()` does not have to be called. (due to limitation of client/server protocol, prepared statements can not reset *individual* stream data on the server side). (Bug #11498)

A.3.6. Changes in MySQL Connector/J 3.1.10 (23 June 2005)

Bugs Fixed

- Fixed connecting without a database specified raised an exception in `MysqlIO.changeDatabaseTo()`.

- Initial implemention of `ParameterMetadata` for `PreparedStatement.getParameterMetadata()`. Only works fully for `CallableStatements`, as current server-side prepared statements return every parameter as a `VARCHAR` type.

A.3.7. Changes in MySQL Connector/J 3.1.9 (22 June 2005)

Bugs Fixed

- Fixed `PreparedStatement.setClob()` not accepting `null` as a parameter. (Bug #11360)

- Actually write manifest file to correct place so it ends up in the binary jar file. (Bug #10144)

- Try to handle `OutOfMemoryErrors` more gracefully. Although not much can be done, they will in most cases close the connection they happened on so that further operations don't run into a connection in some unknown state. When an OOM has happened, any further operations on the connection will fail with a "Connection closed" exception that will also list the OOM exception as the reason for the implicit connection close event. (Bug #10850)

- `autoReconnect` ping causes exception on connection startup. (Bug #11259)

- Reorganized directory layout. Sources now are in `src` folder. Don't pollute parent directory when building, now output goes to `./build`, distribution goes to `./dist`. (Bug #10496)

- `Connection.setCatalog()` is now aware of the `useLocalSessionState` configuration property, which when set to `true` will prevent the driver from sending `USE ...` to the server if the requested catalog is the same as the current catalog. (Bug #11115)

- Setting `cachePrepStmts=true` now causes the `Connection` to also cache the check the driver performs to determine if a prepared statement can be server-side or not, as well as caches server-side prepared statements for the lifetime of a connection. As before, the `prepStmtCacheSize` parameter controls the size of these caches. (Bug #10850)

- Don't send `COM_RESET_STMT` for each execution of a server-side prepared statement if it isn't required. (Bug #10850)

- `3-0-Compat`: Compatibility with Connector/J 3.0.x functionality (Bug #11115)

- Production package doesn't include JBoss integration classes. (Bug #11411)

- `maxPerformance`: Maximum performance without being reckless (Bug #11115)

- Unsigned `SMALLINT` treated as signed for `ResultSet.getInt()`, fixed all cases for `UNSIGNED` integer values and server-side prepared statements, as well as `ResultSet.getObject()` for `UNSIGNED TINYINT`. (Bug #10156)

- `solarisMaxPerformance`: Maximum performance for Solaris, avoids syscalls where it can (Bug #11115)

- Added `maintainTimeStats` configuration property (defaults to `true`), which tells the driver whether or not to keep track of the last query time and the last successful packet sent to the server's time. If set to `false`, removes two syscalls per query. (Bug #11115)

- 0-length streams not sent to server when using server-side prepared statements. (Bug #10850)

- The data type returned for `TINYINT(1)` columns when `tinyInt1isBit=true` (the default) can be switched between `Types.BOOLEAN` and `Types.BIT` using the new configuration property `transformedBitIsBoolean`, which defaults to `false`. If set to `false` (the default), `DatabaseMetaData.getColumns()` and `ResultSet-MetaData.getColumnType()` will return `Types.BOOLEAN` for `TINYINT(1)` columns. If `true`, `Types.BOOLEAN` will be returned instead. Regardless of this configuration property, if `tinyInt1isBit` is enabled, columns with the type `TINY-INT(1)` will be returned as `java.lang.Boolean` instances from `ResultSet.getObject(...)`, and `ResultSet-MetaData.getColumnClassName()` will return `java.lang.Boolean`. (Bug #10485)

- `VARBINARY` data corrupted when using server-side prepared statements and `ResultSet.getBytes()`. (Bug #11115)

- Added `createDatabaseIfNotExist` property (default is `false`), which will cause the driver to ask the server to create the database specified in the URL if it doesn't exist. You must have the appropriate privileges for database creation for this to work. (Bug #10144)

- `SQLException` thrown when retrieving `YEAR(2)` with `ResultSet.getString()`. The driver will now always treat `YEAR` types as `java.sql.Dates` and return the correct values for `getString()`. Alternatively, the `yearIsDateType` connection property can be set to `false` and the values will be treated as `SHORT`s. (Bug #10485)

- Connector/J dumping query into `SQLException` twice. (Bug #11360)

- Made `ServerPreparedStatement.asSql()` work correctly so auto-explain functionality would work with server-side prepared statements. (Bug #10155)

- Double quotation marks not recognized when parsing client-side prepared statements. (Bug #10155)

- Removed nonsensical "costly type conversion" warnings when using usage advisor. (Bug #11411)

- `com.mysql.jdbc.PreparedStatement.ParseInfo` does unnecessary call to `toCharArray()`. (Bug #9064)

- Driver doesn't support `{?=CALL(...)}` for calling stored functions. This involved adding support for function retrieval to `DatabaseMetaData.getProcedures()` and `getProcedureColumns()` as well. (Bug #10310)

- Memory leak in `ServerPreparedStatement` if `serverPrepare()` fails. (Bug #10144)

- Added support/bug hunting feature that generates `.sql` test scripts to `STDERR` when `autoGenerateTestcaseScript` is set to `true`. (Bug #10496)

- Driver detects if you're running MySQL-5.0.7 or later, and does not scan for `LIMIT ?[,?]` in statements being prepared, as the server supports those types of queries now. (Bug #10850)

- `SQLException` is thrown when using property `characterSetResults` with `cp932` or `eucjpms`. (Bug #10496)

- Made JDBC2-compliant wrappers public to enable access to vendor extensions. (Bug #10155)

- Added the following configuration bundles, use one or many using the `useConfigs` configuration property:

  - `maxPerformance`: Maximum performance without being reckless

  - `solarisMaxPerformance`: Maximum performance for Solaris, avoids syscalls where it can

  - `3-0-Compat`: Compatibility with Connector/J 3.0.x functionality

  (Bug #11115)

- `DatabaseMetaData.supportsMultipleOpenResults()` now returns `true`. The driver has supported this for some time, DBMD just missed that fact. (Bug #10155)

- Cleaned up logging of profiler events, moved code to dump a profiler event as a string to `com.mysql.jdbc.log.LogUtils` so that third parties can use it. (Bug #10155)

- Made `enableStreamingResults()` visible on `com.mysql.jdbc.jdbc2.optional.StatementWrapper`. (Bug #10155)

- Overhaul of character set configuration, everything now lives in a properties file.

- Driver now correctly uses CP932 if available on the server for Windows-31J, CP932 and MS932 java encoding names, otherwise it resorts to SJIS, which is only a close approximation. Currently only MySQL-5.0.3 and newer (and MySQL-4.1.12 or .13, depending on when the character set gets backported) can reliably support any variant of CP932.

A.3.8. Changes in MySQL Connector/J 3.1.8 (14 April 2005)

Bugs Fixed

- Added finalizers to `ResultSet` and `Statement` implementations to be JDBC spec-compliant, which requires that if not explicitly closed, these resources should be closed upon garbage collection. (Bug #9319)

- Should accept `null` for name patterns in DBMD (meaning "`%`"), even though it isn't JDBC compliant, for legacy's sake. Disable by setting connection property `nullNamePatternMatchesAll` to `false` (which will be the default value in C/J 3.2.x). (Bug #9769)

- Check for empty strings (`''`) when converting `CHAR`/`VARCHAR` column data to numbers, throw exception if `emptyString-sConvertToZero` configuration property is set to `false` (for backward-compatibility with 3.0, it is now set to `true` by default, but will most likely default to `false` in 3.2). (Bug #8803)

- The performance metrics feature now gathers information about number of tables referenced in a SELECT. (Bug #9704)

- Fixed driver not returning `true` for `-1` when `ResultSet.getBoolean()` was called on result sets returned from server-side prepared statements. (Bug #9778)

- `ServerPreparedStatements` now correctly "stream" `BLOB`/`CLOB` data to the server. You can configure the threshold chunk size using the JDBC URL property `blobSendChunkSize` (the default is 1MB). (Bug #8868)

- The logging system is now automatically configured. If the value has been set by the user, using the URL property `logger` or the system property `com.mysql.jdbc.logger`, then use that, otherwise, autodetect it using the following steps:

  1. Log4j, if it is available,

  2. Then JDK1.4 logging,

  3. Then fallback to our `STDERR` logging.

(Bug #9704)

- Added a `Manifest.MF` file with implementation information to the `.jar` file. (Bug #9778)

- Should accept `null` for catalog (meaning use current) in DBMD methods, even though it is not JDBC-compliant for legacy's sake. Disable by setting connection property `nullCatalogMeansCurrent` to `false` (which will be the default value in C/J 3.2.x). (Bug #9917)

- More tests in `Field.isOpaqueBinary()` to distinguish opaque binary (that is, fields with type `CHAR(n)` and `CHARACTER SET BINARY`) from output of various scalar and aggregate functions that return strings. (Bug #9778)

- `DBMD.getTables()` shouldn't return tables if views are asked for, even if the database version doesn't support views. (Bug #9778)

- Added support for the c3p0 connection pool's (http://c3p0.sf.net/) validation/connection checker interface which uses the lightweight `COM_PING` call to the server if available. To use it, configure your c3p0 connection pool's `connectionTesterClassName` property to use `com.mysql.jdbc.integration.c3p0.MysqlConnectionTester`. (Bug #9320)

- `DATA_TYPE` column from `DBMD.getBestRowIdentifier()` causes `ArrayIndexOutOfBoundsException` when accessed (and in fact, didn't return any value). (Bug #8803)

- `DATE_FORMAT()` queries returned as `BLOB`s from `getObject()`. (Bug #8868)

- A continuation of Bug #8868, where functions used in queries that should return nonstring types when resolved by temporary tables suddenly become opaque binary strings (work-around for server limitation). Also fixed fields with type of `CHAR(n) CHARACTER SET BINARY` to return correct/matching classes for `RSMD.getColumnClassName()` and `ResultSet.getObject()`. (Bug #9236)

- `PreparedStatement.getMetaData()` inserts blank row in database under certain conditions when not using server-side prepared statements. (Bug #9320)

- Better detection of `LIMIT` inside/outside of quoted strings so that the driver can more correctly determine whether a prepared statement can be prepared on the server or not. (Bug #9320)

- Stored procedures with `DECIMAL` parameters with storage specifications that contained "`,`" in them would fail. (Bug #9682)

- Server-side session variables can be preset at connection time by passing them as a comma-delimited list for the connection property `sessionVariables`. (Bug #8868)

- `Connection.canHandleAsPreparedStatement()` now makes "best effort" to distinguish `LIMIT` clauses with placeholders in them from ones without to have fewer false positives when generating work-arounds for statements the server cannot currently handle as server-side prepared statements. (Bug #9320)

- `BlobFromLocator` now uses correct identifier quoting when generating prepared statements. (Bug #8868)

- `Statement.getMoreResults()` could throw NPE when existing result set was `.close()`d. (Bug #9704)

- `PreparedStatement.addBatch()` doesn't work with server-side prepared statements and streaming `BINARY` data. (Bug #9040)

- Fixed `build.xml` to not compile `log4j` logging if `log4j` not available. (Bug #9320)

- Cannot use `UTF-8` for characterSetResults configuration property. (Bug #9206)

- `PreparedStatement.setObject(int, Object, int type, int scale)` now uses scale value for `BigDecimal` instances. (Bug #9682)

- `DBMD.supportsResultSetConcurrency()` not returning `true` for forward-only/read-only result sets (we obviously support this). (Bug #8792)

- Fixed regression in `ping()` for users using `autoReconnect=true`. (Bug #8868)

- Stored procedures with same name in different databases confuse the driver when it tries to determine parameter counts/types. (Bug #9319)

- `DBMD.supportsMixedCase*Identifiers()` returns wrong value on servers running on case-sensitive file systems. (Bug #8800)

- Fixed `DatabaseMetaData.getTables()` returning views when they were not asked for as one of the requested table types.

- Added support for new precision-math `DECIMAL` type in MySQL 5.0.3 and up.

- Made `Connection.ping()` a public method.

- Fixed `ResultSet.getTime()` on a `NULL` value for server-side prepared statements throws NPE.

A.3.9. Changes in MySQL Connector/J 3.1.7 (18 February 2005)

Bugs Fixed

- Infinite recursion when "falling back" to master in failover configuration. (Bug #7952)

- Disable multi-statements (if enabled) for MySQL-4.1 versions prior to version 4.1.10 if the query cache is enabled, as the server returns wrong results in this configuration. (Bug #7952)

- Added support for `BIT` type in MySQL-5.0.3. The driver will treat `BIT(1-8)` as the JDBC standard `BIT` type (which maps to `java.lang.Boolean`), as the server does not currently send enough information to determine the size of a bitfield when < 9 bits are declared. `BIT(>9)` will be treated as `VARBINARY`, and will return `byte[]` when `getObject()` is called. (Bug #8424)

- Added `useLocalSessionState` configuration property, when set to `true` the JDBC driver trusts that the application is well-behaved and only sets autocommit and transaction isolation levels using the methods provided on `java.sql.Connection`, and therefore can manipulate these values in many cases without incurring round-trips to the database server. (Bug #8424)

- Fixed synchronization issue with `ServerPreparedStatement.serverPrepare()` that could cause deadlocks/crashes if connection was shared between threads. (Bug #8096)

- Added `holdResultsOpenOverStatementClose` property (default is `false`), that keeps result sets open over statement.close() or new execution on same statement (suggested by Kevin Burton). (Bug #7715)

- Removed `dontUnpackBinaryResults` functionality, the driver now always stores results from server-side prepared statements as is from the server and unpacks them on demand. (Bug #7952)

- Detect new `sql_mode` variable in string form (it used to be integer) and adjust quoting method for strings appropriately. (Bug #7715)

- Choose correct "direction" to apply time adjustments when both client and server are in GMT time zone when using `Result-Set.get(..., cal)` and `PreparedStatement.set(...., cal)`. (Bug #4718)

- Fixed duplicated code in `configureClientCharset()` that prevented `useOldUTF8Behavior=true` from working properly. (Bug #7952)

- Remove `_binary` introducer from parameters used as in/out parameters in `CallableStatement`. (Bug #4718)

- Always return `byte[]`s for output parameters registered as `*BINARY`. (Bug #4718)

- Timestamps converted incorrectly to strings with server-side prepared statements and updatable result sets. (Bug #7715)

- By default, the driver now scans SQL you are preparing using all variants of `Connection.prepareStatement()` to determine if it is a supported type of statement to prepare on the server side, and if it is not supported by the server, it instead prepares it as a client-side emulated prepared statement. You can disable this by passing `emulateUnsupportedPstmts=false` in your JDBC URL. (Bug #4718)

- Emulated locators corrupt binary data when using server-side prepared statements. (Bug #8096)

- Added `enableStreamingResults()` to `Statement` for connection pool implementations that check `State-ment.setFetchSize()` for specification-compliant values. Call `Statement.setFetchSize(>=0)` to disable the streaming results for that statement. (Bug #8424)

- Added `dontTrackOpenResources` option (default is `false`, to be JDBC compliant), which helps with memory use for non-

well-behaved apps (that is, applications that don't close `Statement` objects when they should). (Bug #4718)

- Fixed NPE in `ResultSet.realClose()` when using usage advisor and result set was already closed. (Bug #8428)

- `ResultSet.getBigDecimal()` throws exception when rounding would need to occur to set scale. The driver now chooses a rounding mode of "half up" if nonrounding `BigDecimal.setScale()` fails. (Bug #8424)

- Send correct value for "boolean" `true` to server for `PreparedStatement.setObject(n, "true", Types.BIT)`. (Bug #4718)

- Timestamp key column data needed `_binary` stripped for `UpdatableResultSet.refreshRow()`. (Bug #7686)

- `PreparedStatements` not creating streaming result sets. (Bug #8487)

- `ResultSet.getString()` doesn't maintain format stored on server, bug fix only enabled when `noDatetimeStringSync` property is set to `true` (the default is `false`). (Bug #8428)

- Fixed bug with Connection not caching statements from `prepareStatement()` when the statement wasn't a server-side prepared statement. (Bug #4718)

- Don't pass `NULL` to `String.valueOf()` in `ResultSet.getNativeConvertToString()`, as it stringifies it (that is, returns `null`), which is not correct for the method in question. (Bug #8487)

A.3.10. Changes in MySQL Connector/J 3.1.6 (23 December 2004)

Bugs Fixed

- `DBMD.getProcedures()` doesn't respect catalog parameter. (Bug #7026)

- Fixed hang on `SocketInputStream.read()` with `Statement.setMaxRows()` and multiple result sets when driver has to truncate result set directly, rather than tacking a `LIMIT n` on the end of it.

A.3.11. Changes in MySQL Connector/J 3.1.5 (02 December 2004)

Bugs Fixed

- Removed unwanted new `Throwable()` in `ResultSet` constructor due to bad merge (caused a new object instance that was never used for every result set created). Found while profiling for Bug #6359. (Bug #6225)

- Use 1MB packet for sending file for `LOAD DATA LOCAL INFILE` if that is < `max_allowed_packet` on server. (Bug #6537)

- Use our own implementation of buffered input streams to get around blocking behavior of `java.io.BufferedInputStream`. Disable this with `useReadAheadInput=false`. (Bug #6399)

- Added experimental configuration property `dontUnpackBinaryResults`, which delays unpacking binary result set values until they're asked for, and only creates object instances for nonnumeric values (it is set to `false` by default). For some usecase/jvm combinations, this is friendlier on the garbage collector. (Bug #5706)

- Fixed batched updates with server prepared statements weren't looking if the types had changed for a given batched set of parameters compared to the previous set, causing the server to return the error "Wrong arguments to mysql_stmt_execute()". (Bug #5235)

- Make auto-deserialization of `java.lang.Objects` stored in `BLOB` columns configurable using `autoDeserialize` property (defaults to `false`). (Bug #6399)

- Don't throw exceptions for `Connection.releaseSavepoint()`. (Bug #5706)

- `ServerSidePreparedStatement` allocating short-lived objects unnecessarily. (Bug #6225)

- Inefficient detection of pre-existing string instances in `ResultSet.getNativeString()`. (Bug #5706)

- `UNSIGNED BIGINT` unpacked incorrectly from server-side prepared statement result sets. (Bug #5729)

- Use null-safe-equals for key comparisons in updatable result sets. (Bug #6225)

- Failing to connect to the server when one of the addresses for the given host name is IPV6 (which the server does not yet bind on). The driver now loops through *all* IP addresses for a given host, and stops on the first one that `accepts()` a `sock-et.connect()`. (Bug #6348)

- `ResultSetMetaData.getColumnDisplaySize()` returns incorrect values for multi-byte charsets. (Bug #6399)

- `SUM()` on `DECIMAL` with server-side prepared statement ignores scale if zero-padding is needed (this ends up being due to conversion to `DOUBLE` by server, which when converted to a string to parse into `BigDecimal`, loses all "padding" zeros). (Bug #6537)

- Use a per-session `Calendar` instance by default when decoding dates from `ServerPreparedStatements` (set to old, less performant behavior by setting property `dynamicCalendars=true`). (Bug #5706)

- Handle case when string representation of timestamp contains trailing "." with no numbers following it. (Bug #5235)

- Use `DatabaseMetaData.getIdentifierQuoteString()` when building DBMD queries. (Bug #6537)

- Re-work `Field.isOpaqueBinary()` to detect `CHAR(n) CHARACTER SET BINARY` to support fixed-length binary fields for `ResultSet.getObject()`. (Bug #6399)

- Server-side prepared statements did not honor `zeroDateTimeBehavior` property, and would cause class-cast exceptions when using `ResultSet.getObject()`, as the all-zero string was always returned. (Bug #5235)

- Fixed too-early creation of `StringBuffer` in `EscapeProcessor.escapeSQL()`, also return `String` when escaping not needed (to avoid unnecessary object allocations). Found while profiling for Bug #6359. (Bug #6225)

- Fix comparisons made between string constants and dynamic strings that are converted with either `toUpperCase()` or `to-LowerCase()` to use `Locale.ENGLISH`, as some locales "override" case rules for English. Also use `StringUtils.indexOfIgnoreCase()` instead of `.toUpperCase().indexOf()`, avoids creating a very short-lived transient `String` instance.

A.3.12. Changes in MySQL Connector/J 3.1.4 (04 September 2004)

Bugs Fixed

- Fixed `ServerPreparedStatement` to read prepared statement metadata off the wire, even though it is currently a placeholder instead of using `MysqlIO.clearInputStream()` which didn't work at various times because data wasn't available to read from the server yet. This fixes sporadic errors users were having with `ServerPreparedStatements` throwing `ArrayIndexOutOfBoundExceptions`. (Bug #5032)

- Optimized integer number parsing, enable "old" slower integer parsing using JDK classes using `useFastIntParsing=false` property. (Bug #4642)

- Track packet sequence numbers if `enablePacketDebug=true`, and throw an exception if packets received out-of-order. (Bug #4689)

- `ResultSet.wasNull()` does not work for primitives if a previous `null` was returned. (Bug #4689)

- Added three ways to deal with all-zero datetimes when reading them from a `ResultSet`: `exception` (the default), which throws an `SQLException` with an SQLState of `S1009`; `convertToNull`, which returns `NULL` instead of the date; and `round`, which rounds the date to the nearest closest value which is `'0001-01-01'`. (Bug #5032)

- Added `useOnlyServerErrorMessages` property, which causes message text in exceptions generated by the server to only contain the text sent by the server (as opposed to the SQLState's "standard" description, followed by the server's error message). This property is set to `true` by default. (Bug #4642)

- The driver is more strict about truncation of numerics on `ResultSet.get*()`, and will throw an `SQLException` when truncation is detected. You can disable this by setting `jdbcCompliantTruncation` to `false` (it is enabled by default, as this functionality is required for JDBC compliance). (Bug #5032)

- `ResultSet.getObject()` returns wrong type for strings when using prepared statements. (Bug #4482)

- `ServerPreparedStatements` dealing with return of `DECIMAL` type don't work. (Bug #5012)

- Calling `MysqlPooledConnection.close()` twice (even though an application error), caused NPE. Fixed. (Bug #4482)

- Connector/J 3.1.3 beta does not handle integers correctly (caused by changes to support unsigned reads in `Buffer.readInt()` -> `Buffer.readShort()`). (Bug #4510)

- You can now use URLs in `LOAD DATA LOCAL INFILE` statements, and the driver will use Java's built-in handlers for retrieving the data and sending it to the server. This feature is not enabled by default, you must set the `allowUrlInLocalInfile` connection property to `true`. (Bug #5032)

- Added support in `DatabaseMetaData.getTables()` and `getTableTypes()` for views, which are now available in MySQL server 5.0.x. (Bug #4510)

- `ServerPreparedStatement.execute*()` sometimes threw `ArrayIndexOutOfBoundsException` when unpacking field metadata. (Bug #4642)

- `ResultSet.getObject()` doesn't return type `Boolean` for pseudo-bit types from prepared statements on 4.1.x (shortcut for avoiding extra type conversion when using binary-encoded result sets obscured test in `getObject()` for "pseudo" bit type). (Bug #5032)

- Use `com.mysql.jdbc.Message`'s classloader when loading resource bundle, should fix sporadic issues when the caller's classloader can't locate the resource bundle. (Bug #5032)

A.3.13. Changes in MySQL Connector/J 3.1.3 (07 July 2004)

Bugs Fixed

- Externalized more messages (on-going effort). (Bug #4119)

- Support new time zone variables in MySQL-4.1.3 when `useTimezone=true`. (Bug #4311)

- Null bitmask sent for server-side prepared statements was incorrect. (Bug #4119)

- Added constants for MySQL error numbers (publicly accessible, see `com.mysql.jdbc.MysqlErrorNumbers`), and the ability to generate the mappings of vendor error codes to SQLStates that the driver uses (for documentation purposes). (Bug #4119)

- Added packet debugging code (see the `enablePacketDebug` property documentation). (Bug #4119)

- Error in retrieval of `mediumint` column with prepared statements and binary protocol. (Bug #4311)

- Use SQL Standard SQL states by default, unless `useSqlStateCodes` property is set to `false`. (Bug #4119)

- Support for unsigned numerics as return types from prepared statements. This also causes a change in `ResultSet.getObject()` for the `bigint unsigned` type, which used to return `BigDecimal` instances, it now returns instances of `java.lang.BigInteger`. (Bug #4311)

- Added support for `INOUT` parameters in `CallableStatements`.

- Mangle output parameter names for `CallableStatements` so they will not clash with user variable names.

A.3.14. Changes in MySQL Connector/J 3.1.2 (09 June 2004)

Bugs Fixed

- Fixed `DatabaseMetaData.getProcedures()` when run on MySQL-5.0.0 (output of `SHOW PROCEDURE STATUS` changed between 5.0.0 and 5.0.1. (Bug #3520)

- Fixed case when no output parameters specified for a stored procedure caused a bogus query to be issued to retrieve out parameters, leading to a syntax error from the server.

- Correctly map output parameters to position given in `prepareCall()` versus. order implied during `registerOutParameter()`. (Bug #3146)

- Added `connectionCollation` property to cause driver to issue `set collation_connection=...` query on connection init if default collation for given charset is not appropriate. (Bug #3520)

- `ServerPreparedStatements` weren't actually de-allocating server-side resources when `.close()` was called.

- Don't enable server-side prepared statements for server version 5.0.0 or 5.0.1, as they aren't compatible with the '4.1.2+' style that the driver uses (the driver expects information to come back that isn't there, so it hangs). (Bug #3804)

- `getProcedures()` does not return any procedures in result set. (Bug #3539)

- Fixed stored procedure parameter parsing info when size was specified for a parameter (for example, `char()`, `varchar()`).

- Removed wrapping of exceptions in `MysqlIO.changeUser()`.

- `getWarnings()` returns `SQLWarning` instead of `DataTruncation`. (Bug #3804)

- Cleaned up detection of server properties. (Bug #3146)

- Correctly detect initial character set for servers >= 4.1.0. (Bug #3146)

- Support placeholder for parameter metadata for server >= 4.1.2. (Bug #3146)

- Added `logSlowQueries` property, along with `slowQueriesThresholdMillis` property to control when a query should be considered "slow."

- Added `.toString()` functionality to `ServerPreparedStatement`, which should help if you're trying to debug a query that is a prepared statement (it shows SQL as the server would process).

- Fixed sending of split packets for large queries, enabled nio ability to send large packets as well.

- Enabled callable statement caching using `cacheCallableStmts` property.

- Fixed case when no parameters could cause a `NullPointerException` in `CallableStatement.setOutputParameters()`.

- Added `gatherPerformanceMetrics` property, along with properties to control when/where this info gets logged (see docs for more info).

- `DBMD.getSQLStateType()` returns incorrect value. (Bug #3520)

- `getProcedureColumns()` doesn't work with wildcards for procedure name. (Bug #3540)

A.3.15. Changes in MySQL Connector/J 3.1.1 (14 February 2004, alpha)

Bugs Fixed

- Fixed charset conversion issue in `getTables()`. (Bug #2502)

- Fixed `ConnectionProperties` that weren't properly exposed through accessors, cleaned up `ConnectionProperties` code. (Bug #2623)

- Prepared `Statements` will be re-prepared on auto-reconnect. Any errors encountered are postponed until first attempt to re-execute the re-prepared statement.

- Reduced number of methods called in average query to be more efficient.

- Refactored how connection properties are set and exposed as `DriverPropertyInfo` as well as `Connection` and `DataSource` properties.

- Fixed stack overflow in `Connection.prepareCall()` (bad merge).

- Enabled streaming of result sets from server-side prepared statements. (Bug #2606)

- Allow contents of `PreparedStatement.setBlob()` to be retained between calls to `.execute*()`.

- Fixed `IllegalAccessError` to `Calendar.getTimeInMillis()` in `DateTimeValue` (for JDK < 1.4).

- Default result set type changed to `TYPE_FORWARD_ONLY` (JDBC compliance).

- Fixed bug with `UpdatableResultSets` not using client-side prepared statements.

- Class-cast exception when using scrolling result sets and server-side prepared statements. (Bug #2623)

- Display where/why a connection was implicitly closed (to aid debugging). (Bug #1673)

- `DatabaseMetaData.getColumns()` is not returning correct column ordinal info for non-`'%'` column name patterns. (Bug #1673)

- Centralized setting of result set type and concurrency.

- Implemented `Connection.prepareCall()`, and `DatabaseMetaData`. `getProcedures()` and `getProcedure-Columns()`. (Bug #2359)

- Implemented multiple result sets returned from a statement or stored procedure. (Bug #2502)

- Fixed `NullPointerException` in `ServerPreparedStatement.setTimestamp()`, as well as year and month discrepancies in `ServerPreparedStatement.setTimestamp()`, `setDate()`. (Bug #1673)

- Fixed character encoding issues when converting bytes to ASCII when MySQL doesn't provide the character set, and the JVM is set to a multi-byte encoding (usually affecting retrieval of numeric values).

- Support "old" `profileSql` capitalization in `ConnectionProperties`. This property is deprecated, you should use `pro-fileSQL` if possible.

- Added ability to have multiple database/JVM targets for compliance and regression/unit tests in `build.xml`. (Bug #1673)

- Support for transaction savepoints (MySQL >= 4.0.14 or 4.1.1).

- Check for closed connection on delete/update/insert row operations in `UpdatableResultSet`.

- Support for NIO. Use `useNIO=true` on platforms that support NIO.

- Fixed sending of queries larger than 16M. (Bug #1673)

- Use DocBook version of docs for shipped versions of drivers. (Bug #2671)

- Merged fix of data type mapping from MySQL type `FLOAT` to `java.sql.Types.REAL` from 3.0 branch. (Bug #1673)

- Merged prepared statement caching, and `.getMetaData()` support from 3.0 branch. (Bug #2359)

- Fixed NPE and year/month bad conversions when accessing some datetime functionality in `ServerPreparedStatements` and their resultant result sets. (Bug #1673)

- Removed `useFastDates` connection property.

- Added named and indexed input/output parameter support to `CallableStatement`. MySQL-5.0.x or newer. (Bug #1673)

- Fixed off-by-1900 error in some cases for years in `TimeUtil.fastDate`/`TimeCreate()` when unpacking results from server-side prepared statements. (Bug #2359)

- Support for `mysql_change_user()`. See the `changeUser()` method in `com.mysql.jdbc.Connection`.

- `DatabaseMetaData` now reports `supportsStoredProcedures()` for MySQL versions >= 5.0.0

- Deal with 0-length tokens in `EscapeProcessor` (caused by callable statement escape syntax).

- Reset `long binary` parameters in `ServerPreparedStatement` when `clearParameters()` is called, by sending `COM_RESET_STMT` to the server. (Bug #2359)

- Server-side prepared statements were not returning data type `YEAR` correctly. (Bug #2606)

- `NULL` fields were not being encoded correctly in all cases in server-side prepared statements. (Bug #2671)

- Implemented long data (Blobs, Clobs, InputStreams, Readers) for server prepared statements.

- Correctly initialize datasource properties from JNDI Refs, including explicitly specified URLs.

- Ensure that warnings are cleared before executing queries on prepared statements, as-per JDBC spec (now that we support warnings).

- Implemented `Statement.getWarnings()` for MySQL-4.1 and newer (using `SHOW WARNINGS`).

- Unpack "unknown" data types from server prepared statements as `Strings`.

- `NULL` values for numeric types in binary encoded result sets causing `NullPointerExceptions`. (Bug #2359)

- `CommunicationsException` implemented, that tries to determine why communications was lost with a server, and displays possible reasons when `.getMessage()` is called. (Bug #1673)

- Fixed rare buffer underflow when writing numbers into buffers for sending prepared statement execution requests. (Bug #2671)

- Fix support for table aliases when checking for all primary keys in `UpdatableResultSet`.

- Optimized `Buffer.readLenByteArray()` to return shared empty byte array when length is 0.

- Detect collation of column for `RSMD.isCaseSensitive()`. (Bug #1673)

- Merged unbuffered input code from 3.0. (Bug #2623)

A.3.16. Changes in MySQL Connector/J 3.1.0 (18 February 2003, alpha)

Bugs Fixed

- Track open `Statements`, close all when `Connection.close()` is called (JDBC compliance).

- Added `requireSSL` property.

- Added `useServerPrepStmts` property (default `false`). The driver will use server-side prepared statements when the server version supports them (4.1 and newer) when this property is set to `true`. It is currently set to `false` by default until all bind/fetch functionality has been implemented. Currently only DML prepared statements are implemented for 4.1 server-side prepared statements.

A.4. Changes in MySQL Connector/J 3.0.x

A.4.1. Changes in MySQL Connector/J 3.0.17 (23 June 2005)

Bugs Fixed

- `PreparedStatement.fixDecimalExponent()` adding extra +, making number unparseable by MySQL server. (Bug #7061)

- Workaround for server Bug #9098: Default values of `CURRENT_*` for `DATE`, `TIME`, `DATETIME`, and `TIMESTAMP` columns can't be distinguished from `string` values, so `UpdatableResultSet.moveToInsertRow()` generates bad SQL for inserting default values. (Bug #8812)

- Handle streaming result sets with more than 2 billion rows properly by fixing wraparound of row number counter. (Bug #7601)

- Fixed duplicated code in `configureClientCharset()` that prevented `useOldUTF8Behavior=true` from working properly. (Bug #7952)

- Statements created from a pooled connection were returning physical connection instead of logical connection when `getConnection()` was called. (Bug #7316)

- `MS932`, `SHIFT_JIS`, and `Windows_31J` not recognized as aliases for `sjis`. (Bug #7607)

- `EUCKR` charset is sent as `SET NAMES euc_kr` which MySQL-4.1 and newer doesn't understand. (Bug #8629)

- Support new protocol type `MYSQL_TYPE_VARCHAR`. (Bug #7081)

- Connections starting up failed-over (due to down master) never retry master. (Bug #6966)

- `PreparedStatements` don't encode Big5 (and other multi-byte) character sets correctly in static SQL strings. (Bug #7033)

- `PreparedStatement.fixDecimalExponent()` adding extra +, making number unparseable by MySQL server. (Bug #7601)

- Backported SQLState codes mapping from Connector/J 3.1, enable with `useSqlStateCodes=true` as a connection property, it defaults to `false` in this release, so that we don't break legacy applications (it defaults to `true` starting with Connector/J 3.1). (Bug #7686)

- Added `useOldUTF8Behavior`' configuration property, which causes JDBC driver to act like it did with MySQL-4.0.x and earlier when the character encoding is `utf-8` when connected to MySQL-4.1 or newer. (Bug #7081)

- Escape sequence {fn convert(..., type)} now supports ODBC-style types that are prepended by `SQL_`. (Bug #7601)

- Added support for the `EUC_JP_Solaris` character encoding, which maps to a MySQL encoding of `eucjpms` (backported from 3.1 branch). This only works on servers that support `eucjpms`, namely 5.0.3 or later. (Bug #8629)

- `NON_UNIQUE` column from `DBMD.getIndexInfo()` returned inverted value. (Bug #8812)

- Use hex escapes for `PreparedStatement.setBytes()` for double-byte charsets including "aliases" `Windows-31J`, `CP934`, `MS932`. (Bug #8629)

- Timestamp key column data needed `_binary` stripped for `UpdatableResultSet.refreshRow()`. (Bug #7686)

- `DatabaseMetaData.getIndexInfo()` ignored `unique` parameter. (Bug #7081)

- `DatabaseMetaData.supportsSelectForUpdate()` returns correct value based on server version. (Bug #8629)

- Which requires hex escaping of binary data when using multi-byte charsets with prepared statements. (Bug #8064)

- `Timestamp`/`Time` conversion goes in the wrong "direction" when `useTimeZone=true` and server time zone differs from client time zone. (Bug #5874)

- Adding `CP943` to aliases for `sjis`. (Bug #6549, Bug #7607)

A.4.2. Changes in MySQL Connector/J 3.0.16 (15 November 2004)

Bugs Fixed

- Made `TINYINT(1)` -> `BIT`/`Boolean` conversion configurable using `tinyInt1isBit` property (default `true` to be JDBC compliant out of the box). (Bug #5664)

- Off-by-one bug in `Buffer.readString(`*`string`*`)`. (Bug #5664)

- `ResultSet.updateByte()` when on insert row throws `ArrayOutOfBoundsException`. (Bug #5664)

- Fixed regression where `useUnbufferedInput` was defaulting to `false`. (Bug #5664)

- `ResultSet.getTimestamp()` on a column with `TIME` in it fails. (Bug #5664)

- Fixed `DatabaseMetaData.getTypes()` returning incorrect (this is, nonnegative) scale for the `NUMERIC` type. (Bug #5664)

- Only set `character_set_results` during connection establishment if server version >= 4.1.1. (Bug #5664)

- Re-issue character set configuration commands when re-using pooled connections or `Connection.changeUser()` when connected to MySQL-4.1 or newer.

- Fixed `ResultSetMetaData.isReadOnly()` to detect nonwritable columns when connected to MySQL-4.1 or newer, based on existence of "original" table and column names.

A.4.3. Changes in MySQL Connector/J 3.0.15 (04 September 2004)

Bugs Fixed

- Calling `.close()` twice on a `PooledConnection` causes NPE. (Bug #4808)

- "Production" is now "GA" (General Availability) in naming scheme of distributions. (Bug #4860, Bug #4138)

- `DOUBLE` mapped twice in `DBMD.getTypeInfo()`. (Bug #4742)

- Removed redundant calls to `checkRowPos()` in `ResultSet`. (Bug #4334)

- `DBMD.getColumns()` returns incorrect JDBC type for unsigned columns. This affects type mappings for all numeric types in the `RSMD.getColumnType()` and `RSMD.getColumnTypeNames()` methods as well, to ensure that "like" types from `DBMD.getColumns()` match up with what `RSMD.getColumnType()` and `getColumnTypeNames()` return. (Bug #4860, Bug #4138)

- `ResultSet.getMetaData()` should not return incorrectly initialized metadata if the result set has been closed, but should instead throw an `SQLException`. Also fixed for `getRow()` and `getWarnings()` and traversal methods by calling `checkClosed()` before operating on instance-level fields that are nullified during `.close()`. (Bug #5069)

- `RSMD.getPrecision()` returning 0 for nonnumeric types (should return max length in chars for nonbinary types, max length in bytes for binary types). This fix also fixes mapping of `RSMD.getColumnType()` and `RSMD.getColumnTypeName()` for the `BLOB` types based on the length sent from the server (the server doesn't distinguish between `TINYBLOB`, `BLOB`, `MEDIUMBLOB` or `LONGBLOB` at the network protocol level). (Bug #4880)

- Added FLOSS license exemption. (Bug #4742)

- `ResultSet` should release `Field[]` instance in `.close()`. (Bug #5022)

- Failover for `autoReconnect` not using port numbers for any hosts, and not retrying all hosts.

  > **Warning**
  >
  > This required a change to the `SocketFactory connect()` method signature, which is now `public Socket connect(String host, int portNumber, Properties props)`; therefore, any third-party socket factories will have to be changed to support this signature.

  (Bug #4334)

- Use `_binary` introducer for `PreparedStatement.setBytes()` and `set*Stream()` when connected to MySQL-4.1.x or newer to avoid misinterpretation during character conversion. (Bug #5069)

- Logical connections created by `MysqlConnectionPoolDataSource` will now issue a `rollback()` when they are closed and sent back to the pool. If your application server/connection pool already does this for you, you can set the `rollbackOnPooledClose` property to `false` to avoid the overhead of an extra `rollback()`. (Bug #4334)

- Parse new time zone variables from 4.1.x servers. (Bug #5069)

- `StringUtils.escapeEasternUnicodeByteStream` was still broken for GBK. (Bug #4010)

A.4.4. Changes in MySQL Connector/J 3.0.14 (28 May 2004)

Bugs Fixed

- Fixed URL parsing error.

A.4.5. Changes in MySQL Connector/J 3.0.13 (27 May 2004)

Bugs Fixed

- Using a `MySQLDatasource` without server name fails. (Bug #3848)

- `PreparedStatement.getGeneratedKeys()` method returns only 1 result for batched insertions. (Bug #3873)

- `No Database Selected` when using `MysqlConnectionPoolDataSource`. (Bug #3920)

A.4.6. Changes in MySQL Connector/J 3.0.12 (18 May 2004)

Bugs Fixed

- Backported "change user" and "reset server state" functionality from 3.1 branch, to enable clients of `MysqlConnection-PoolDataSource` to reset server state on `getConnection()` on a pooled connection.

- Made `StringRegressionTest` 4.1-unicode aware. (Bug #3520)

- Use `SET character_set_results` during initialization to enable any charset to be returned to the driver for result sets. (Bug #2670)

- Fixed regression in `PreparedStatement.setString()` and eastern character encodings. (Bug #3520)

- Don't truncate `BLOB` or `CLOB` values when using `setBytes()` and `setBinary/CharacterStream()`. (Bug #2670)

- Map duplicate key and foreign key errors to SQLState of `23000`.

- Allow `java.util.Date` to be sent in as parameter to `PreparedStatement.setObject()`, converting it to a `Timestamp` to maintain full precision. . (Bug #103)

- Add unsigned attribute to `DatabaseMetaData.getColumns()` output in the `TYPE_NAME` column.

- Not specifying database in URL caused `MalformedURL` exception. (Bug #3554)

- Added `failOverReadOnly` property, to enable the user to configure the state of the connection (read-only/writable) when failed over.

- Dynamically configure character set mappings for field-level character sets on MySQL-4.1.0 and newer using `SHOW COLLATION` when connecting. (Bug #2670)

- Map `binary` character set to `US-ASCII` to support `DATETIME` charset recognition for servers >= 4.1.2. (Bug #2670)

- Inconsistent reporting of data type. The server still doesn't return all types for *BLOBs *TEXT correctly, so the driver won't return those correctly. (Bug #3570)

- Auto-convert MySQL encoding names to Java encoding names if used for `characterEncoding` property. (Bug #3554)

- Backport documentation tooling from 3.1 branch.

- Use `junit.textui.TestRunner` for all unit tests (to enable them to be run from the command line outside of Ant or Eclipse). (Bug #3554)

- Renamed `StringUtils.escapeSJISByteStream()` to more appropriate `escapeEasternUnicodeByteStream()`. (Bug #3511)

- Use `charsetnr` returned during connect to encode queries before issuing `SET NAMES` on MySQL >= 4.1.0. (Bug #2670)

- `StringUtils.escapeSJISByteStream()` not covering all eastern double-byte charsets correctly. (Bug #3511)

- `DBMD.getSQLStateType()` returns incorrect value. (Bug #3520)

- Add helper methods to `ResultSetMetaData` (`getColumnCharacterEncoding()` and `getColumnCharacterSet()`) to permit end users to see what charset the driver thinks it should be using for the column. (Bug #2670)

- Don't escape SJIS/GBK/BIG5 when using MySQL-4.1 or newer.

- Added encoding names that are recognized on some JVMs to fix case where they were reverse-mapped to MySQL encoding names incorrectly. (Bug #3554)

- Only set `character_set_results` for MySQL >= 4.1.0. (Bug #2670)

- Return creating statement for `ResultSets` created by `getGeneratedKeys()`. (Bug #2957)

- `UpdatableResultSet` not picking up default values for `moveToInsertRow()`. (Bug #3557)

- Allow `url` parameter for `MysqlDataSource` and `MysqlConnectionPool DataSource` so that passing of other properties is possible from inside appservers.

A.4.7. Changes in MySQL Connector/J 3.0.11 (19 February 2004)

Bugs Fixed

- Return `java.lang.Integer` for `TINYINT` and `SMALLINT` types from `ResultSet-MetaData.getColumnClassName()`. (Bug #2852)

- Return `java.lang.Double` for `FLOAT` type from `ResultSetMetaData.getColumnClassName()`. (Bug #2855)

- Added `useUnbufferedInput` parameter, and now use it by default (due to JVM issue http://developer.java.sun.com/developer/bugParade/bugs/4401235.html) (Bug #2578)

- `AutoReconnect` time was growing faster than exponentially. (Bug #2447)

- Fixed failover always going to last host in list. (Bug #2578)

- Detect `on`/`off` or `1`, `2`, `3` form of `lower_case_table_names` value on server. (Bug #2578)

- Return `[B` instead of `java.lang.Object` for `BINARY`, `VARBINARY` and `LONGVARBINARY` types from `ResultSet-MetaData.getColumnClassName()` (JDBC compliance). (Bug #2855)

- Issue connection events on all instances created from a `ConnectionPoolDataSource`. (Bug #2855)

- Trigger a `SET NAMES utf8` when encoding is forced to `utf8` *or* `utf-8` using the `characterEncoding` property. Previously, only the Java-style encoding name of `utf-8` would trigger this.

A.4.8. Changes in MySQL Connector/J 3.0.10 (13 January 2004)

Bugs Fixed

- Fixed regression of `Statement.getGeneratedKeys()` and `REPLACE` statements. (Bug #1576)

- Implement `ResultSet.updateClob()`. (Bug #1913)

- Enable caching of the parsing stage of prepared statements using the `cachePrepStmts`, `prepStmtCacheSize`, and `prepStmtCacheSqlLimit` properties (disabled by default). (Bug #2006)

- Foreign Keys column sequence is not consistent in `DatabaseMetaData.getImported/Exported/CrossReference()`. (Bug #1731)

- Subsequent call to `ResultSet.updateFoo()` causes NPE if result set is not updatable. (Bug #1630)

- Backported fix for aliased tables and `UpdatableResultSets` in `checkUpdatability()` method from 3.1 branch. (Bug

#1534)

- Fixed security exception when used in Applets (applets can't read the system property `file.encoding` which is needed for `LOAD DATA LOCAL INFILE`). (Bug #2006)

- Speed up parsing of `PreparedStatements`, try to use one-pass whenever possible. (Bug #2006)

- Support escape sequence {fn convert ... }. (Bug #1914)

- Autoreconnect code didn't set catalog upon reconnect if it had been changed. (Bug #1913)

- Fixed exception `Unknown character set 'danish'` on connect with JDK-1.4.0 (Bug #2006)

- Fixed mappings in SQLError to report deadlocks with SQLStates of `41000`. (Bug #2006)

- `ResultSet.getObject()` on `TINYINT` and `SMALLINT` columns should return Java type `Integer`. (Bug #1913)

- `ArrayIndexOutOfBounds` when parameter number == number of parameters + 1. (Bug #1958)

- Added more descriptive error message `Server Configuration Denies Access to DataSource`, as well as retrieval of message from server. (Bug #1913)

- "Friendlier" exception message for `PacketTooLargeException`. (Bug #1534)

- Don't count quoted IDs when inside a 'string' in `PreparedStatement` parsing. (Bug #1511)

- Removed static synchronization bottleneck from instance factory method of `SingleByteCharsetConverter`. (Bug #2006)

- Removed static synchronization bottleneck from `PreparedStatement.setTimestamp()`. (Bug #2006)

- `ResultSet.findColumn()` should use first matching column name when there are duplicate column names in `SELECT` query (JDBC-compliance). (Bug #2006)

- `maxRows` property would affect internal statements, so check it for all statement creation internal to the driver, and set to 0 when it is not. (Bug #2006)

- Barge blobs and split packets not being read correctly. (Bug #1576)

- `ResultSetMetaData.isCaseSensitive()` returned wrong value for `CHAR`/`VARCHAR` columns. (Bug #1913)

- `DatabaseMetaData.getSystemFunction()` returning bad function `VResultsSion`. (Bug #1775)

- Use constants for SQLStates. (Bug #2006)

- Fix for 4.1.1-style authentication with no password. (Bug #1630)

- Cross-database updatable result sets are not checked for updatability correctly. (Bug #1592)

- Statements being created too many times in `DBMD.extractForeignKeyFromCreateTable()`. (Bug #1925)

- Map charset `ko18_ru` to `ko18r` when connected to MySQL-4.1.0 or newer. (Bug #2006)

- Fix for `ArrayIndexOutOfBounds` exception when using `Statement.setMaxRows()`. (Bug #1695)

- Connection property `maxRows` not honored. (Bug #1933)

- Added `alwaysClearStream` connection property, which causes the driver to always empty any remaining data on the input stream before each query. (Bug #1913)

- `DatabaseMetaData.getColumns()` should return `Types.LONGVARCHAR` for MySQL `LONGTEXT` type. (Bug #1592)

- Ensure that `Buffer.writeString()` saves room for the `\0`. (Bug #2006)

A.4.9. Changes in MySQL Connector/J 3.0.9 (07 October 2003)

Bugs Fixed

- Issue exception on `ResultSet.getXXX()` on empty result set (wasn't caught in some cases). (Bug #848)

- Made `databaseName`, `portNumber`, and `serverName` optional parameters for `MysqlDataSourceFactory`. (Bug #1246)

- `ResultSet.get/setString` mashing char 127. (Bug #1247)

- Double-escaping of `'\'` when charset is SJIS or GBK and `'\'` appears in nonescaped input. (Bug #879)

- Don't hide messages from exceptions thrown in I/O layers. (Bug #848)

- Support `InnoDB` constraint names when extracting foreign key information in `DatabaseMetaData` (implementing ideas from Parwinder Sekhon). (Bug #664, Bug #517)

- Fixed `CLOB.truncate()`. (Bug #1130)

- Fixed `ResultSet.previous()` behavior to move current position to before result set when on first row of result set. (Bug #496)

- Backported 4.1 protocol changes from 3.1 branch (server-side SQL states, new field information, larger client capability flags, connect-with-database, and so forth). (Bug #664, Bug #517)

- Fixed `Statement` and `PreparedStatement` issuing bogus queries when `setMaxRows()` had been used and a `LIMIT` clause was present in the query. (Bug #496)

- Added property to "clobber" streaming results, by setting the `clobberStreamingResults` property to `true` (the default is `false`). This will cause a "streaming" `ResultSet` to be automatically closed, and any outstanding data still streaming from the server to be discarded if another query is executed before all the data has been read from the server. (Bug #1247)

- When emptying input stream of unused rows for "streaming" result sets, have the current thread `yield()` every 100 rows to not monopolize CPU time. (Bug #879)

- Fixed deadlock issue with `Statement.setMaxRows()`. (Bug #1099)

- Don't wrap `SQLExceptions` in `RowDataDynamic`. (Bug #688)

- Added `com.mysql.jdbc.util.BaseBugReport` to help creation of testcases for bug reports. (Bug #1247)

- Clip +/- INF (to smallest and largest representative values for the type in MySQL) and NaN (to 0) for `setDouble`/`setFloat()`, and issue a warning on the statement when the server does not support +/- INF or NaN. (Bug #884)

- Don't try and reset isolation level on reconnect if MySQL doesn't support them. (Bug #688)

- `DatabaseMetaData.getColumns()` getting confused about the keyword "set" in character columns. (Bug #1099)

- Fixed regression in large split-packet handling. (Bug #848)

- Fixed test for end of buffer in `Buffer.readString()`.

- The `insertRow` in an `UpdatableResultSet` is now loaded with the default column values when `moveToInsertRow()` is called. (Bug #688)

- Better diagnostic error messages in exceptions for "streaming" result sets. (Bug #848)

- Backported authentication changes for 4.1.1 and newer from 3.1 branch. (Bug #1247)

- `DatabaseMetaData.getColumns()` wasn't returning `NULL` for default values that are specified as `NULL`. (Bug #688)

- Don't change timestamp TZ twice if `useTimezone==true`. (Bug #774)

- `refreshRow` didn't work when primary key values contained values that needed to be escaped (they ended up being doubly escaped). (Bug #661)

- Fix `UpdatableResultSet` to return values for `getXXX()` when on insert row. (Bug #675)

- Don't fire connection closed events when closing pooled connections, or on `PooledConnection.getConnection()` with already open connections. (Bug #884)

- Change default statement type/concurrency to `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY` (spec compliance). (Bug #688)

- Faster date handling code in `ResultSet` and `PreparedStatement` (no longer uses `Date` methods that synchronize on static calendars).

- Optimized `CLOB.setChracterStream()`. (Bug #1131)

A.4.10. Changes in MySQL Connector/J 3.0.8 (23 May 2003)

Bugs Fixed

- Fixed SJIS encoding bug, thanks to Naoto Sato. (Bug #378)

- Allow bogus URLs in `Driver.getPropertyInfo()`.

- Fix problem detecting server character set in some cases. (Bug #378)

- Allow multiple calls to `Statement.close()`. (Bug #378)

- Return correct number of generated keys when using `REPLACE` statements. (Bug #378)

- Unicode character 0xFFFF in a string would cause the driver to throw an `ArrayOutOfBoundsException`. . (Bug #378)

- Return list of generated keys when using multi-value `INSERTS` with `Statement.getGeneratedKeys()`.

- Fixed result set not getting set for `Statement.executeUpdate()`, which affected `getGeneratedKeys()` and `getUpdateCount()` in some cases.

- Changed Ant target `compile-core` to `compile-driver`, and made testsuite compilation a separate target.

- Fix infinite loop with `Connection.cleanup()`.

- Fix row data decoding error when using *very* large packets. (Bug #378)

- Optimized row data decoding. (Bug #378)

- Use JVM charset with file names and `LOAD DATA [LOCAL] INFILE`.

- Issue exception when operating on an already closed prepared statement. (Bug #378)

- Optimized usage of `EscapeProcessor`. (Bug #378)

A.4.11. Changes in MySQL Connector/J 3.0.7 (08 April 2003)

Bugs Fixed

- Throw `SQLExceptions` when trying to do operations on a forcefully closed `Connection` (that is, when a communication link failure occurs).

- Fixed `ResultSet.getTimestamp()` when underlying field is of type `DATE`.

- Fixed escaping of 0x5c (`'\'`) character for GBK and Big5 charsets.

- Don't reset `Connection.isReadOnly()` when autoReconnecting.

- Ensure that packet size from `alignPacketSize()` does not exceed `max_allowed_packet` (JVM bug)

- Don't pick up indexes that start with `pri` as primary keys for `DBMD.getPrimaryKeys()`.

- Fixed `LOAD DATA LOCAL INFILE` bug when file > `max_allowed_packet`.

- Updatable `ResultSets` can now be created for aliased tables/columns when connected to MySQL-4.1 or newer.

- Fixed missing conversion for `YEAR` type in `ResultSetMetaData.getColumnTypeName()`.

- Remove synchronization from `Driver.connect()` and `Driver.acceptsUrl()`.

- `IOExceptions` during a transaction now cause the `Connection` to be closed.

- Fixed `StringIndexOutOfBoundsException` in `PreparedStatement.setClob()`.

- Fixed `MysqlPooledConnection.close()` calling wrong event type.

- 4.1 Column Metadata fixes.

- You can now toggle profiling on/off using `Connection.setProfileSql(boolean)`.

- Fixed charset issues with database metadata (charset was not getting set correctly).

A.4.12. Changes in MySQL Connector/J 3.0.6 (18 February 2003)

Bugs Fixed

- Add "window" of different `NULL` sorting behavior to `DBMD.nullsAreSortedAtStart` (4.0.2 to 4.0.10, true; otherwise, no).

- Implemented `Blob.setBytes()`. You still need to pass the resultant `Blob` back into an updatable `ResultSet` or `Prepared-Statement` to persist the changes, because MySQL does not support "locators".

- More checks added in `ResultSet` traversal method to catch when in closed state.

- Fixed `ResultSetMetaData.isWritable()` to return correct value.

- Clean up `Statement` query/method mismatch tests (that is, `INSERT` not permitted with `.executeQuery()`).

- Allow ignoring of warning for "non transactional tables" during rollback (compliance/usability) by setting `ignoreNonTxTables` property to `true`.

- Fixed `ResultSetMetaData` to return `""` when catalog not known. Fixes `NullPointerExceptions` with Sun's `Cached-RowSet`.

- Fixed `SQLExceptions` getting swallowed on initial connect.

- Fixed `DBMD.getTypeInfo()` and `DBMD.getColumns()` returning different value for precision in `TEXT` and `BLOB` types.

- Fixed `Statement.setMaxRows()` to stop sending `LIMIT` type queries when not needed (performance).

- Backported 4.1 charset field info changes from Connector/J 3.1.

A.4.13. Changes in MySQL Connector/J 3.0.5 (22 January 2003)

Bugs Fixed

- Greatly reduce memory required for `setBinaryStream()` in `PreparedStatements`.

- Retrieve `TX_ISOLATION` from database for `Connection.getTransactionIsolation()` when the MySQL version supports it, instead of an instance variable.

- Quote table names in `DatabaseMetaData.getColumns()`, `getPrimaryKeys()`, `getIndexInfo()`, `getBestRow-Identifier()`.

- Fixed `Buffer.fastSkipLenString()` causing `ArrayIndexOutOfBounds` exceptions with some queries when unpack-

ing fields.

- Added update options for foreign key metadata.

- Implemented an empty `TypeMap` for `Connection.getTypeMap()` so that some third-party apps work with MySQL (IBM WebSphere 5.0 Connection pool).

- Added missing `LONGTEXT` type to `DBMD.getColumns()`.

- Fixed `ResultSet.isBeforeFirst()` for empty result sets.

A.4.14. Changes in MySQL Connector/J 3.0.4 (06 January 2003)

Bugs Fixed

- Added support for quoted identifiers in `PreparedStatement` parser.

- Added `strictUpdates` property to enable control of amount of checking for "correctness" of updatable result sets. Set this to `false` if you want faster updatable result sets and you know that you create them from `SELECT` statements on tables with primary keys and that you have selected all primary keys in your query.

- Reduce memory footprint of `PreparedStatements` by sharing outbound packet with `MysqlIO`.

- Added support for 4.0.8-style large packets.

- Added quoted identifiers to database names for `Connection.setCatalog`.

- Fixed `PreparedStatement.executeBatch()` parameter overwriting.

- Streamlined character conversion and `byte[]` handling in `PreparedStatements` for `setByte()`.

A.4.15. Changes in MySQL Connector/J 3.0.3 (17 December 2002)

Bugs Fixed

- Changed `SingleByteCharConverter` to use lazy initialization of each converter.

- Fixed `ResultSetMetaData.getPrecision()` returning incorrect values for some floating-point types.

- Check for connection closed in more `Connection` methods (`createStatement`, `prepareStatement`, `setTransactionIsolation`, `setAutoCommit`).

- Some MySQL-4.1 protocol support (extended field info from selects).

- More robust implementation of updatable result sets. Checks that *all* primary keys of the table have been selected.

- `DBMD.getImported/ExportedKeys()` now handles multiple foreign keys per table.

- Honor `lower_case_table_names` when enabled in the server when doing table name comparisons in `DatabaseMetaData` methods.

- Added `CLIENT_LONG_FLAG` to be able to get more column flags (`isAutoIncrement()` being the most important).

- Substitute `'?'` for unknown character conversions in single-byte character sets instead of `'\0'`.

- Because of above, implemented `ResultSetMetaData.isAutoIncrement()` to use `Field.isAutoIncrement()`.

- Fixed charset handling in `Fields.java`.

- Fixed `ResultSetMetaData.getColumnTypeName()` returning `BLOB` for `TEXT` and `TEXT` for `BLOB` types.

- Implemented `Connection.nativeSQL()`.

- LOAD DATA LOCAL INFILE ... now works, if your server is configured to permit it. Can be turned off with the allow-LoadLocalInfile property (see the README).

- Use nonaliased table/column names and database names to fully qualify tables and columns in UpdatableResultSet (requires MySQL-4.1 or newer).

- Changed charsToByte in SingleByteCharConverter to be nonstatic.

- NamedPipeSocketFactory now works (only intended for Windows), see README for instructions.

- Fixed Buffer.isLastDataPacket() for 4.1 and newer servers.

- More robust escape tokenizer: Recognize -- comments, and permit nested escape sequences (see test-suite.EscapeProcessingTest).

- Allow user to alter behavior of Statement/PreparedStatement.executeBatch() using continueBatchOnError property (defaults to true).

A.4.16. Changes in MySQL Connector/J 3.0.2 (08 November 2002)

Bugs Fixed

- Implemented Clob.truncate().

- Properly restore connection properties when autoReconnecting or failing-over, including autoCommit state, and isolation level.

- Fixed issue when calling Statement.setFetchSize() when using arbitrary values.

- Added queriesBeforeRetryMaster property that specifies how many queries to issue when failed over before attempting to reconnect to the master (defaults to 50).

- Fixed ResultSet.setFetchDirection(FETCH_UNKNOWN).

- Added driver property useHostsInPrivileges. Defaults to true. Affects whether or not @hostname will be used in DB-MD.getColumn/TablePrivileges.

- Fixed ResultSet.isLast() for empty result sets (should return false).

- Fixed various non-ASCII character encoding issues.

- Added connectTimeout parameter that enables users of JDK-1.4 and newer to specify a maximum time to wait to establish a connection.

- Fixed start position off-by-1 error in Clob.getSubString().

- Fixed issue with updatable result sets and PreparedStatements not working.

- PreparedStatement now honors stream lengths in setBinary/Ascii/Character Stream() unless you set the connection property useStreamLengthsInPrepStmts to false.

- Escape 0x5c character in strings for the SJIS charset.

- Removed some not-needed temporary object creation by smarter use of Strings in EscapeProcessor, Connection and DatabaseMetaData classes.

- Implemented ResultSet.updateBlob().

- Fixed incorrect conversion in ResultSet.getLong().

- Fixed UnsupportedEncodingException thrown when "forcing" a character encoding using properties.

- Removed duplicate code from UpdatableResultSet (it can be inherited from ResultSet, the extra code for each method to handle updatability I thought might someday be necessary has not been needed).

- Implemented `Clob.setAsciiStream()`.

- Fixed `DBMD.supportsResultSetConcurrency()` so that it returns `true` for `Result-Set.TYPE_SCROLL_INSENSITIVE` and `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`.

- Failover and `autoReconnect` work only when the connection is in an `autoCommit(false)` state, to stay transaction-safe.

- Implemented `Clob.setCharacterStream()`.

- Use `SHOW CREATE TABLE` when possible for determining foreign key information for `DatabaseMetaData`. Also enables cascade options for `DELETE` information to be returned.

- All `DBMD` result set columns describing schemas now return `NULL` to be more compliant with the behavior of other JDBC drivers for other database systems (MySQL does not support schemas).

- Added SSL support. See `README` for information on how to use it.

- Added `com.mysql.jdbc.MiniAdmin` class, which enables you to send `shutdown` command to MySQL server. This is intended to be used when "embedding" Java and MySQL server together in an end-user application.

- Implemented `Clob.setString()`.

A.4.17. Changes in MySQL Connector/J 3.0.1 (21 September 2002)

Bugs Fixed

- Added `paranoid` parameter, which sanitizes error messages by removing "sensitive" information from them (such as host names, ports, or user names), as well as clearing "sensitive" data structures when possible.

- Added `LOCAL TEMPORARY` to table types in `DatabaseMetaData.getTableTypes()`.

- Added `socketTimeout` parameter to URL.

- `Connection.close()` issues `rollback()` when `getAutoCommit()` is `false`.

- `Connection.isClosed()` no longer "pings" the server.

- Added limited `Clob` functionality (`ResultSet.getClob()`, `PreparedStatement.setClob()`, `PreparedStatement.setObject(Clob)`.

- Implemented `ResultSet.getCharacterStream()`.

- Massive code clean-up to follow Java coding conventions (the time had come).

- Fixed `ResultSet.getRow()` off-by-one bug.

- Fixed `RowDataStatic.getAt()` off-by-one bug.

- Fixed `ResultSetMetaData.isSigned()` for `TINYINT` and `BIGINT`.

- Charsets now automatically detected. Optimized code for single-byte character set conversion.

A.4.18. Changes in MySQL Connector/J 3.0.0 (31 July 2002)

Bugs Fixed

- Performance enhancements: Driver is now 50–100% faster in most situations, and creates fewer temporary objects.

- **!!! LICENSE CHANGE !!!** The driver is now GPL. If you need non-GPL licenses, please contact me `<mark@mysql.com>`.

- Overall speed improvements using controlling transient object creation in `MysqlIO` class when reading packets.

- JDBC-3.0 functionality including `Statement/PreparedStatement.getGeneratedKeys()` and `Result-Set.getURL()`.

- Performance improvements in string handling and field metadata creation (lazily instantiated) contributed by Alex Twisleton-Wykeham-Fiennes.

- Better checking for closed connections in `Statement` and `PreparedStatement`.

- Support for large packets (new addition to MySQL-4.0 protocol), see `README` for more information.

- Repackaging: New driver name is `com.mysql.jdbc.Driver`, old name still works, though (the driver is now provided by MySQL-AB).

- Added multi-host failover support (see `README`).

- `ResultSet.getDate/Time/Timestamp` now recognizes all forms of invalid values that have been set to all zeros by MySQL (SF bug 586058).

- JDBC Compliance: Passes all tests besides stored procedure tests.

- Testsuite now uses Junit (which you can get from http://www.junit.org.

- Support for streaming (row-by-row) result sets (see `README`) Thanks to Doron.

- Float types now reported as `java.sql.Types.FLOAT` (SF bug 579573).

- `ResultSet.getTimestamp()` now works for `DATE` types (SF bug 559134).

- Fix and sort primary key names in `DBMetaData` (SF bugs 582086 and 582086).

- The driver now only works with JDK-1.2 or newer.

- General source-code cleanup.

A.5. Changes in MySQL Connector/J 2.0.x

A.5.1. Changes in MySQL Connector/J 2.0.14 (16 May 2002)

Bugs Fixed

- Quoted identifiers not used if server version does not support them. Also, if server started with `--ansi` or `--sql-mode=ANSI_QUOTES`, """ will be used as an identifier quote character, otherwise "'" will be used.

- More code cleanup.

- `PreparedStatement` now releases resources on `.close()`. (SF bug 553268)

- Added SQL profiling (to `STDERR`). Set `profileSql=true` in your JDBC URL. See `README` for more information.

- `LogicalHandle.isClosed()` calls through to physical connection.

- Fixed typo for `relaxAutoCommit` parameter.

- `ResultSet.getDouble()` now uses code built into JDK to be more precise (but slower).

A.5.2. Changes in MySQL Connector/J 2.0.13 (24 April 2002)

Bugs Fixed

- `ResultSetMetaData.getColumnClassName()` now implemented.

- `DBMetaData.getIndexInfo()` - bad PAGES fixed. (SF BUG 542201)

- Rudimentary version of `Statement.getGeneratedKeys()` from JDBC-3.0 now implemented (you need to be using JDK-1.4 for this to work, I believe).

- More code cleanup.

- `PreparedStatement.toString()` fixed. (SF bug 534026)

- Added `set`/`getPortNumber()` to `DataSource(s)`. (SF bug 548167)

- Added `setURL()` to `MySQLXADataSource`. (SF bug 546019)

- Faster blob escaping for `PrepStmt`.

- Fixed unicode chars being read incorrectly. (SF bug 541088)

A.5.3. Changes in MySQL Connector/J 2.0.12 (07 April 2002)

Bugs Fixed

- General code cleanup.

- Fixed `getRow()` bug (527165) in `ResultSet`.

- `ResultSet.refreshRow()` implemented.

- Added `getIdleFor()` method to `Connection` and `MysqlLogicalHandle`.

- Added `getTable/ColumnPrivileges()` to DBMD (fixes 484502).

- `ResultSet`: Fixed updatability (values being set to `null` if not updated).

- Fixes for `ResultSet` updatability in `PreparedStatement`.

- Added support for `YEAR` type (533556).

- `DataSources` - fixed `setUrl` bug (511614, 525565), wrong datasource class name (532816, 528767).

- Relaxed synchronization in all classes, should fix 520615 and 520393.

- `ResultSet.insertRow()` should now detect auto_increment fields in most cases and use that value in the new row. This detection will not work in multi-valued keys, however, due to the fact that the MySQL protocol does not return this information.

- Added support for `BIT` types (51870) to `PreparedStatement`.

- Added identifier quoting to all `DatabaseMetaData` methods that need them (should fix 518108).

- Fixed time zone off-by-1-hour bug in `PreparedStatement` (538286, 528785).

- Added new types to `getTypeInfo()`, fixed existing types thanks to Al Davis and Kid Kalanon.

- Fixed `testsuite.Traversal afterLast()` bug, thanks to Igor Lastric.

A.5.4. Changes in MySQL Connector/J 2.0.11 (27 January 2002)

Bugs Fixed

- More changes to fix `Unexpected end of input stream` errors when reading `BLOB` values. This should be the last fix.

- Fixed missing `DELETE_RULE` value in `DBMD.getImported/ExportedKeys()` and `getCrossReference()`.

- Full synchronization of `Statement.java`.

A.5.5. Changes in MySQL Connector/J 2.0.10 (24 January 2002)

Bugs Fixed

- Fixed spurious `Unexpected end of input stream` errors in `MysqlIO` (bug 507456).

- Fixed null-pointer-exceptions when using `MysqlConnectionPoolDataSource` with Websphere 4 (bug 505839).

A.5.6. Changes in MySQL Connector/J 2.0.9 (13 January 2002)

Bugs Fixed

- Full synchronization on methods modifying instance and class-shared references, driver should be entirely thread-safe now (please let me know if you have problems).

- Fixed off-by-one-hour error in `PreparedStatement.setTimestamp()` (bug 491577).

- Implementation of `DatabaseMetaData.getExported/ImportedKeys()` and `getCrossReference()`.

- Report batch update support through `DatabaseMetaData` (bug 495101).

- `Ant` build was corrupting included `jar` files, fixed (bug 487669).

- Removed concatenation support from driver (the `||` operator), as older versions of VisualAge seem to be the only thing that use it, and it conflicts with the logical `||` operator. You will need to start `mysqld` with the `--ansi` flag to use the `||` operator as concatenation (bug 491680).

- Fixed quoting error with escape processor (bug 486265).

- `DataSource` implementations moved to `org.gjt.mm.mysql.jdbc2.optional` package, and (initial) implementations of `PooledConnectionDataSource` and `XADataSource` are in place (thanks to Todd Wolff for the implementation and testing of `PooledConnectionDataSource` with IBM WebSphere 4).

- Fixed casting bug in `PreparedStatement` (bug 488663).

- Added detection of network connection being closed when reading packets (thanks to Todd Lizambri).

- Fixed extra memory allocation in `MysqlIO.readPacket()` (bug 488663).

A.5.7. Changes in MySQL Connector/J 2.0.8 (25 November 2001)

Bugs Fixed

- `PreparedStatement.setAnyNumericType()` now handles positive exponents correctly (adds + so MySQL can understand it).

- Batch updates now supported (thanks to some inspiration from Daniel Rall).

- `DatabaseMetaData.getPrimaryKeys()` and `getBestRowIdentifier()` are now more robust in identifying primary keys (matches regardless of case or abbreviation/full spelling of `Primary Key` in `Key_type` column).

- `XADataSource`/`ConnectionPoolDataSource` code (experimental)

A.5.8. Changes in MySQL Connector/J 2.0.7 (24 October 2001)

Bugs Fixed

- Fixed `ResultSet.isAfterLast()` always returning `false`.

- PreparedStatement.setBoolean() will use 1/0 for values if your MySQL version is 3.21.23 or higher.

- Updatable result sets now correctly handle `NULL` values in fields.

- Fixed `DatabaseMetaData.supportsTransactions()`, and `supportsTransactionIsolationLevel()` and `getTypeInfo() SQL_DATETIME_SUB` and `SQL_DATA_TYPE` fields not being readable.

- PreparedStatement.setDouble() now uses full-precision doubles (reverting a fix made earlier to truncate them).

- Fixed time zone issue in `PreparedStatement.setTimestamp()`. (thanks to Erik Olofsson)

- Fixed dangling socket problem when in high availability (`autoReconnect=true`) mode, and finalizer for `Connection` will close any dangling sockets on GC.

- Fixed `ResultSetMetaData.getPrecision()` returning one less than actual on newer versions of MySQL.

- `ResultSet.getBlob()` now returns `null` if column value was `null`.

- Capitalize type names when `capitalizeTypeNames=true` is passed in URL or properties (for WebObjects. (thanks to Anjo Krank)

- `PreparedStatement.setCharacterStream()` now implemented

- Fixed `PreparedStatement` generating SQL that would end up with syntax errors for some queries.

- Initial transaction isolation level read from database (if available). (thanks to Dmitry Vereshchagin)

- Character sets read from database if `useUnicode=true` and `characterEncoding` is not set. (thanks to Dmitry Vereshchagin)

A.5.9. Changes in MySQL Connector/J 2.0.6 (16 June 2001)

Bugs Fixed

- Fixed case-sensitive column names in `ResultSet.java`.

- Fixed `PreparedStatement` parameter checking.

A.5.10. Changes in MySQL Connector/J 2.0.5 (13 June 2001)

Bugs Fixed

- Fixed `ResultSet.getBlob() ArrayIndex` out-of-bounds.

- Fixed NPE on `PreparedStatement.executeUpdate()` when all columns have not been set.

- Fixed `ArrayIndexOutOfBounds` when sending large `BLOB` queries. (Max size packet was not being set)

- Fixed `ResultSetMetaData.getColumnTypeName` for `TEXT`/`BLOB`.

- `getObject()` on `ResultSet` correctly does `TINYINT`->`Byte` and `SMALLINT`->`Short`.

- `ResultSet` has +/-Inf/inf support.

- `ResultSet.getBoolean()` now recognizes `-1` as `true`.

- `DataBaseMetaData.getCrossReference()` no longer `ArrayIndexOOB`.

- Added `ISOLATION` level support to `Connection.setIsolationLevel()`

- Fixed data parsing of `TIMESTAMP` values with 2-digit years.

- Added `Byte` to `PreparedStatement.setObject()`.

- `ResultSet.insertRow()` works now, even if not all columns are set (they will be set to `NULL`).

A.5.11. Changes in MySQL Connector/J 2.0.3 (03 December 2000)

Bugs Fixed

- Implemented `getBigDecimal()` without scale component for JDBC2.

- Added `ultraDevHack` URL parameter, set to `true` to enable (broken) Macromedia UltraDev to use the driver.

- Added detection of -/+INF for doubles.

- Fixed incorrect detection of `MAX_ALLOWED_PACKET`, so sending large blobs should work now.

- Fixed off-by-one error in `java.sql.Blob` implementation code.

- Faster ASCII string operations.

- Fixed composite key problem with updatable result sets.

A.5.12. Changes in MySQL Connector/J 2.0.1 (06 April 2000)

Bugs Fixed

- Fixed some issues with updatability support in `ResultSet` when using multiple primary keys.

- No escape processing is done on `PreparedStatements` anymore per JDBC spec.

- Fixed many JDBC-2.0 traversal, positioning bugs, especially with respect to empty result sets. Thanks to Ron Smits, Nick Brook, Cessar Garcia and Carlos Martinez.

- `DatabaseMetaData.getPrimaryKeys()` now works correctly with respect to `key_seq`. Thanks to Brian Slesinsky.

- Fixed `RSMD.isWritable()` returning wrong value. Thanks to Moritz Maass.

- Cleaned up exception handling when driver connects.

- Columns that are of type `TEXT` now return as `Strings` when you use `getObject()`.

A.5.13. Changes in MySQL Connector/J 2.0.0pre5 (21 February 2000)

- Fixed Bad Handshake problem.

A.5.14. Changes in MySQL Connector/J 2.0.0pre4 (10 January 2000)

- Fixes to ResultSet for insertRow() - Thanks to Cesar Garcia

- Fix to Driver to recognize JDBC-2.0 by loading a JDBC-2.0 class, instead of relying on JDK version numbers. Thanks to John Baker.

- Fixed ResultSet to return correct row numbers

- Statement.getUpdateCount() now returns rows matched, instead of rows actually updated, which is more SQL-92 like.

10-29-99

- Statement/PreparedStatement.getMoreResults() bug fixed. Thanks to Noel J. Bergman.

- Added Short as a type to PreparedStatement.setObject(). Thanks to Jeff Crowder

- Driver now automagically configures maximum/preferred packet sizes by querying server.

- Autoreconnect code uses fast ping command if server supports it.

- Fixed various bugs with respect to packet sizing when reading from the server and when alloc'ing to write to the server.

A.5.15. Changes in MySQL Connector/J 2.0.0pre (17 August 1999)

- Now compiles under JDK-1.2. The driver supports both JDK-1.1 and JDK-1.2 at the same time through a core set of classes. The driver will load the appropriate interface classes at runtime by figuring out which JVM version you are using.

- Fixes for result sets with all nulls in the first row. (Pointed out by Tim Endres)

- Fixes to column numbers in SQLExceptions in ResultSet (Thanks to Blas Rodriguez Somoza)

- The database no longer needs to specified to connect. (Thanks to Christian Motschke)

A.6. Changes in MySQL Connector/J 1.2.x and lower

A.6.1. Changes in MySQL Connector/J 1.2b (04 July 1999)

- Better Documentation (in progress), in doc/mm.doc/book1.html

- DBMD now permits null for a column name pattern (not in spec), which it changes to '%'.

- DBMD now has correct types/lengths for getXXX().

- ResultSet.getDate(), getTime(), and getTimestamp() fixes. (contributed by Alan Wilken)

- EscapeProcessor now handles \{ \} and { or } inside quotation marks correctly. (thanks to Alik for some ideas on how to fix it)

- Fixes to properties handling in Connection. (contributed by Juho Tikkala)

- ResultSet.getObject() now returns null for NULL columns in the table, rather than bombing out. (thanks to Ben Grosman)

- ResultSet.getObject() now returns Strings for types from MySQL that it doesn't know about. (Suggested by Chris Perdue)

- Removed DataInput/Output streams, not needed, 1/2 number of method calls per IO operation.

- Use default character encoding if one is not specified. This is a work-around for broken JVMs, because according to spec, EVERY JVM must support "ISO8859_1", but they do not.

- Fixed Connection to use the platform character encoding instead of "ISO8859_1" if one isn't explicitly set. This fixes problems people were having loading the character- converter classes that didn't always exist (JVM bug). (thanks to Fritz Elfert for pointing out this problem)

- Changed MysqlIO to re-use packets where possible to reduce memory usage.

- Fixed escape-processor bugs pertaining to {} inside quotation marks.

A.6.2. Changes in MySQL Connector/J 1.2a (14 April 1999)

- Fixed character-set support for non-Javasoft JVMs (thanks to many people for pointing it out)

- Fixed ResultSet.getBoolean() to recognize 'y' & 'n' as well as '1' & '0' as boolean flags. (thanks to Tim Pizey)

- Fixed ResultSet.getTimestamp() to give better performance. (thanks to Richard Swift)

- Fixed getByte() for numeric types. (thanks to Ray Bellis)

- Fixed DatabaseMetaData.getTypeInfo() for DATE type. (thanks to Paul Johnston)

- Fixed EscapeProcessor for "fn" calls. (thanks to Piyush Shah at locomotive.org)

- Fixed EscapeProcessor to not do extraneous work if there are no escape codes. (thanks to Ryan Gustafson)

- Fixed Driver to parse URLs of the form "jdbc:mysql://host:port" (thanks to Richard Lobb)

A.6.3. Changes in MySQL Connector/J 1.1i (24 March 1999)

- Fixed Timestamps for PreparedStatements

- Fixed null pointer exceptions in RSMD and RS

- Re-compiled with jikes for valid class files (thanks ms!)

A.6.4. Changes in MySQL Connector/J 1.1h (08 March 1999)

- Fixed escape processor to deal with unmatched { and } (thanks to Craig Coles)

- Fixed escape processor to create more portable (between DATETIME and TIMESTAMP types) representations so that it will work with BETWEEN clauses. (thanks to Craig Longman)

- MysqlIO.quit() now closes the socket connection. Before, after many failed connections some OS's would run out of file descriptors. (thanks to Michael Brinkman)

- Fixed NullPointerException in Driver.getPropertyInfo. (thanks to Dave Potts)

- Fixes to MysqlDefs to allow all *text fields to be retrieved as Strings. (thanks to Chris at Leverage)

- Fixed setDouble in PreparedStatement for large numbers to avoid sending scientific notation to the database. (thanks to J.S. Ferguson)

- Fixed getScale() and getPrecision() in RSMD. (contrib'd by James Klicman)

- Fixed getObject() when field was DECIMAL or NUMERIC (thanks to Bert Hobbs)

- DBMD.getTables() bombed when passed a null table-name pattern. Fixed. (thanks to Richard Lobb)

- Added check for "client not authorized" errors during connect. (thanks to Hannes Wallnoefer)

A.6.5. Changes in MySQL Connector/J 1.1g (19 February 1999)

- Result set rows are now byte arrays. Blobs and Unicode work bidirectonally now. The useUnicode and encoding options are implemented now.

- Fixes to PreparedStatement to send binary set by setXXXStream to be sent untouched to the MySQL server.

- Fixes to getDriverPropertyInfo().

A.6.6. Changes in MySQL Connector/J 1.1f (31 December 1998)

- Changed all ResultSet fields to Strings, this should allow Unicode to work, but your JVM must be able to convert between the character sets. This should also make reading data from the server be a bit quicker, because there is now no conversion from StringBuffer to String.

- Changed PreparedStatement.streamToString() to be more efficient (code from Uwe Schaefer).

- URL parsing is more robust (throws SQL exceptions on errors rather than NullPointerExceptions)

- PreparedStatement now can convert Strings to Time/Date values using setObject() (code from Robert Currey).

- IO no longer hangs in Buffer.readInt(), that bug was introduced in 1.1d when changing to all byte-arrays for result sets. (Pointed out by Samo Login)

A.6.7. Changes in MySQL Connector/J 1.1b (03 November 1998)

- Fixes to DatabaseMetaData to allow both IBM VA and J-Builder to work. Let me know how it goes. (thanks to Jac Kersing)

- Fix to ResultSet.getBoolean() for NULL strings (thanks to Barry Lagerweij)

- Beginning of code cleanup, and formatting. Getting ready to branch this off to a parallel JDBC-2.0 source tree.

- Added "final" modifier to critical sections in MysqlIO and Buffer to allow compiler to inline methods for speed.

9-29-98

- If object references passed to setXXX() in PreparedStatement are null, setNull() is automatically called for you. (Thanks for the suggestion goes to Erik Ostrom)

- setObject() in PreparedStatement will now attempt to write a serialized representation of the object to the database for objects of Types.OTHER and objects of unknown type.

- Util now has a static method readObject() which given a ResultSet and a column index will re-instantiate an object serialized in the above manner.

A.6.8. Changes in MySQL Connector/J 1.1 (02 September 1998)

- Got rid of "ugly hack" in MysqlIO.nextRow(). Rather than catch an exception, Buffer.isLastDataPacket() was fixed.

- Connection.getCatalog() and Connection.setCatalog() should work now.

- Statement.setMaxRows() works, as well as setting by property maxRows. Statement.setMaxRows() overrides maxRows set using properties or url parameters.

- Automatic re-connection is available. Because it has to "ping" the database before each query, it is turned off by default. To use it, pass in "autoReconnect=true" in the connection URL. You may also change the number of reconnect tries, and the initial timeout value using "maxReconnects=n" (default 3) and "initialTimeout=n" (seconds, default 2) parameters. The timeout is an exponential backoff type of timeout; for example, if you have initial timeout of 2 seconds, and maxReconnects of 3, then the driver will timeout 2 seconds, 4 seconds, then 16 seconds between each re-connection attempt.

A.6.9. Changes in MySQL Connector/J 1.0 (24 August 1998)

- Fixed handling of blob data in Buffer.java

- Fixed bug with authentication packet being sized too small.

- The JDBC Driver is now under the LGPL

8-14-98

- Fixed Buffer.readLenString() to correctly read data for BLOBS.

- Fixed PreparedStatement.stringToStream to correctly read data for BLOBS.

- Fixed PreparedStatement.setDate() to not add a day. (above fixes thanks to Vincent Partington)

- Added URL parameter parsing (?user=... and so forth).

A.6.10. Changes in MySQL Connector/J 0.9d (04 August 1998)

- Big news! New package name. Tim Endres from ICE Engineering is starting a new source tree for GNU GPL'd Java software. He's graciously given me the org.gjt.mm package directory to use, so now the driver is in the org.gjt.mm.mysql package scheme. I'm "legal" now. Look for more information on Tim's project soon.

- Now using dynamically sized packets to reduce memory usage when sending commands to the DB.

- Small fixes to getTypeInfo() for parameters, and so forth.

- DatabaseMetaData is now fully implemented. Let me know if these drivers work with the various IDEs out there. I've heard that they're working with JBuilder right now.

- Added JavaDoc documentation to the package.

- Package now available in .zip or .tar.gz.

A.6.11. Changes in MySQL Connector/J 0.9 (28 July 1998)

- Implemented getTypeInfo(). Connection.rollback() now throws an SQLException per the JDBC spec.

- Added PreparedStatement that supports all JDBC API methods for PreparedStatement including InputStreams. Please check this out and let me know if anything is broken.

- Fixed a bug in ResultSet that would break some queries that only returned 1 row.

- Fixed bugs in DatabaseMetaData.getTables(), DatabaseMetaData.getColumns() and DatabaseMetaData.getCatalogs().

- Added functionality to Statement that enables executeUpdate() to store values for IDs that are automatically generated for AUTO_INCREMENT fields. Basically, after an executeUpdate(), look at the SQLWarnings for warnings like "LAST_INSERTED_ID = 'some number', COMMAND = 'your SQL query'". If you are using AUTO_INCREMENT fields in your tables and are executing a lot of executeUpdate()s on one Statement, be sure to clearWarnings() every so often to save memory.

A.6.12. Changes in MySQL Connector/J 0.8 (06 July 1998)

- Split MysqlIO and Buffer to separate classes. Some ClassLoaders gave an IllegalAccess error for some fields in those two classes. Now mm.mysql works in applets and all classloaders. Thanks to Joe Ennis <jce@mail.boone.com> for pointing out the problem and working on a fix with me.

A.6.13. Changes in MySQL Connector/J 0.7 (01 July 1998)

- Fixed DatabaseMetadata problems in getColumns() and bug in switch statement in the Field constructor. Thanks to Costin Mano-lache <costin@tdiinc.com> for pointing these out.

A.6.14. Changes in MySQL Connector/J 0.6 (21 May 1998)

- Incorporated efficiency changes from Richard Swift <Richard.Swift@kanatek.ca> in `MysqlIO.java` and `ResultSet.java`:

- We're now 15% faster than gwe's driver.

- Started working on `DatabaseMetaData`.

- The following methods are implemented:

  - `getTables()`

  - `getTableTypes()`

  - `getColumns()`

  - `getCatalogs()`

Appendix B. Licenses for Third-Party Components

MySQL Connector/J

B.1. Ant-Contrib License

The following software may be included in this product: Ant-Contrib

```
Ant-Contrib
Copyright (c) 2001-2003 Ant-Contrib project. All rights reserved.
Licensed under the Apache 1.1 License Agreement, a copy of which is reproduced below.

The Apache Software License, Version 1.1

Copyright (c) 2001-2003 Ant-Contrib project.  All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in
    the documentation and/or other materials provided with the
    distribution.

 3. The end-user documentation included with the redistribution, if
    any, must include the following acknowlegement:
        "This product includes software developed by the
        Ant-Contrib project (http://sourceforge.net/projects/ant-contrib)."
    Alternately, this acknowlegement may appear in the software itself,
    if and wherever such third-party acknowlegements normally appear.

 4. The name Ant-Contrib must not be used to endorse or promote
    products derived from this software without prior written
    permission. For written permission, please contact
    ant-contrib-developers@lists.sourceforge.net.

 5. Products derived from this software may not be called "Ant-Contrib"
    nor may "Ant-Contrib" appear in their names without prior written
    permission of the Ant-Contrib project.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED.  IN NO EVENT SHALL THE ANT-CONTRIB PROJECT OR ITS
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

B.2. Simple Logging Facade for Java (SLF4J) License

The following software may be included in this product:

```
Simple Logging Facade for Java (SLF4J)

Copyright (c) 2004-2008 QOS.ch
All rights reserved.

Permission is hereby granted, free of charge,
to any person obtaining a copy of this software
and associated documentation files (the "Software"),
to deal in the Software without restriction, including
without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell
```

```
copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the
following conditions:

The above copyright notice and this permission notice
shall be included in all copies or substantial portions
of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN
AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
OR OTHER DEALINGS IN THE SOFTWARE.
```