Contents

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference

Abstract

This document describes how to use the JDBC/MX Driver for NonStop SQL/MX, a type 2 driver, on HP Integrity NonStop™ servers. The JDBC/MX driver provides HP NonStop Server for Java applications with JDBC access to HP NonStop SQL/MX. Where applicable, the JDBC/MX driver conforms to the standard JDBC 3.0 API from Sun Microsystems, Inc.

Product Version

JDBC/MX Driver for NonStop SQL/MX H50

Supported Hardware

All HP Integrity NonStop NS-series servers

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs and H06.04 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
540388-004	August 2009

Document History

Part Number	Product Version	Published
529777-001	JDBC Driver for SQL/MX (JDBC/MX) H10 and V32	May 2005
540388-001	JDBC/MX Driver for NonStop SQL/MX H50	January 2006
540388-002	JDBC/MX Driver for NonStop SQL/MX H50	November 2007
540388-003	JDBC/MX Driver for NonStop SQL/MX H50	November 2008

540388-004	JDBC/MX Driver for NonStop SQL/MX H50	August 2009

Legal Notices

Contents

JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference

Contents

Home

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

About This Document

This section explains these subjects:

New and Changed Information

Changes added to this revision - part number 540388-004:

- Added information on Module File Caching (Module File Caching).

Changes added to this revision - part number 540388-003:

- Supported release statements have been updated to include J-series RVUs.
- Added information about result sets support under Stored Procedures.
- Added unsupported features of the stored procedures in Java under Limitations.

Changes added to this revision - part number 540388-002:

- Added these properties under Setting Properties in the Command Line and Using Additional JDBC/MX Properties:
  - BatchUpdate Exception handling Improvements
  - Statement Level Atomicity
- Updated information about using JDBC connection under Managing Nonblocking JDBC/MX.

Is This Document for You?

This JDBC/MX Driver for NonStop SQL/MX Programmer's Reference is for experienced Java programmers who want to use the JDBC API to access SQL databases with **NonStop SQL/MX**.

This document assumes you are already familiar with **NonStop Server for Java 5**—the Java implementation for use in enterprise Java applications on HP Integrity NonStop servers. NonStop Server for Java 5 is based on the reference Java implementation for Solaris, licensed by HP from Sun Microsystems, Inc. The NonStop Server for Java is a conformant version of a Sun Microsystems JDK as described in the *NonStop Server for Java Programmer's Reference*.

This document also assumes that you are already familiar with the JDBC API from reading literature in the field.

Document Structure

This document is a set of linked **HTML** files (Web pages). Each file corresponds to one of the sections listed and described in the following table.

Document Sections

Section	Description
Table of Contents	Shows the structure of this document in outline form. Each section and subsection name is a link to that section or subsection.
About This Document	Describes the intended audience and the document structure, lists related documents, explains notation conventions and abbreviations, and invites comments.
Introduction to JDBC/MX Driver	Describes the JDBC/MX driver architecture and the API package.
Installing and Verifying JDBC/MX	Describes where to find information about the installation requirements and explains how to verify the JDBC/MX driver the installation.
Accessing SQL Databases with SQL/MX	Explains how to access SQL databases with SQL/MX from the NonStop Server for Java 4 by using the JDBC/MX driver.
Working with BLOB and CLOB Data	Describes working with `BLOB` and `CLOB` data in JDBC applications using the standard interface described in the JDBC 3.0 API specification.
Managing the SQL/MX Tables for BLOB and CLOB Data	Describes the database management (administrative) tasks for adding and managing the tables for `BLOB` and `CLOB` data. The JDBC/MX driver uses SQL/MX tables in implementing support for `BLOB` and `CLOB` data access.
Module File Caching (MFC)	Describes the Module File Caching (MFC) feature of the JDBC/ MX T2 Driver.
JDBC/MX Compliance	Explains how JDBC/MX differs from the Sun Microsystems JDBC standard because of limitations of SQL/MX and the JDBC/MX driver.
JDBC Trace Facility	Explains how to use the JDBC trace facility and how to log SQL statement IDs and corresponding JDBC SQL statements.
Migration	Describes any code changes needed by applications to migrate from using an earlier JDBC/MX driver PVU. to H50.
Messages	Lists messages in numerical SQLCODE order. The descriptions include the SQLCODE, SQLSTATE, message-text, the cause, the effect, and recovery information.

Appendix A. Sample Programs Accessing CLOB and BLOB Data	Shows sample program accessing CLOB and BLOB data.
Glossary	Defines many terms that this document uses.
Index	Lists this document's subjects alphabetically. Each index entry is a link to the appropriate text.
List of Examples	Lists the examples in this document. Each example name is a link to that example.
List of Figures	Lists the figures in this document. Each figure name is a link to that figure.
List of Tables	Lists the tables in this document. Each table name is a link to that table.

Printing This Document

Although reading this document on paper sacrifices the HTML links to other documentation that you can use when viewing this document on your computer screen, you can print this document one file at a time, from either the **NonStop Technical Library** or your Web **browser**. For a list of the sections that make up this document, see Document Structure.

Note: Some browsers require that you reduce the print size to print all the text displayed on the screen.

Related Reading

For background information about the features described in this guide, see the following documents:

- HP NonStop JDBC/MX Driver for NonStop SQL/MX API Reference (javadoc information about the JDBC/MX APIs available in the NonStop Technical Library at docs.hp.com)
- NonStop System Computing Documents
- Sun Microsystems Documents

NonStop System Computing Documents

The following NonStop system computing documents are available in the NonStop Technical Library at docs.hp.com.

- Additional Java-Oriented Products. These documents are available in the Java category under Independent Products in the NonStop Technical Library at docs.hp.com.
  - NonStop Server for Java Programmer's Reference

    This documentation describes NonStop Server for Java 5, a Java environment that supports compact, concurrent, dynamic and portable programs for the enterprise server.
  - NonStop Server for Java Tools Reference Page

This documentation consists of a title page, a table of contents, and the Tools Reference Pages for NonStop Server for Java 5.

- NonStop Server for Java API and Reference

  This documentation contains the documentation for these packages:

  - `com.tandem.os`
  - `com.tandem.tmf`
  - `com.tandem.util`

- JToolkit for NonStop Servers Programmer's Reference

  This documentation describes the JToolkit for NonStop Servers, a set of additional features that work in conjunction with NonStop Server for Java 5.

- JDBC driver for NonStop SQL/MP Programmer's Reference

  This documentation describes how to use the JDBC Driver for SQL/MP (JDBC/MP), which provides Java applications with access to HP NonStop SQL/MP.

- HP NonStop JDBC Type 4 Driver Programmer's Reference

  The HP NonStop JDBC Type 4 documentation describes the JDBC Type 4 driver that allows Java programmers to remotely develop applications deployed on PCs to access NonStop SQL databases through NonStop SQL/MX.

- Inspect Manual

  Documents the Inspect interactive symbolic debugger for **HP NonStop systems**. You can use Inspect to debug Java Native Interface (JNI) code running in a **Java virtual machine (VM)**.

- SQL/MX Documents

  NonStop Server for Java 5 includes JDBC drivers that enable Java programs to interact with NonStop SQL databases with SQL/MX.

  - SQL/MX Programming Manual for Java

    Explains how to use embedded SQL in Java programs (SQLJ programs) to access NonStop SQL databases with SQL/MX.

  - SQL/MX Guide to Stored Procedures in Java

    Describes how to develop and deploy stored procedures in Java (SPJs) in SQL/MX.

  - SQL/MX Quick Start

    Describes basic techniques for using SQL in the SQL/MX conversational interface (MXCI). Also includes information about installing the sample database.

  - SQL/MX Comparison Guide for SQL/MP Users

    Compares SQL/MP and SQL/MX.

  - SQL/MX Installation and Management Guide

    Describes how to install and manage SQL/MX on a NonStop server.

  - SQL/MX Glossary

    Explains the terminology used in SQL/MX documentation.

  - SQL/MX Query Guide

    Explains query execution plans and how to write optimal queries for SQL/MX.

    - SQL/MX Reference Manual

      Describes SQL/MX language elements (such as expressions, predicates, and functions) and the SQL statements that can be run in MXCI or in embedded programs. Also describes MXCI commands and utilities.
    - SQL/MX Messages Manual

      Describes SQL/MX messages.
    - SQL/MX Programming Manual for C and COBOL

      Describes the SQL/MX programmatic interface for ANSI C and COBOL.
    - SQL/MX Data Mining Guide

      Describes the SQL/MX data structures and operations needed for the knowledge-discovery process.
    - SQL/MX Queuing and Publish/Subscribe Services

      Describes how SQL/MX integrates transactional queuing and publish/subscribe services into its database infrastructure.
- TMF Documents
    - TMF Introduction

      Introduces the concepts of transaction processing and the features of the **HP NonStop Transaction Management Facility (TMF)** product.
    - TMF Application Programmer's Guide

      Explains how to design requester and server modules for execution in the TMF programming environment and describes system procedures that are helpful in examining the content of TMF audit trails.

Sun Microsystems Documents

The following documents were available on Sun Microsystems Web sites when the JDBC/MX driver was released.

Note: Sun Microsystems Java 2 Platform Standard Edition JDK 5.0 Documentation is provided on the NonStop Server for Java 5 product distribution CD in an executable file for your convenience in case you cannot get Java documentation from the Sun Microsystems Web sites. The links to Sun Java documentation in the JDBC/MX driver documentation go to the Sun Microsystems Web sites, which provide more extensive documentation than JDK 5.0. HP cannot guarantee the availability of the JDK 5.0 documentation on the Sun Web sites. Also, HP is not responsible for the links or content in the documentation from Sun Microsystems.

- JDBC 3.0 Specification, available for downloading from Sun Microsystems (`http://java.sun.com/products/jdbc/download.html`).
- JDBC API Documentation, includes links to APIs and Tutorials (`http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/index.html`)
- JDBC Data Access API general information (`http://java.sun.com/products/jdbc/index.html`)
- JDBC Data Access API FAQs for JDBC 3.0 (`http://java.sun.com/products/jdbc/faq.html`)
- JDBC API Javadoc Comments
    - Core JDBC 3.0 API in the java.sql package

(`http://java.sun.com/j2se/1.5.0/docs/api/java/sql/package-summary.html`)

- Optional JDBC 3.0 API in the javax.sql package (`http://java.sun.com/j2se/1.5.0/docs/api/javax/sql/package-summary.html`)

Notation Conventions

Bold Type

Bold type within text indicates terms defined in the Glossary. For example:

abstract class

Computer Type

`Computer type` letters within text indicate keywords, reserved words, command names, class names, and method names; enter these items exactly as shown. For example:

```
myfile.c
```

Italic Computer Type

`Italic computer type` letters in syntax descriptions or text indicate variable items that you supply. For example:

```
pathname
```

[ ] Brackets

Brackets enclose optional syntax items. For example:

```
jdb [options]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. Items are separated by vertical lines. For example:

```
where [threadID|all]
```

{ } Braces

A group of items enclosed in braces is a list from which you must choose one item. For example:

```
-c identity {true|false}
```

| Vertical Line

A vertical line separates alternatives in a list that is enclosed in brackets or braces. For example:

```
where [threadID|all]
```

... Ellipsis

An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
print {objectID|objectName} ...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
dump objectID ...
```

Punctuation

Parentheses, commas, equal signs, and other symbols not previously described must be entered as shown. For example:

```
-D propertyName=newValue
```

Item Spacing

Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis

or comma. If there is no space between two items, spaces are not permitted. In the following example, spaces are not permitted before or after the period:

```
subvolume-name.filename
```

Line Spacing

If the syntax of a command is too long to fit on a single line, each line that is to be continued on the next line ends with a backslash (\) and each continuation line begins with a greater-than symbol (>). For example:

```
/usr/bin/c89 -c -g -I /usr/tandem/java/include \
> -I /usr/tandem/java/include/oss  -I . \
> -Wextensions -D_XOPEN_SOURCE_EXTENDED=1 jnative01.c
```

Abbreviations

ANSI. American National Standards Institute

API. application program interface

ASCII. American Standard Code for Information Interchange

BLOB. Binary Large OBject

CD. compact disk

CLOB. Character Large OBject

COBOL. Common Business-Oriented Language

CPU. central processing unit

DCL. Data Control Language

DDL. Data Definition Language

DML. Data Manipulation Language

HTML. Hypertext Markup Language

HTTP. Hypertext Transfer Protocol

IEC. International Electrotechnical Committee

ISO. International Organization for Standardization

JAR. Java Archive

JCK. Java Conformance Kit

JFC. Java Foundation Classes

JDBC. Java Database Connectivity

JDBC/MP. JDBC Driver for SQL/MP

JDBC/MX. JDBC Driver for NonStop SQL/MX

JNDI. Java Naming and Directory Interface

JNI. Java Native Interface

JRE. Java Run-time Environment

LAN. local area network

LOB. Large OBject

MBCS. Multibyte Character Set

MFC. Module File Caching

NonStop TS/MP. NonStop Transaction Services/MP

OSS. Open System Services

POSIX. portable operating system interface x

RISC. reduced instruction set computing

RVU. Release Version Update

SPJ. stored procedure in Java

SQLJ. embedded SQL in Java programs

SQL/MP. Structured Query Language/MP

SQL/MX. Structured Query Language/MX

TCP/IP. Transmission Control Protocol/Internet Protocol

TMF. Transaction Management Facility

URL. uniform resource locator

VM. virtual machine

WWW. World Wide Web

Introduction to JDBC/MX Driver

The HP JDBC/MX Driver for NonStop SQL/MX implements the JDBC technology that conforms to the standard JDBC 3.0 Data Access API. This JDBC/MX driver enables Java applications to use HP NonStop SQL/MX to access NonStop SQL databases.

For more information on the JDBC APIs associated with the JDBC/MX implementation, see Sun Microsystems Documents earlier in this document. To obtain detailed information on the standard JDBC API, you should download the JDBC API documentation provided by Sun Microsystems (http://java.sun.com/products/jdbc/download.html).

The JDBC/MX driver together with HP NonStop Server for Java 5 is a Java environment that supports compact, **concurrent**, dynamic, **portable** programs for the enterprise server. The JDBC/MX driver requires NonStop Server for Java 5 and SQL/MX, which both require the HP NonStop Open System Services (OSS) environment. The NonStop Server for Java 5 uses the HP NonStop operating system to add the NonStop system fundamentals of **scalability** and program **persistence** to the Java environment.

This section explains these subjects:

- JDBC/MX Architecture
- JDBC API Packages
- Sample Programs Summary

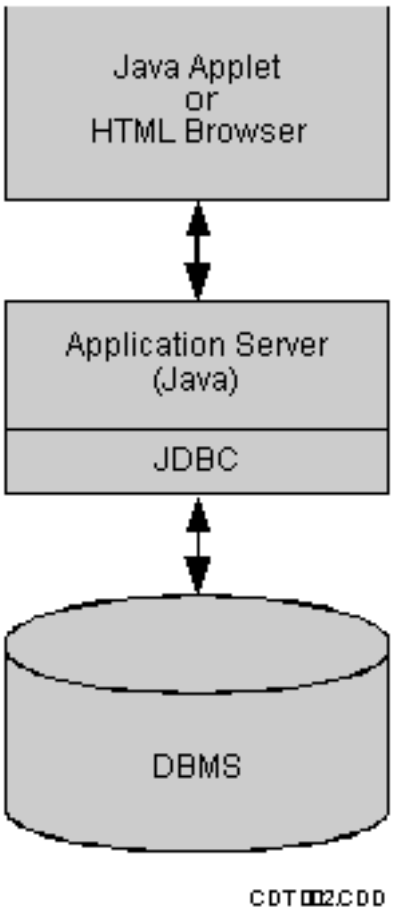
JDBC/MX Architecture

The JDBC/MX driver is a Type 2 driver; it employs proprietary native APIs to use SQL/MX to access NonStop SQL databases. The native API of SQL/MX cannot be called from client systems. For this reason, the JDBC/MX driver runs on NonStop servers only.

The JDBC/MX driver is best suited for a three-tier model. In the three-tier model, commands are sent to a middle tier of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. The middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

The following figure illustrates a three-tier architecture for database access:

Architecture of the JDBC/MX Driver



CDTD2CDD

JDBC API Packages

The JDBC/MX API packages are shipped with the JDBC/MX driver software. For the API documentation, see the JDBC/MX Driver for NonStop SQL/MX for H50 API Reference in the H-series library in the NonStop Technical Library at docs.hp.com.

The `java.sql` and `javax.sql` packages are included as part of Java 2, Standard Edition (J2SE) SDK 1.4.2 and, therefore, are available with the core APIs delivered with the NonStop Server for Java 5 product.

Sample Programs Summary

The JDBC/MX driver includes sample Java programs that illustrate the features of the product. The programs are described in the following table.

Sample Program	Comments

`sampleJdbcMx.java`	Illustrates loading the JDBC/MX driver and obtaining a JDBC connection using the `DriverManager` interface.
`CreateDataSource.java` and `TestDataSource.java`	Illustrates making a connection by using the `DataSource` interface, thereby avoiding embedding driver-specific codes in the Java programs. `CreateDataSource.java` creates the `SQLMXDataSource` object and registers the object with the Java Naming and Directory Interface (JNDI).
`MultiThreadTest.java`	Demonstrates the nonblocking JDBC/MX driver feature. By default, this program creates two threads. In nonblocking mode, these two threads run concurrently. This program displays the thread ID and status of the SQL operation before and after each operation. When the program runs in blocking mode, you observe only one thread switch because the begin-transaction operation starts a transaction in SQL nowait mode. When the program runs in nonblocking mode, you can observe many thread switches.
`holdJdbcMx.java`	Illustrates the holdable cursor support in the JDBC/MX driver. The program creates a subscriber thread that subscribes to a message queue. When all the rows in the message queue are read, the subscriber times out after five seconds.
`TestConnectionPool.java`	Demonstrates the benefits of connection pooling and statement pooling. This program performs a loop for a 100 times that makes a JDBC connection, runs a few SQL statements, and closes the connection. You use the OSS `time()` command to measure the performance benefits of connection pooling and statement pooling in this program.
`CreateTraceDS.java` `TestTraceDS.java`	Demonstrates tracing by creating a wrapper around the driver-specific data source to be traced. These demonstration programs are located in the `/demo` directory of the product installation directory.
`JdbcRowSetSample.java`	Demonstrates how to create an `SQLMXJdbcRowSet` object and invoke several `JdbcRowSet methods`.
`LobSample.java`	Demonstrates the LOB feature in the JDBC/MX driver.
`TransactionMode.java`	Demonstrates internal, external, and mixed transaction modes.

`ISO88591Sample.java`	Demonstrates the `ISO88591` property.

For information on running these sample programs, see the README file provided with the JDBC/MX driver software.

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

Installing and Verifying JDBC/MX

This section explains these subjects:

Installation Requirements

Hardware and software requirements for the JDBC/MX Driver for NonStop SQL/MX are described in the Softdoc file on the NonStop Server for Java 5 product CD, with which the JDBC/MX driver is delivered. Read that document before installing the product.

The JDBC/MX driver version is HP JDBC/MX driver for NonStop SQL/MX H50(also identified as product T1275)

The JDBC/MX driver requires the following software:

- NonStop SQL/MX Release 2.0 or subsequent 2.x releases
- NonStop Server for Java, based on Java 2 Platform Standard Edition 5.0 (T2766H50)or subsequent product updates

Note: For the most current statement of the software requirements, see the Softdoc file for the list of earliest acceptable versions of the required software. You can substitute later versions of the same products.

JDBC/MX Driver File Locations

The JDBC/MX driver installation directory location for the JDBC/MX driver software is:

install_dir/jdbcMx/T1275H50

For example, the default installation directory is

/usr/tandem/jdbcMx/T1275H50

The files installed include:

.../demo
: Demo programs

.../lib/libjdbcMx.so
: JDBC/MX driver library

.../lib/jdbcMx.jar
: JDBC/MX Java archive file, which includes the JDBC trace facility

.../bin/jdbcMxInstall
: JDBC/MX installation script

.../bin/jdbcMxUninstall
: JDBC/MX uninstall script

Verifying the JDBC/MX Driver

To verify the version of the JDBC/MX driver, use these commands:

- `java` command, which displays the version of the java code portion of the JDBC/MX driver
- `vproc` command, which displays the version of the C code portion of the JDBC/MX driver

To use the `java` command, type the following at the OSS prompt:

```
java JdbcMx -version
```

This command displays output similar to:

```
JDBC driver for NonStop(TM) SQL/MX Version
T127H50_23DEC2005_JDBCMX...
```

Compare the output with the product numbers in the Softdoc file on the NonStop Server for Java 5 distribution CD.

Use the `vproc` command to check for the correct library. Issue the following at the OSS prompt:

```
vproc
```

```
/jdbcmx-installation-directory/T1275H50/lib/libjdbcMx.so

...
      Binder timestamp: 20DEC2005 18:10:14
     Version procedure: T1275H50_23DEC2005_JDBCMX_1220
     TNS/E Native Mode: runnable file
```

The version procedure that corresponds to the JDBC/MX Driver (T1275) product should match in the output of both the `java` and `vproc` commands.

Setting CLASSPATH

For running JDBC applications, ensure the `CLASSPATH` environment variable includes the `jdbcMx.jar` file. Given the default installation, the path is

`/usr/tandem/jdbcMx/current/lib/jdbcMx.jar`

Setting the _RLD_LIB_PATH

For running JDBC applications, ensure the `_RLD_LIB_PATH` environment variable path is set to TNS/E jdbcMx PIC file. Given the default installation, the path is:

`/usr/tandem/jdbcMx/T1275H50/lib`

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

Accessing SQL Databases with SQL/MX

This section describes the following subjects:

Connection to SQL/MX

A Java application can obtain a JDBC connection to SQL/MX in two ways:

- Using the `DriverManager` class
- Using the `DataSource` interface

Connection Using the DriverManager Class

This is the traditional way to establish a connection to the database. The `DriverManager` class works with the `Driver` interface to manage the set of drivers loaded. When an application issues a request for a connection using the `DriverManager.getConnection` method and provides a URL, the `DriverManager` is responsible for finding a suitable driver that recognizes this URL and obtains a database connection using that driver.

`com.tandem.sqlmx.SQLMXDriver` is the JDBC/MX driver class that implements the `Driver` interface. The application can load the JDBC/MX driver in one of the following ways, except as noted in the `DriverManager` Object Properties table:

- Specifying the JDBC/MX driver class in the `-Djdbc.drivers` option in the command line
- Using the `Class.ForName` method within the application
- Adding the JDBC/MX driver class to the `jdbc.drivers` property within the application

The `DriverManager.getConnection` method accepts a string containing a JDBC URL. The JDBC URL for the JDBC/MX driver is `jdbc:sqlmx:`.

When connecting by using the DriverManager class, use the information in the following topics:

- JDBC/MX Driver Properties Used with the DriverManager Class
- Guidelines for Using Connections with the DriverManager Class

JDBC/MX Driver Properties Used with the DriverManager Class

JDBC/MX driver defines the following set of properties that you can use to configure the driver:

Property Name	Type	Value	Description
`contBatchOnError`	`String`	`on` or `off`	Communicates with JDBC driver to continue the remaining jobs in the batch even after any BatchUpdateExceptions. See contBatchOnError Property.
`catalog`	`String`	See Default Catalog and Schema.	If the default catalog and schema are not specified, the JDBC/MX driver allows SQL/MX to follow its own rules for defaults.
`schema`	`String`	See Default Catalog and Schema.	See `catalog` above.
`mploc`	`String`	See mploc Property.	The location (in `$volume.subvolume` format) in which SQL/MP tables are created. (The default location is the default subvolume of the logged-on user.)

`enableLog`	`boolean`	`on` or `off`	Enables logging of SQL statement IDs and the corresponding JDBC SQL statements. See enableLog Property.
`idMapFile`	`String`	A valid OSS filename	Specifies the file to which the trace facility logs SQL statement IDs and the corresponding JDBC SQL statements. See idMapFile Property.
`ISO88591`	`String`	See ISO88591 Property.	Specifies the Java encoding used when accessing and writing to ISO88591 columns.
`maxStatements`	`int`	See maxStatements Property.	The total number of `PreparedStatement` objects that the connection pool should cache. See maxStatements Property.
`minPoolSize`	`int`	See minPoolSize Property.	Limits the number of physical connections that can be in the free connection pool. See minPoolSize Property.
`maxPoolSize`	`int`	See maxPoolSize Property.	Sets maximum number of physical connections that the pool should contain. This number includes both free connections and connections in use. See maxPoolSize Property.
`blobTableName`	`String`	See LOB Table Name Properties.	Specifies the LOB table for using BLOB columns.

`clobTableName`	`String`	See LOB Table Name Properties.	Specifies the LOB table for using CLOB columns.
`transactionMode`	`String`	See transactionMode Property.	Sets the transaction mode, which provides control over how and when transactions are performed. See transactionMode Property.
Note: Do not add the jdbcmx. prefix to the property name when the properties are given as a parameter to the connection method or when using the data source. The prefix is not needed to identify the property type because the property is being passed to a JDBC/MX driver object. Use the jdbcmx. prefix only in the command line as described under Setting Properties in the Command Line.			

Guidelines for Using Connections with the DriverManager Class

- Java applications can specify the properties in the following ways:
  - Using JDBC/MX properties with the `-D` option in the command line. If used, this option applies to all JDBC connections using the `DriverManager` within the Java application. The format is to enter the following in the command line:

    ```
    -Djdbcmx.property_name=property_value
    ```

    For example in a command line, `-Djdbcmx.maxStatements=1024`
  - Using the `java.util.properties` parameter in the `getConnection` method of `DriverManager`.
- The properties passed through the `java.util.properties` parameter have a higher precedence over the command-line properties.
- The connection pooling feature is available when the Java application uses the `DriverManager` class to obtain a JDBC connection. The connection pool size is determined by the `maxPoolSize` property value and `minPoolSize` property value.
- The JDBC/MX driver has a connection-pool manager for a combination of catalog and schema; therefore, connections with the same catalog and schema combinations are pooled together. The connection pooling property values that are used at the time of obtaining the first connection for a given catalog and schema combination is effective throughout the life of the process. An application cannot change these property values subsequent to the first connection for a given catalog and schema combination.
- As in the basic `DataSource` object implementation, a Java application can enable statement pooling by setting the property to a non-zero positive value.

Connection Using the DataSource Interface

The `DataSource` interface, introduced in the JDBC 2.0 optional package, is the preferred way to establish a connection to the database because it enhances the application portability. The JDBC/MX driver implements the `DataSource` interface and returns a connection object when an application requests a connection using the `getConnection` method in the `DataSource` interface.

Using a `DataSource` object increases the application portability by allowing the application to use a logical name for a data source instead of providing driver-specific information in the application. A logical name is mapped to a `DataSource` object by means of a naming service that uses the Java Naming and Directory Interface (JNDI).

The following table describes the properties that you can use to identify a JDBC/MX data source object:

DataSource Object Properties

Property Name	Type	Value	Description
`contBatchOnError`	`String`	`on` or `off`	Communicates with JDBC driver to continue the remaining jobs in the batch even after any BatchUpdateExceptions. See contBatchOnError Property.
`catalog`	`String`	See Default Catalog and Schema.	If the default catalog and schema are not specified, the JDBC/MX driver allows SQL/MX to follow its own rules for defaults.
`schema`	`String`	See Default Catalog and Schema.	See `catalog` above.
`dataSourceName`	`String`		The registered `ConnectionPoolDataSource` name. When this string is empty, connection pooling is used by default with the pool size determined by the maxPoolSize property and minPoolSize property of the basic `DataSource` object. For more information, see Connection Using the Basic DataSource API.

`description`	`String`	Any valid identifier	The description of the data source.
`enableLog`	`boolean`	`on` or `off`	Enables logging of SQL statement IDs and the corresponding JDBC SQL statements. See enableLog Property.
`idMapFile`	`String`	A valid OSS filename	Specifies the file to which the trace facility logs SQL statement IDs and the corresponding JDBC SQL statements. See idMapFile Property.
`ISO88591`	`String`	See ISO88591 Property.	Specifies the Java encoding used when accessing and writing to ISO88591 columns.
`maxPoolSize`	`int`	See maxPoolSize Property.	Sets maximum number of physical connections that the pool should contain. This number includes both free connections and connections in use. See maxPoolSize Property.
`maxStatements`	`int`	See `maxStatements` Property.	The total number of `PreparedStatement` objects that the connection pool should cache. See maxStatements Property.
`minPoolSize`	`int`	See minPoolSize Property.	Limits the number of physical connections that can be in the free connection pool. See minPoolSize Property.
`mploc`	`String`	See `mploc` Property.	The location (in `$volume.subvolume` format) in which SQL/MP tables are created (The default location is the default subvolume of the logged-on user.)
`blobTableName`	`String`	See LOB Table Name Properties.	Specifies the LOB table for using BLOB columns.

`clobTableName`	`String`	See LOB Table Name Properties.	Specifies the LOB table for using CLOB columns.
`transactionMode`	`String`	See transactionMode Property.	Sets the transaction mode, which provides control over how and when transactions are performed. See transactionMode Property.
Note: Do not add the jdbcmx. prefix to the property name when the properties are given as a parameter to the connection method or when using the data source. The prefix is not needed to identify the property type because the property is being passed to a JDBC/MX driver object. Use the jdbcmx prefix only in the command line, as described under Setting Properties in the Command Line.			

JdbcRowSet Implementation

An implementation of the `JdbcRowSet` interface, `SQLMXJdbcRowSet`, is provided within the `com.tandem.sqlmx` package. A `JdbcRowSet` object maintains a connection to the database, similar to a `ResultSet` object. However, a `JdbcRowSet` object maintains a set of properties and listener notification mechanisms that make it a JavaBeans™ component.

The `SQLMXJdbcRowSet` object can be created using these `SQLMXJdbcRowSet` constructors:

- The default constructor that does not require any parameters.
  Note: This constructor does not attempt to connect to the database until the `execute` method is invoked.
- The constructor that takes a `Connection` object.
- The constructor that takes a `ResultSet` object.
- The constructor that takes a url, username, and password
  Note: Username and password attributes are currently not supported. This constructor has been provided for future use after the username and password support has been provided by both SQL/MX and JDBC/MX.

Refer to the `JdbcRowSetSample.java` demo program as an example of instantiating and manipulating an `SQLMXJdbcRowSet` object. Also, refer to the Unsupported and Deviations sections for specific implementation details of the `SQLMXJdbcRowSet` object.

For additional details, refer to the `JdbcRowSet` Interface specification at http://java.sun.com/j2se/1.5.0/docs/api/javax/sql/rowset/JdbcRowSet.html.

JDBC/MX Properties

JDBC/MX properties included in both the `DriverManager` object properties table and `DataSource` object properties table are described in topics as follows:

- Default Catalog and Schema
- LOB Table Name Properties
- ISO88591 Property
- mploc Property
- maxStatements Property
- minPoolSize Property
- maxPoolSize Property
- transactionMode Property

These properties and additional properties can be specified in a command line, as described under Setting Properties in the Command Line.

For information about using features provided by various JDBC/MX properties, see the topic, Using Additional JDBC/MX Properties.

Default Catalog and Schema

The default catalog and schema are used to access SQL objects referenced in SQL statements if the SQL objects are not fully qualified. The three-part fully qualified name for SQL/MX objects is of the form:

```
[[catalog.]schema.]object-name
```

The catalog and schema names can be any arbitrary strings that conform to SQL identifiers. These names conform to ANSI SQL:99 catalog and schema names.

For example, using the default catalog and schema properties for a table referenced as `CAT.SCH.TABLE`, the options are:

```
-Djdbcmx.catalog=CAT -Djdbcmx.schema=SCH
```

For more information, see the *SQL/MX Reference Manual*.

LOB Table Name Properties

LOB tables store data for LOB columns. The properties you use to specify the LOB table for using `BLOB` columns or `CLOB` columns are:

For the `BLOB` columns

```
blobTableName
```

For the `CLOB` columns

```
clobTableName
```

The property value is of the form:

```
catalog_name.schema_name.lob_table_name
```

You can specify the name of the LOB table using properties in the following ways:

- By using the `-Djdbcmx.property_name=property_value option` in the `java` command line. For example:

  ```
  -Djdbcmx.clobTableName=mycat.myschema.myLobTable
  ```

- By using the `java.util.Properties` parameter in the `getConnection` method of `DriverManager` class. The properties passed through the Properties parameter have precedence over the command line properties.
- By setting either of these properties in the `DataSource`. See Connection Using the DataSource Implementation.

ISO88591 Property

The ISO88591 character set mapping property corresponds to the SQL/MX ISO88591 character set, which is a single-byte 8-bit character set for character data types. The ISO88591 character set supports English and other Western European languages. Specify the `ISO88591` property as

```
String
```

The default value is `DEFAULT` which uses the default Java encoding when accessing and writing to ISO88591 columns. The value can be any valid Java Canonical Name as listed in the "Canonical Name for `java.io` and `java.lang` API" column of the Sun documentation, Supported Encodings (http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html).

For example, if KANJI data has been stored in an ISO88591 column in an SQL/MP table (accessed through SQL/MX) and has been read and written to the database using the column character set, you can specify the following property to ensure the correct encoding:

```
-Djdbcmx.ISO88591=SJIS
```

mploc Property

The property `mploc` specifies the Guardian location in which SQL tables are created. The format for `mploc` is:

```
[\node].$volume.subvolume
```

Java applications using the JDBC/MX driver can specify `mploc` by using the system property `mploc` with the `-D` option in the command line.

```
-Djdbcmx.mploc=mploc
```

For example with the `DriverManager` object, in the OSS environment, specify the `mploc` property in either of the following forms:

```
-Djdbcmx.mploc=[\\node.]\$volume.subvolume
```

or

```
-Djdbcmx.mploc='[\node.]$volume.subvolume'
```

For more information, see the *SQL/MX Reference Manual*.

maxStatements Property

Sets the total number of `PreparedStatement` objects that the connection pool should cache. This total includes both free objects and objects in use. Specify the `maxStatements` property as:

```
int
```

The integer can be 0 through 2147483647. Any negative value is treated like 0. The default is 0, which disables statement pooling. HP recommends that you enable statement pooling for your JDBC applications, because this pooling can dramatically help the performance of many applications.

minPoolSize Property

Limits the number of physical connections that can be in the free connection pool. Specify the `minPoolSize` property as:

```
int
```

The integer can be 0 through 2147483647, but less than `maxPoolSize`. The default is 0. Any negative value is treated like 0. Any value greater than `maxPoolSize` is changed to the `maxPoolSize` value. This value is ignored when `maxPoolSize` is -1. The value determines connection pool use as follows:

- When the number of physical connections in the free pool reaches the `minPoolSize` value, the JDBC/MX driver closes subsequent connections by physically closing them—not by adding them to the free pool.
- 0 means the connections are not physically closed; the connections are always added to the free pool when the connection is closed.

maxPoolSize Property

Sets the maximum number of physical connections that the pool can contain. These connections include both free connections and connections in use. When the maximum number of physical connections is reached, the JDBC/MX driver throws an `SQLException` with the message, "Maximum pool size is reached." Specify the `maxPoolSize` property as:

`int`

The integer can be -1, 0 through 2147483647, but greater than `minPoolSize`. The default is 0. Any negative value is treated like -1. Any positive value less than `minPoolSize` is changed to the `minPoolSize` value. The value determines connection pool use as follows:

- 0 means no maximum pool size.
- -1 for the basic `DataSource` object means connection pooling is disabled. -1 is invalid for the `ConnectionPoolDatasource` object.

transactionMode Property

The `transactionMode` property provides control over how and when transactions are performed. Specify the `transactionMode` property as:

`String`

The default is `mixed`. The allowed values are:

`internal`

Specifies that transactions are always performed within a JDBC/MX driver-managed transaction. If an external transaction exists when internal transaction mode is in effect, the external transaction is suspended and the SQL statement is executed within a JDBC/MX driver-managed transaction. Upon completion of the driver s internally managed transaction, the existing external transaction is resumed. The Connection autoCommit flag maintains a value of true when in internal transaction mode.

Note: Using `internal` transactionMode for select statements performed in external transactions causes JDBC/MX to throw an "invalid transaction state" exception. Therefore, do not specify `internal` transactionMode under these conditions.

`mixed`

Specifies that the driver inherits any active transaction in the current thread. The autocommit setting of the transaction is ignored. The application must either commit or rollback the transaction in this mode. If there is no active transaction, the driver creates one and begins the transaction, or aborts it if there is an SQL error. In this mode, the driver supports both autocommit and non-autocommit. The application ends the transaction in non-autocommit mode.

`external`

Specifies that if an external transaction exists, transactions are performed within the external transaction. If an external transaction does not exist, the SQL statement is executed without a transaction. This allows SQL statements that do not require an existing transaction to be performed without one, providing an improvement in performance. If an SQL command requires a transaction and no external transaction exists, an SQL exception is thrown.

Note: Using `external` transactionMode for SQL statements that require execution within a transaction results in an SQL exception. Therefore, do not specify `external` transactionMode under these conditions.

Considerations:

- If any other string is specified for the value of transaction mode, `mixed` is used.
- Using the external or mixed transaction mode can improve performance.
- Using the internal transaction mode can affect performance for applications because of the overhead of TMF transactions under a heavy load.
- This property can be set within a JDBC/MX driver properties file, defined within a `DataSource` object, or passed in through the `java` command line.
- The transaction mode can only be changed for new connections; therefore, it cannot be dynamically changed within a connection.

This property can be specified in a data source, in the JDBC/MX properties file, or in the `java` command line.

Setting Properties in the Command Line

JDBC/MX driver property names used on the command line in the `java -D` option must include the prefix:

```
jdbcmx.
```

This notation, which includes the period (`.`), ensures that all the JDBC/MX driver property names are unique for a Java application. For example: the `maxStatements` property becomes

```
jdbcmx.maxStatements
```

JDBC/MX Driver Properties Allowed in the Command Line

JDBC/MX Prefix	Property Name	Description
`jdbcmx.`	`contBatchOnError`	Communicates with JDBC driver to continue the remaining jobs in the batch even after any BatchUpdateExceptions. See contBatchOnError Property.
`jdbcmx.`	`stmtatomicity`	Allows the user to enable atomicity of SQL statements at statement level. See stmtatomicity Property.

jdbcmx.	batchBinding	Specifies that statements are batched together in the executeBatch() operation. See Setting Batch Processing for Prepared Statements.
jdbcmx.	blobTableName	Specifies the LOB table for using BLOB columns. See LOB Table Name Properties.
jdbcmx.	catalog	Sets the default catalog. See Default Catalog and Schema.
jdbcmx.	clobTableName	Specifies the LOB table for using CLOB columns. See LOB Table Name Properties.
jdbcmx.	enableLog	Enables logging of SQL statement IDs and the corresponding JDBC SQL statements. See enableLog Property.
jdbcmx.	idMapFile	Specifies the file to which the trace facility logs SQL statement IDs and the corresponding JDBC SQL statements. See idMapFile Property.
jdbcmx.	ISO88591	Specifies the encoding to be used when accessing or writing to data stored in IS088591 columns. See ISO88591 Property.
jdbcmx.	maxPoolSize	Sets the maximum pool size. See maxPoolSize Property.
jdbcmx.	maxStatements	Sets the total number of PreparedStatement objects that the connection pool should cache. See maxStatements Property.
jdbcmx.	minPoolSize	Sets the minimum pool size. See minPoolSize Property.
jdbcmx.	mploc	Sets the location in SQL/MP tables. See mploc Property.

`jdbcmx.`	`reserveDataLocators`	Sets the number of data locators to be reserved. See Setting the reserveDataLocators Property.
`jdbcmx.`	`schema`	Sets the default schema. See Default Catalog and Schema.
`jdbcmx.`	`sqlmx_nowait`	See Managing Nonblocking JDBC/MX.
`jdbcmx.`	`traceFile`	Specifies the trace file for logging. See Enabling Tracing for Application Servers.
`jdbcmx.`	`traceFlag`	Sets the trace flag for logging. See Enabling Tracing for Application Servers.
`jdbcmx.`	`transactionMode`	Sets the transaction mode, which provides control over how and when transactions are performed. See transactionMode Property.

For example, using the `mploc` property in the OSS environment, specify the `mploc` property including the prefix in either of the following forms:

```
-Djdbcmx.mploc=[\\node.]\$volume.subvolume
```

or

```
-Djdbcmx.mploc='[\node.]$volume.subvolume'
```

Transactions

The JDBC/MX driver provides transaction support to maintain data integrity and consistency. To allow the application to interleave transactions between SQL/MX objects and the traditional file system, the JDBC/MX driver checks if a transaction is active whenever it needs to interact with SQL/MX.

The transactionMode property determines transaction processing behavior. If you use transactionMode in a typical environment, with the default value `mixed`,

- When an active transaction exists, the autocommit setting is ignored, and the JDBC/MX driver lets the application manage the transaction.
- When no active transaction exists, the JDBC/MX driver manages the transactions.

This implementation differs from JDBC/MP. In the JDBC/MP driver, two different types of URLs decide which component manages the transaction.

If you are accessing `BLOB` and `CLOB` data, see also Transactions Involving `Blob` and `Clob` Access.

Autocommit Mode and Transaction Boundaries

When JDBC/MX manages the transactions, the driver decides to start a new transaction. A new transaction is started when no transaction is associated with the `Connection`. When there is a transaction associated with the `Connection`, that transaction is resumed. The `Connection` attribute `autocommit` specifies when to end the transaction. Enabling autocommit causes the JDBC/MX driver to end the transaction in accordance with the following rules:

- The JDBC/MX driver rolls back the transaction for any SQL error in SQL statements other than SELECT statements.
- In the case of non-SELECT SQL statements, the JDBC/MX driver commits the transaction if the current transaction was started for this SQL statement.
- In the case of SELECT statements, the JDBC/MX driver commits the transaction at the time of closing the result set.
- In the case of concurrent multiple SELECT statements, the JDBC/MX driver commits the transaction only when the result set of the SELECT statement or the statement that started the transaction is closed.

Disabling Autocommit Mode

When the autocommit mode is disabled, the application must explicitly commit or roll back each transaction by calling the `Connection` methods `commit` and `rollback`, respectively. When any SQL error occurs in SQL statements other than SELECT statement, SQL/MX flags the transactions for aborting. In such a case, the transaction is rolled back without regard to whether the application commits or rolls back the transaction.

Stored Procedures

SQL/MX provides support for stored procedures with result sets, which are written in Java and run under an SQL/MX execution environment.

Stored procedures can be run in SQL/MX by using the CALL statement. The JDBC/MX driver allows stored procedures to be called by using the standard JDBC API escape syntax for stored procedures. The escape SQL syntax is:

```
{call procedure-name([arg1,arg2, ...])}
```

where `argn` refers to the parameters sequentially, with the first parameter being `arg1`. For more

information about the non-escape syntax of the CALL statement, see the *SQL/MX Reference Manual*.

Java applications can use the JDBC standard `CallableStatement` interface to run stored procedures in SQL/MX by using the CALL statement. For more information, see the *SQL/MX Guide to Stored Procedures in Java*.

Limitations

Limitations of the stored procedures in Java (SPJs) are:

- The stored procedures in Java (SPJs) do not support result sets returned from the Java method that contain CLOB or BLOB data types.
- SPJs do not support SHORTANSI names.

Note: Do not use the SHORTANSI name type with SPJs.

SQL Context Management

NonStop SQL/MX allows you to manage SQL/MX contexts. An SQL/MX context can be considered as an instance of the SQL/MX executor that has its own execution environment that contains the following:

- CONTROL and SET information
- A transaction
- An SQL/MX compiler process (MXCMP)
- Set of SQL/MX executive server processes (ESPs)
- User-created SQL statements

The JDBC/MX driver maps a JDBC connection to an SQL/MX context. Therefore, in a multithreaded application, a JDBC application has multiple SQL/MX compiler processes (MXCMP processes) associated with the application. An SQL/MX context is created when the application obtains a JDBC connection. An SQL/MX context is destroyed when the application explicitly or implicitly closes the JDBC connection.

The following JDBC connection attributes are passed to the SQL/MX context by the JDBC/MX driver by executing the corresponding SQL statements:

Connection Attributes Passed to the SQL/MX Context

Attribute	SQL Statement
`catalog`	`SET CATALOG default-catalog-name`
`schema`	`SET SCHEMA default-schema-name`
`mploc`	`SET MPLOC default-location`

`transaction isolation`	`SET TRANSACTION isolation-level`

A process (JVM process) can have multiple SQL/MX contexts within a process.

Holdable Cursors

JDBC/MX driver supports the holdablity attribute for the `ResultSet`. To use holdable cursors in your JDBC applications, follow these guidelines:

- Use one of the following constants for the holdablity attribute:

  `com.tandem.sqlmx.SQLMXResultSet.HOLD_CURSORS_OVER_COMMIT`

  Ensure that when the application calls the method `Connection.commit` or `Connection.rollback`, the `HOLD_CURSORS_OVER_COMMIT` constant indicates that `ResultSet` objects are not closed.

  `com.tandem.sqlmx.SQLMXResultSet.CLOSE_CURSORS_AT_COMMIT`

  Ensure that when the application calls the method `Connection.commit` or `Connection.rollback`, the `CLOSE_CURSORS_AT_COMMIT` constant indicates that `ResultSet` objects are closed.

- For the `ResultSet` objects to be holdable over a commit operation, ensure that the SQL statement that generates the `ResultSet` has either stream access mode, or embedded `update` or `delete` for table references.

- Use either of the following methods in `SQLMXConnection` objects to create result sets with holdable cursors over commit:

  ```
  createStatement(int resultSetType, int resultSetConcurrency,
  int resultSetHoldability)
  ```

  ```
  prepareStatement(String sql, int resultSetType,
  int resultSetConcurrency, int resultSetHoldability)
  ```

  For a demonstration in a sample program, see the holdJdbcMx.java program description.

Connection Pooling

JDBC/MX provides an implementation of connection pooling, where a cache of physical database connections are assigned to a client connection session and reused for the database activity. Once the client session is closed, the physical connection is put back into cache for subsequent use. This implementation contrasts to the basic `DataSource` object implementation, where a one-to-one correspondence exists between client `Connection` object and the physical database connection.

Your applications can use connection pooling in the following ways:

- Connection Pooling by an Application Server
- Connection Pooling Using the Basic DataSource API
- Connection Pooling with the DriverManager Class

Connection Pooling by an Application Server

Usually, in a three-tier environment, the application server implements the connection pooling component. How to implement this component is described in these topics:

- Guidelines for Implementing an Application Server to Use Connection Pooling
- Standard ConnectionPoolDataSource Object Properties

Guidelines for Implementing an Application Server to Use Connection Pooling

- The application server maintains a cache of the `PooledConnection` objects created by using `ConnectionPoolDataSource` interface. When the client requests a connection object, the application looks for the suitable `PooledConnection` object. The lookup criteria and other methods are specific to the application server.
- The application server implements the `ConnectionEventListener` interface and registers the listener object with the `PooledConnection` object. The JDBC/MX driver notifies the listener object with a `connectionClosed` event when the application is finished using the Connection object. Then, the connection pooling component can reuse this `PooledConnection` object for future requests. The JDBC/MX driver also notifies the listener object with `connectionErrorOccurred` event when the `PooledConnection` object fails to initialize the connection. The application server's connection pooling component should discard the `PooledConnection` when such a connection error event occurs.
- The application server manages the connection pool by using the `SQLMXConnectionPoolDataSource`, which implements the `ConnectionPoolDataSource` interface. Use the getter and setter methods, provided by JDBC/MX, to set the connection pool configuration properties listed in the table of Standard ConnectionPoolDataSource Object Properties. In addition to these standard properties, the

`ConnectionPoolDataSource` includes the JDBC/MX driver-specific properties as described under Connection Using the DataSource Interface.

Standard ConnectionPoolDataSource Object Properties

Note: The application server defines the meaning of these properties.

Property Name	Type	Description
`maxStatements`	`int`	The total number of `PreparedStatement` objects that the pool should cache. This total includes both free objects and objects in use. 0 (zero) disables statement pooling.
`initialPoolSize`	`int`	The number of physical connections the pool should contain when it is created.
`minPoolSize`	`int`	The number of physical connections the pool should keep available at all times. 0 (zero) indicates no maximum size.
`maxPoolSize`	`int`	The maximum number of physical connections that the pool should contain. 0 (zero) indicates no maximum size.
`maxIdleTime`	`int`	The number of seconds that a physical connection should remain unused in the pool before the connection is closed. 0 (zero) indicates no limit.
`propertyCycle`	`int`	The interval, in seconds, that the pool should wait before enforcing the current policy defined by the values of the above connection pool properties.

Connection Pooling Using the Basic DataSource API

For your JDBC application to enable connection pooling, use the basic `DataSource` interface, which includes the following properties that control connection pooling:

- `maxPoolSize`
- `minPoolSize`
- `maxStatements`

Your application can enable connection pooling in the following two ways:

- By setting the `dataSourceName` property in the basic `DataSource` object to the previously registered `ConnectionPoolDataSource` object. When the connection pooling is enabled, the JDBC/MX driver-specific properties in the `ConnectionPoolDataSource` object are effective, and the JDBC/MX driver-specific properties in the `DataSource` object are ignored. The connection is initialized with the JDBC/MX driver-specific properties when the `PooledConnection` is obtained.
- By using the properties in the `DataSource` object, when the `dataSourceName` property is empty. Connection pooling is enabled by default. Note that the default value for the `maxPoolSize` property is 0, which enables connection pooling. See the `DataSource` interface for the details on using these properties.

For troubleshooting application connection pooling, note the following details on how the feature is implemented. JDBC/MX looks for the first available `PooledConnection` object and assigns the object to the client requests for a connection. JDBC/MX ensures that the SQL/MX execution environment and compilation environment remain the same for all the connections in the connection pooling environment; that is, the environment is the same as when the initial connection was obtained by the client session either from the pool or from a new physical connection.

Connection Pooling with the DriverManager Class

Connection pooling is available by default when your JDBC application uses the `DriverManager` class for connections. You can manage connection pooling by using the following properties listed in the `DriverManager` Object Properties table and described as under JDBC/MX Properties:

- `maxPoolSize`
- `minPoolSize`
- `maxStatements`

Set these properties in either of two ways:

- Using the option `-Dproperty_name=property_value` in the command line
- Using the `java.util.Properties` parameter in the `getConnection()` method of the `DriverManager` class

Use these guidelines when setting properties for connection pooling with the `DriverManager` class:

- To enable connection pooling, set the `maxPoolSize` property to an integer value greater than 0 (zero).
- The properties passed through the `Properties` parameter have a higher precedence over the command-line properties.
- Connections with the same catalog-schema combination are pooled together and managed by the JDBC/MX driver. The connection-pooling property values that the application process uses when it obtains the first connection for a given catalog-schema combination are effective for that combination through the life of the application process.

Statement Pooling

The statement pooling feature allows applications to reuse the `PreparedStatement` object in same way that they can reuse a connection in the connection pooling environment. Statement pooling is done completely transparent to the application. Using statement pooling is described in the following topics:

- Guidelines for Statement Pooling
- Controlling the Performance of ResultSet Processing
- Troubleshooting Statement Pooling

Guidelines for Statement Pooling

- Enable statement pooling by setting the `DataSource` object `maxStatements` property to an integer value greater than 0 and, also, by enabling connection pooling. See Connection Pooling for more information.
- Enabling statement pooling for your JDBC applications might dramatically improve the performance.
- Explicitly close a prepared statement by using the `Statement.close` method because `PreparedStatement` objects that are not in scope are also not reused unless the application explicitly closes them.
- To ensure that your application reuses a `PreparedStatement`, call either of the following:
  - `Statement.close` method—called by the application
  - `Connection.close` method—called by the application. All the `PreparedStatement` objects that were in use are ready to be reused when the connection is reused.

Controlling the Performance of ResultSet Processing

To improve JDBC application performance of result fetches for statements that are expected to return more than two rows, the application should set the fetch size before executing the statement. This operation works because the `ResultSet` getter methods have been modified in the JDBC/MX driver to optimize database interactions. The JDBC/MX driver uses the fetch-size setting to determine the size of memory used for reading and buffering data.

The application can control the `ResultSet` fetch size by using the `setFetchSize()` method of the `Statement` class, `PreparedStatement` class, and `ResultSet` class.

Considerations:

- Applications that use SQL/MX tables, rather than SQL/MP tables, have improved performance only for result fetches that have greater than two rows returned. The default JDBC/MX fetch size is set to 1.
- Once the application sets the fetch size to a value greater than 2 for a statement, the application should not reset the value back to 2 or less. If the application does so, the application will experience a slight degradation in performance as compared to using the default value.
- Setting the fetch size greater than 2 for statements that return fewer than two rows causes a slight performance degradation, as compared to using the default fetch size.
- Setting the fetch size to a value greater than the number of rows returned by a statement causes the JDBC/MX driver to use more memory, but does not affect the API's functionality.

Troubleshooting Statement Pooling

Note the following JDBC/MX driver implementation details if you are troubleshooting statement pooling:

- JDBC/MX driver looks for a matching `PreparedStatement` object in the statement pool and reuses the `PreparedStatement`. The matching criteria include the SQL string, current catalog, current schema, current transaction isolation, and `resultSetHoldability`. If JDBC/MX driver finds the matching `PreparedStatement` object, JDBC/MX driver returns the same `preparedStatement` object to the application for reuse and marks the `PreparedStatement` object as in use.
- The algorithm, "earlier used are the first to go," is used to make room for caching subsequently generated `PreparedStatement` objects when the number of statements reaches the `maxStatements` limit.
- JDBC/MX driver assumes that any SQL CONTROL statements in effect at the time of execution or reuse are the same as those in effect at the time of SQL/MX compilation. If this condition is not true, reuse of a `PreparedStatement` object might result in unexpected behavior.
- You should avoid SQL/MX recompilation to yield performance improvements from statement pooling. The SQL/MX executor automatically recompiles queries when certain conditions are met. Some of these conditions are:
    - A run-time version of a table has a different redefinition timestamp than the compile-time version of the same table.
    - An existing open operation on a table was eliminated by a DDL or SQL utility operation.
    - The transaction isolation level and access mode at execution time is different from that at the compile time.

  For more information on SQL/MX recompilation, see the *SQL/MX Programming Manual for C and COBOL* or the *SQL/MX Programming Manual for Java*.
- When a query is recompiled, the SQL/MX executor stores the recompiled query; therefore, the query is recompiled only once until any of the previous conditions are met again.

- JDBC/MX driver pools the `CallableStatement` objects in the same way as `PreparedStatement` objects when the statement pooling is activated.
- JDBC/MX driver does not cache `Statement` objects.

Using Additional JDBC/MX Properties

You can use JDBC/MX properties for the following application features:

- BatchUpdate Exception handling Improvements
- Statement Level Atomicity
- Managing Nonblocking JDBC/MX
- Setting Batch Processing for Prepared Statements
- Setting the reserveDataLocators Property

In addition to these topics, also see Enabling Tracing for Application Servers.

BatchUpdate Exception handling Improvements

When a command in the batch fails, the remaining commands of the batch are not executed resulting in re-execution of entire batch. But, with this Batch Update Exception handling support, the remaining elements of the batch after the error prone statement can be executed and hence re-execution of the entire batch jobs can be avoided.

contBatchOnError property

The contBatchOnError property communicates with JDBC driver to continue the remaining jobs in the batch even after any BatchUpdateExceptions. This java property can be set from the command line as:

```
Djdbcmx.contBatchOnError={ON|OFF}
```

where

`ON`

continues batch execution even after any other batch exception

`OFF`

terminates the batch execution on any other batch exception. The default is set to `OFF.`

Note: This property can be set either through java command line option or through property file of Datasource.

Statement Level Atomicity

To maintain the database consistency, transactions must be controlled so that they either complete successfully or are aborted. With the prior release versions of JDBC/MX (before H10 AAB and V32 AAU on G-series), the transaction is automatically aborted on any error while performing an SQL statement.

This version of JDBC/MX driver follows up with the SQL/MX 2.0 Statement Atomicity feature and guarantees that an individual SQL statement within a transaction either completes successfully or has no effect on the database. When this statement level atomicity is followed, with the auto commit mode set to false, any failure occurred during the Insert, Update, or Delete operations will not abort the current transaction and this helps in execution of all the statements under this current transaction until a commit or rollback is issued. This feature is optional and can be enabled by setting the system property ‘stmtatomicity’.

stmtatomicity property

Enabling the stmtatomicity property, allows the JDBC driver to set the transactions atomicity at statement level.

This java property can be set from the command line as:

```
Djdbcmx.stmtatomicity={ON|OFF}
```

where

`ON`

statement level atomicity

`OFF`

transaction level atomicity. The default is set to `OFF.`

Note: This functionality is already available in JDBC/MX H10AAB and V32AAU versions.

Managing Nonblocking JDBC/MX

Blocking mode with the JDBC/MX driver causes the whole JVM process to be blocked when an SQL operation occurs. Nonblocking mode causes the JDBC/MX driver to block only the thread that invokes the SQL operation and not the whole JVM process. In a multithreaded Java application, the nonblocking JDBC/MX feature enables the JVM to schedule other threads concurrently while each SQL operation is being done by a thread.

By default, JDBC/MX uses the nonblocking mode. You can disable nonblocking JDBC/MX in a Java application by setting the `sqlmx_nowait` property to `OFF` by using the `-Djdbcmx.sqlmx_nowait` option in the command line. The syntax is:

```
-Djdbcmx.sqlmx_nowait={ ON | OFF }
```

where

`ON`

specifies nonblocking JDBC/MX. The default is `ON`.

`OFF`

specifies process blocking JDBC/MX.

You can also programmatically disable or enable nonblocking JDBC/MX by setting the `sqlmx_nowait` property within the program. Depending on your application, set this property as follows:

- In JDBC/MX applications that obtain a JDBC connection by using the `DriverManager` class, set this property before the JDBC/MX driver is loaded.
- In JDBC/MX applications that obtain a JDBC connection by using JNDI API with the `DataSource` interface, set this property before the `DataSource` object is created.

JDBC connection can now be simultaneously used from multiple threads. Multiple threads working on SQL statements are allowed to share the same connection. Therefore, single connection context is used across multiple threads and the operations associated with the connection object are made thread safe.

If you are an application developer writing multithreaded Java applications that use nonblocking JDBC, follow these recommendations:

- Create only one JDBC connection per thread. Applications obtaining multiple JDBC connections in single thread do not run the SQL/MX operations concurrently and can waste system resources because each connection requires its own SQL/MX compiler process.
- Do not share JDBC Java objects--such as `Statement` or `ResultSet` objects--across threads for purposes other than canceling the SQL operation with the `Cancel`() method.
- Be aware of the non-preemptive nature of the thread implementation in NonStop Server for Java 4. A CPU-bound thread runs to its completion without providing an opportunity for the thread scheduler to schedule a different thread.
- If an application is written to share connection across multiple threads, then the connection properties should not be modified.

Setting Batch Processing for Prepared Statements

You can improve the performance of batch processing when using the `PreparedStatement.executeBatch()` method by setting the `batchBinding` property. When the `batchBinding` property is set, the statements are batched in the `executeBatch()` operation.

When a JDBC application sets the `batchBinding` property, the JDBC/MX driver allocates resources relative to the specified binding size.

To set the `batchBinding` size, specify the `batchBinding` property in the command line. The syntax is:

```
-Djdbcmx.batchBinding=binding_size
```

where `binding_size` is a positive, signed, long integer that specifies the maximum number of `PreparedStatement.executeBatch()` method statements that the JDBC/MX driver can bind together for execution. The integer value can be in the range of 0 to 2 gigabytes.

Considerations

- The values allowed for `binding_size` can result in your application running out of memory. Check that you set the `binding_size` to a size appropriate for the memory limits.
- If the number of statements is greater than the binding size, the JDBC/MX driver breaks the execution of statements into blocks whose sizes are based on the binding size.
- Even if the JDBC application does not call for batch execution, setting the `jdbcmx.batchBinding` property causes the allocation of database resources relative to the specified binding size.
- When the `jdbcmx.batchBinding` property is not set, the `PreparedStatement.executeBatch()` method returns a row-count array that contains the number of rows affected by the corresponding statement for each item in the array. By default, the JDBC/MX driver performs batch processing by returning a row-count array.
- When the `jdbcmx.batchBinding` property is set, the detailed information indicated in the preceding bulleted item is no longer available. If the statement execution succeeds, the row-count item is set to `Statement.SUCCESS_NO_INFO` in compliance with the JDBC 3.0 specification. The `PreparedStatement.getUpdateCount()` method returns the total number of rows affected by all the statements executed by the `PreparedStatement.executeBatch()` method.

Setting the reserveDataLocators Property

The reserveDataLocators property sets the number of data locators to be reserved for a process for storing data in a LOB table. The default value for reserving data locators is 100. The property is of the form:

```
jdbcmx.reserveDataLocators=n
```

where `n` is an integer value of the number of data locators to be reserved. Do not set a value much greater than the number of data locators actually needed. For more information about data locator use, see Reserving Data Locators.

To change this value for a JDBC application, specify this property from the command line. For example, the following command reserves 150 data locators for program `myProgramClass.`

```
java -Djbcmx.reserveDataLocators=150 myProgramClass
```

Supported Character Set Encodings

Java applications using the JDBC/MX driver can specify the Java `file.encoding` property to set the default encoding to any character set supported by Java if no SQL literals exist in the program. If the program has SQL literals, the program should use only the Java encoding sets that correspond to SQL/MX supported sets.

The JDBC/MX driver supports the reading and writing of SQL CHAR, VARCHAR, VARCHAR_LONG, and VARCHAR_WITH_LENGTH data types only when using the SQL/MX supported character sets listed in the subsequent table.

The JDBC/MX driver encodes and decodes String data types as a function of the associated character set name for the particular SQL database column independent of the default encoding.

The format of the Java `file.encoding` property is:

```
-Dfile.encoding=encoding
```

Note: SQL/MX supports a subset of encoding sets supported by NonStop Server for Java 4.

Corresponding SQL/MX Character Sets and Java Encoding Sets

SQL/MX Character Set	Corresponding Java Encoding Set—Canonical Name for java.io API	Corresponding Java Encoding Set—Canonical Name for java.io and java.lang API	Description
ISO88591	ISO-8859-1	ISO8859_1	Single-character, 8-bit, character set for character-data type. It supports English and other Western European languages.
UCS2	UTF-16BE	UnicodeBigUnmarked	Universal Character Set encoded in 2 bytes. Double-character Unicode character set in UTF16 big-endian encoding. **NOTE:** UCS2 is supported in SQL/MX tables only.

KANJI	Shift_JIS	SJIS	The multibyte character set widely used on Japanese mainframes. It is composed for a single-byte character set and a double-byte character set. It is a subset of Shift JIS (the double character portion). Its encoding is big-endian. **NOTE:** KAJNI is supported in SQL/MP tables only.
KSC5601	EUC-KR (Code Set 1)	EUC_KR	Double-character character set required on systems used by government and banking within Korea. Its encoding is big endian. **NOTE:** KSC5601 is supported in SQL/MP tables only.

For complete information on character sets supported by SQL/MX and any additional limitations on support for SQL/MP tables, see the SQL/MX Reference Manual.

For complete information about NonStop Server for Java 5 support for encodings see Supported Encodings (http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html).

Working with BLOB and CLOB Data

This section describes working with BLOB and CLOB data in JDBC applications. You can use the standard interface described in the JDBC 3.0 API specification to access BLOB and CLOB data in NonStop SQL/MX tables with support provided by the JDBC/MX driver.

BLOB and CLOB are not native data types in an SQL/MX database. But, database administrators can create SQL/MX tables that have BLOB and CLOB columns by using the JDBC/MX driver or special SQL syntax in MXCI as described in the next section, Managing the SQL/MX Tables for BLOB and CLOB Data.

For management purposes, CLOB and BLOB data is referred to as large object (LOB) data, which can represent either data type.

> **Note:** Support for BLOB and CLOB data requires SQL/MX tables.

The section is organized in topics by category as follows:

Category	Topic
The Physical Files	• Architecture for LOB Support • Setting Properties for the LOB Table • Specify the LOB Table Name
Accessing CLOB Data	• Storing CLOB Data • Reading CLOB Data • Updating CLOB Data • Deleting CLOB Data
Accessing BLOB Data	• Storing BLOB data • Reading Binary Data from a BLOB Column • Updating BLOB Data • Deleting BLOB Data

Miscellaneous	• NULL and Empty BLOB or CLOB Value • Transactions Involving Blob and Clob Access • Access Considerations for Clob and Blob Objects

For full working examples showing how to access `BLOB` and `CLOB` data, see Appendix A.

For information about creating and managing tables for `BLOB` and `CLOB` data, see Managing the SQL/MX Tables for `BLOB` and `CLOB` Data.

Architecture for LOB Support

The tables that support LOB data are:

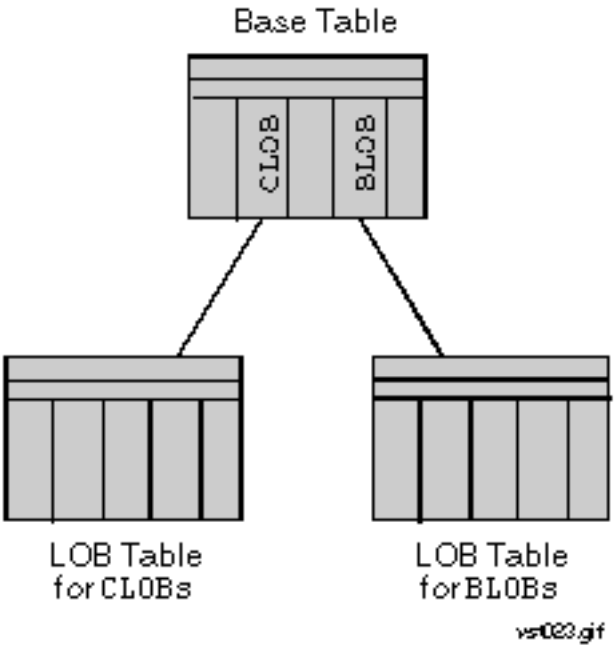
Base table

Referenced by JDBC applications to insert, store, read, and update `BLOB` and `CLOB` data. In the base table, the JDBC/MX driver maps the `BLOB` and `CLOB` columns into a data-locator column. The data-locator column points to the actual LOB data that is stored in a separate user table called the LOB table.

LOB table

Actually contains the `BLOB` and `CLOB` data in chunks. A `Clob` or `Blob` object is identified by a data locator. LOB tables have two formats: LOB table for `BLOB` data and a LOB table for `CLOB` data.

LOB Architecture: Tables for LOB Data Support



Setting Properties for the LOB Table

Before running the JDBC application that uses BLOB and CLOB data through the JDBC API, the database administrator must create the LOB tables. For information on creating LOB tables, see Managing LOB Data with the JDBC/MX Lob Admin Utility.

The JDBC applications that access BLOB or CLOB data must specify the associated LOB table names and, optionally, configure the `reserveDataLocator` property. These tasks are described in the topics:

- Specifying the LOB Table
- Reserving Data Locators

Specifying the LOB Table

At run time, a user JDBC application notifies the JDBC/MX driver of the name, or names, of the LOB tables associated with the `CLOB` or `BLOB` columns of the base tables being accessed by the application. One LOB table, or separate tables, can be used for `BLOB` and `CLOB` data.

The JDBC application specifies a LOB table name either through a system parameter or through a Java `Property` object by using one of the following properties, depending on the LOB column type:

LOB Column Type	Property name
`BLOB`	`blobTableName`
`CLOB`	`clobTableName`

For more information about using these properties, see LOB Table Name Properties.

Reserving Data Locators

A data locator is the reference pointer value (SQL `LARGEINT` data type) that is substituted for the `BLOB` or `CLOB` column in the base table definition. Each object stored into the LOB table is assigned a unique data locator value. Because the LOB table is a shared resource among all accessors that use the particular LOB table, reserving data locators reduces contention for getting the next value. By using a default setting of 100 reserved data locators, each JVM instance can insert 100 large objects (not chunks) before needing a new allocation.

You can specify the number of data locators (n) to reserve for your application by using the JDBC/MX system property `jdbcmx.reserveDataLocators` in the command line.

For information about specifying this property, see Setting the reserveDataLocators Property.

Storing CLOB Data

Inserting CLOB Columns by Using the Clob Interface

When you insert a row containing a `CLOB` data type, and before the column can be updated with real `CLOB` data, you can insert a row that has an "empty" `CLOB` value . You can insert an empty `CLOB` value in a NonStop SQL/MX database by specifying `EMPTY_CLOB()` function for the `CLOB` column in the insert statement.

The JDBC/MX driver scans the SQL string for the `EMPTY_CLOB()` function and substitutes the next-available data locator.

Then, you must obtain the handle to the empty `CLOB` column by selecting the `CLOB` column for update.

Note the limitation: Do not rename the CLOB column in the select query.

The following code illustrates how to obtain the handle to an empty `CLOB` column:

```
Clob myClob = null;
Statement s = conn.createStatement();
ResultSet rs = s.executeQuery("Select myClobColumn
      from myTable where ….for update");
if (rs.next())
     myClob = rs.getClob(1);
```

You can now write data to the `CLOB` column. See Writing ASCII or Unicode Data to a `CLOB` Column.

Writing ASCII or Unicode Data to a CLOB Column

You can write ASCII or Unicode data to a CLOB column as follows.

ASCII Data

You can write ASCII or Unicode data to the `CLOB` column by using the `Clob` interface. The following code illustrates using the `setAsciiStream` method of the `Clob` interface to write `CLOB` data.

```
Clob myClob = null;
// stream begins at position 1
long pos = 1;
// Example string containing data
String s = "TEST_CLOB";
for (int i=0; i<5000; i++) s = s + "DATA";
// Obtain the output stream to write Clob data
OutputStream os = myClob.setAsciiStream(pos);
// write Clob data using OutputStream
byte[] myClobData = s.getBytes();
os.write(myClobData);
```

The JDBC/MX driver splits the output stream into chunks and stores the chunks in the LOB table.

Unicode Data

The following code illustrates how to write Unicode data to a `CLOB` column after obtaining the handle to the empty `CLOB` column.

```
Clob myClob = null;
// stream begins at position 1
long pos = 1;
// Example string containing the Unicode data
String s =  TEST_UNICODE_DATA ;
// Obtain the output stream to write Clob data
Writer cw = myClob.setCharacterStream(pos);
// write Clob data using Writer
char[] myClobData = s.toCharArray();
cw.write(myClobData);
```

Inserting CLOB Data by Using the PreparedStatement Interface

You can use the `PreparedStatement` interface to insert a CLOB column with data as follows:

- ASCII Data
- Unicode Data

ASCII Data

You can insert a `CLOB` column with ASCII or Unicode data from a `FileInputSteam`. You must use the `PreparedStatement` interface to insert the `CLOB` column.

```
FileInputStream inputAsciiStream = new
FileInputStream(myClobTestFile);
int clobLen = inputAsciiStream.available();
PreparedStatement ps = conn.prepareStatement("insert
      into myTable (myClobColumn) values (?)");
ps.setAsciiStream(1, inputAsciiStream, clobLen);
ps.executeUpdate();
```

The JDBC/MX driver reads the data from `FileInputSteam` and writes the data to the LOB table. The JDBC/MX driver substitutes the next-available data locator for the parameter of the `CLOB` column in the table.

Unicode Data

You can insert a `CLOB` column with Unicode data from a `FileReader`. You must use the `PreparedStatement` interface to insert the `CLOB` column.

```
FileReader inputReader = new FileReader(myClobTestFile);
PreparedStatement ps = conn.prepareStatement("insert
      into myTable (myClobColumn) values (?)");
ps.setCharacterStream(1, inputReader, (int)myClobTestFile.length());
ps.executeUpdate();
```

The JDBC/MX driver reads the data from `FileReader` and writes the data to the LOB table. The JDBC/MX driver substitutes the next available-data locator for the parameter of the `CLOB` column in the table.

Inserting a Clob Object by Using the setClob Method

Your JDBC application cannot directly instantiate a `Clob` object. To perform an equivalent operation:

1. Obtain a `Clob` object by using the `getClob` method of the `ResultSet` interface.
2. Insert the `Clob` object into another row by using the `setClob` method of the `PreparedStatement` interface.

In this situation, the JDBC/MX driver generates a new data locator and, when the `PreparedStatement` is executed, copies the contents of the source `Clob` into the new `Clob` object.

Reading CLOB Data

Reading ASCII Data from a CLOB Column

You can read ASCII or Unicode data from a `CLOB` column by using the `Clob` interface or `InputStream`.

The following code illustrates how to read the ASCII data from the `CLOB` column by using the `Clob` interface:

```
// Obtain the Clob from ResultSet
Clob myClob = rs.getClob("myClobColumn");
// Obtain the input stream to read Clob data
InputStream is = myClob.getAsciiStream();
// read Clob data using the InputStream
byte[] myClobData = new byte[length];
int readLen = is.read(myClobData, offset, length);
```

To read ASCII or Unicode data from the `CLOB` column by using `InputStream`:

```
// obtain the InputStream from ResultSet
InputStream is = rs.getAsciiStream("myClobColumn");
// read Clob data using the InputStream
byte[] myClobData = new byte[length];
int readLen = is.read(myClobData, offset, length);
```

Reading Unicode Data from a CLOB Column

You can read Unicode data from the `CLOB` column by using the `Clob` interface or `Reader`. The following code illustrates how to read the Unicode data from the `CLOB` column by using the `Clob` interface.

```
// Obtain the Clob from ResultSet
Clob myClob = rs.getClob("myClobColumn");
// Obtain the input stream to read Clob data
Reader cs = myClob.getCharacterStream();
// read Clob data using Reader
char[] myClobData = new char[length];
int readLen = cs.read(myClobData, offset, length);
```

To read Unicode data from the `CLOB` column by using a `Reader`:

```
// obtain the Reader from ResultSet
Reader cs = rs.getCharacterStream("myClobColumn");
// read Clob data using the InputStream
char[] myClobData = new char[length];
int readLen = cs.read(myClobData, offset, length);
```

Updating CLOB Data

You can make updates to `CLOB` data by using the methods in the `Clob` interface or by using the `updateClob` method of the `ResultSet` interface. The JDBC/MX driver makes changes directly to the `CLOB` data. Therefore, the JDBC/MX driver returns `false` to the `locatorsUpdateCopy` method of the `DatabaseMetaData` interface. Applications do not need to issue a separate update statement to update the `CLOB` data.

Make updates to `CLOB` data in the following ways:

- Updating `Clob` Objects with the updateClob Method
- Replacing Clob Objects

Updating Clob Objects with the updateClob Method

Unlike some LOB support implementations, the JDBC/MX driver updates the `CLOB` data directly in the database. So, when the `Clob` object is the same in the `updateClob` method as the `Clob` object obtained using `getClob`, the `updateRow` method of the `ResultSet` interface does nothing with the `Clob` object.

When the `Clob` objects differ, the `Clob` object in the `updateClob` method behaves as if the `setClob` method was issued. See Inserting a Clob Object with the setClob Method.

Replacing Clob Objects

You can replace Clob objects in the following ways:

- Use the `EMPTY_CLOB()` function to replace the `Clob` object with the empty `Clob` object, then insert new data as described under Inserting CLOB Columns by Using the Clob Interface.
- Use the `PreparedStatement.setAsciiStream()` or `setCharacterStream()` method to replace the existing `Clob` object with new `CLOB` data.
- Use the `setClob` or `updateClob` method to replace the existing `CLOB` objects as explained earlier under Inserting a Clob Object with the setClob Method and Updating Clob Objects with the updateClob Method.

Deleting CLOB Data

To delete CLOB data, the JDBC application uses the SQL DELETE statement to delete the row in the base table.

When the row containing the CLOB column is deleted by the JDBC application, the corresponding CLOB data is automatically deleted by the delete trigger associated with the base table. For information about triggers, see Using an SQL/MX Trigger to Delete LOB Data.

See also NULL and Empty BLOB or CLOB Value.

Storing BLOB Data

You can perform operations similar to those used on CLOB columns as those used on BLOB columns by using the Blob interface. You can:

- Use the EMPTY_BLOB() function in the insert statement to create an empty BLOB column in the database.
- Use setBinaryStream method of the Blob interface to obtain the InputStream to read BLOB data.
- Use getBinaryStream method of the Blob interface to obtain the OutputStream to write BLOB data.
- Use setBinaryStream of the PreparedStatement interface to write the data to the BLOB column.

The details of these operations are discussed in the following topics:

- Inserting a BLOB Column Using the Blob Interface
- Writing Binary Data to a BLOB Column
- Inserting a BLOB Column by Using the PreparedStatement Interface
- Inserting a Blob Object by Using the setBlob Method

Inserting a BLOB Column by Using the Blob Interface

When you insert a row containing a BLOB data type, you can insert the row with an "empty" BLOB value before the column can be updated with real BLOB data. You can insert an empty BLOB value in an SQL/MX database by specifying EMPTY_BLOB() function for the BLOB column in the insert statement.

The JDBC/MX driver scans the SQL string for the EMPTY_BLOB() function and substitutes the next-available data locator.

Then, you must obtain the handle to the empty BLOB column by selecting the BLOB column for update.

The following code illustrates how to obtain the handle to an empty BLOB column:

```
Blob myBlob = null;
Statement s = conn.createStatement();
ResultSet rs = s.executeQuery("Select myBlobColumn
      from myTable where …For update");
if (rs.next())
     myBlob = rs.getBlob(1);
```

You can now write data to the BLOB column. See Writing Binary Data to a BLOB Column.

Writing Binary Data to a BLOB Column

You can write data to the BLOB column by using Blob interfaces. The following code illustrates using the setBinaryStream method of the Blob interface to write BLOB data.

```
Blob myBlob = null
// Stream begins at position 1
long pos = 1;
// Example string containing binary data
String s = "BINARY_DATA";
for (int i=0; i<5000; i++) s = s + "DATA";
// Obtain the output stream to write Blob data
OutputStream os = myBlob.setBinaryStream(pos);
// write Blob data using OutputStream
byte[] myBlobData = s.getBytes();
os.write(myBlobData);
```

The JDBC/MX driver splits the output stream into chunks and stores the chunks in the LOB table.

Inserting a BLOB Column by Using the PreparedStatement Interface

You can also insert a BLOB column that has binary data from a FileInputSteam. You must use PreparedStatement interface to insert the BLOB column.

```
FileInputStream inputBinary = new FileInputStream(myBlobTestFile);
int blobLen = inputBinary.available();
PreparedStatement ps = conn.prepareStatement("insert
      into myTable (myBlobColumn) values (?)");
ps.setBinaryStream(1, inputBinary, blobLen);
ps.executeUpdate();
```

The JDBC/MX driver reads the data from FileInputSteam and writes the data to the LOB table. The

JDBC/MX driver substitutes the next-available data locator for the parameter of the BLOB column in the table.

Inserting a Blob Object by Using the setBlob Method

Your JDBC application cannot directly instantiate a Blob object. To perform an equivalent operation:

1. Obtain a Blob object by using the getClob method of the ResultSet interface.
2. Insert the Blob object into another row by using the setBlob method of the PreparedStatement interface.

In this situation, the JDBC/MX driver generates a new data locator and copies the contents of the source Blob into the new Blob object when the application issues the setBlob method of the PreparedStatement interface.

Reading Binary Data from a BLOB Column

You can read binary data from the BLOB column by using the Blob interface or InputStream. The following code illustrates how to read the binary data from the BLOB column by using the Blob interface:

```
// Obtain the Blob from ResultSet
Blob myBlob = rs.getBlob("myBlobColumn");
// Obtain the input stream to read Blob data
InputStream is = myBlob.getBinaryStream();
// read Blob data using the InputStream
byte[] myBlobData = new byte[length];
is.read(myBlobData, offset, length);
```

To read binary data from the BLOB column by using InputStream

```
// obtain the InputStream from ResultSet
InputStream is = rs.getBinaryStream("myBlobColumn");
// read Blob data using the InputStream
byte[] myBlobData = new byte[length];
is.read(myBlobData, offset, length);
```

Updating BLOB Data

You can update BLOB data by using the methods in the Blob interface or by using the updateClob method of the ResultSet interface. The JDBC/MX driver makes changes to the BLOB data directly. Hence, the JDBC/MX driver returns false to the locatorsUpdateCopy method of the DatabaseMetaData interface. Applications do not need to issue a separate update statement to update the BLOB data.

Update BLOB data in the following ways.

- Updating Blob Objects by using the updateBlob Method
- Replacing Blob Objects

Updating Blob Objects by Using the updateBlob Method

Unlike some LOB support implementations, the JDBC/MX driver updates the BLOB data directly in the database. So, when the Blob object is the same in the updateBlob method as the object obtained using getBlob, the updateRow method of the ResultSet interface does nothing with the Blob object.

When the Blob objects differ, the Blob object in the updateBlob method behaves as if the setBlob method was issued. See Inserting a Blob Object with the setBlob Method.

Replacing Blob Objects

You can replace Blob objects in the following ways:

- Use the EMPTY_BLOB() function to replace the Blob object with the empty Blob object.
- Replace an existing Blob object in a row by inserting the Blob with new data as described under Inserting a BLOB Column Using the Blob Interface.
- Use the setBinaryStream() method to of the PreparedStatement interface replace the existing Blob object with new BLOB data.
- Use the setBlob or updateBlob methods to replace the existing BLOB objects as explained earlier under Inserting a Blob Object with the setBlob Method and Updating Blob Objects with the UpdateBlob Method.

Deleting BLOB Data

To delete `BLOB` data, the JDBC application uses the SQL DELETE statement to delete the row in the base table.

When the row containing the `BLOB` column is deleted by the application, the corresponding `BLOB` data is automatically deleted by the delete trigger associated with the base table. For information about triggers, see Using an SQL/MX Trigger to Delete LOB Data.

See also NULL and Empty BLOB or CLOB Value.

NULL and Empty BLOB or CLOB Value

The data locator can have a NULL value if the `BLOB` or `CLOB` column is omitted in the insert statement. The JDBC/MX driver returns NULL when the application retrieves the value for such a column.

When the application uses the `EMPTY_BLOB()` method or `EMPTY_CLOB()` method to insert empty `BLOB` or `CLOB` data into the `BLOB` or `CLOB` column, JDBC/MX driver returns the `Blob` or `Clob` object with no data.

Transactions Involving Blob and Clob Access

HP recommends that your JDBC applications control the transactions when the `BLOB` columns or `CLOB` columns are accessed either by using the external transaction or by setting the connection to manual commit mode.

If executing a prepared statement involving BLOB or CLOB data with autocommit mode enabled and an external transaction does not exist, the JDBC/MX driver mimics autocommit mode. This operation ensures that inserts or updates of LOB data are committed only after both the base table and LOB tables are modified.

In some instances an `outputStream` or `Writer` object is returned to the application when the object can be held for an unknown period of time. Therefore, the following interfaces throw the exception, `Autocommit is on and LOB objects are involved`, exception when LOB data is involved, autocommit is enabled, and an external transaction does not exist:

- `Clob.setAsciiStream`
- `Clob.setCharacterStream`
- `Blob.setBinaryStream`

If an SQL/MX or FS exception occurs while the base table and LOB table are being updated, the internal transaction used for this operation is rolled back, and an exception is thrown.

When an SQL/MX or file system exception occurs while JDBC/MX mimics autocommit mode for the base table and the insert or update operations on a LOB table, the internal transaction used for this operation is rolled back and the following exception is thrown:

```
Transaction error {0} - {1} while updating LOB tables
```

For the description, see the message information under sqlcode 29070.

The JDBC/MX driver reserves data locators in its own transaction to improve the concurrency among the different processes trying to reserve the data locators.

For more information, see Transactions.

Access Considerations for Clob and Blob Objects

The JDBC/MX driver allows all the valid operations on the current `Clob` object or `Blob` object, called a LOB object. LOB objects are current as long as the row that contains these LOB objects is the current row. The JDBC/MX driver throws an SQLException, issuing the following message, when the application attempts to perform operations on a LOB object that is not current:

```
Lob object {object-id} is not current
```

Only one `InputStream` or `Reader` and one `OutputStream` or `Writer` can be associated with the current LOB object.

- When the application obtains the `InputStream` or `Reader` from the LOB object, the JDBC/MX driver closes the `InputStream` or `Reader` that is already associated with the LOB object.
- Similarly, when the application obtains the `OutputStream` or `Writer` from the LOB object, the JDBC/MX driver closes the `OutputStream` or `Writer` that is already associated with the LOB object.

Managing the SQL/MX Tables for BLOB and CLOB Data

BLOB and CLOB are not native data types in an SQL/MX database. But, database administrators can create SQL/MX tables that have BLOB and CLOB columns by using the JDBC/MX driver or special SQL syntax in MXCI as described in this section. For management purposes, CLOB and BLOB data is referred to as large object (LOB) data, which can represent either data type.

Note: Support for BLOB and CLOB data requires SQL/MX tables.

Before using this section, be sure to see the file descriptions for the tables that contain LOB data. This information is under the topic Architecture for LOB Support in the preceding section.

With the exception above, this section provides the information that database administrators need to create and manage the tables required to support LOB data. The topics are:

- Creating Base Tables that Have LOB Columns
- Managing LOB Data by Using the JDBC/MX Lob Admin Utility
- Using SQL/MX Triggers to Delete LOB Data
- Limitations of the BLOB and CLOB Data Types

Creating Base Tables that Have LOB Columns

You can write JDBC programs to create base tables that have LOB columns or you can use the SQL/MX conversational interface MXCI as described in the following topics:

- Data Types for LOB Columns
- Using MXCI to Create Base Tables that Have LOB Columns
- Using JDBC Programs to Create Base Tables that Have LOB Columns

Data Types for LOB Columns

The data types for the LOB columns are:

CLOB
: Character large object data

BLOB
: Binary large object data

Note: The `CLOB` and `BLOB` data type specification is special syntax that is allowed for use in base tables accessed by JDBC/MX driver as described in this manual.

Using MXCI To Create Base Tables that Have LOB Columns

Before using the procedure to create the tables, note that when using MXCI to create base tables, you must enter the following special command in the MXCI session to enable the base table creation of tables that have LOB (`BLOB` or `CLOB`) columns:

```
CONTROL QUERY DEFAULT JDBC_PROCESS 'TRUE'
```

Follow these steps to create a base table that has LOB columns:

1. At the OSS prompt, invoke the SQL/MX utility MXCI. Type:

   MXCI
2. Type the following command to enable creating tables that have LOB columns:

   ```
   CONTROL QUERY DEFAULT JDBC_PROCESS 'TRUE'
   ```
3. Type the CREATE TABLE statement; for example, you might use the following simple form of the statement:

   ```
   CREATE TABLE table1 (c1 INTEGER NOT NULL, c2 CLOB, c3 BLOB, PRIMARY
   KEY(c1))
   ```

where;

table1
: The name of the base table.

c1
: Column 1, defined as the `INTEGER` data type with the NOT NULL constraint.

c2
: Column 2, defined as the `CLOB` data type.

c3
: Column 3, defined as the `BLOB` data type.

PRIMARY KEY
: Specifies c1 as the primary key.

Use this example as the archetype for creating base tables. For information about valid names for tables (`table1`) and columns (`c1`, `c2`, and `c3`) and for information about the CREATE TABLE statement, see the *SQL/MX Reference Manual*.

Using JDBC Programs To Create Base Tables that Have LOB Columns

When using a JDBC Program to create base tables that have LOB columns, simply put the CREATE TABLE statements in the program as you would any other SQL statement. For an example of the CREATE TABLE

statement, see the preceding discussion Using MXCI to Create Base Tables that Have LOB Columns.

Managing LOB Data by Using the JDBC/MX Lob Admin Utility

The JDBC/MX driver provides the JDBC/MX Lob Admin Utility that you can use for these tasks:

- Creating the LOB table (a table that holds LOB data).
- Creating the SQL/MX triggers for the LOB columns in the base tables to ensure that orphan LOB data does not occur in a LOB table.

Information about using the JDBC/MX Lob Admin Utility is provided in these topics.

- Running the JDBC/MX Lob Admin Utility
- Help Listing from the JDBC/MX Lob Admin Utility
- Using SQL/MX Triggers to Delete LOB Data

Running the JDBC/MX Lob Admin Utility

Run the JDBC/MX Lob Admin utility in the OSS environment.

The format of the command is:

```
java [java_options] JdbcMxLobAdmin [prog_options] [table_name]
```

java_options

The java_options are properties that can be specified on the `java` command line in the `-D` option.

Property Specification	Description
`jdbcmx.blobTableName`	Specifies the LOB table for using `BLOB` columns. Required if `BLOB` columns are involved. See LOB Table Name Properties.
`jdbcmx.clobTableName`	Specifies the LOB table for using `CLOB` columns. Required if `CLOB` columns are involved. See LOB Table Name Properties.
`jdbcmx.catalog`	Sets the default catalog. See Default Catalog and Schema.
`jdbcmx.schema`	Sets the default schema. See Default Catalog and Schema.

program_options

prog_option	Description
-help	Displays help information
-exec	Runs the SQL statements that are generated.
-create	Generates SQL statements to create LOB tables. These statements describe the architecture of the tables and, therefore, provide a description of the LOB tables.
-trigger	Generates SQL statements to create triggers for the designated table. The table must exist.
-unicode	Generates SQL statements to create unicode LOB tables. Use only for `CLOB` data.
-drop	Generate SQL statements to drop triggers for the designated table. The table must exist.
-out	Writes the SQL statements to a specified file in OSS file space.

table_name

The *table_name* represents a base table that contains LOB columns. The *table_name* is of the form:

```
[catalogName.][schemaName.]baseTableName
```

For information about catalog, schema, and table names, see the *SQL/MX Reference Manual*.

Help Listing from the JDBC/MX Lob Admin Utility

The command to display JDBC/MX Lob Admin utility help appears below followed by the help listing.

```
java JdbcMxLobAdmin -help
```

Hewlett-Packard JDBC/MX Lob Admin Utility 2.0 (c) Copyright 2004, 2005 Hewlett-Packard Development Company, LP.

```
java [<java_options>] JdbcMxLobAdmin [<prog_options>] [<table_name>]

<java_options> is:
    [-Djdbcmx.clobTableName=<clobTableName>]
    [-Djdbcmx.blobTableName=<blobTableName>]
    [-Djdbcmx.catalog=<catalog>]
    [-Djdbcmx.schema=<schema>]
```

```
<prog_options> is:
    [-exec] [-create] [-trigger] [-help] [-drop] [-out <filename>]
where -help    - Display this information.
      -exec    - Execute the SQL statements that are generated.
      -create  - Generate SQL statements to create LOB tables.
      -trigger - Generate SQL statements to create triggers for <table_name>.
      -unicode - Generate SQL statements to create unicode LOB tables
                 (CLOB only).
      -drop    - Generate SQL statements to drop triggers for <table_name>.
      -out     - Write the SQL statements to <filename>.

<clobTableName> | <blobTableName> is:
    <catalogName>.<schemaName>.<lobTableName>

<table_name> is:
    [<catalogName>.][<schemaName>.]<baseTableName>

<baseTableName> is the table that contains LOB column(s).
<lobTableName> is the table that contains the LOB data.
```

Using SQL/MX Triggers to Delete LOB Data

Use the JDBC/MX Lob Admin Utility to generate triggers to delete LOB data from the LOB table when the base row is deleted. These triggers ensure that orphan LOB data does not occur in the LOB table. To manage the triggers, use these JDBC/MX Lob Admin Utility options:

`-trigger`

Generates SQL statements to create triggers.

`-drop`

Generates SQL statements drop triggers.

For example, the following command (typed on one line) generates the SQL statements to create the triggers for the base table `sales.paris.pictures,` which contains a `BLOB` column, and executes those statements.

```
java -Djdbcmx.blobTableName=sales.paris.lobTable4pictures JdbcMxLobAdmin
-trigger
     -exec sales.paris.pictures
```

Limitations of the CLOB and BLOB Data Types

Limitations of the CLOB and BLOB data types, collectively referred to as LOB data, are:

- LOB columns can only be in the target column list of these SQL statements:
  - INSERT statement,
  - Select list of a SELECT statement

    - Column name in the SET clause of an UPDATE statement
- LOB columns cannot be referenced in the SQL/MX functions and expressions.
- LOB data is not deleted from the LOB table when the base row is deleted unless a trigger is established. For information about triggers, see Using an SQL/MX Trigger to Delete LOB Data.
- LOB data is not accessible if the base table name is changed.
- LOB columns cannot be copied to another table by using the SQL/MX utility commands.
- The name of a base table that has `CLOB` or `BLOB` columns must be unique across all catalogs and schemas when more than one of these base tables share a single LOB table.

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

Module File Caching (MFC)

The Module File Caching (MFC) feature shares the SQL/MX prepared statement plans among the JDBC/MX T2 database connections and JVM processes. It helps in reducing the SQL/MX compilation time during the steady state of the JDBC/MX T2 application, thereby reducing resource consumption.

> **Note:** Module File Caching is supported only on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

The topics discussed in this chapter are:

Design of MFC

For information on the MFC design, see the *HP NonStop SQL/MX Connectivity Service Manual*.

Enabling MFC

The following are the two new properties which are required for using MFC in an application that uses the JDBC/MX T2 Driver.

- `modulecaching` Property:

  To enable MFC, the value of this property must be set to ON.

- `compiledmodulelocation` Property:

  The value for this property must be a valid directory name. For example: `/usr/temp`. This is the location where the intermediary files such as `*`.mdf for MFC are generated.

Both these properties must be set to enable MFC.

Limitations of MFC

- MFC cache should be used only on production systems. It should not be used on development or User Acceptance Testing (UAT) systems where SQL/MX undergoes changes.
- It does not handle session-specific SQL/MX Control Query Defaults (CQD) and SQL/MX Control Query Shape (CQS).
- It is recommended to set the CQD in SQL/MX for disabling the auto-recompilation feature of the SQL/MX while using with MFC. This ensures that automatic recompilations are avoided due to changes in SQL/MX objects because the plans are generated in the module file. The application will receive an SQL/MX exception if there is an auto-recompilation required for the query. You must clean the stale module files before continuing with the application.
- For lightweight queries, MFC performs only marginally better than the SQL/MX compile.
- Combining external statement cache with MFC does not yield memory benefits. The WebLogic Server (WLS) statement cache is an example of external statement cache. It is recommended that you use the JDBC/MX T2 statement cache.
- Some scalar functions such as ABS, SUM, and AVG are not handled through the MFC in the first release. For information on the scalar functions, see the *HP NonStop SQL/MX Reference Manual*.

Troubleshooting MFC

The troubleshooting of MFC includes:

Benefits of MFC

JDBC applications using the `java.sql.PreparedStatement` object result in lower processor utilization, lower memory consumption, and better response time.

Setting an Environment for MFC

See EnablingMFC.

.lock Files

The *.lock files are generated for every query that pass through the MFC module file creation process. These files are also used for synchronizing, so that different connections do not re-create the same module file. These *.lock files are deleted once the binary module in the `/usr/tandem/sqlmx/USERMODULES` directory is created successfully.

The *.lock files are not deleted for the queries that cannot create module files.

.mdf Files

These temporary files are generated during preprocessing. These .mdf files are retained for easier support and troubleshooting.

Disk Activity

The MFC access plans, stored in the disk OSH location (`/usr/tandem/sqlmx/USERMODULE`), increases the processor utilization of the disks. To overcome this problem, use the fileset for that directory. It is beneficial to have OSS caching on data volumes. To enable fileset and OSS caching, see Enable Fileset and OSS Caching.

Note: If a DDL alters, it is recommended that you run the management script (`mgscript`) to delete module files associated with that table or catalog. For information on the management script, see *Managing MFC* in the *HP NonStop SQL/MX Connectivity Service Manual*.

Enable Fileset and OSS Caching

To add a fileset pointing to the `USERMODULES` directory, perform the following steps:

1. At a TACL prompt, enter:

   ```
   SCF
   ```

   and then enter:

   ```
   assume $zpmon
   ```

2. At an SCF prompt, enter the SCF command:

   ```
   add server #zpns1,cpu 1,backupcpu 2
   ```

3. Add a fileset:

   ```
   add fileset mxc1,nameserver #zpns1,catalog $oss, pool
   mxcpool, mntpoint "/usr/tandem/sqlmx/USERMODULES"
   ```

4. Verify the status of the fileset:

   ```
   info fileset mxc1,detail
   ```

5. Start the fileset:

   ```
   start fileset mxc1
   ```

To enable OSS caching, perform the following steps:

1. At a TACL prompt, enter:

   ```
   SCF
   ```

   and then enter:

   ```
   assume $zpmon
   ```

2. At an SCF prompt, enter the following SCF command to stop all filesets on your system:

   ```
   STOP FILESET $ZPMON.*
   ```

   This command begins with the last fileset mounted and stops the filesets in the reverse order in which they were last started.

3. Stop the OSS Monitor process:

   If the OSS Monitor is running as a standard process, enter the `STOP` command at a TACL prompt:

   ```
   STOP $ZPMON
   ```

   If the OSS Monitor is running as a persistent process, enter the `ABORT` command at an SCF prompt:

   ```
   ABORT PROCESS $ZZKRN.#ZPMON
   ```

4. At the SCF prompt, enter the following set of commands for each disk volume in the fileset:

   ```
   STOP DISK diskname
   ALTER DISK diskname, OSSCACHING ON
   START DISK diskname
   ```

   `diskname` is the name of a disk volume that contains OSS files.

5. Restart the OSS Monitor as a normal or persistent process with the appropriate command.

6. Restart the OSS file system by entering the SCF command:

   ```
   START FILESET $ZPMON.filesetname
   ```

   where, `filesetname` is the name of each fileset that contains OSS files, beginning with the root and specified in the order in which mount points occur.

Known Issues

Scenario 1

MFC plans become obsolete when the base table is altered or dropped. The following sequence of operations illustrates the issue.

Operation	Expected Result	Actual Result	Remarks
`Create table testing(info int);`	Success	Success	Table testing is created.
`Stmt1 = Prepare("select * from testing")`	Success	Success	Stmt1 is prepared with MXCMP.
`Stmt1.execute()`	Success	Success	Stmt1 is executed.
`Stmt1.fetch()`	Success	Success	Data in the table testing is retrieved.
`Stmt1 = Prepare("select * from testing")`	Success	Success	Compiled plan is retrieved from MFC.

`Stmt1.execute()`	Success	Success	MFC statement works as expected.
`Stmt1.executeUpdate("drop table testing")`	Success	Success	The table testing is dropped.
`Stmt1.executeUpdate("create table testing (mycol varchar(10))")`	Success	Success	The table testing is created with varchar column.
`Stmt1 = Prepare("select * from testing")`	Success	Success	Compiled plan is retrieved from MFC, which is not correct because the table datatype is changed when the MFC plan is created.
`Stmt1.execute()`	Success	Failure	MXOSRVR turned the SQL/MX CQD recompilation_warnings ON. SQL/MX throws SQL exception upon similarity check failure and MXOSRVR drops the invalid module file from the `/usr/tandem/sqlmx/USERMODULES` location.
`Stmt1 = Prepare("select * from testing")`	Success	Success	A new plan is created in the MFC location.
`Stmt1.execute()`	Success	Success	MFC statement works as expected.

When performing the above operations, the execute() call fails when an invalid module file is found in the MFC. However, subsequent prepare() calls create a new module file. This open issue is similar to the driver side cache present in the JDBC/MX T2 driver.

Scenario 2

When scalar functions such as, Sum(), Avg(), ABS(), Count() appear in the selected columns, the list of the SQL query is not cached as MFC. For example,

```
SELECT sum(col1 + col2) from TAB WHERE col3 = ?
```

The scalar functions are supported in the INSERT, UPDATE, and DELETE queries and in the WHERE clause. For example:

```
SELECT col1 from TAB WHERE sum(col2 + col3) = ?
```


HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

JDBC/MX Compliance

The JDBC/MX Driver for NonStop SQL/MX conforms where applicable to the Sun Microsystems JDBC 3.0 API specification. However, the JDBC/MX driver differs from the JDBC standard in some ways because of limitations of NonStop SQL/MX and the JDBC/MX driver. This subsection describes the JDBC methods that are not supported, the methods and features that deviate from the specification, and features that are HP extensions to the JDBC standard. JDBC features that conform to the specification are not described in this subsection.

The topics are:

Unsupported Features

The following interfaces in the `java.sql` package are not implemented in the JDBC/MX driver because the data types involved are not supported by NonStop SQL/MX:

- `java.sql.Array`
- `java.sql.Ref`
- `java.sql.Savepoint`
- `java.sql.SQLData`
- `java.sql.SQLInput`
- `java.sql.SQLOutput`
- `java.sql.Struct`

Note: Support for `java.sql.Blob` and `java.sql.Clob` packages require the use of SQL/MX user tables as described in **Working with BLOB and CLOB Data**. These packages are not supported for access of SQL/MP user tables.

The following methods in the `java.sql` package throw an `SQLException` with the message "Unsupported feature - *method-name*":

Method	Comments

`CallableStatement.getArray(int parameterIndex)`
`CallableStatement.getArray(String parameterName)`
`CallableStatement.getBlob(int parameterIndex)`
`CallableStatement.getBlob(String parameterName)`
`CallableStatement.getClob(int parameterIndex)`
`CallableStatement.getClob(String parameterName)`
`CallableStatement.getObject(int parameterIndex, Map map)`
`CallableStatement.getObject(String parameterName, Map map)`
`CallableStatement.getRef(int parameterIndex)`
`CallableStatement.getRef(String`
	The particular `CallableStatement` method is not supported.

parameterName) CallableStatement.getURL(int parameterIndex) CallableStatement.getURL(String parameterName) CallableStatement.executeBatch()	
Connection.releaseSavepoint(Savepoint savepoint) Connection.rollback(Savepoint savepoint) Connection.setSavepoint() Connection.setSavepoint(String name)	The particular Connection methods are not supported.
PreparedStatement.setArray(int parameterIndex, Array x) PreparedStatement.setRef(int parameterIndex, Ref x) PreparedStatement.setURL(int parameterIndex, URL x)	The particular PreparedStatement method is not supported.
ResultSet.getArray(int columnIndex) ResultSet.getArray(String columnName) ResultSet.getObject(int columnIndex, Map map) ResultSet.getObject(String columnName, Map map) ResultSet.getRef(int columnIndex) ResultSet.getRef(String columnName) ResultSet.getURL(int columnIndex) ResultSet.getURL(String columnName) ResultSet.updateArray(int columnIndex) ResultSet.updateArray(String columnName) ResultSet.updateRef(int columnIndex) ResultSet.updateRef(String columnName)	The particular ResultSet methods are not supported.

The following methods are not supported when used for access of SQL/MP user tables:

Method	Comments
PreparedStatement.setBlob(int parameterIndex, Blob x) PreparedStatement.setClob(int parameterIndex, Clob x)	The particular PreparedStatement methods are not supported for access of SQL/MP user tables only.
ResultSet.getBlob(int columnIndex) ResultSet.getBlob(String columnName) ResultSet.getClob(int columnIndex) ResultSet.getClob(String columnName) ResultSet.updateBlob(int columnIndex) ResultSet.updateBlob(String columnName) ResultSet.updateClob(int columnIndex) ResultSet.updateClob(String columnName)	The particular ResultSet methods are not supported for access of SQL/MP user tables only.

The following methods in the java.sql package throw an SQLException with the message "Auto generated keys not supported":

Method	Comments

`Connection.prepareStatement(String sql, int autoGeneratedKeys)`
`Connection.prepareStatement(String sql, int[] columnIndexes)`
`Connection.prepareStatement(String sql, String[] columnNames)`
	Automatically generated keys are not supported.

`Statement.execute(String sql, int autoGeneratedKeys)`
`Statement.execute(String sql, int[] columnIndexes)`
`Statement.execute(String sql, String[] columnNames)`
`Statement.executeUpdate(String sql, int autoGeneratedKeys)`
`Statement.executeUpdate(String sql, int[] columnIndexes)`
`Statement.executeUpdate(String sql, String[] columnNames)`
`Statement.getGeneratedKeys()`
	Automatically generated keys are not supported.

The following methods in the `java.sql` package throw an `SQLException` with the message "Data type not supported:"

Method	Comments
`CallableStatement.getBytes(int parameterIndex)` `CallableStatement.getBytes(String parameterName)`	The particular data type is not supported.
`CallableStatement.setBytes(String parameterIndex, bytes[] x)`	Supports only BLOB, VARCHAR, BINARY, LONGVARCHAR, VARBINARY, and LONGVARBINARY; otherwise, the particular data type is not supported.
`PreparedStatement.setBytes(int ColumnIndex, bytes[] x)`	Supports only BLOB, CHAR, DATE, TIME, TIMESTAMP, VARCHAR, BINARY, LONGVARCHAR, VARBINARY, and LONGVARBINARY; otherwise, the particular data type is not supported.
`PreparedStatement.setObject(int parameterIndex, Object x int targetSqlType)` `PreparedStatement.setString(int parameterIndex, String x)`	Does not support the ARRAY, BINARY, BIT, DATALINK, JAVA_OBJECT, and REF types.
`ResultSet.getBytes(int ColumnIndex)` `ResultSet.getBytes(String ColumnName)`	Supports only BLOB, CHAR, VARCHAR, BINARY, LONGVARCHAR, VARBINARY, and LONGVARBINARY; otherwise, the particular data type is not supported.

The following optional interfaces in the `javax.sql` package are not implemented in the JDBC/MX driver:

Method	Comments

javax.sql.XAConnection javax.sql.XADataSource	Distributed Transactions, as described in the JDBC 3.0 API specification, are not yet implemented.
javax.sql.RowSet javax.sql.RowSetInternal javax.sql.RowSetListener javax.sql.RowSetMetaData javax.sql.RowSetReader javax.sql.RowSetWriter	RowSet is not implemented in the JDBC/MX driver. You can, however, download reference implementation of RowSet from Sun Microsystems (http://developer.java.sun.com/developer/earlyAccess/crs/).
javax.sql.JdbcRowSet.getArray(int columnIndex) javax.sql.JdbcRowSet.getArray(String columnName) javax.sql.JdbcRowSet.getObject(int columnIndex, Map map) javax.sql.JdbcRowSet.getObject(String columnName, Map map) javax.sql.JdbcRowSet.getRef(int columnIndex) javax.sql.JdbcRowSet.getRef(String columnName) javax.sql.JdbcRowSet.getURL(int columnIndex) javax.sql.JdbcRowSet.getURL(String columnName) javax.sql.JdbcRowSet.rollback(Savepoint savepoint) javax.sql.JdbcRowSet.setArray(int parameterIndex, Array x) javax.sql.JdbcRowSet.setRef(int parameterIndex, Ref x) javax.sql.JdbcRowSet.updateArray(int columnIndex) javax.sql.JdbcRowSet.updateArray(String columnName) javax.sql.JdbcRowSet.updateRef(int columnIndex) javax.sql.JdbcRowSet.updateRef(int columnIndex)	The JdbcRowSet API methods are supported except for these that throw an Unsupported feature - method-name SQLMXException.

For additional information about deviations for some methods, see Deviations.

Deviations

The following table lists methods that differ in execution from the JDBC specification. When an argument in a method is ignored, the JDBC/MX driver does not throw an SQLException, thus allowing the application to continue processing. The application might not obtain the expected results, however. Other methods listed do not necessarily throw an SQLException, unless otherwise stated, although they differ from the specification.

Note: java.sql.DatabaseMetaData.getVersionColumns() method mimics the java.sql.DatabaseMetaData.getBestRowIdentifier() method because SQL/MX does not support SQL_ROWVER (a columns function that returns the column or columns in the specified table, if any, that are automatically updated by the data source when any value in the row is updated by any transaction).

Method	Comments

`java.sql.DatabaseMetaData.getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)`	The column is added to the column data, but its value is set to NULL because SQL/MX does not support the column type for types as follows: SCOPE_CATALOG, SCOPE_SCHEMA, SCOPE_TABLE, and SOURCE_DATA_TYPE.
`java.sql.DatabaseMetaData.getSchemas()`	TABLE_CATALOG is added to the column data and returns the catalog name.
`java.sql.DatabaseMetaData.getTables(String catalog, String schemaPattern, String[] types)`	The column is added to the column data, but its value is set to NULL because SQL/MX does not support the column type for types as follows: TYPE_CAT, TYPE_SCHEMA, TYPE_NAME, SELF_REFERENCING_COL_NAME, and REF_GENERATION.
`java.sql.DatabaseMetaData.getUDTs(String catalog, String schemaPattern, String tableNamePattern, int[] types)`	BASE_TYPE is added to the column data, but its value is set to NULL because SQL/MX does not support the base type.
`java.sql.DatabaseMetaData.getVersionColumns()`	Mimics the `DatabaseMetaData.getBestRowIdentifier()` method because SQL/MX does not support SQL_ROWVER (a columns function that returns the column or columns in the specified table, if any, that are automatically updated by the data source when any value in the row is updated by any transaction).

`java.sql.DriverManager.getConnection(String url, String usr, String password)`
`java.sql.DriverManager.getConnection(String url, Properties info)`
`javax.sql.DataSource.getConnection(String username, String password)`
	User name and password arguments are ignored. All connections have the same security privileges as the user who invoked the Java VM.

`java.sql.DriverManager.setLoginTImeout(...)` `javax.sql.DataSource.setLoginTimeout(...)`	Login time-out is ignored.
`javax.sql.DataSource.setLogWriter`	This method has no effect unless the JDBC trace facility is enabled; for information on the JDBC trace facility, see the NonStop Server for Java Programmer's Reference.
`java.sql.Connection.createStatement(...)` `java.sql.Connection.prepareStatement(...)`	The JDBC/MX driver does not support the scroll-sensitive result set type, so an `SQLWarning` is issued if an application requests that type. The result set is changed to a scroll-insensitive type.
`java.sql.Connection.setReadOnly(...)`	The read-only attribute is ignored.
`java.sql.ResultSetMetaData.getPrecision(int column)` `java.sql.ResultSetMetaData.getColumnDisplaySize(int column)`	For `CLOB` and `BLOB` columns, these methods return 0 to denote an unlimited value. According to the standard API, the `getPrecision`() method and `getColumnDisplatSize`() method return an integer value, but LOB data larger than the maximum integer value can be stored in the database.
`java.sql.ResultSet.setFetchDirection(...)`	The fetch direction attribute is ignored.

`java.sql.Statement.setEscapeProcessing(...)`	Because SQL/MX parses the escape syntax, disabling escape processing has no effect.
`java.sql.Statement.setFetchDirection(...)`	The fetch direction attribute is ignored.
`java.sql.Statement.setQueryTimeout(...)`	The query time-out value is ignored. The JDBC/MX driver does not abort execution when the query time-out period has expired.

`javax.sql.JdbcRowSet.setUsername(String username)`
`javax.sql.JdbcRowSet.setPassword(String password)`
`javax.sql.JdbcRowSet(String url, String username, String password)`
	User name and password arguments are ignored. Security privileges are the same as for the user who invoked the Java VM.

`javax.sql.JdbcRowSet.setReadOnly(...)`	The read-only attribute is ignored.
`javax.sql.JdbcRowSet.setEscapeProcessing(...)`	Disabling escape processing has no effect because SQL/MX parses the escape syntax.
`javax.sql.JdbcRowSet.setFetchDirection(...)`	The fetch direction attribute is ignored.
`javax.sql.JdbcRowSet.setQueryTimeout(...)`	The query time-out value is ignored. The JDBC/MX driver does not abort execution when the query time-out period has expired.

The following features are implemented in the JDBC/MX driver but might differ in implementation from other drivers:

Updatable Result Set

The JDBC/MX driver supports both read-only and updatable concurrency modes. The JDBC/MX driver expects the following criteria for a result set to be updatable:

- The table name of the first column in the result set is assumed to be the table to be updated. This assumption allows queries from multiple tables also to be updatable.
- The query selects the primary key columns of the table to be updated.

The JDBC/MX driver throws an `SQLException` when any of the following conditions occur:

- The primary key columns are updated.
- Any selected column has been updated since the most recent time it was read. (However, the JDBC/MX driver does not ensure that columns in the table that are not part of the select query remain unchanged since the row was most recently read.)
- A query does not select nonnullable columns or columns that do not have a default value.

The result set is also affected in the following ways:

- A deleted row is removed from the result set. The method `databaseMetaData.deletesAreDetected()` returns `false`.
- An inserted row is added to the result set at the current cursor position. The method `databaseMetaData.insertsAreDetected()` returns `true`.

Batch Updates

The batch update facility allows a `Statement` object to submit a set of heterogeneous update, insert, or delete commands together as a single unit to the database. This facility also allows multiple sets of parameters to be associated with a `PreparedStatement` object.

When the autocommit mode is enabled, the JDBC/MX driver commits the updates only when all commands in the batch succeed. If any command in the batch fails, the updates are rolled back in both autocommit and nonautocommit mode.

With the BatchUpdate Exception handling improvements support, JDBC driver now continues processing the remaining jobs in the batch even after BatchUpdateExceptions. If there is any Batch exceptions encountered during the execution, the exception is queued up and the

remaining batch commands are executed. At the execution completion of all elements in the batch the queued exceptions are thrown. The user application must handle, commit, or rollback of batch transaction on an exception. By this, re-execution of entire jobs is avoided. However, for any TMF errors, that results in transaction failure, cannot be addressed by this enhancement.

HP Extensions

The following HP extensions to the JDBC standard are implemented in the JDBC/MX driver.

Interval Data Type

The `interval` data type is not a generic SQL type defined in the Java 2 JDBC 3.0 Specification, but SQL/MX supports the `interval` data type. To allow JDBC applications for SQL/MX to access the `interval` data type, the JDBC/MX driver maps it to the `Types.OTHER` data type. The JDBC/MX driver enables the `getObject()` and `getString()` methods of the `ResultSet` interface, and the `setObject()` and `setString()` methods of the `PreparedStatement` interface, to access this data type. The `interval` data type is always accessed as a `String` object. The JDBC/MX driver also allows escape syntax for `interval` literals.

Internationalization

The JDBC/MX driver is designed so that Java messages can be adopted for various languages. The error messages in JDBC/MX components are stored outside the source code in a separate property file and retrieved dynamically based on the locale setting. The error messages in different languages are stored in separate property files based on the language and country. This extension does not apply to all messages that can occur when running JDBC applications.

SQL Conformance

JDBC/MX conforms to the SQL language entry level of SQL:1999. This subsection describes the JDBC/MX support for:

- SQL Scalar Functions
- CONVERT Function
- JDBC Data Types
- SQL Escape Clauses

SQL Scalar Functions

JDBC/MX maps JDBC scalar functions to their equivalent SQL/MX functions, as shown in the following tables:

Numeric Functions

JDBC Function	SQL/MX Equivalent Function
ABS	ABS
ACOS	ACOS
ASIN	ASIN
ATAN	ATAN
ATAN2	ATAN2
CEILING	CEILING
COS	COS
DEGREES	DEGREES
EXP	EXP

FLOOR	FLOOR
LOG	LOG
LOG10	LOG10
MOD	MOD
PI	PI
POWER	POWER
RADIANS	RADIANS
SIGN	SIGN
SIN	SIN
SINH	SINH
SQRT	SQRT
TAN	TAN

String Functions

JDBC Function	SQL/MX Equivalent Function
ASCII	ASCII
CHAR	CHAR
CHAR_LENGTH	CHAR_LENGTH
CONCAT	CONCAT
INSERT	INSERT
LCASE	LOWER
LEFT	SUBSTRING
LENGTH	LENGTH
LOCATE	LOCATE (JDBC `LOCATE start` parameter is not supported)
LOWER	LOWER
LPAD	LPAD
LTRIM	LTRIM
OCTET_LENGTH	OCTET_LENGTH
POSITION	POSITION
REPEAT	REPEAT
REPLACE	REPLACE
RIGHT	RIGHT
RTRIM	TRIM...TRAILING
SPACE	SPACE
SUBSTRING	SUBSTRING

UCASE	UPPER | UPSHIFT

Note: JDBC string functions in queries can return unexpected results for fixed-length (CHAR) column names because SQL/MX pads a fixed-length string with blanks up to the length of the definition, so the results from some JDBC string functions can include trailing blanks at the end of the string. Use the RTRIM function in queries to cause SQL/MX to trim extra blanks from the column names.

Time and Date Functions

JDBC Function	SQL/MX Equivalent Function
CONVERTTIMESTAMP	CONVERTTMESTAMP
CURRENT	CURRENT
CURRENT_TIMESTAMP	CURRENT_TIMESTAMP
CURDATE, CURRENT_DATE	CURRENT_DATE
CURTIME, CURRENT_TIME	CURRENT_TIME
DATEFORMAT	DATEFORMAT
DAY	DAY
DAYNAME	DAYNAME
DAYOFMONTH	DAYOFMONTH
DAYOFWEEK	DAYOFWEEK
DAYOFYEAR	DAYOFYEAR
EXTRACT	EXTRACT
HOUR	HOUR
JULIANTIMESTAMP	JULIANTIMESTAMP
MINUTE	MINUTE
MONTH	MONTH
MONTHNAME	MONTHNAME
QUARTER	QUARTER
SECOND	SECOND
WEEK	WEEK
YEAR	YEAR

System Functions

JDBC Function	SQL/MX Equivalent Function
CURRENT_USER	CURRENT_USER
SYSTEM_USER	SYSTEM_USER
USER	USER

CONVERT Function

JDBC/MX uses the SQL/MX CAST function to support the JDBC CONVERT function. The JDBC CONVERT function has the following format:

```
{ fn CONVERT( value_exp, data_type ) }
```

The SQL/MX CAST has this format:

```
CAST( { value_exp | NULL } AS data_type )
```

SQL/MX translates the CONVERT syntax to the CAST syntax, converting the data type argument to its equivalent SQL/MX value. For example, if the JDBC data type parameter for character data is an integer value (SQL_CHAR or 1), the equivalent SQL/MX data type is a string literal with a value of CHARACTER.

JDBC Data Types

The following table shows the JDBC data types that are supported by JDBC/MX and their corresponding SQL/MX data types:

JDBC Data Type	Supported by JDBC/MX	SQL/MX Data Type
Types.Array	No	
Types.BIGINT	Yes	LARGEINT
Types.BINARY	No	
Types.BIT	No	
Types.BLOB	Yes	
Types.CHAR	Yes	CHAR(n)
Types.CLOB	Yes	
Types.DATE	Yes	DATE
Types.DECIMAL	Yes	DECIMAL(p,s)
Types.DISTINCT	No	
Types.DOUBLE (*)	Yes	DOUBLE PRECISION
Types.FLOAT (*)	Yes	FLOAT(p)
Types.INTEGER	Yes	INTEGER
Types.JAVA_OBJECT	No	
Types.LONGVARBINARY	No	
Types.LONGVARCHAR	Yes**	VARCHAR[(n)]
Types.NULL	No	
Types.NUMERIC	Yes	NUMERIC(p,s)
Types.REAL	Yes	FLOAT(p)
Types.REF	No	
Types.SMALLINT	Yes	SMALLINT

Types.STRUCT	No	
Types.TIME	Yes	TIME
Types.TIMESTAMP	Yes	TIMESTAMP
Types.TINYINT	No	
Types.VARBINARY	No	
Types.VARCHAR	Yes	VARCHAR(n)

* See Floating Point Support.
** For details about maximum length, see the *SQL/MX Reference Manual*.

The JDBC/MX driver maps the following SQL/MX data types to the JDBC data type `Types.OTHER`:

```
DATETIME YEAR
DATETIME YEAR TO MONTH
DATETIME YEAR TO DAY
DATETIME YEAR TO HOUR
DATETIME YEAR TO MINUTE
DATETIME MONTH
DATETIME MONTH TO DAY
DATETIME MONTH TO HOUR
DATETIME MONTH TO SECOND
DATETIME DAY
DATETIME DAY TO HOUR
DATETIME DAY TO MINUTE
DATETIME DAY TO SECOND
DATETIME HOUR
DATETIME HOUR TO MINUTE
DATETIME MINUTE
DATETIME MINUTE TO SECOND
DATETIME SECOND
DATETIME FRACTION


INTERVAL YEAR(p)
INTERVAL YEAR(p) TO MONTH
INTERVAL MONTH(p)
INTERVAL DAY(p)
INTERVAL DAY(p) TO HOUR
INTERVAL DAY(p) TO MINUTE
INTERVAL DAY(p) TO SECOND
INTERVAL HOUR(p)
INTERVAL HOUR(p) TO MINUTE
INTERVAL HOUR(p) TO SECOND
INTERVAL MINUTE(p)
INTERVAL MINUTE(p) TO SECOND
INTERVAL SECOND(p)
```

Floating-Point Support

The JDBC/MX driver and the NonStop Server for Java pass any FLOAT (32-bit) number or DOUBLE (64-bit) number in the IEEE 754 floating-point format.

Floating-point values are stored in SQL/MX tables as IEEE 754 values.

Floating-point values are stored in SQL/MP tables in Tandem format (called TNS format in OSS terminology. For floating-point values stored in SQL/MP tables in the Tandem format, SQL/MX performs the conversion from the IEEE 754 format to the Tandem format when

storing the values and from the Tandem format to the IEEE 754 format when retrieving and passing the values.

Since SQL/MX tables store IEEE 754 floating-point values, JDBC applications accessing floating-point data do not receive floating-point exceptions. The JDBC applications should check for plus (+) or minus (-) infinity conditions to determine if an overflow or underflow has occurred. Applications can also encounter a not-a-number value being passed back, for example, for numbers divided by zero. This processing is done according to the IEEE 754 standard.

SQL/MP tables can generate floating-point exceptions.

For the range of floating-point values and double-precision values for IEEE 754 format and TNS format, *see the NonStop Server for Java Programmer's Reference*. For information on floating-point formats in SQL/MX, see "Data Types" in the *SQL/MX Reference Manual*.

SQL Escape Clauses

JDBC/MX accepts SQL escape clauses and translates them into equivalent SQL/MX clauses, as shown in the following table:

SQL Escape Clause	SQL/MX Equivalent Clause
`{ d 'date-literal' }`	`DATE 'date-literal'`
`{ t 'time-literal' }`	`TIME 'time-literal'`
`{ ts 'timestamp-literal' }`	`TIMESTAMP 'timestamp-literal'`
`{ oj join-expression }`	`join-expression *`
`{ INTERVAL sign interval-string interval-qualifier }`	`INTERVAL sign interval-string interval-qualifier`
`{ fn scalar-function }`	`scalar-function`
`{ escape 'escape-character' }`	`escape 'escape-character'`
`{ call procedure-name... }`	`CALL procedure-name...`
`{ ?=call procedure-name... }`	Not supported in the current release

* JDBC syntax does not include nested joins, while SQL/MX does. JDBC/MX extends the SQL escape syntax for an outer join.

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

JDBC Trace Facility

The JDBC trace facility traces the entry point of all JDBC methods called from the Java applications. To make this facility generic, it is implemented as a JDBC driver wrapper.

The JDBC trace facility can be enabled in any of the following ways in which a JDBC connection to a database can be obtained:

Tracing Using the DriverManager Class

Java applications can use the `DriverManager` class to obtain the JDBC connection and enable the JDBC trace facility by loading the JDBC trace driver. `com.tandem.jdbc.TDriver` is the trace driver class that implements the Driver interface. The application can load the JDBC trace driver in one of the following ways:

- Specify the JDBC trace driver class in the `-Djdbc.drivers` option in the command line.
- Use the `Class.ForName` method within the application.
- Add the JDBC trace class to the `jdbc.drivers` property within the application.

The JDBC URL passed in the `getConnection` method of the driver class determines which JDBC driver obtains the connection. Use the following URL and JDBC driver to obtain the JDBC connection:

```
jdbc:sqlmx:
```

Java applications should turn on tracing using the `DriverManager.setLogWriter` method, for example by using the following JDBC API call in your application:

```
DriverManager.setLogWriter(new PrintWriter(new FileWriter("FileName")));
```

Tracing Using the DataSource Implementation

This is preferred way to establish a JDBC connection and to enable the JDBC trace facility. In this way, a logical name is mapped to a trace data source object by means of a naming service that uses the Java Naming and Directory Interface (JNDI).

The following table describes the set of properties that are required for a trace data source object:

Property Name	Type	Description

`dataSourceName`	`String`	The data source name
`description`	`String`	Description of this data source
`traceDataSource`	`String`	The name of the `DataSource` object to be traced

The `traceDataSource` object is used to obtain the JDBC connection to the database. Java applications should turn on tracing using the `setLogWriter` method of the `DataSource` interface.

Tracing Using the java Command

Enable tracing by specifying the tracing system property by using the following arguments when starting your Java program:

```
java -Djdbcmx.traceFile=logFile -Djdbcmx.traceFlag=n
```

The `logFile` is the file name that is to contain the tracing information. The `n` value for the `traceFlag` can be the following values:

Value for n	Description
0	No tracing.
1	Traces connection and statement pooling calls only.
2	Traces the LOB-code path only.
3	Traces the entry point of all JDBC methods.

Note: Only one `traceFlag` value can be in effect at a time.

Tracing Using the system.setProperty Method

Enable tracing by using the `System.setProperty(key, value)` to set the same value as described above. For example:

```
System.setProperty("traceFile", "myLogFile.log");
System.setProperty("traceFlag", "2");
```

Set the system property before the program makes any JDBC API calls.

Tracing by Loading the Trace Driver Within the Program

Enable tracing by loading the JDBC trace driver within the program by using the `Class.forName("com.tandem.jdbc.TDriver")` method. This method also requires that you set the `DriverManager.setLogWriter` method.

Tracing Using a Wrapper Data Source

Enable tracing by creating a wrapper data source around the data source to be traced. The wrapper data source contains the `TraceDataSource` property that you can set to the data source to be traced. For information about demonstration programs that show using this method, see JDBC Trace Facility Demonstration Programs.

Enabling Tracing for Application Servers

Typically, tracing output is written to the `PrintWriter` object that the application sets by using either the `DataSource.setLogWriter()` method or `DriverManager.setLogWriter()` method. User-written Java applications can use these methods with the JDBC Trace Facility.

Application servers, however, might not enable the JDBC tracing with the `setLogWriter()` method. Instead application servers can enable tracing and set the tracing level by using the following JDBC/MX properties:

- `jdbcmx.traceFile`
- `jdbcmx.traceFlag`

jdbcmx.traceFile Property

To enable tracing for application servers, use the `jdbcmx.traceFile` property specified in the command line:

```
-Djdbcmx.traceFile=trace_file_name
```

where `jdbcmx.trace_file_name` is an OSS filename. If the file exists, the tracing output is appended to the existing file.

The `PrintWriter` object that is set using `setLogWriter()` method has higher precedence over the `jdbcmx.traceFile` system property setting. This property can be specified in the command line or programmatically before the first connection.

jdbcmx.traceFlag Property

To set the tracing level for application servers that use the `jdbcmx.traceFile` property, use the `traceFlag` property specified in the command line:

```
-Djdbcmx.traceFlag=n
```

where `n` is an integer that specifies the tracing level. The value can be 0, 1, or 2. The default level is 0. Any value greater than 2 is treated like 2. The tracing levels are:

Level	Meaning
0	No tracing.
1	Traces connection and statement pooling information.
2	Traces the LOB-code path only.
3	Traces the entry point of all JDBC methods.

Note: Only one `traceFlag` value can be in effect at a time.

Trace-File Output Format

A trace entry appears at the start of the trace file that shows the vproc of the JDBC/MX driver being traced. This entry appears only when the `traceFlag` value is 1, 2, or 3. For example,

```
jdbcTrace:[08/02/05 04:02:49]:TRACING JDBC/MX VERSION: T1275H50_23DEC2005_JDBCMX_10220
```

The format of the trace output has two types where the second type is used only where the JDBC/MX driver has an object to map to. The formats are:

Format 1

```
jdbcTrace:[timestamp] [thread-id]:[object-id] :className.method(param...)
```

Format 2

```
jdbcTrace:[timestamp] [thread-id]:[object-id] :className.method(param...)
returns [return-object] [return-object-id]
```

where

`timestamp`
: is the day and time representation in the form: `mm/dd/yy hr:min:sec`
where `mm` is month; `dd`, day; `yy`, year; `hr`, hour; `min`, minute; `sec`, seconds.

`thread-id`
: is the String representation of the current thread

`object-id`
: is the `hashcode` of the JDBC object

`classname`
: is the JDBC implementation class name.

`return-object`
: is the object returned by the traced method. The `return-object` can be one of the following interface types: `CallableStatement`, `Connection`, `PooledConnection`, `ResultSet`, `Statement`, `DatabaseMetaData`, `ParameterMetaData`, or `ResultSetMetaData`.

`return-object-id`
: is the `hashcode` of the object returned by the traced method.

Trace output is sent to the `PrintWriter` specified in the `setLogWriter` method.

Example 1

```
jdbcTrace:[10/12/05 10:04:39]
[Thread[main,5,main]]:[5256233]:com.tandem.sqlmx.SQLMXPreparedStatement.executeQuery()
```

Example 2

Some traced methods will have two trace statements, one for the method entry point and the other for return object mapping. Some code paths might log additional tracing statements between method entry and the return. For example, between SQLMXConnection.prepareStatement () trace entries, you might see:

```
jdbcTrace:[10/12/05 10:04:39]
[Thread[main,5,main]]:[10776760]:SQLMXConnection.prepareStatement("select c1, c2 from
tconpool where c1 = ?")
```

<additional trace entries>

```
<jdbcTrace:[10/12/05 10:04:39]
[Thread[main,5,main]]:[10776760]:SQLMXConnection.prepareStatement("select c1, c2 from
tconpool where c1 = ?") returns PreparedStatement [23276589]
```

Logging SQL Statement IDs and Corresponding JDBC SQL Statements

The JDBC/MX driver can write a supplemental log file that shows the SQL statement ID (STMID) of executed SQL statements mapped with the corresponding JDBC SQL statements.

The `idMapFile` contains a list of all the SQL statements issued by the application, and correlates them to the internal driver STMTID (a `hashcode`). The trace-file output (see Trace-File Output) lists the STMID (the `object-id` in the trace output), which can be used to reference the SQL statements in the idMapFile trace file.

The statement-ID is logged in the `idMapFile` to avoid replacing the `object-id` in the trace-file output with the verbose and potentially large SQL statement for every entry.

Mapping statement-IDs to SQL statements applies to any interface that prepares or executes a statement, for example, `PreparedStatement`, `Connection`, `ResultSet`, `JdbcRowSet`, and `Statement`.

- Specifying Statement-ID Logging
- Properties for Statement-ID Logging
- Statement-ID Log Output

Specifying Statement-ID Logging

To specify supplemental logging:

1. Set the `enableLog` property to `on` to enable logging.
2. Set the `idMapFile` property to specify the log file. By default, the log is written to the screen.

For additional information about these properties, see enableLog Property and idMapFile Property.

You can specify these properties either in the command line or in the program similar to setting tracing described earlier under Tracing Using the java Command and Tracing Using the system.setProperty Method.

> Specify Logging in the Command Line

```
java  Djdbcmx.idMapFile=logFile  Djdbcmx.enableLog=on
```

Specify Logging in a Program

```
System.setProperty("enableLog", "on");
System.setProperty("("idMapFile", "myMapFile.log");
```

Properties for Statement-ID Logging

enableLog Property

Enables logging of SQL statement IDs and the corresponding JDBC SQL statements. The format for `enableLog` property is:

`-Djdbcmx.enableLog=boolean`

Data type: `boolean`
Default: `off`

Valid values are either `on` or `off`. You can specify this property only in the `java` command line.

The following specification in the `java` command line enables the logging:

```
-Djdbcmx.enableLog=on
```

For more information, see Logging SQL Statement IDs and Corresponding JDBC SQL Statements.

idMapFile Property

Specifies the file to which the JDBC trace facility logs SQL statement IDs and the corresponding JDBC SQL statements. The format for the `idMapFile` property is:

`-Djdbcmx.idMapFile=filename`

Data type: `string`
Default: logs to the screen

Specify a valid OSS file name. You can specify this property only in the `java` command line.

The following entry in the `java` command line specifies logging to file `/sales/app5/STMID-Log`.

```
-Djdbcmx.idMapFile=/sales/app5/STMID-log
```

To enable logging, use the `enableLog` property. For more information, see Logging SQL Statement IDs and Corresponding JDBC SQL Statements.

Statement-ID Log Output

The format of a statement-ID log output entry is:

```
[timestamp] STMTobject-id (sql-statement)
```

where

`timestamp`
is the day and time representation in the form: `mm/dd/yy hr:min:sec`
where `mm` is month; `dd`, day; `yy`, year; `hr`, hour; `min`, minute; `sec`, seconds.

`object-id`
is the `hashcode` of the JDBC object.

`sql-statement`
is the actual SQL statement mapped to the statement ID.

Example

```
[08/05/05 10:32:38] STMT16399041 ("insert into TST_TBL (c1) values = ?")
```

JDBC Trace Facility Demonstration Program

The JDBC/MX driver provides jdbcTrace demonstration programs in the installation directory. The programs are described in the `README_JDBCTrace` file. For the location, see JDBC/MX Driver File Locations. These programs demonstrate tracing by creating a wrapper around the driver-specific data source to be traced. For additional information, see Sample Programs Summary.

Migration

This section describes the considerations and application changes required to migrate applications from the JDBC/MX V30, V31, V32, H10 drivers to the JDBC/MX H50 driver. These topics are:

Summary of Migration Changes for JDBC/MX Driver Versions

Migration Topic	Migrating from V30	Migrating from V31	Migrating from V32 (TNS/R) or H10 (TNS/E)
Transactions	Applies	N/A	N/A
nametype Property	Applies	N/A	N/A
Deprecated Property-Name Specification	Applies	N/A	N/A
Deprecated Methods According to the J2SE 5.0 API	Applies	Applies	Applies
Row Count Array of the PreparedStatement.executeBatch Method	Applies	N/A	N/A
Using Character Encoding Sets and SQL Databases	Applies	N/A	N/A
Connection sharing across multiple threads	Applies	Applies	Applies
Location Change for Installed Files	Applies	Applies	Applies
Version of NonStop Server for Java	Applies	Applies	Applies
Release of NonStop SQL/MX	Applies	N/A	N/A

If you are migrating from JDBC/MX V30, V31, or V32 you might want to see the new and changed information in the JDBC Driver for SQL/MX Programmer's Reference for the later products.

This section also includes the topics

- Migrating to TNS/E Systems
- Migrating from JDBC/MP Applications

Transactions

Transaction semantics changed in the V31 product from the previous versions of the JDBC/MX driver when the connection is set to autocommit mode.

In previous releases, when multiple select statements were involved in a transaction in autocommit mode, the JDBC/MX driver ended the transaction when any select statement result set was closed. In this release, the JDBC/MX driver ends the transaction only when the result set of the select statement that started the transaction is closed.

If your application depends on the previous transaction semantics, you need to re-code the application.

---

nametype Property

Use of the `nameType` property was removed in the JDBC/MX V31 driver. This property allowed you to specify the use of either ANSI or SHORTANSI names. SHORTANSI names are no longer allowed. The names are ANSI names. Remove use of this feature from your applications.

Deprecated Property-Name Specification

With the JDBC/MX V31 and V32 drivers, property names used on the command line in the `java -D` option should now include the prefix:

```
jdbcmx.
```

This notation, which includes the period (`.`), ensures that all the JDBC/MX driver property names are unique for a Java application. For example: `maxStatements` becomes

```
jdbcmx.maxStatements
```

For application migration purposes, the JDBC/MX V31 and V32 drivers allow the deprecated property-name specification on the command line.

The property names passed to JDBC/MX V31 and V32 driver methods in a Properties object do not require the prefix.

Summary of Deprecated Property-Name Specifications for Use in the Command Line

Deprecated Property-Name Specification	New Property-Name Specification
catalog	jdbcmx.catalog
schema	jdbcmx.schema
mploc	jdbcmx.mploc
maxPoolSize	jdbcmx.maxPoolSize
minPoolSize	jdbcmx.minPoolSize
maxStatements	jdbcmx.maxStatements
traceFile	jdbcmx.traceFile
traceFlag	jdbcmx.traceFlag
sqlmx_nowait	jdbcmx.sqlmx_nowait

Note: Support for the deprecated property-name specification will end in a future JDBC/MX driver release. HP recommends that you migrate your JDBC applications to use the new property-name specification.

Deprecated Methods According to the J2SE 5.0 API

The following methods are marked as deprecated according to the J2SE 5.0 API, but functionality remains unchanged to minimize impact on existing user applications.

```
SQLMXCallableStatement.getBigDecimal()
SQLMXCallableStatement.setUnicodeStream()
SQLMXConnectionPoolDataSource.setNameType()
SQLMXConnectionPoolDataSource.getNameType()
SQLMXDataSource.setNameType()
SQLMXDataSource.getNameType()
SQLMXJdbcRowSet.getBigDecimal()
SQLMXJdbcRowSet.getUnicodeStream()
SQLMXPreparedStatement.setUnicodeStream()
```

```
SQLMXResultSet.getBigDecimal()
SQLMXResultSet.getUnicodeStream()
TCallableStatement.getBigDecimal()
TPreparedStatement.setUnicodeStream()
TResultSet.getBigDecimal()
TResultSet.getUnicodeStream()
```

Row Count Array of the PreparedStatement.executeBatch Method

With the release of the JDBC/MX V31 and V32 drivers, you can improve the performance of batch processing when using the `PreparedStatement.executeBatch()` method by setting the `batchBinding` property.

If you do not set the `batchBinding` property, your JDBC applications operate without batch array binding (the default setting).

If you update your application to use the `batchBinding` property, you must consider the change in information returned on the `PreparedStatement.executeBatch()` method.

For detailed information, see Setting Batch Processing for Prepared Statements.

Using Character Encoding Sets and SQL Databases

If your application uses Java character encoding sets and accesses SQL databases, consider the change in the JDBC/MX V31 and V32 drivers' support of multibyte character sets and how the change might affect your application.

The JDBC/MX driver now supports the reading and writing of CHAR, VARCHAR, VARCHAR_LONG, and VARCHAR_WITH_LENGTH data types that utilize a double-byte character set. The double-byte character sets supported by JDBC/MX are ISO88591, UCS2, KANJI, and KSC5601.

Previously, String type column data was always encoded using the default character set encoding, which was typically ISO88591, but KANJI and KSC5061 were also supported.

Now the JDBC/MX driver encodes and decodes String data types as a function of the associated character set name for the particular SQL table column independent of the default encoding. For the

currently supported character sets, see Multibyte Character Set (MBCS) Support.

Connection sharing across multiple threads

- Applications that do not share connections across multiple threads can be used with the new JDBC version (H50AAD) without any changes.
- The existing multi-threaded application on other platforms with connection objects shared across multiple threads can be directly ported to work with enhanced JDBC/MX driver.
- For a new application, to utilize the connection sharing enhancement the application has to be redesigned to share the connection across multiple threads.

Fallback provisions

If the application is modified for sharing connections across multiple threads, consider either of the following steps:

- Explicitly synchronize the connection object usage in the application.
- Revert back the changes done for sharing the connection.

Location Change for Installed Files

With the JDBC/MX V30 driver, the driver software was installed to the default location of the `/usr/tandem/java_public_lib` directory, which was the public library directory for NonStop Server for Java 4.

Now for the V31 and subsequent PVUs, the JDBC/MX driver must be installed in its own space. For the current installation location for the JDBC/MX driver, see JDBC/MX Driver File Locations.

Version of NonStop Server for Java

JDBC/MX requires these versions of NonStop Server for Java:

- JDBC/MX V30 requires NonStop Server for Java 3.1.1 or subsequent 3.x release (product number T0083).
- JDBC/MX V31 requires version 1 of NonStop Server for Java 4 (product number T2766), which is based on J2SE SDK 1.4.1.
- JDBC/MX V32 (TNS/R system) and H10 (TNS/E system) require NonStop Server for Java 4

(product number T2766), which is based on J2SE SDK 1.4.2.

- JDBC/MX H50 requires NonStop Server for Java 5 (product number T2766), which is based on J2SE 5.0.
- JDBC/MX H50 AAD requires NonStop Server for Java 5 (product number T2766H51 or T2766H50), which is based on J2SE 5.0.

For Java migration issues, see the NonStop Server for Java Programmer's Reference.

Release of NonStop SQL/MX

JDBC/MX requires these versions of NonStop SQL/MX:

- JDBC/MX V30 requires NonStop SQL/MX 1.8.5.
- JDBC/MX V31 requires NonStop SQL/MX 2.0.
- JDBC/MX V32 requires NonStop SQL/MX 2.0 or all subsequent 2.x versions until otherwise indicated in a replacement publication.

For SQL/MX migration issues, see the SQL/MX Installation and Management Guide.

Migrating to TNS/E Systems

For information about migrating Java applications from TNS/R systems to TNS/E systems, see the NonStop Server for Java Programmer's Reference.

Migrating from JDBC/MP Applications

For extensive information on migrating applications from NonStop SQL/MP to NonStop SQL/MX, see the SQL/MX Database and Application Migration Guide.

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

Messages

JDBC/MX returns sqlcode and file-system error codes as error codes for the `getErrorCode()` method of SQLException.

Messages from the Java Portion of the JDBC Driver (range 29000 through 29079)		Messages from the Native-interface Portion of the JDBC Driver (range 29250 through 29499)
29001-29009	29050-29059	29251-29259
29010-29019	29060-29069	29260-29267
29020-29029	29070-29079	
29030-29039	29080-29089	
29040-29049		

Messages are listed in numerical SQLCODE order. Descriptions include the following:

```
SQLCODE  SQLSTATE  message-text

   Cause    [ What occurred to trigger the message.]
   Effect   [ What is the result when this occurs. ]
   Recovery [ How to diagnose and fix the problem. ]
```

For information about error codes outside these ranges, see the *SQL/MX Messages Manual*.

Messages From the Java Side of the JDBC/MX Driver

29001 HYC00 Unsupported feature - {0}

Cause: The feature listed is not supported by the JDBC driver.
Effect: An unsupported exception is throw, and null `resultSet` is returned.
Recovery: Remove the feature functionality from the program.

[back to the top]

29002 08003 Connection does not exist

Cause: An action was attempted when the connection to the database was closed.
Effect: The database is inaccessible.
Recovery: Retry the action after the connection to the database is established.

[back to the top]

29003 HY000 Statement does not exist

Cause: A validation attempt was made on the getter or exec invocation on a closed statement.
Effect: The getter or exec invocation validation fails.
Recovery: Issue `validateGetInvocation()` or `validateExecDirectInvocation` when the statement is open.

[back to the top]

29004 HY024 Invalid transaction isolation value

Cause: An attempt was made to set the transaction isolation level to an invalid value.
Effect: `SQLMXConnection.setTransactionIsolation` does not set the transaction isolation value.
Recovery: Valid isolation values are: `SQL_TXN_READ_COMMITTED`, `SQL_TXN_READ_UNCOMMITTED`, `SQL_TXN_REPEATABLE_READ`, and `SQL_TXN_SERIALIZABLE`.
If no isolation value is specified, the default is `SQL_TXN_READ_COMMITTED`.

[back to the top]

29005 HY024 Invalid ResultSet type

Cause: An attempt was made to set an invalid `ResultSet` Type value.
Effect: The SQLMXStatement call with the `resultSetType` parameter fails.
Recovery: Valid `ResultSet` types are: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, and `TYPE_SCROLL_SENSITIVE`.

[back to the top]

29006 HY000 Invalid Result Set concurrency

Cause: An attempt was made to set an invalid result-set concurrency value.
Effect: The `SQLMXStatement` call with `resultSetConcurrency` fails.
Recovery: Valid `resultSetConcurrency` values are: `CONCUR_READ_ONLY` and

CONCUR_UPDATABLE.

[back to the top]

29007 07009 Invalid descriptor index

Cause: A `ResultSetMetadata` column parameter or a `ParameterMetaData` param parameter is outside of the descriptor range.
Effect: The `ResultSetMetadata` or `ParameterMetaData` method data is not returned as expected.
Recovery: Validate the column or parameter that is supplied to the method.

[back to the top]

29008 24000 Invalid cursor state

Cause: The `ResultSet` method was called when the connection was closed.
Effect: The method call does not succeed.
Recovery: Make sure the connection is open before making the `ResultSet` method call.

[back to the top]

29009 HY109 Invalid cursor position

Cause: An attempt was made to perform a `deleteRow()` method or `updateRow()` method or `cancelRowUpdates` method when the `ResultSet` row cursor was on the insert row. Or an attempt was made to perform the `insertRow()` method when the `ResultSet` row cursor was not on the insert row.
Effect: The row changes and cursor manipulation do not succeed.
Recovery: To insert a row, move the cursor to the insert row. To delete, cancel, or update a row, move the cursor from the insert row.

[back to the top]

29010 07009 Invalid column name

Cause: A column search does not contain `columnName` string.
Effect: The column comparison or searches do not succeed.
Recovery: Supply a valid `columnName` string to the `findColum()`, `validateGetInvocation()`, and `validateUpdInvocation()` methods.

[back to the top]

29011 07009 Invalid column index or descriptor index

Cause: A `ResultSet` method was issued that has a column parameter that is outside of the valid range.
Effect: The `ResultSet` method data is not returned as expected.
Recovery: Make sure to validate the column that is supplied to the method.

[back to the top]

29012 07006 Restricted data type attribute violation

Cause: An attempt was made to execute a method either while an invalid data type was set or the data type did not match the SQL column type.
Effect: The interface method is not executed.
Recovery: Make sure the correct method and Java data type is used for the column type.

[back to the top]

29013 HY024 Fetch size is less than 0

Cause: The size set for `ResultSet.setFetchSize` rows to fetch is less than zero.
Effect: The number of rows that need to be fetched from the database when more rows are needed for a `ResultSet` object is not set.
Recovery: Set the `setFetchSize()` method rows parameter to a value greater than zero.

[back to the top]

29014 HY000 SQL data type not recognized

Cause: An unrecognized SQL data type was detected by JDBC.
Effect: An exception is thrown; data is not updated.
Recovery: Make sure that the SQL data type is supported by JDBC. The error is internal to the JDBC/MX driver.

[back to the top]

29015 HY024 Invalid fetch direction

Cause: The `setFetchDirection()` method direction parameter is set to an invalid value.
Effect: The direction in which the rows in this `ResultSet` object are processed is not set.
Recovery: Valid fetch directions are: `ResultSet.FETCH_FORWARD`, `ResultSet.FETCH_REVERSE`, and `ResultSet.FETCH_UNKNOWN`.

[back to the top]

29016 22018 SQL column {0,number,integer} data type cannot be converted to the specified Java data type

Cause: Attempted to convert a non-numeric string to `BigDecimal` using the `ResultSet.getLong()` method.
Effect: An exception is reported and no data is obtained.
Recovery: Ensure that the column is a valid type to be converted.

[back to the top]

29017 HY004 SQL data type not supported

Cause: An unsupported SQL data type was detected in a setter method.
Effect: `ARRAY`, `BINARY`, `BIT`, `DATALINK`, `JAVA_OBJECT`, and `REF` data types are not supported.
Recovery: Use a supported data type with the JDBC setter method.

[back to the top]

29018 22018 Invalid character value in cast specification

Cause: An attempt was made to convert a string to a numeric type but the string does not have the appropriate format.
Effect: Strings that are obtained through a getter method cannot be cast to the method type.
Recovery: Validate the string in the database to make sure it is a compatible type.

[back to the top]

29019 07002 Parameter {0, number, integer} for {1, number, integer} set of parameters is not set

Cause: An input descriptor contains a parameter that does not have a value set.
Effect: The `method checkIfAllParamsSet()` reports the parameter that is not set.
Recovery: Set a value for the listed parameter.

[back to the top]

29020 07009 Invalid parameter index

Cause: A getter or setter method parameter count index is outside of the valid input-descriptor range, or the input-descriptor range is null.
Effect: The getter and setter method invocation validation fails.
Recovery: Change the getter or setter parameter index to a valid parameter value.

[back to the top]

29021 HY004 Object type not supported

Cause: A prepared-statement `setObject()` method call contains an unsupported Object Type.
Effect: The `setObject()` method does not set a value for the designated parameter.
Recovery: Informational message only; no corrective action is needed. Valid Object Types are: `null`, `BigDecimal`, `Date`, `Time`, `Timestamp`, `Double`, `Float`, `Long`, `Short`, `Byte`, `Boolean`, `String, byte[]`, `Clob`, and `Blob`.

[back to the top]

29022 HY010 Function sequence error

Cause: The `PreparedStatement.execute()` method does not support the use of the `PreparedStatement.addBatch()` method.
Effect: An exception is reported; the operation is not completed.
Recovery: Use the `PreparedStatement.executeBatch()` method.

[back to the top]

29023 HY109 The cursor is before the first row, therefore no data can be retrieved.

Cause: `getCurrentRow()` is called when the cursor is before the first row.
Effect: An exception is reported; no data is retrieved.
Recovery: Validate the application call to the `getCurrentRow()` method.

[back to the top]

29024 HY109 The cursor is after last row, which could be due to the result set containing no rows, or all rows have been retrieved.

Cause: `getCurrentRow()` is called when the cursor is after the last row.
Effect: An exception is reported; no data is retrieved.
Recovery: Validate the application call to the `getCurrentRow()` method.

[back to the top]

29025 22003 The data value ({0}) is out of range for column/parameter number {1,number,integer}

Cause: An attempt was made to set or get a value to or from the database when the value is outside the valid range for the column data type.

Effect: An exception is thrown; data is not retrieved or updated.
Recovery: Make sure that the value is within the valid range for the column type.

[back to the top]

29026 HY000 Transaction can't be committed or rolled back when AutoCommit mode is on

Cause: An attempt was made to commit a transaction while `AutoCommit` mode is enabled.
Effect: The transaction is not committed.
Recovery: Disable `AutoCommit`. Use the method only when the `AutoCommit` mode is disabled.

[back to the top]

29027 HY011 SetAutoCommit not possible, since a transaction is active

Cause: An attempt was made to call the `setAutoCommit()` mode while a transaction was active.
Effect: The current `AutoCommit` mode is not modified.
Recovery: Complete the transaction, then attempt to set the `AutoCommit` mode.

[back to the top]

29028 22003 The data value ({0}) is negative, but the column/parameter number {1,number,integer} is unsigned

Cause: An attempt was made to set a negative value into an unsigned column.
Effect: An exception is thrown; data is not updated.
Recovery: Make sure that the value is within the valid range for the column type.

[back to the top]

29030 22003 The data value ({0}) had to be rounded up for column/parameter number {1,number,integer}

Cause: The `setBigDecimal()`method rounded up a value to be inserted into a column.
Effect: An SQLWarning is issued to indicate that a value is rounded up. Data is entered into database column.
Recovery: None. This is a warning condition.

[back to the top]

29031 HY000 SQL SELECT statement in batch is illegal

Cause: A `SELECT` SQL statement was used in the `executeBatch()` method.
Effect: An exception is reported; the `SELECT` SQL query cannot be used in batch queries.
Recovery: Use the `executeQuery()` method to issue the `SELECT` SQL statement.

[back to the top]

29032 23000 Row has been modified since it is last read

Cause: An attempt was made to update or delete a `ResultSet` object row while the cursor was on the insert row.
Effect: The `ResultSet` row modification does not succeed.
Recovery: Move the `ResultSet` object cursor away from the row before updating or deleting the row.

[back to the top]

29033 23000 Primary key column value can't be updated

Cause: An attempt was made to update the primary-key column in a table.
Effect: The column is not updated.
Recovery: Columns in the primary-key definition cannot be updated and cannot contain null values, even if you omit the `NOT NULL` clause in the column definition.

[back to the top]

29035 HY000 IO Exception occurred {0}

Cause: An ASCII or Binary or Character stream setter or an updater method resulted in a `java.io.IOException`.
Effect: The designated setter or updater method does not modify the ASCII or Binary or Character stream.
Recovery: Informational message only; no corrective action is needed.

[back to the top]

29036 HY000 Unsupported encoding {0}

Cause: The character encoding is not supported.
Effect: An exception is thrown when the requested character encoding is not supported.
Recovery: ASCII (ISO88591), KANJI, KSC5601, and UCS2 are the only supported character encodings. SQL/MP tables do not support UCS2 character encoding.

[back to the top]

29037 HY106 ResultSet type is TYPE_FORWARD_ONLY

Cause: An attempt was made to point a `ResultSet` cursor to a previous row when the object type is set as `TYPE_FORWARD_ONLY`.
Effect: The `ResultSet` object cursor manipulation does not occur.
Recovery: `TYPE_FORWARD_ONLY` `ResultSet` object type cursors can move forward only. `TYPE_SCROLL_SENSITIVE` and `TYPE_SCROLL_INSENSITIVE` types are scrollable.

[back to the top]

29038 HY107 Row number is not valid

Cause: A `ResultSet absolute()` method was called when the row number was set to 0.
Effect: The cursor is not moved to the specified row number.
Recovery: Supply a positive row number (specifying the row number counting from the beginning of the result set), or supply a negative row number (specifying the row number counting from the end of the result set).

[back to the top]

29039 HY092 Concurrency mode of the ResultSet is CONCUR_READ_ONLY

Cause: An action was attempted on a `ResultSet` object that cannot be updated because the concurrency is set to `CONCUR_READ_ONLY`.
Effect: The `ResultSet` object is not modified.
Recovery: For updates, you must set the `ResultSet` object concurrency to `CONCUR_UPDATABLE`.

[back to the top]

29040 HY000 Operation invalid. Current row is the insert row

Cause: An attempt was made to retrieve update, delete, or insert information on the current insert row.
Effect: The `ResultSet` row information retrieval does not succeed.
Recovery: To retrieve row information, move the `ResultSet` object cursor away from the insert row.

[back to the top]

29041 HY000 Operation invalid. No primary key for the table

Cause: The `getKeyColumns()` method failed on a table that was created without a primary-key column defined.
Effect: No primary-key data is returned for the table.

Recovery: Change the table to include a primary-key column.

[back to the top]

29042 HY000 Fetch size value is not valid

Cause: An attempt was made to set the fetch-row size to a value that is less than 0.
Effect: The number of rows that are fetched from the database when more rows are needed is not set.
Recovery: For the `setFetchSize()` method, supply a valid row value that is greater than or equal to 0.

[back to the top]

29043 HY000 Max rows value is not valid

Cause: An attempt was made to set a limit of less than 0 for the maximum number of rows that any `ResultSet` object can contain.
Effect: The limit for the maximum number of rows is not set.
Recovery: For the `setMaxRows()` method, use a valid value that is greater than or equal to 0.

[back to the top]

29044 HY000 Query timeout value is not valid

Cause: An attempt was made to set a value of less than 0 for the number of seconds the driver waits for a `Statement` object to execute.
Effect: The query timeout limit is not set.
Recovery: For the `setQueryTimeout()` method, supply a valid value that is greater than or equal to 0.

[back to the top]

29045 01S07 Fractional truncation

Cause: The data retrieved by the `ResultSet` getter method has been truncated.
Effect: The data retrieved is truncated.
Recovery: Make sure that the data to be retrieved is within a valid data-type range.

[back to the top]

29046 22003 Numeric value out of range

Cause: A value retrieved from the `ResultSet` getter method is outside the range for the data type.
Effect: The `ResultSet` getter method does not retrieve the data.
Recovery: Make sure the data to be retrieved is within a valid data-type range.

[back to the top]

29047 HY000 Batch update failed. See next exception for details

Cause: One of the commands in a batch update failed to execute properly.
Effect: Not all the batch-update commands succeed. See the subsequent exception for more information.
Recovery: View the subsequent exception for possible recovery actions.

[back to the top]

29048 HY009 Invalid use of null

Cause: A parameter that has an expected table name is set to null.
Effect: The `DatabaseMetadata` method does not report any results.
Recovery: For the `DatabaseMetaData` method, supply a valid table name that is not null.

[back to the top]

29049 25000 Invalid transaction state

Cause: The `begintransaction()` method was called when a transaction was in progress.
Effect: A new transaction is not started.
Recovery: Before calling the `begintransaction()` method, validate whether other transactions are currently started.

[back to the top]

29050 HY107 Row value out of range

Cause: A call to `getCurrentRow` retrieved is outside the first and last row range.
Effect: The current row is not retrieved.
Recovery: It is an informational message only; no recovery is needed. Report the entire message to your service provider.

[back to the top]

29051 01S02 ResultSet type changed to TYPE_SCROLL_INSENSITIVE

Cause: The Result Set Type was changed.
Effect: None.
Recovery: This message is reported as an SQL Warning. It is an informational message only; no recovery is needed.

[back to the top]

29052 22003 The Timestamp ({0}) is not in format yyyy-mm-dd hh:mm:ss.ffffffff for column/parameter number {1,number,integer}

Cause: An attempt was made to enter an invalid timestamp format into a `TIMESTAMP` column type.
Effect: An exception is thrown; data is not updated.
Recovery: Make sure that a timestamp in the form of `yyyy-mm-dd hh:mm:ss.fffffffff` is used.

[back to the top]

29053 HY000 SQL SELECT statement is invalid in executeUpdate() method

Cause: A select SQL statement was used in the `executeUpdate()` method.
Effect: The `SQL query not performed` exception is reported.
Recovery: Use the `executeQuery()` method to issue the select SQL statement.

[back to the top]

29054 HY000 Only SQL SELECT statements are valid in executeQuery() method

Cause: A non-select SQL statement was used in the `executeQuery()` method.
Effect: The exception reported is "`SQL query not performed`".
Recovery: Use the `executeUpdate()` method to issue the non-select SQL statement.

[back to the top]

29055 22003 The Date ({0}) is not in format yyyy-mm-dd for column/parameter number {1,number,integer}

Cause: An attempt was made to enter an invalid date format into a `DATE` column type.
Effect: An exception is thrown; is not updated.
Recovery: Make sure that a date in the form of `yyyy-mm-dd` is used.

[back to the top]

29056 HY000 Statement is already closed

Cause: A `validateSetInvocation()` or `validateExecuteInvocation` method was used on a closed statement.
Effect: The validation on the statement fails and returns an exception.
Recovery: Use the `validateSetInvocation()` or `validateExecuteInvocation` method

prior to the statement close.

[back to the top]

29057 HY000 Auto generated keys not supported

Cause: An attempt was made to use the Auto-generated keys feature.
Effect: The attempt does not succeed.
Recovery: The Auto-generated keys feature is not supported.

[back to the top]

29058 HY000 Connection is not associated with a PooledConnection object

Cause: The `getPooledConnection()` method was invoked before the `PooledConnection` object was established.
Effect: A connection from the pool cannot be retrieved.
Recovery: Make sure a `PooledConnection` object is established before using the `getPooledConnection()` method.

[back to the top]

29059 HY000 'blobTableName' property is not set or set to null value or set to invalid value

Cause: Attempted to access a `BLOB` column without setting the property `blobTableName`, or the property is set to an invalid value.
Effect: The application cannot access `BLOB` columns.
Recovery: Set the `blobTableName` property to a valid LOB table name. The LOB table name is of format `catalog.schema.lobTableName`.

[back to the top

29060 HY000 'clobTableName' property is not set or set to null value or set to invalid value

Cause: `clobTableName` property is not set or is set to null value or set to an invalid value.
Effect: The application cannot access `CLOB` columns.
Recovery: Set the `clobTableName` property to a valid LOB table name. The LOB table name is of format `catalog.schema.lobTableName.`

[back to the top]

29061 HY000 Lob object {0} is not current

Cause: Attempted to access a `CLOB` column without setting the property `jdbcmx.clobTableName` or the property is set to an invalid value.
Effect: The application cannot access `CLOB` columns.
Recovery: Set the `jdbcmx.clobTableName` property to a valid LOB table name. The LOB table name is of format `catalog.schema.lobTableName.`

[back to the top]

29062 HY000 Operation not allowed since primary key columns are not in the select list

Cause: `getKeyColumns()` fails on table created without a primary key column.
Effect: No primary key data is returned for table.
Recovery: Alter the table to include a primary key column, or remove the `getKeyColumns()` method call from the program.

[back to the top]

29063 HY000 Transaction error {0} - {1} while obtaining start data locator

Cause: A transaction error occurred when the JDBC/MX driver attempted to reserve the data locators for the given process while inserting or updating a LOB column.
Effect: The application cannot insert or update the LOB columns.
Recovery: Check the file-system error in the message and take recovery action accordingly.

[back to the top]

29064 22018 Java data type does not match SQL data type for column

Cause: Attempted to call a `PreparedStatement` setter method with an invalid column data type.
Effect: An exception is thrown; data is not updated.
Recovery: Make sure that the column data type is valid for the `PreparedStatement` setter method.

[back to the top]

29065 22018 Java data type cannot be converted to the specified SQL data type

Cause: A `PreparedStatement` setter method Java object conversion to the given SQL data type is invalid.
Effect: An exception is thrown; data is not updated.
Recovery: Make sure that the column data type is valid for the `PreparedStatement` setter method.

[back to the top]

29066 22018 The String data {0} cannot be converted to a numeric value

Cause: A `PreparedStatement` setter method could not convert a string to an integer.
Effect: An exception is thrown and the string data is not converted.
Recovery: Make sure that the data to be converted to an integer is valid.

[back to the top]

29067 07009 Invalid input value in the method {0}

Cause: One or more input values in the given method is invalid.
Effect: The given input method failed.
Recovery: Check the input values for the given method.

[back to the top]

29068 07009 The value for position can be any value between 1 and one more than the length of the LOB data

Cause: The position input value in `Blob.setBinaryStream`, `Clob.setCharacterStream`, or `Clob.setAsciiStream` can be between 1 and one more than the length of the LOB data.
Effect: The application cannot write the LOB data at the specified position.
Recovery: Correct the position input value.

[back to the top]

29069 HY000 Autocommit is on and LOB objects are involved

Cause: LOB data is involved with autocommit enabled and an external transaction does not exist.
Effect: An exception is reported; the LOB columns are not set.
Recovery: Start an external transaction or disable the autocommit mode when using the `Clob.setASciiStream()`, `Clob.setCharacterStream()`, or `Blob.setBinaryStream()` method.

[back to the top]

29070 HY000 Transaction error {0} - {1} while updating LOB tables

Cause: An SQL or file system (FS) exception occurred during insert or update operations on the base and LOB tables within an internal transaction.
Effect: An exception is reported; the internal transaction is rolled back.
Recovery: See the SQL or FS error message.

[back to the top]

29071 HY000 Internal programming error - {0}

Cause: The JNI layer (get Object method) always returns a byte array and, therefore, any other instance is considered a programming error.
Effect: An exception is reported.
Recovery: None. The error is internal to the JDBC/MX driver.

[back to the top]

29072 HY000 Attempting to exceed the maximum connection pool size ({0,number,integer})

Cause: An attempt was made to obtain a connection outside the set connection pool size limit.
Effect: An exception is thrown.
Recovery: Increase the connection pool size by using the `maxPoolSize` command-line property.

[back to the top]

29073 22003 The Time ({0}) is not in format hh:mm:ss for column/parameter number {1,number,integer}

Cause: An attempt was made to enter an invalid time format into a `TIME` column type.
Effect: An exception is thrown; data is not updated.
Recovery: Make sure that a time in the form of `hh:mm:ss` is used.

[back to the top]

29074 42821 The getter method, {0}, cannot be used to retrieve data for column/parameter number {1,number,integer}

Cause: Attempted to use an unsupported column type in the `ResultSet.getString` method.
Effect: An exception is reported; no data is obtained.
Recovery: Use a supported column data type other than `BLOB`, `ARRAY`, `REF`, `STRUCT`, `DATALINK`, or

JAVA_OBJECT with the ResultSet.getString method.

[back to the top]

29075 HY000 'transactionMode' property is set to a null value or set to an invalid value

Cause: Called SQLMXDataSource.setTransactionMode() or SQLMXConnectionPoolDataSource.setTransactionMode() using an invalid transaction mode.
Effect: The application cannot set the transaction mode.
Recovery: Use a valid transaction mode: external, internal, or mixed.

[back to the top]

29076 HY000 Exceeded 'maxStatements' ({0,number,integer}) -- performance may be compromised

Cause: The cached statement count has reached the limit set by the maxStatements property and all statements are in use.
Effect: An SQL warning condition. Statements continue to be added to the internal cache.
Recovery: An SQL warning condition. Use the maxStatements property (or -Djdbcmx.maxStatements command-line property) to increase the number of statements allowed.

[back to the top]

29077 HY000 HY000 Max rows value cannot be less than the fetch size

Cause: The row value passed to the JdbcRowSet.setMaxRows method is less than the current fetch-size setting
Effect: The maximum number of rows that the JdbcRowSet object can contain is not set.
Recovery: Increase the fetch-size value by using the JdbcRowSet.setFetchSize, or increase the maximum-rows value passed to the JdbcRowSet.setMaxRows method.

[back to the top]

29078 HY000 Invalid JdbcRowSet state - {0} {1}

Cause: The Connection, ResultSet, or PreparedStatement value associated with the JdbcRowSet operation is null.
Effect: The method call fails.
Recovery: Make sure a call to the JdbcRowSet.execute() method is performed.

[back to the top]

29079 HY000 Match Columns are not the same as those set

Cause: The designated column passed to `unsetMatchColumn()` method was not previously set as a match column**.**
Effect: The designated column is not unset for this `JdbcRowSet` object.
Recovery: Use the `setMatchColumn()` method to set the designated column as a match column.

[back to the top]

29080 HY000 Set the match columns before getting them

Cause: A call to `getMatchColumnNames()`, `getMatchColumnIndexex()`, or `unsetMatchColumn()` method returns a null or match column.
Effect: A match column value is not retrieved for this `JdbcRowSet` object.
Recovery: Use `setMatchColumn()` to set the designated column as a match column.

[back to the top]

29081 HY000 Match columns should be greater than 0

Cause: The program passed a column index value less than zero to the `setMatchColumn()` method.
Effect: The designated match column for this `JdbcRowSet` object is not set.
Recovery: Call the `setMatchColumn()` method with a valid column index value greater than zero.

[back to the top]

29082 HY000 Match columns cannot be null or empty string

Cause: A null or empty column name string is passed to the `setMatchColumn()` method.
Effect: The designated match column for this `JdbcRowSet` object is not set.
Recovery: Call the `setMatchColumn()` method with a valid non-null column-name string.

[back to the top]

29083 HY000 Columns being unset are not the same as those set

Cause: The designated column passed to the `unsetMatchColumn()` method was not previously set as a match column.
Effect: The designated column is not unset for this `JdbcRowSet` object.
Recovery: Use the `setMatchColumn()` method to set the designated column as a match column.

[back to the top]

29084 HY000 Use column name as argument to unsetMatchColumn

Cause: A column-name string value is passed to the `unsetMatchColumn(integer i)` method.
Effect: An exception is thrown. The designated match column for this `JdbcRowSet` object is not unset.
Recovery: Call the `unsetMatchColumn(integer i)` method with an integer column ID value, or use the `unsetMatchColumn(String s)` method.

[back to the top]

29085 HY000 Use column ID as argument to unsetMatchColumn

Cause: An column name integer value is passed to the `unsetMatchColumn(String s)` method.
Effect: An exception is thrown. The designated match column for this `JdbcRowSet` object is not unset.
Recovery: Call the `unsetMatchColumn(String s)` method with a string column-name value, or use the `unsetMatchColumn(integer i)` method.

[back to the top]

29086 HY000 Missing JdbcRowSet parameter ({0,number,integer})

Cause: An internal driver condition detects that a `JdbcRowSet` parameter does not have a value set.
Effect: The `JdbcRowSet.execute()` method fails.
Recovery: None. The error is internal to the JDBC/MX driver.

[back to the top]

29087 HY000 JdbcRowSet setProperties error {0} - {1}

Cause:An internal driver condition detects that the `JdbcRowSet` property reported in the message could not be set for `JdbcRowSet` prepared statement.
Effect: The `JdbcRowSet.execute()` method fails.
Recovery: The error is internal to the JDBC/MX driver.

[back to the top]

29088 HY000 JdbcRowSet prepare error - {0}

Cause: The driver encountered an internal error when preparing a `JdbcRowSet` prepared statements.
Effect: An exception is reported.
Recovery: None. The error is internal to the JDBC/MX driver.

[back to the top]

29089 HY000 JdbcRowSet connect error - {0} {1}

Cause: The driver encountered an internal error when attempting to establish a connection.
Effect: An exception is reported.
Recovery: None. The error is internal to the JDBC/MX driver.

[back to the top]

Messages From the JNI Side of the JDBC/MX Driver

29251 HY000 Programming Error

Cause: Either SQL has detected an error in one of the SQL parameters for a statement or SQL returned an error for an operation that was attempted but that is not handled by JDBC.
Effect: An exception is reported.
Recovery: For SQL parameter errors, the exception-message text usually identifies the problem to be corrected. For unhandled SQL errors, the Error Code of the exception identifies the SQL error that was caught. Refer to the SQL error-message documentation for details about the error code.

[back to the top]

29252 HY008 Operation Cancelled

Cause: An SQL operation was cancelled by a break
Effect: An exception is reported; the operation is not completed.
Recovery: This message is application-specific. Issue the statement again.

[back to the top]

29253 22003 Numeric value out of range

Cause: A numeric value is not within the range of its target column.
Effect: An exception is reported; the operation is not completed.
Recovery: Adjust the numeric value to a valid range for the SQL column type.

[back to the top]

29254 22001 String data right-truncated

Cause: An attempt was made to place a string in a database but the string exceeds the database limits.
Effect: Some of the data is not placed in the database.
Recovery: Shorten the length of the string.

[back to the top]

29255 HY000 TMF error has occurred : [*tmf-error*]

Cause: An internal transaction request failed.
Effect: An exception is reported; the operation is not completed.
Recovery: Refer to the TMF error message *tmf-error*.

[back to the top]

29256 HY000 Error while obtaining the system catalog name : [*error*]

Cause: During initialization of the JDBC driver, an error occurred when attempting to determine a system catalog name.
Effect: The JDBC driver is not registered with the Driver Manager.
Recovery: Make sure that SQL is installed and that a system catalog exists.

[back to the top]

29257 07002 All parameters are not set

Cause: A parameter that was read was null.
Effect: An exception is reported; the operation is not completed.
Recovery: Enter a non-null parameter value.

[back to the top]

29258 25000 Invalid Transaction State

Cause: A transaction-state problem was detected when attempting to begin or resume a transaction through TMF.
Effect: An exception is reported; the operation is not completed.
Recovery: Informational message only; no corrective action is needed. Report the entire message to your service provider.

[back to the top]

29259 HY000 Module Error

Cause: An invalid parameter was detected when attempting to get catalog information or attempting to prepare a statement from a module.
Effect: An exception is reported; the operation is not completed.
Recovery: See the exception message for recovery details.

[back to the top]

29260 HY000 Invalid Statement/Connection Handle

Cause: An invalid SQL statement handle was detected.
Effect: An exception is reported; the operation is not completed.
Recovery: Informational message only; no corrective action is needed. Report the entire message to your service provider.

[back to the top]

29261 HY000 No error message in SQL/MX diagnostics area, but sqlcode is non-zero

Cause: An SQL error was detected but no error message was reported by SQL/MX.
Effect: An SQL exception or warning is thrown without a diagnostic message.
Recovery: Unknown.

[back to the top]

29262 HY090 Invalid or null sql string

Cause: A stored-procedure or prepared-statement call contains an invalid SQL string.
Effect: The stored procedure or prepared statement is not executed.
Recovery: Make sure that the stored procedure or prepared statement contains a valid SQL command.

[back to the top]

29263 HY000 Invalid or null statement label or name

Cause: A calling database stored a procedure or a prepared statement that has an invalid statement-label input parameter.
Effect: The stored procedure or prepared statement is not executed.
Recovery: Make sure that the statement-label parameter is valid.

[back to the top]

29264 HY000 Invalid or null module name

Cause: A calling database stored a procedure or a prepared statement that has an invalid module-name input parameter.
Effect: The stored procedure or prepared statement is not executed.
Recovery: Make sure that the module-name parameter is valid.

[back to the top]

29265 HY000 Unsupported character set encoding

Cause: The character-set type for a `CHAR`, `VARCHAR`, `VARCHAR_LONG`, or `VARCHAR_WITH_LENGTH` column is not supported by the JDBC/MX driver setter or getter methods.
Effect: An SQL exception is thrown.
Recovery: Change the column character-set type to a type supported by the JDBC/MX driver.

[back to the top]

29266 HY000 Data type not supported : [*data type*]

Cause: An unsupported data type was retrieved from SQL.
Effect: An exception is thrown.
Recovery: None. `BIT, BITVAR, BPINT_UNSIGNED, SQLTYPECODE_FLOAT, SQLTYPECODE_REAL`, and `SQLTYPECODE_DOUBLE` data types are not expected to be returned from SQL/MX. The JDBC/MX driver does not support these data types.

[back to the top]

29267 HY000 Exceeded JVM allocated memory

Cause: JDBC attempted to internally allocate JVM memory after it has been exhausted.
Effect: The condition is a function of the JVM heap size. An exception is thrown.
Recovery: Configure the maximum JVM heap size accordingly.

[back to the top]

Messages from the Java Portion of the JDBC Driver (range 29000 through 29249)	Messages from the Native-interface Portion of the JDBC Driver (range 29250 through 29499)

Appendix A. Sample Programs Accessing CLOB and BLOB Data

This appendix shows two working programs:

Sample Program Accessing CLOB Data

This sample program shows operations that can be performed through the `CLOB` interface or through the `PreparedStatement` interface. The sample program shows examples of both interfaces taking a variable and putting the variable's value into a base table that has a `CLOB` column.

```
//  LOB operations can be performed through the Clob interface,
//  or the PreparedStatement interface.
//  This program shows examples of both interfaces taking a
//  variable and putting it into the cat.sch.clobbase table.
//
//  The LOB base table for this example is created as:
//     >> create table clobbase
//               (col1 int not null not droppable,
//                col2 clob, primary key (col1));
//
//  The LOB table for this example is created through
//  the JdbcMxLobAdmin utility as:
//     >> create table cat.sch.clobdatatbl
//               (table_name char(128) not null not droppable,
//                data_locator largeint not null not droppable,
//                chunk_no int not null not droppable,
//                lob_data varchar(3880),
//                primary key(table_name, data_locator, chunk_no))
//                attributes extent(1024), maxextents 768 ;
//
//  ***** The following is the Clob interface...
//     - insert the base row with EMPTY_CLOB() as value for
//       the LOB column
```

```
//     - select the LOB column 'for update'
//     - load up a byte[] with the data
//     - use Outputstream.write(byte[])
//
//  ***** The following is the PreparedStatement interface...
//     - need an Inputstream object that already has data
//     - need a PreparedStatement object that contains the
//            'insert...' DML of the base table
//     - ps.setAsciiStream() for the lob data
//     - ps.executeupdate(); for the DML
//
// To run this example, issue the following:
//    # java TestCLOB 1 TestCLOB.java 1000
//

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

public class TestCLOB
{
 public static void main (String[] args)
        throws java.io.FileNotFoundException,
      java.io.IOException
 {
  int           length = 500;
  int           recKey;
  long          start;
  long          end;
  Connection    conn1 = null;

  // Set jdbcmx.clobTableName System Property. This property
  // can also be added to the command line through
  // "-Djdbcmx.clobTableName=...", or a
  // java.util.Properties object can be used and passed to
  // getConnection.
  System.setProperty( "jdbcmx.clobTableName","cat.sch.clobdatatbl" );

  if (args.length < 2) {
        System.out.println("arg[0]=; arg[1]=file;
        arg[2]=");
        return;
  }

  String  k = "K";
  for (int i=0; i<5000; i++) k = k + "K";
```

```
  System.out.println("string length = " + k.length());

  FileInputStream clobFs = new FileInputStream(args[1]);
  int clobFsLen = clobFs.available();

  if (args.length == 3)
        length = Integer.parseInt(args[2]);
  recKey = Integer.parseInt(args[0]);

  System.out.println("Key: " + recKey +"; Using "
     + length + " of file " + args[1]);

  try {
    Class.forName("com.tandem.sqlmx.SQLMXDriver");
    start = System.currentTimeMillis();
    conn1 = DriverManager.getConnection("jdbc:sqlmx:");

    System.out.println("Cleaning up test tables...");
    Statement stmt0 = conn1.createStatement();
    stmt0.execute("delete from clobdatatbl");
    stmt0.execute("delete from clobbase");

    conn1.setAutoCommit(false);

  }
  catch (Exception e1) {
    e1.printStackTrace();
  }

// PreparedStatement interface example - This technique
// is suitable if the LOB data is already on the NonStop
// system disk.
  try {
    System.out.println("PreparedStatement interface
        LOB insert...");
    String stmtSource1 = "insert into clobbase
        values (?,?)";
    PreparedStatement stmt1
        = conn1.prepareStatement(stmtSource1);
    stmt1.setInt(1,recKey);
    stmt1.setAsciiStream(2,clobFs,length);
    stmt1.executeUpdate();
    conn1.commit();
  }
  catch (SQLException e) {
     e.printStackTrace();
```

```
    SQLException next = e;
    do {
        System.out.println("Messge : " + e.getMessage());
        System.out.println("Error Code : " + e.getErrorCode());
        System.out.println("SQLState : " + e.getSQLState());
    }  while ((next = next.getNextException()) != null);
  }

// Clob interface example - This technique is suitable when
// the LOB data is already in the app, such as having been
// transferred in a msgbuf.
  try {
    // insert a second base table row with an empty LOB column
    System.out.println("CLOB interface EMPTY LOB insert...");
    String stmtSource2 = "insert into clobbase
                          values (?,EMPTY_CLOB())";
    PreparedStatement stmt2
                        = conn1.prepareStatement(stmtSource2);
    stmt2.setInt(1,recKey+1);
    stmt2.executeUpdate();

    Clob clob = null;

    System.out.println("Obtaining CLOB data to
        update (EMPTY in this case)...");
    PreparedStatement stmt3
        = conn1.prepareStatement("select col2
        from clobbase where col1 = ? for update");
    stmt3.setInt(1,recKey+1);
    ResultSet rs = stmt3.executeQuery();
    if (rs.next())  clob = rs.getClob(1);  // has to be there
//                             else the base table insert fails

    System.out.println("Writing data to previously empty CLOB...");
    OutputStream os = clob.setAsciiStream(1);
    byte[] bData = k.getBytes();
    os.write(bData);
    os.close();
    conn1.commit();
  }
  catch (SQLException e) {
    e.printStackTrace();
    SQLException next = e;
    do {
        System.out.println("Messge : " + e.getMessage());
        System.out.println("Vendor Code : " + e.getErrorCode());
```

```
        System.out.println("SQLState : " + e.getSQLState());
    }  while ((next = next.getNextException()) != null);
  }

 } // main
} // class
```

Sample Program Accessing BLOB Data

This sample program shows the use of both the `Blob` interface and the `PreparedStatement` interface to take a byte variable and put the variable's value into a base table that has a `BLOB` column.

```
//  LOB operations may be performed throubh the Blob, or
//  PreparedStatement interface. This program shows examples of
//  using both interfaces taking a byte[] variable and putting
//  it into the cat.sch.blobtiff table.
//
//  The LOB base table for this example is created as:
//     >> create table blobtiff
//               (col1 int not null not droppable,
//                tiff blob, primary key (col1));
//
//  The LOB table for this example is created through the
//  JdbcMxLobAdmin utility as:
//     >> create table cat.sch.blobdatatbl
//               (table_name char(128) not null not droppable,
//                data_locator largeint not null not droppable,
//                chunk_no int not null not droppable,
//                lob_data varchar(3880),
//                primary key(table_name, data_locator, chunk_no))
//                attributes extent(1024), maxextents 768 ;
//
//  ***** The following is the blob interface...
//     - insert the base row with EMPTY_BLOB() as value for
//       the LOB column
//     - select the lob column 'for update'
//     - load up a byte[] with the data
//     - use Outputstream.write(byte[])
//
//  ***** The following is the prep stmt interface...
//     - need an Inputstream object that already has data
//     - need a PreparedStatement object that contains the
//           'insert...' DML of the base table
```

```
//     - ps.setAsciiStream() for the lob data
//     - ps.executeupdate(); for the DML
//
// To run this example, issue the following:
//    # java TestBLOB 1 TestBLOB.class 1000
//

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

public class TestBLOB
{
 public static void main (String[] args)
        throws java.io.FileNotFoundException, java.io.IOException
 {
  int           numBytes;
  int           recKey;
  long          start;
  long          end;
  Connection    conn1 = null;

  // Set jdbcmx.blobTableName System Property. This property
  // can also be added to the command line through
  // "-Djdbcmx.blobTableName=...", or a
  // java.util.Properties object can be used and passed to
  // getConnection.
  System.setProperty( "jdbcmx.blobTableName","cat.sch.blobdatatbl" );

  if (args.length < 2) {
        System.out.println("arg[0]=; arg[1]=file; arg[2]=");
        return;
  }

  // byte array for the blob
  byte[] whatever = new byte[5000];
  for (int i=0; i<5000; i++) whatever[i] = 71;   // "G"

  String  k = "K";
  for (int i=0; i<5000; i++) k = k + "K";
  System.out.println("string length = " + k.length());

  java.io.ByteArrayInputStream iXstream
      = new java.io.ByteArrayInputStream(whatever);

  numBytes = iXstream.available();
  if (args.length == 3)
```

```
        numBytes = Integer.parseInt(args[2]);
    recKey = Integer.parseInt(args[0]);

    System.out.println("Key: " + recKey +"; Using "
                       + numBytes + " of file " + args[1]);

    try {
      Class.forName("com.tandem.sqlmx.SQLMXDriver");
      start = System.currentTimeMillis();
      conn1 = DriverManager.getConnection("jdbc:sqlmx:");

      System.out.println("Cleaning up test tables...");
      Statement stmt0 = conn1.createStatement();
      stmt0.execute("delete from blobdatatbl");
      stmt0.execute("delete from blobtiff");

      conn1.setAutoCommit(false);

    }
    catch (Exception e1) {
      e1.printStackTrace();
    }

// PreparedStatement interface example - This technique is
// suitable if the LOB data is already on the
// NonStop system disk.
    try {
      System.out.println("PreparedStatement interface LOB insert...");
      String stmtSource1 = "insert into blobtiff values (?,?)";
      PreparedStatement stmt1 = conn1.prepareStatement(stmtSource1);
      stmt1.setInt(1,recKey);
      stmt1.setBinaryStream(2,iXstream,numBytes);
      stmt1.executeUpdate();
      conn1.commit();
    }
    catch (SQLException e) {
       e.printStackTrace();
       SQLException next = e;
       do {
           System.out.println("Messge : " + e.getMessage());
           System.out.println("Error Code : " + e.getErrorCode());
           System.out.println("SQLState : " + e.getSQLState());
       }  while ((next = next.getNextException()) != null);
    }

// Blob interface example - This technique is suitable when
```

```
// the LOB data is already in the app, such as having been
// transfered in a msgbuf.
  try {
    // insert a second base table row with empty LOB column
    System.out.println("BLOB interface LOB insert...");
    String stmtSource2 = "insert into blobtiff
                          values (?,EMPTY_BLOB())";
    PreparedStatement stmt2 = conn1.prepareStatement(stmtSource2);
    stmt2.setInt(1,recKey+1);
    stmt2.executeUpdate();

    Blob tiff = null;

    System.out.println("Obtaining BLOB data to
                        update (EMPTY in this case)...");
    PreparedStatement stmt3 = conn1.prepareStatement("select tiff
        from blobtiff where col1 = ? for update");
    stmt3.setInt(1,recKey+1);
    ResultSet rs = stmt3.executeQuery();
    if (rs.next())  tiff = rs.getBlob(1);  // has to be there
                                else the base table insert failed

    System.out.println("Writing data to previously
                       empty BLOB...");
    OutputStream os = tiff.setBinaryStream(1);
    byte[] bData = k.getBytes();
    os.write(bData);
    os.close();
    conn1.commit();
  }
  catch (SQLException e) {
    e.printStackTrace();
    SQLException next = e;
    do {
        System.out.println("Messge : " + e.getMessage());
        System.out.println("Vendor Code : " + e.getErrorCode());
        System.out.println("SQLState : " + e.getSQLState());
    }  while ((next = next.getNextException()) != null);
  }

 } // main
} // class
```

HP JDBC/MX 5.0 Driver for SQL/MX Programmer's Reference (540388-004)

Glossary

A B C D E F G H I J L M N O P Q R S T U V W X Y Z

A

abstract class

In Java, a class designed only as a parent from which subclasses can be derived, which is not itself suitable for instantiation. An abstract class is often used to "abstract out" incomplete sets of features, which can then be shared by a group of sibling subclasses that add different variations of the missing pieces.

American Standard Code for Information Interchange (ASCII)

The predominant character set encoding of present-day computers. ASCII uses 7 bits for each character. It does not include accented letters or any other letter forms not used in English (such as the German sharp-S or the Norwegian ae-ligature). Compare with Unicode.

American National Standards Institute (ANSI)

The United States government body responsible for approving US standards in many areas, including computers and communications. ANSI is a member of ISO. ANSI sells ANSI and ISO (international) standards.

ANSI

See American National Standards Institute (ANSI).

API

See application program interface (API).

application program

One of the following:

- A software program written for or by a user for a specific purpose
- A computer program that performs a data processing function rather than a control function

application program interface (API)

A set of functions or procedures that are called by an application program to communicate with other software components.

ASCII

See American Standard Code for Information Interchange (ASCII).

autocommit mode

A mode in which a JDBC driver automatically commits a transaction without the programmer's

calling `commit()`.

B

base table

A table that has physical existence: that is, a table stored in a file.

BLOB

Short for Binary Large OBject, a collection of binary data stored as a single entity in a database management system. These entities are primarily used to hold multimedia objects such as images, videos, and sound. They can also be used to store programs or even fragments of code. A Java `Blob` object (Java type, `java.sql.Blob`) corresponds to the SQL `BLOB` data type.

branded

A Java virtual machine that Sun Microsystems, Inc. has certified as conformant.

browser

A program that allows you to read hypertext. The browser gives some means of viewing the contents of nodes and of navigating from one node to another. Internet Explorer, Netscape Navigator, NCSA Mosaic, Lynx, and W3 are examples for browsers for the WWW. They act as clients to remote servers.

bytecode

The code that `javac`, the Java compiler, produces. When the Java virtual machine loads this code, it either interprets it or just-in-time compiles it into native RISC code.

C

catalog

In SQL/MP and SQL/MX, a set of tables containing the descriptions of SQL objects such as tables, views, columns, indexes, files, and partitions.

class path

The location where the Java VM and other Java programs that are located in the `/usr/tandem/java/bin` directory search for class libraries (such as classes.zip). The JDBC/MX driver programs are in `/usr/tandem/jdbcMX/current/lib/jdbcMx.jar`. You can set the class path explicitly or with the `CLASSPATH` environment variable.

CLOB

Short for Character Large OBject, text data stored as a single entity in a database management system. A Java `Clob` object (Java type, `java.sql.Clob`) corresponds to the SQL `CLOB` data type.

client

A software process, hardware device, or combination of the two that requests services from a server. Often, the client is a process residing on a programmable workstation and is the part of a

program that provides the user interface. The workstation client might also perform other portions of the program logic. Also called a requester.

command

The operation demanded by an operator or program; a demand for action by, or information from, a subsystem. A command is typically conveyed as an interprocess message from a program to a subsystem.

concurrency

A condition in which two or more transactions act on the same record in a database at the same time. To process a transaction, a program must assume that its input from the database is consistent, regardless of any concurrent changes being made to the database. TMF manages concurrent transactions through concurrency control.

concurrency control

Protection of a database record from concurrent access by more than one process. TMF imposes this control by dynamically locking and unlocking affected records to ensure that only one transaction at a time accesses those records.

connection pooling

A framework for pooling JDBC connections.

Core Packages

The required set of APIs in a Java platform edition which must be supported in any and all compatible implementations.

D

Data Control Language (DCL)

The set of data control statements within the SQL/MP language.

Data Manipulation Language (DML)

The set of data-manipulation statements within the SQL/MP language. These statements include INSERT, DELETE, and UPDATE, which cause database modifications that Remote Duplicate Database Facility (RDF) can replicate.

DCL

See Data Control Language (DCL).

DML

See Data Manipulation Language (DML).

driver

A class in JDBC that implements a connection to a particular database management system such as NonStop SQL/MX. The NonStop Server for Java 5 has these driver implementations: JDBC/MP Driver for NonStop SQL/MP and JDBC/MX Driver for NonStop SQL/MX.

`DriverManager`

The JDBC class that manages drivers.

E

exception

An event during program execution that prevents the program from continuing normally; generally, an error. Java methods raise exceptions using the `throw` keyword and handle exceptions using `try`, `catch`, and `finally` blocks.

Expand

The NonStop operating system network that extends the concept of fault tolerance to networks of geographically distributed NonStop systems. If the network is properly designed, communication paths are constantly available even if there is a single line failure or component failure.

expandability

See scalability.

F

fault tolerance

The ability of a computer system to continue processing during and after a single fault (the failure of a system component) without the loss of data or function.

G

`get()` method

A method used to read a data item. For example, the `SQLMPConnection.getAutoCommit()` method returns the transaction mode of the JDBC driver's connection to an SQL/MP or SQL/MX database. Compare to `set()` method.

Guardian

An environment available for interactive and programmatic use with the NonStop operating system. Processes that run in the Guardian environment use the Guardian system procedure calls as their API. Interactive users of the Guardian environment use the HP Tandem Advanced Command Language (TACL) or another HP product's command interpreter. Compare to OSS.

H

Hotspot virtual machine

See Java Hotspot virtual machine.

HP JDBC Driver for NonStop SQL/MP (JDBC/MP)

The product that provides access to NonStop SQL/MP and conforms to the JDBC API.

HP JDBC Driver for NonStop SQL/MX (JDBC/MX)

The product that provides access to NonStop SQL/MX and conforms to the JDBC API.

HP NonStop ODBC Server

The HP implementation of ODBC for NonStop systems.

HP NonStop operating system

The operating system for NonStop systems.

HP NonStop Server for Java, based on Java 2 Platform Standard> Edition 5.0

The formal name of the NonStop Server for Java product whose Java virtual machine conforms to the Java 2 Platform, Standard Edition (J2SE) 5.0. See also NonStop Server for Java 5.

HP NonStop SQL/MP (SQL/MP)

HP NonStop Structured Query Language/MP, the HP relational database management system for NonStop servers.

HP NonStop SQL/MX (SQL/MX)

HP NonStop Structured Query Language/MX, the HP next-generation relational database management system for business-critical applications on NonStop servers.

HP NonStop system

HP computers (hardware and software) that support the NonStop operating system.

HP NonStop Transaction Management Facility (TMF)

An HP product that provides transaction protection, database consistency, and database recovery. The NonStop Server for Java's NonStop SQL/MX drivers call procedures in the TMF subsystem.

hyperlink

A reference (link) from a point in one hypertext document to a point in another document or another point in the same document. A browser usually displays a hyperlink in a different color, font, or style. When the user activates the link (usually by clicking on it with the mouse), the browser displays the target of the link.

hypertext

A collection of documents (nodes) containing cross-references or links that, with the aid of an interactive browser, allow a reader to move easily from one document to another.

Hypertext Mark-up Language (HTML)

A hypertext document format used on the World Wide Web.

Hypertext Transfer Protocol (HTTP)

The client-server Transmission Control Protocol/Internet Protocol (TCP/IP) used on the World Wide Web for the exchange of HTML documents.

I

IEC

See International Electrotechnical Commission (IEC).

IEEE

Institute for Electrical and Electronic Engineers (IEEE).

interactive

Question-and-answer exchange between a user and a computer system.

interface

In general, the point of communication or interconnection between one person, program, or device and another, or a set of rules for that interaction. See also API.

International Electrotechnical Commission (IEC)

A standardization body at the same level as ISO.

International Organization for Standardization (ISO)

A voluntary, nontreaty organization founded in 1946, responsible for creating international standards in many areas, including computers and communications. Its members are the national standards organizations of 89 countries, including ANSI.

Internet

The network of many thousands of interconnected networks that use the TCP/IP networking communications protocol. It provides e-mail, file transfer, news, remote login, and access to thousands of databases. The Internet includes three kinds of networks:

- High-speed backbone networks such as NSFNET and MILNET
- Mid-level networks such as corporate and university networks
- Stub networks such as individual LANs

interoperability

One of the following:

- The ability to communicate, execute programs, or transfer data between dissimilar environments, including among systems from multiple vendors or with multiple versions of operating systems from the same vendor. HP documents often use the term *connectivity* in this context, while other vendors use *connectivity* to mean hardware compatibility.
- Within a NonStop system node, the ability to use the features or facilities of one environment from another. For example, the `gtacl` command in the OSS environment allows an interactive user to start and use a Guardian tool in the Guardian environment.

interpreter

The component of the Java VM that interprets bytecode into native machine code.

ISO

See International Organization for Standardization (ISO).

J

J2SE Development Kit (JDK)

The development kit delivered with the J2SE platform. Contrast with J2SE Runtime Environment (JRE). See also, Java 2 Platform Standard Edition (J2SE).

J2SE Runtime Environment (JRE)

The Java virtual machine and the Core Packages. This is the standard Java environment that the `java` command invokes. Contrast with J2SE Development Kit (JDK). See also, Java 2 Platform Standard Edition (J2SE).

`jar`

The Java Archive tool, which combines multiple files into a single Java Archive (JAR) file. Also, the command to run the Java Archive Tool.

JAR file

A Java Archive file, produced by the Java Archive Tool, **`jar`**.

`java`

The Java interpreter, which executes Java bytecode. Also, the command to run the Java interpreter. The Java command invokes the Java runtime.

Java Database Connectivity (JDBC)

An industry standard for database-independent connectivity between the Java platform and relational databases such as SQL/MP or SQL/MX. JDBC provides a call-level API for SQL-based database access.

Java HotSpot virtual machine

The Java virtual machine implementation designed to produce maximum program-execution speed for applications running in a server environment. The Java HotSpot virtual machine is a run-time environment that features an adaptive compiler that dynamically optimizes the performance of running applications. NonStop Server for Java 5 implements the Java HotSpot virtual machine.

Java Naming and Directory Interface (JNDI)

A standard extension to the Java platform, which provides Java technology-enabled application programs with a unified interface to multiple naming and directory services.

Java Native Interface (JNI)

The C-language interface used by C functions called by Java classes. Includes an Invocation API that invokes a Java VM from a C program.

Java Runtime

The JVM and the Core Packages. This is the standard Java environment that the **`java`** command invokes.

Java virtual machine (JVM)

The process that loads, links, verifies, and interprets Java bytecode. The NonStop Server for Java 5 implements the Java HotSpot virtual machine.

JDBC

See Java Database Connectivity (JDBC).

JDBC API

The programmatic API in Java to access relational databases.

JDBC Trace Facility

A utility designed to trace the entry point of all the JDBC methods called from the Java applications.

JDBC/MP

See HP JDBC Driver for SQL/MP (JDBC/MP).

JDBC/MX

See HP JDBC Driver for SQL/MX (JDBC/MX).

JNDI

See Java Naming and Directory Interface (JNDI).

JNI

See Java Native Interface (JNI).

`jre`

The Java run-time environment, which executes Java bytecode. Also, the command to run the Java run-time environment.

L

LAN

See local area network (LAN).

local area network (LAN)

A data communications network that is geographically limited (typically to a radius of 1 kilometer), allowing easy interconnection of terminals, microprocessors, and computers within adjacent buildings. Ethernet is an example of a LAN.

LOB

Short for Large OBject. Represents either `CLOB` or `BLOB` data.

M

MXCI

SQL/MX Conversational Interface.

N

native

In the context of Java programming, something written in a language other than Java (such as C or C++) for a specific platform.

node

One of the following:

- An addressable device attached to a computer network.
- A hypertext document.

NonStop Server for Java 5

The informal name of the NonStop Server for Java, based on the Java 2 Platform Standard Edition 5.0 products. This product is a Java environment that supports compact, concurrent, dynamic, and portable programs for the enterprise server.

NonStop Technical Library (NTL)

The browser-based interface to NonStop computing technical information. NTL replaces HP Total Information Manager (TIM).

O

ODBC

See Open Database Connectivity (ODBC).

Open Database Connectivity (ODBC)

The standard Microsoft product for accessing databases.

Open System Services (OSS)

An environment available for interactive and programmatic use with the NonStop operating system. Processes that run in the OSS environment use the OSS API. Interactive users of the OSS environment use the OSS shell for their command interpreter. Compare to Guardian.

OSS

See Open System Services (OSS).

P

package

A collection of related classes; for example, JDBC.

persistence

A property of a programming language where created objects and variables continue to exist and retain their values between runs of the program.

portability

The ability to transfer programs from one platform to another without reprogramming. A characteristic of open systems. Portability implies use of standard programming languages such as C.

Portable Operating System Interface X (POSIX)

A family of interrelated interface standards defined by ANSI and Institute for Electrical and Electronic Engineers (IEEE). Each POSIX interface is separately defined in a numbered ANSI/IEEE standard or draft standard. The standards deal with issues of portability, interoperability, and uniformity of user interfaces.

POSIX

See Portable Operating System Interface X (POSIX).

protocol

A set of formal rules for transmitting data, especially across a network. Low-level protocols define electrical and physical standards, bit-ordering, byte-ordering, and the transmission, error detection, and error correction of the bit stream. High-level protocols define data formatting, including the syntax of messages, the terminal-to-computer dialogue, character sets, sequencing of messages, and so on.

R

_RLD_LIB_PATH

The location where the Java VM and other Java programs search for the TNS/E jdbcMx PIC file. Set _RLD_LIB_PATH explicitly or with the `_RLD_LIB_PATH` environment variable.

RDF

See Remote Duplicate Database Facility (RDF).

Remote Duplicate Database Facility (RDF)

The HP software product that does the following:

- Assists in disaster recovery for online transaction processing (OLTP) production databases
- Monitors database updates audited by the TMF subsystem on a primary system and applies those updates to a copy of the database on a remote system

S

scalability

The ability to increase the size and processing power of an online transaction processing system by adding processors and devices to a system, systems to a network, and so on, and to do so easily and transparently without bringing systems down. Sometimes called expandability.

server

One of the following:

- An implementation of a system used as a stand-alone system or as a node in an Expand network.
- The hardware component of a computer system designed to provide services in response to requests received from clients across a network. For example, NonStop system servers provide transaction processing, database access, and other services.
- A process or program that provides services to a client. Servers are designed to receive request messages from clients; perform the desired operations, such as database inquiries or updates, security verifications, numerical calculations, or data routing to other computer systems; and return reply messages to the clients.

`set()` method

A method used to modify a data item. For example, the `SQLMPConnection.setAutoCommit()` method changes the transaction mode of the JDBC driver's connection to an SQL/MP or SQL/MX database. Compare to `get()` method.

SQL context

An instantiation of the SQL executor with its own environment.

SQLJ

Also referred to as SQLJ Part 0, the "Database Language SQL—Part 10: Object Language Bindings (SQL/OLB)" part of the ANSI SQL-2002 standard that allows static SQL statements to be embedded directly in a Java program.

SQL/MP

See HP NonStop SQL/MP.

SQL/MX

See HP NonStop SQL/MX.

statement pooling

A framework for pooling `PreparedStatement` objects.

stored procedure

A procedure registered with SQL/MX and invoked by SQL/MX during execution of a CALL statement. Stored procedures are especially important for client/server database systems because storing the procedure on the server side means that it is available to all clients. And when the procedure is modified, all clients automatically get the new version.

stored procedure in Java (SPJ)

A stored procedure whose body is a static Java method.

stub

One of the following:

- A dummy procedure used when linking a program with a run-time library. The stub need not contain any code. Its only purpose is to prevent "undefined label" errors at link time.
- A local procedure in a remote procedure call (RPC). A client calls the stub to perform a task, not necessarily aware that the RPC is involved. The stub transmits parameters over the network to the server and returns results to the caller.

T

thread

A task that is separately dispatched and that represents a sequential flow of control within a process.

`threads`

The nonnative thread package that is shipped with Sun Microsystems JDK.

`throw`

Java keyword used to raise an exception.

`throws`

Java keyword used to define the exceptions that a method can raise.

TMF

See HP NonStop Transaction Management Facility (TMF)

transaction

A user-defined action that a client program (usually running on a workstation) requests from a server.

Transaction Management Facility (TMF)

A set of HP software products for NonStop systems that assures database integrity by preventing incomplete updates to a database. It can continuously save the changes that are made to a database (in real time) and back out these changes when necessary. It can also take online "snapshot" backups of the database and restore the database from these backups.

trigger

A trigger defines a set of actions that are executed automatically whenever a delete, insert, or update operation occurs on a specified base table.

U

Unicode

A character-coding scheme designed to be an extension of ASCII. By using 16 bits for each character (rather than ASCII's 7), Unicode can represent almost every character of every language and many symbols (such as "&") in an internationally standard way, eliminating the complexity of incompatible extended character sets and code pages. Unicode's first 128 codes correspond to those of standard ASCII.

uniform resource locator (URL)

A draft standard for specifying an object on a network (such as a file, a newsgroup, or, with JDBC, a database). URLs are used extensively on the World Wide Web. HTML documents use them to specify the targets of hyperlinks.

URL

See uniform resource locator (URL).

V

virtual machine (VM)

A self-contained operating environment that behaves as if it is a separate computer. See also Java virtual machine and Java Hotspot virtual machine.

W

World Wide Web (WWW)

An Internet client-server hypertext distributed information retrieval system that originated from the CERN High-Energy Physics laboratories in Geneva, Switzerland. On the WWW everything (documents, menus, indexes) is represented to the user as a hypertext object in HTML format. Hypertext links refer to other documents by their URLs. These can refer to local or remote resources accessible by FTP, Gopher, Telnet, or news, as well as those available by means of the HTTP protocol used to transfer hypertext documents. The client program (known as a browser) runs on the user's computer and provides two basic navigation operations: to follow a link or to send a query to a server.

WWW

See World Wide Web (WWW).

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W
