Contents

HP JDBC/MP Driver for NonStop SQL/MP Programmer's Reference for H10

Abstract

This document describes how to use the JDBC/MP Driver for NonStop SQL/MP on HP Integrity NonStop™ NS-series servers. JDBC/MP provides NonStop Server for Java applications with JDBC access to HP NonStop SQL/MP. JDBC/MP driver conforms where applicable to the standard JDBC 3.0 API from Sun Microsystems, Inc.

Product Version

HP JDBC/MP Driver for NonStop SQL/MP H10

Supported Hardware

All HP Integrity NonStop NS-series servers

Supported Release Version Updates (RVUs)

This publication supports H06.04 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
529851-001	January 2006

Document History

Part Number	Product Version	Published

526349-002	JDBC Driver for SQL/MP (JDBC/MP) V21	September 2003
527401-001	JDBC Driver for SQL/MP (JDBC/MP) V30	October 2003
527401-002	JDBC Driver for SQL/MP (JDBC/MP) V30	July 2004
527401-003	JDBC Driver for SQL/MP (JDBC/MP) V30 and H10	May 2005
529851-001	JDBC/MP Driver for NonStop SQL/MP H10	January 2006

Legal Notices

© Copyright 2006 Hewlett-Packard Development Company L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Export of this documentation may require authorization from the U.S. Department of Commerce.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Itanium, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks, and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE

OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. This documentation and the software to which it relates are derived in part from materials supplied by the following:

© 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation. © 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991 Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

JDBC Driver for SQL/MP Programmer's Reference

Contents

- **About This Document**
  - New and Changed Information
  - Is This Document for You?
  - Document Structure
  - Printing This Document
  - Related Reading
    - NonStop System Computing Documents
    - Sun Microsystems Documents
  - Notation Conventions
  - Abbreviations
- **Introduction to JDBC/MP Driver**
  - JDBC Driver Types
  - JDBC/MP Architecture
  - JDBC/MP API Packages
  - Limitation

- **Installing and Verifying JDBC/MP**
  - JDBC/MP Driver Requirements
  - Verifying the JDBC/MP Driver

- **Accessing SQL Databases with SQL/MP**
  - Setting a Column in an SQL Table to Null
  - Transactions and the JDBC Driver for SQL/MP
    - Autocommit Mode and the Standard Driver
    - Nonautocommit Mode and the Standard Driver
    - Autocommit Mode and the Transaction-Aware Driver
    - Nonautocommit Mode and the Transaction-Aware Driver
  - JDBC `DriverManager`
  - Loading a Driver
    - Specifying the JDBC Driver Class on the `java` Command Line
    - Adding the JDBC Driver Class to the `jdbc.drivers` Property
    - Loading the JDBC Driver Class Directly into the JVM
  - Connecting a Program to a Database
    - Standard Driver Example
    - Transaction-Aware Driver Example
  - Passing SQL/MP Statements to a Database
  - Compatible Java and SQL/MP Data Types
  - Referring to Database Objects
    - When You Can Use Aliases
    - Creating and Using Aliases
    - Which SQL Statements Support Aliases
  - Connection Pooling and the Application Server
    - Guidelines for Implementing an Application Server to Use Connection Pooling
    - Standard `ConnectionPoolDataSource` Object Properties
    - Registering the `ConnectionPoolDataSource` and DataSource with a JNDI-Based Name Service
  - Connection Pooling in a Basic DataSource Implementation

        - Basic DataSource Object Properties
        - Requesting Connection and Statement Pooling
        - Determining of the Pool is Large Enough
        - Using an SQLMPDataSource with an SQLMPConnectionPoolDataSource
    - Statement Pooling
    - JDBC/MP Connection-Pooling and Statement-Pooling Properties
        - maxStatements Property
        - minPoolSize Property
        - maxPoolSize Property
        - poolLogging Property
        - TransactionMode Property
        - jdbcPropfile Property
    - Troubleshooting
        - Tracing
        - No suitable driver Error
        - Data Truncation
        - Dangling Statements
    - Sample SQL/MP Program

- **JDBC/MP Compliance**
    - JDBC API Call Exception Summary
    - Compliance Information
    - JDBC Driver for SQL/MP Support for JDBC 3.0
        - Result Set Support
        - Metadata Support
        - Batch Update Support
        - BLOBs and CLOBs
        - JDBC 2.0 Standard Extensions
    - Floating Point Support
    - HP Extensions
- **JDBC Trace Facility**
    - Tracing Using the DriverManager Class

- Tracing Using the DataSource Implementation
- Tracing by Loading the Trace Driver Within the Program
- Tracing Using a Wrapper Data Source
- Output Format
- JDBC Trace Facility Demonstration Programs

- **Glossary**
- **Index**
- **List of Examples**
- **List of Figures**
- **List of Tables**

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

About This Document

This section explains these subjects:

- New and Changed Information
- Is This Document for You?
- Document Structure
- Printing This Document
- Related Reading
- Notation Conventions
- Abbreviations

New and Changed Information

This revision (part number 529851-001) of the HP JDBC/MP Driver for NonStop SQL/MP Programmer's Reference for H10 is an update to the manual about using the JDBC/MP H10 driver. This manual replaces the previous revision, part number 527401-003 for H-series information only.

Changes added to this revision—part number 529851-001:

- Updated the Supported Hardware information.
- Updated the Supported Release Version Updates (RVUs) information.
- Moved the API information about the JDBC Driver for SQL/MP to a new manual, HP JDBC/MP Driver for NonStop SQL/MP API Reference for H10 (540496-001).
- Updated links to Sun Microsystems J2SE documentation from version 1.4.2 to 1.5.0.

Changes added to the previous revision—part number 527401-003:

- Added a Note in the Installing and Verifying JDBC/MP chapter.
- Added information about the JDBC/MP driver limitation.
- Added and changed these glossary definitions:
  - HP NonStop operating system (previously HP NonStop Kernel operating system)
  - TNS/E
  - TNS/R

Is This Document for You?

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

This document assumes you are already familiar with NonStop Server for Java—the Java implementation for use in enterprise Java applications on HP NonStop servers. NonStop Server for Java 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 SDK, for example, NonStop Server for Java, based on Java™ 2 Platform Standard

Edition, SDK 1.5.0.

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 this 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 Driver for SQL/MP	Describes the JDBC/MP architecture and the API package.
Installing and Verifying JDBC/MP	Describes where to find information about the installation requirements and explains how to verify the JDBC/MP the installation.
Accessing SQL Databases with SQL/MP	Explains how to access SQL databases with SQL/MP from the NonStop Server for Java by using JDBC/MP 3.0.
JDBC/MP Compliance	Explains how JDBC/MP differs from the Sun Microsystems JDBC standard because of limitations of SQL/MP and the JDBC/MP driver.
JDBC Trace Facility	Explains how to use the JDBC trace facility
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 **HP NonStop Technical Library (NTL)** 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 these documents:

- HP JDBC/MP Driver for NonStop SQL/MP API Reference for H10 (javadoc information about the JDBC/MP APIs are available on the NonStop Technical Library)
- NonStop System Computing Documents
- Sun Microsystems Documents

NonStop System Computing Documents

These NonStop system computing documents are available on the NonStop Technical Library (NTL).

- Additional Java-Oriented Products. These documenents are available in the Java category under Independent Products in the NonStop Technical Library (http://www.hp.com/go/ntl).
  - NonStop Server for Java Programmer's Reference

    This documentation describes NonStop Server for Java, a Java environment that supports compact, concurrent, dynamic, and portable programs for the enterprise server. Also, describes how to use the JDBC Trace Facility that can trace the entry point of all the JDBC methods called from the Java applications.
  - NonStop Server for Java Tools Reference Pages

    This documentation consists of a title page, a table of contents, and the Tools Reference Pages for NonStop Server for Java.
  - NonStop Server for Java API 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 with NonStop Server for Java.
  - HP NonStop JDBC Server Programmer's Reference

    The HP NonStop JDBC server documentation describes the JDBC driver that lets Java programmers to remotely (on a PC or UNIX machine) develop and debug applications that is deployed on NonStop servers to access NonStop SQL databases with SQL/MP or SQL/MX.
  - Using Oracle JDeveloper With NonStop Enterprise Application Server

    This paper describes the considerations for using Oracle JDeveloper to develop servlets and Enterprise Java Beans applications in the NonStop Enterprise Application Server environment.
  - JDBC/MX Driver for NonStop SQL/MX Programmer's Reference for H50

    This document describes how to use the JDBC/MX Driver for NonStop SQL/MX on NonStop servers. JDBC/MX provides NonStop Server for Java applications with JDBC access to HP 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 (JVM)**.

- NonStop ODBC Server Documents

  The **HP NonStop ODBC Server** enables programs written for the Microsoft Open Database Connectivity (ODBC) product to access SQL/MP databases. If the ODBC Server is installed on a NonStop server, and an ODBC driver is installed on a **client**, the client can remotely access an SQL/MP database by means of a JDBC program that uses a JDBC-ODBC bridge driver.

  - ODBC Server Installation and Management Manual

    Explains the installation, configuration, management, and tuning of the ODBC Server and its components.

  - ODBC Server Reference Manual

    Contains reference information for the NonStop ODBC Server, describes the NonStop ODBC or SQL/MP Server features and the statements that the NonStop ODBC Server supports, and explains how the NonStop ODBC Server accommodates the differences between the ODBC or SQL/MP Server and NonStop SQL/MP.

- SQL/MP Documents

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

  - Introduction to NonStop SQL/MP

    Provides an overview of the SQL/MP product.

  - SQL/MP Glossary

    Explains the terminology used in the SQL/MP documentation.

  - SQL/MP Installation and Management Guide

    Explains how to plan, install, create, and manage an SQL/MP database; describes the syntax of installation and management commands; and describes SQL/MP **catalogs** and file structures.

  - SQL/MP Messages Manual

    Explains SQL/MP messages for the conversational interface, the **application programming interface (API)**, and the utilities.

  - SQL/MP Programming Manual for COBOL

    Describes the SQL/MP programmatic interface for ANSI COBOL. Also describes embedded SQL statements used in COBOL applications.

  - SQL/MP Programming Manual for C

    Describes the SQL/MP programmatic interface for ANSI C. Also describes embedded SQL statements used in C applications.

  - SQL/MP Query Guide

    Explains how to retrieve and modify data from an SQL/MP database and how to analyze and improve query performance.

  - SQL/MP Reference Manual

    Explains the SQL/MP language elements, expressions, functions, and statements.

  - SQL/MP Report Writer Guide

    Explains how to use report writer commands and **SQLCI** options to design and produce reports.

  - SQL/MP Version Management Guide

    Explains the rules that govern version management for the NonStop SQL software, catalogs, objects, messages, programs, and data structures.

- 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

These documents were available on Sun Microsystems Web sites when the JDBC Driver for SQL/MP was released, but HP cannot guarantee their continuing availability. If a link to a Sun Microsystems document fails, use the Sun Microsystems Web site search engine (`http://java.sun.com/share/search.htm`) or contact the Sun Microsystems webmaster (`http://java.sun.com/cgi-bin/feedback.pl`).

- 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`)

Note: Sun Microsystems Java 2 Platform Standard Edition 5.0 documentation is provided under the Java documentation from the Sun Microsystems Web sites. For more information, see the NonStop Server for Java Programmer's Reference.

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 this 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 disc

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

JVM. Java Virtual Machine

JCK. Java Conformance Kit

JFC. Java Foundation Classes

JDBC. Java Database Connectivity

JDBC/MP. JDBC Driver for SQL/MP

JDBC/MX. JDBC Driver for SQL/MX

JDK. Java Development Kit

JNDI. Java Naming and Directory Interface

JNI. Java Native Interface

JRE. Java Run-time Environment

LAN. local area network

NonStop TS/MP. NonStop Transaction Services/MP

NTL. NonStop Technical Library

OSS. Open System Services

POSIX. Portable Operating System Interface

RISC. reduced instruction-set computing

RVU. release version update

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

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

Introduction to JDBC/MP Driver

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

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

The JDBC/MP driver and HP NonStop Server for Java is a Java environment that supports compact, **concurrent**, dynamic, **portable** programs for the enterprise server. The JDBC/MP driver requires NonStop Server for Java and SQL/MP, which both require the HP NonStop Open System Services (OSS) environment. The NonStop Server for Java 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 Driver Types
- JDBC/MP Architecture
- JDBC/MP API Packages

JDBC Driver Types

This table describes the types of JDBC drivers:

Type		Written in Java	Description
No.	Name		
1	JDBC-to-ODBC bridge	Possibly	Uses a JDBC-to-ODBC bridge to translate JDBC calls to **ODBC** calls, which are handled by an ODBC driver. This requires an ODBC driver on the client side and an ODBC server on the server side.

2	Native protocol	Partially	Translates JDBC calls into the native API for a particular database. The driver must run on a machine where this API is available. (`sqlmp` is a Type 2 driver.)
3	Pure Java	Entirely	Translates JDBC calls to a database-independent protocol for transmission over the network. Server middleware then translates this protocol into a database's native protocol.
4	Native protocol, Pure Java	Entirely	Translates JDBC calls directly to the network protocol of a particular database. Therefore, a JDBC program on the client can directly call to the database server.

Note: HP does not support third-party JDBC drivers for NonStop systems, but any JDBC driver that is written entirely in Java (that is, a Type 3 or a Type 4 driver) can run on top of the NonStop Server for Java. To use such a driver to connect your Java program to a third-party database, see the third-party instructions for that driver.

JDBC/MP Architecture

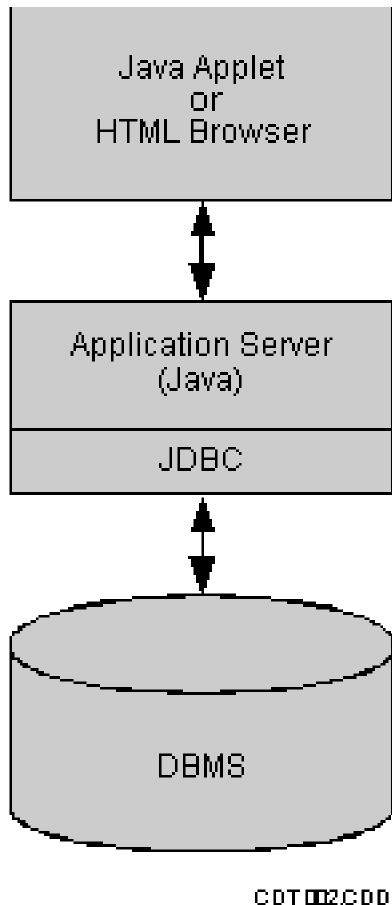
The JDBC/MP driver is a Type 2 driver; it employs proprietary, native APIs to use SQL/MP to access NonStop SQL databases. The native API of SQL/MP cannot be called from client systems. Therefore, the JDBC/MP driver runs on NonStop servers only.

The JDBC/MP 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 the results to the user. This use of the middle tier

- Allows you to maintain control over access and the kinds of updates that can be made to corporate data
- Simplifies the deployment of applications
- Can provide performance advantages

This figure illustrates a three-tier architecture for database access:

Architecture of the JDBC Driver for SQL/MP



JDBC/MP API Packages

The JDBC/MP API packages are shipped with the JDBC Driver for SQL/MP software.

The `java.sql` and `javax.sql` packages are included as part of Java 2, Standard Edition (J2SE) 1.5.0 and, therefore, are available with the core APIs delivered with the NonStop Server for Java product.

Limitation

In TNS/E, applications cannot have their own embedded SQL code. An application using the JDBC/MP driver cannot have any other object with embedded SQL code linked to it.

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

Installing and Verifying JDBC/MP

This section explains these subjects:

- JDBC/MP Driver Requirements
- Verifying the JDBC/MP Driver

JDBC/MP Driver Requirements

Before using the JDBC Driver for SQL/MP, the product components must be installed and the **Java Virtual Machine (JVM)** must be compiled by the SQL compiler. In TNS/E, JVM must not be recompiled by the SQL compiler.

For information about these tasks, see the softdoc or the README file that accompanies the NonStop Server for Java product.

> **Note:** The JDBC driver for SQL/MP has the same hardware requirement as NonStop Server for Java, which can run on all Integrity NonStop NS-series servers. For Java product for current requirements, see the softdoc for NonStop server.

Verifying the JDBC/MP Driver

To verify the JDBC/MP driver after installation, run the Sample SQL/MP Program.

> **Note for TNS/E:** The libsqlmp.lib file no longer exists in JDBC/MP 3.0. The libsqlmp.lib file is replaced by the libsqlmp.so file that is copied in the $JDBCHOME/T1277H10/current/lib directory. JDBC installation dependency on the java_public_lib directory is now removed. The SQL compile operation is now executed on the libsqlmp.so file when installing JDBC/MP.

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

Accessing SQL Databases with SQL/MP

Java programs interact with NonStop SQL databases using **NonStop SQL/MP** and the **Java Database Connectivity (JDBC) API**.

The NonStop Server for Java and the JDBC Driver for SQL/MP include:

- The JDBC 3.0 API
- The JDBC **`DriverManager`** in the Java Development Kit (JDK) 1.5.0
- These Type 2 **drivers**, which behave identically except where stated otherwise:
  - The standard driver, `sqlmp`, which manages transactions as described in the JDBC API documentation provided by Sun Microsystems.
  - The transaction-aware driver, `sqlmptx` that lets you to define transactions outside of JDBC using the methods described in the NonStop Server for Java Programmer's Reference under "Transactions".

This section describes these subjects:

- Setting a Column in an SQL Table to Null
- Transactions and the JDBC Driver for SQL/MP
- JDBC/MP Driver Requirements
- Verifying the JDBC/MP Driver
- JDBC Driver Types
- JDBC `DriverManager`
- Loading a Driver
- Connecting a Program to a Database
- Passing SQL/MP Statements to a Database
- Compatible Java and SQL/MP Data Types
- Referring to Database Objects
- Connection Pooling and the Application Server
- Connection Pooling in a Basic `DataSource` Implementation
- Statement Pooling
- JDBC/MP Connection-Pooling and Statement-Pooling Properties
- Troubleshooting
- Sample SQL/MP Program

Setting a Column in an SQL Table to Null

If you want to set a column in a SQL/MP table to null, you must supply a null indicator when you create the update statement for the table. For example, if a table is defined as:

```
create table foo( empnum smallint unsigned not null,
   lastname character(20) not null,
   deptnum smallint unsigned,
   primary key (empnum) ) organization key sequenced;
```

and you want to insert a new `foo` row and to set `deptnum` to null (which is permissible because it is not defined as `not null`), you must include a null indicator in the insert statement:

```
PreparedStatement stmt;
stmt = conn.prepareStatement("insert into foo values(?,?,??)");
```

Note that the last two question marks are not separated by commas; the first question mark of the pair indicates that a value can be set for this column (the `deptnum` column, as these question marks appear in the third position) and the second is the null indicator, showing that the column can be set to null. If the second question mark is not present, the column could not be set to null.

Transactions and the JDBC Driver for SQL/MP

The **Transaction Management Facility (TMF)** is the system entity that manages transactions. You cannot make procedure calls to TMF directly from any Java application. You must use either the `com.tandem.tmf.Current` class (described in the NonStop Server for Java API Reference) or the Java Transaction API (JTA) to manage transactions.

Autocommit Mode and the Standard Driver

In autocommit mode, the standard driver automatically manages transactions for you. You do not need to explicitly begin and end transactions. In fact, if you have used the Current class method or JTA to begin a transaction before calling the standard driver, the driver throws an exception.

Each statement forms a separate transaction that is committed when the statement is closed or the next time an `executeQuery`, `executeUpdate`, or `execute` method is called. When a `ResultSet` is returned, the standard driver commits the transaction when the last row of the `ResultSet` is retrieved, when the `ResultSet` is closed, or when the statement is closed (which automatically closes the `ResultSet`). If you invoke another `executeQuery`, `executeUpdate`, or `execute` method when a `ResultSet` is open, the driver closes the result set andcommits the transaction. The driver then begins a new transaction for the newly called method.

Autocommit mode is the default for the standard SQL/MP driver.

Nonautocommit Mode and the Standard Driver

In nonautocommit mode, you must manually commit or roll back a transaction. In this mode, a transaction is defined as all the statements preceding an invocation of `commit()` or `rollback()`. Both `commit()` and `rollback()` are methods in the `Connection` class.

Autocommit Mode and the Transaction-Aware Driver

Requesting autocommit mode (that is, specifying `setAutoCommit(true)`) with the transaction-aware driver causes an `SQLException`.

Nonautocommit Mode and the Transaction-Aware Driver

Nonautocommit mode is the default for the transaction-aware driver. You must manage all transactions using `com.tandem.tmf.Current` or the JTA.

JDBC `DriverManager`

The JDBC `DriverManager` maintains a list of available JDBC drivers. When a program calls the `getConnection` method, the JDBC `DriverManager` searches this list and connects the calling program to the appropriate JDBC driver.

Loading a Driver

To load a JDBC driver, load its JDBC driver class into the Java Virtual Machine (JVM) in one of these ways:

- Specify the JDBC driver class on the `java` command line.
- Add the JDBC driver class to the `jdbc.drivers` property.
- Load the JDBC driver class directly into the JVM.

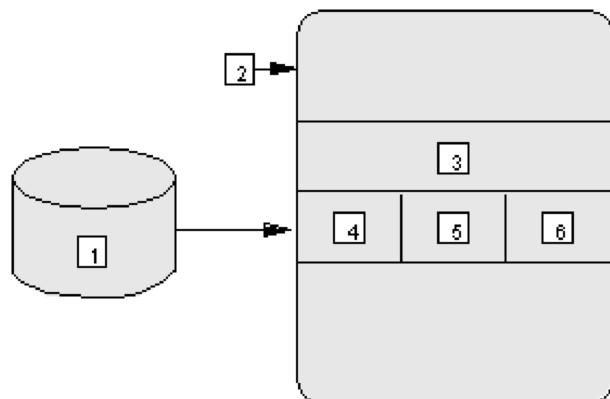
The first two of the preceding ways enable you to change drivers without modifying your program code; the third way does not.

The standard SQL/MP driver and the transaction-aware driver share the same code, so you use the same syntax to load either of them.

Note: The SQL/MP driver file, `com/tandem/sqlmp/SQLMPDriver.class`, is stored in the JAR file `sqlmp.jar`, which contains HP additions to the JVM. The SQL/MP driver file belongs to the JDBC driver class `com.tandem.sqlmp.SQLMPDriver`.

This figure shows a JDBC driver class being loaded into the JVM, which already contains the JDBC `DriverManager` and three databases.

Loading a JDBC Driver Class Into the JVM



Legend

1. JDBC driver class
2. Java Virtual Machine (JVM)
3. JDBC Driver Manager
4. SQL/MP database
5. Oracle database
6. JavaDB database

Specifying the JDBC Driver Class on the `java` Command Line

To specify the JDBC driver class on the `java` command line, use the `-Djdbc.drivers` option. For example, this `java` command loads the SQL/MP driver, whose class name is `com.tandem.sqlmp.SQLMPDriver`:

```
java -Djdbc.drivers=com.tandem.sqlmp.SQLMPDriver
```

To specify multiple JDBC driver classes on the `java` command line, separate them with semicolons. For example, this `java` command loads the SQL/MP driver and a driver whose class name is `com.javaco.purejavajdbc.Driver`:

```
java -Djdbc.drivers=com.tandem.sqlmp.SQLMPDriver;com.javaco.purejavajdbc.Driver
```

Adding the JDBC Driver Class to the `jdbc.drivers` Property

To add the JDBC driver class to the `jdbc.drivers` property:

1. Create a new `Properties` object using this command:
   `Properties prop=new Properties();`
2. Add the JDBC driver class to the `jdbc.drivers` property using this command:
   `prop.put("jdbc.drivers","`*`driver_class_name`*`[;`*`driver_class_name`*`]");`
3. Set the system properties using this command:
   `System.setProperties(prop);`

The first call to a JDBC `DriverManager` method loads the driver or drivers.

This code adds the SQL/MP driver and the JDBC driver whose class name is `com.javaco.purejavajdbc.Driver` to the `jdbc.drivers` property:

```
Properties prop=new Properties();
prop.put("jdbc.drivers","com.tandem.sqlmp.SQLMPDriver;"+
"com.javaco.purejavajdbc.Driver");
System.setProperties(prop);
```

Loading the JDBC Driver Class Directly Into the JVM

To load the JDBC driver class directly into the JVM, use the `Class.forName()` method.

This code loads the SQL/MP driver and the JDBC driver whose class name is `com.javaco.purejavajdbc.Driver` directly into the JVM:

```
Class.forName("com.tandem.sqlmp.SQLMPDriver");
Class.forName("com.javaco.purejavajdbc.Driver");
```

Connecting a Program to a Database

After a driver is loaded, you can use it to connect a Java program to a database by passing the URL of the database to the `DriverManager.getConnection()` method.

You do not need to specify a username and a password to connect to an SQL/MP database.

Standard Driver Example

In a Java program, this code uses the standard driver, `sqlmp`, to create a connection (`myConnection`) between the program and the local SQL/MP database:

```
import java.sql.*;
...
Connection myConnection = DriverManager.getConnection("jdbc:sqlmp:");
```

Note: You must specify the URL for the driver exactly as it is shown in the following examples.

Transaction-Aware Driver Example

In a Java program, this code uses the transaction-aware driver, `sqlmptx`, to create a connection (`myConnection`) between the program and the local SQL/MP database:

```
import java.sql.*;
...
Connection myConnection = DriverManager.getConnection("jdbc:sqlmptx:");
```

Passing SQL/MP Statements to a Database

When there is a connection between the Java program and an SQL/MP database, you can create an SQL statement and pass it to the database that returns a result. To create an SQL statement, use the connection's `createStatement()` method. To pass the statement to the database, use the statement's `executeQuery()` method.

Each SQL statement runs as a waited operation, so executing an SQL statement within a thread blocks the JVM.

Note: If your SQL query includes the name of an SQL/MP table, fully qualify the table name in the query. Otherwise, the volume and subvolume of the table name defaults to the current volume and subvolume.

In this example, the result set is not cached, so each call to `next()` retrieves more information from the database:

```
import java.sql.*;
...
// Create a statement:

Statement myStatement = myConnection.createStatement();

// Pass the string argument directly to the database;
// results are returned in the ResultSet object, r:

ResultSet r = myStatement.executeQuery("SELECT * from $DATA1.DBASE.EMPL");
```

```
// Retrieve successive rows from the result set and
// print the first column of each row, which is a String:

while (r.next())
        System.out.println("Column 1:" + r.getString(1));
```

For information about how database transactions are delimited, committed, and rolled back when using JDBC, see [Transactions and the JDBC Driver for SQL/MP](#).

Compatible Java and SQL/MP Data Types

This table shows which Java data types can hold which SQL/MP data types. Use this table to determine which **`get()` method** to call to read a particular type of data from a result set or which **`set()` method** to call to define the data type of a prepared statement's input parameter. For example, if you select CHAR data from a database, use `getString()` to retrieve the CHAR data from the result set.

Compatible Java and SQL/MP Data Types

		SQL Data Types													
		SMALLINT	INTEGER	LARGEINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	CHAR	VARCHAR	DATE	TIME	TIMESTAMP	DATETIME
Java Data Types	`Byte`	**R**	s	s	s	s	s	s	s	s	s	n	n	n	n
	`Short`	**R**	s	s	s	s	s	s	s	s	s	n	n	n	n
	`Int`	s	**R**	s	s	s	s	s	s	s	s	n	n	n	n
	`Long`	s	s	**R**	s	s	s	s	s	s	s	n	n	n	n
	`Float`	s	s	s	**R**	s	s	s	s	s	s	n	n	n	n
	`Double`	s	s	s	s	s	**R**	s	s	s	s	n	n	n	n
	`BigDecimal`	s	s	s	s	s	s	**R**	s	s	s	n	n	n	n
	`BigInteger`	s	s	s	s	s	s	s	s	**R**	s	n	n	n	n
	`Boolean`	**R**	s	s	s	s	s	s	s	s	s	n	n	n	n
	`String`	s	s	s	s	s	s	s	s	**R**	**R**	n	n	n	n
	`Bytes`	n	n	n	n	n	n	n	n	n	n	n	n	n	n
	`Date`	n	n	n	n	n	n	n	n	n	n	**R**	n	n	s
	`Time`	n	n	n	n	n	n	n	n	n	n	n	**R**	n	s
	`Timestamp`	n	n	n	n	n	n	n	n	n	n	n	n	**R**	**R**

AsciiStream	n	n	n	n	n	n	s	n	**R**	**R**	n	n	n	n
UnicodeStream	n	n	n	n	n	n	n	n	n	n	n	n	n	n
BinaryStream	n	n	n	n	n	n	n	n	n	n	n	n	n	n
Object	s	s	s	s	s	s	s	s	s	s	n	n	n	n

Legend

R	Recommended
s	Supported
n	Not supported for current release

Referring to Database Objects

By default, the SQL/MP driver uses Guardian file names for SQL/MP database objects, as this table shows.

Default SQL/MP Database Object Names

Database Object	**Default Name ***	**Examples**
Catalog	`[\`***`node`***`.][$`*`volume`*`.]`*`subvol`*	`$sqldata.sales` `\tokyo.$disk02.sqlcat`

Index	`[\`*`node`*`.][[$`*`volume`*`.]`*`subvol`*`.]`*`fileID`*	`emptab`
		`$sqldisk.fy02.xsalary`
		`\newyork.$sqldisk.fy02.salary`

Table		
* If you omit *node*, *volume*, or *subvol*, the SQL/MP driver uses your current node, volume, or subvolume to fully qualify the database object name. For more information, see the SQL/MP Reference Manual.		

If you are trying to obtain information about the SQL/MP system catalog and that catalog does not reside in the default location (`$system.sql`), you must specify this DEFINE before you start your Java application:

```
set defmode on
add_define =SQL_SYSTEMCAT_LOC class=map file=\$vol.subvol.catalogs
```

In your Java source code, you can always refer to SQL/MP database objects by their default Guardian file names. In some cases, you can also refer to them by names that you choose (aliases). Aliases are useful when you port an application that can be used on SQL databases having table names that do not conform to the table names used with NonStop SQL/MP.

These subsections explain:

- When You can Use Aliases
- How to Create and Use Aliases
- Which SQL Statements Support Aliases

When You Can Use Aliases

You can use aliases for these SQL/MP database objects:

- Catalogs
- Collations
- Tables (including table qualifiers for columns)
- Views

If an SQL/MP database object is not in the preceding list, you must refer to it by its default Guardian file name.

Creating and Using Aliases

To create and use aliases:

1. Create a `java.util.Properties` object.
2. Put each alias-Guardian file-name pair (called a substitution property) in the `java.util.Properties` object. Prefix each alias with a plus sign (+). The individual `put()` calls for this step can be either in your program.

   Note: Previous versions of the NonStop Server for Java used an equal sign (=) instead of a plus sign. For compatibility, the JDBC Driver for SQL/MP supports the equal sign, but HP recommends the plus sign for new programs.

   ---
3. Connect your program to the database, passing the `java.util.Properties` object to `DriverManager.getConnection()`.
4. In SQL statements that require a property to alias an SQL name, use the aliases instead of their Guardian file names. Do not include the plus sign (+) prefixes in the SQL statements.

Specifying Properties Using the `put()` Method

This example creates and uses two aliases: `longname1`, for the Guardian file name `$VOL1.MYVOL.MYTAB`, and `longname2`, for the Guardian file name `$VOL1.MYVOL.MYTAB2`. The `put()` calls that create the aliases are in the program, not in a separate file.

```
// Create a java.util.Properties object named prop
java.util.Properties prop=new java.util.Properties();

// Create the alias longname1 for the Guardian filename $VOL1.MYVOL.MYTAB
prop.put("+longname1", "$VOL1.MYVOL.MYTAB");

// Create the alias longname2 for the Guardian filename $VOL1.MYVOL.MYTAB2
prop.put("+longname2", "$VOL1.MYVOL.MYTAB2");

// Connect the program to the database, passing prop
conn = DriverManager.getConnection("jdbc:sqlmp:", prop);

// Create an SQL statement
stmt = conn.createStatement();

// Use the aliases instead of their Guardian filenames
rslt = stmt.executeQuery("select * from longname1 where col1='pickme'");
rslt = stmt.executeQuery("select * from longname2 where col1='pickme'");
```

Specifying Properties in a Properties File

This program has the same effect as the preceding program, except that the properties are in the file `jdbc-properties`. The file `jdbc-properties` contains these lines:

```
+longname1,$VOL1.MYVOL.MYTAB
+longname2,$VOL1.MYVOL.MYTAB2
```

To use this file within your program, you can either :

(a) create a properties object and load the file within your program or

(b) provide a System property called `jdbcPropfile` to tell JDBC the name of your properties file.

Loading a Properties File Programmatically

This sample code loads a properties file:

```
// Create a java.util.Properties object named prop
java.util.Properties prop=new java.util.Properties();

// Load the file that creates the aliases
prop.load("jdbc-properties");

// Connect the program to the database, passing prop
conn = DriverManager.getConnection("jdbc:sqlmp:", prop);

// Create an SQL statement
stmt = conn.createStatement();

// Use the aliases instead of their Guardian filenames
rslt = stmt.executeQuery("select * from longname1 where col1='pickme'");
rslt = stmt.executeQuery("select * from longname2 where col1='pickme'");
```

Loading a Properties File Using a System Property

The JDBC driver (optionally) looks for a System property called `jdbcPropfile`. If you set this System property to the name of a file containing alias properties, the JDBC driver automatically loads that file.

This sample code uses that System property:

```
//Start your program specifying the System property jdbcPropfile
java -DjdbcPropfile=/h/mydir/jdbc-properties
   -Djdbc.drivers=com.tandem.sqlmp.SQLMPDriver test1

//Create an SQL statement
stmt = conn.createStatement();

// Use the aliases instead of their Guardian filenames
rslt = stmt.executeQuery("select * from longname1 where col1='pickme'");
rslt = stmt.executeQuery("select * from longname2 where col1='pickme'");
```

Which SQL Statements Support Aliases

These SQL statements support alias substitution:

- In the Data Control Language (DCL):

- `CONTROL TABLE`
- `LOCK TABLE`
- `UNLOCK TABLE`

- In the Data Definition Language (DDL):
  - `ALTER CATALOG`
  - `ALTER COLLATION`
  - `ALTER INDEX`
  - `ALTER PROGRAM`
  - `ALTER TABLE`
  - `ALTER VIEW`
  - `COMMENT`
  - `CREATE CATALOG`
  - `CREATE COLLATION`
  - `CREATE CONSTRAINT`
  - `CREATE INDEX`
  - `CREATE SYSTEM CATALOG`
  - `CREATE TABLE`
  - `CREATE VIEW`
  - `DROP`
  - `HELP TEXT`
  - `UPDATE STATISTICS`
- In the Data Manipulation Language (DML):
  - `DECLARE CURSOR`
  - `DELETE`
  - `INSERT`
  - `SELECT`
  - `UPDATE`

Substitution also works on subqueries.

The alias substitution mechanism does not substitute Guardian file names and Guardian DEFINEs. If your SQL query uses a DEFINE, you cannot use this alias substitution mechanism.

In this example, no alias substitution is possible:

```
select =jobcode, =deptnum, =first_name, =last_name from persnl.employee
where =jobcode > 500 and =deptnum <= 3000 order by =jobcode browse access
```

In this example of a nested statement, the candidates for substitution are printed in **`bold type`**:

```
update employee set salary = salary*1.1
where deptnum in (select deptnum from dept where location = "San Francisco")
```

In this example, the candidates for substitution are printed in **`bold type`**. Candidates include the correlation names, `o` and `p`, because they can be user-defined names that can be replaced by Guardian table names (such as `parts`).

```
select ordernum,
sum(qty_ordered*price) from parts p, odetail o
where o.partnum = parts.partnum
and ordernum in (select ordernum from orders o, customer c
where o.custnum = c.custnum and state = "CALIFORNIA" group by ordernum)
```

Note: Mapping `parts` to a fully qualified Guardian name (such as `$sys1.vol.subvol.parts`) results in an invalid SQL statement because the substitution procedure also replaces the second occurrence of `parts`.

To check the effect of alias substitution during development, use the `SQLMPConnection.nativeSQL(String str)` function. This function returns the SQL statement that results from performing alias substitutions on the SQL statement *str*.

Connection Pooling and the 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
- Registering the `ConnectionPoolDataSource` and DataSource with a JNDI-Based Name Service

Guidelines for Implementing an Application Server to Use Connection Pooling

- The application server maintains a cache of the `PooledConnection` objects created by using the `ConnectionPoolDataSource` interface. When the client requests a connection object, the application looks for the proper `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/MP driver notifies the listener object with a `connectionClosed` event when the application is finished using the `PooledConnection` object. Then, the application server can reuse this `PooledConnection` object for future requests.

  With the JDBC/MP driver, a pooled connection cannot fail to be initialized; therefore, the `connectionErrorOccurred` event cannot occur.

- The application server manages the connection pool by using the `SQLMPConnectionPoolDataSource`, which implements the `ConnectionPoolDataSource` interface. Use the getter and setter methods, provided by JDBC/MP, to set the connection-pool configuration properties listed in the table of Standard `ConnectionPoolDataSource` Object Properties.

- In an environment that uses an application server, deployment of an `SQLMPConnectionPoolDataSource` object requires that both an application-visible `DataSource` object and the underlying `SQLMPConnectionPoolDataSource` object be registered with a JNDI-based naming service. For more information, see Registering the `SQLMPConnectionPoolDataSource` and `DataSource` Objects with a JNDI-Based Naming Service.

Standard `ConnectionPoolDataSource` Object Properties

Property Name	Type	Default Value*	Description

`maxStatements`	`int`	0	The maximum number of `PreparedStatement` objects that the pool should cache. A value of 0 (zero) disables statement pooling.
`minPoolSize`	`int`	0	The number of physical connections the pool should keep available at all times.
`maxPoolSize`	`int`	0	The maximum number of physical connections that the pool should contain. A value of 0 (zero) indicates no maximum size.
`initialPoolSize`	`int`	0	The number of physical connections the pool should keep workable at all times. A value of 0 (zero) indicates that connections should be created as needed.
`maxIdleTime`	`int`	0	The number of seconds that a physical connection should remain unused in the pool before the connection is closed.
`propertyCycle`	`int`	0	The interval, in seconds, that the pool should wait before enforcing the current policy defined by the values of the previous connection-pool properties.
* **Note:** The application server defines the semantics of how these properties are used.			

Registering the `ConnectionPoolDataSource` and `DataSource` Objects with a JNDI-Based Naming Service

In an environment that uses an application server, deployment of an `SQLMPConnectionPoolDataSource` object requires that both an application-visible `DataSource` object and the underlying `SQLMPConnectionPoolDataSource` object be registered with a JNDI-based naming service.

The first step is to deploy the `SQLMPConnectionPoolDataSource` implementation, as is done in this code example:

```
com.tandem.sqlmp.SQLMPConnectionPoolDataSource cpds =
    new com.tandem.sqlmp.SQLMPConnectionPoolDataSource();

cpds.setDatabaseName("booklist");
cpds.setMaxPoolSize(40);
cpds.setMaxStatements(100);
cpds.setTransactionMode("INTERNAL");
   ...

// Register the SQLMPConnectionPoolDataSource with JNDI,
// using the logical name "jdbc/pool/bookserver_pool"
```

```
Context ctx = new InitialContext();
ctx.bind("jdbc/pool/bookserver_pool", cpds);
```

After this step is complete, the SQLMPConnectionPoolDataSource implementation is available as a foundation for the client-visible application-server DataSource implementation. The user deploys the application server DataSource implementation such that it references the SQLMPConnectionPoolDataSource implementation, as shown:

```
// Three Star Server DataSource implements the DataSource
// interface. Create an instance and set properties.


com.threeStar.appserver.PooledDataSource ds =
new com.threeStar.appserver.PooledDataSource();
ds.setDescription("Datasource with connection pooling");

// Reference the registered SQLMPConnectionPoolDataSource
ds.setDataSourceName("jdbc/pool/bookserver_pool");

// Register the DataSource implementation with JNDI, by using the
// logical name "jdbc/bookserver".

Context ctx = new InitialContext();
ctx.bind("jdbc/bookserver", Ds);
```

For more information about connection pooling and the application server, see the JDBC 3.0 specification.

Connection Pooling in a Basic DataSource Implementation

For your NonStop Server for Java application to enable connection pooling, use the JDBC basic DataSource interface. The JDBC/MP driver implements the DataSource interface with a class called SQLMPDataSource.

You can use SQLMPDataSource implementation to provide:

- Both connection and statement pooling
- Only connection pooling
- Only statement pooling
- A connection with neither connection pooling nor statement pooling

The connection pooling implementation is described in these topics:

- Basic DataSource Object Properties
- Requesting Connection and Statement Pooling
- Determining of the Pool is Large Enough
- Using an SQLMPDataSource with an SQLMPConnectionPoolDataSource

Basic DataSource Object Properties

The basic DataSource class includes a set of properties that control connection pooling.

To set the property values for your application, you use the SQLMPDataSource class, which provides getter and setter methods. The properties, listed below, are described under JDBC/MP Connection-Pooling and Statement-Pooling Properties.

- maxStatements
- maxPoolSize
- minPoolSize
- poolLogging
- TransactionMode
- jdbcPropfile

Note: The properties defined for the `SQLMPDataSource` class and the `SQLMPConnectionPoolDataSource` class are similar but the default values are slightly different.

Requesting Connection and Statement Pooling

Before deploying an `SQLMPDataSource`, set the properties that request both connection and statement pooling. This code example shows these properties being set:

```
SQLMPDataSource ds = new SQLMPDataSource();
ds.setMaxPoolSize(0);
ds.setMaxStatements(100);
```

The values that you use when setting these properties depend on your application. Notice that the `minPoolSize` property was not set. Because creating a connection is not an expensive operation for the JDBC/MP driver, only the `maxPoolSize` property needs to be set.

Determining if the Pool is Large Enough

If the values specified for the connection pool are not large enough for a specific application, the JDBC/MP driver throws an `SQLException` when a user application tries to obtain a connection and the pool is full. If this error occurs, consider setting `maxPoolSize` to 0 (zero) to make the size of the connection pool unlimited.

If the values specified for the prepared statement pool are not large enough for a specific application, the JDBC/MP driver still returns a `PreparedStatement`. In this case, the JDBC/MP driver might not pool all the prepared statements used by the application when the application in a steady state.

To determine if the pool is large enough for an application, `SQLMPDataSource` provides the standard methods `setLogWriter` and `getLogWriter`, which you can use with the `setPoolLogging` method (which is not according to the JDBC 3.0 standard). To get the status of the connection pool or statement pool, deploy the `SQLMPDataSource` with `setPoolLogging` set to true. An application uses the `setLogWriter` method to provide an output writer and runs the application normally. The statement pool then writes information about the pool to the output writer. For example:

```
// When deploying the data source
SQLMPDataSource sds = new SQLMPDataSource();
sds.setPoolLogging(true);


//In application code
DataSource ds;


//After obtaining the data source
PrintWriter writer = new PrintWriter(System.out,true);
ds.setLogWriter(writer);
ds.setPoolLogging(true);
```

When `poolLogging` is set to true, the pool writes information about the number of connections or statements currently

in the pool. The pool also writes a message whenever the statement pool is full and a `PreparedStatement` is requested that is not in the free pool. In this case, the pool removes the least recently used statement from the free pool and creates the requested `PreparedStatement`. If these messages occur frequently, consider increasing the size of the statement pool.

Note: Because the pool can write a large amount of data to the output writer when `setPoolLogging` is true, do not turn on `poolLogging` in a production environment unless you need to debug issues that involve the pool size.

Using an `SQLMPDataSource` with an `SQLMPConnectionPoolDataSource`

An `SQLMPDataSource` object can be deployed that refers to an already deployed `SQLMPConnectionPoolDataSource` object. To do this deployment, set the `dataSourceName` property in the `SQLMPDataSource` object to the previously registered `SQLMPConnectionPoolDataSource` object.

Note: When an `SQLMPDataSource` object refers to an `SQLMPConnectionPoolDataSource` object, the properties in the `SQLMPConnectionPoolDataSource` object are used as default values for the `SQLMPDataSource` object. This means that any values that were not set in the `SQLMPDataSource` object when it was deployed use the values set in the `SQLMPConnectionPoolDataSource` object.

Statement Pooling

Statement pooling allows applications to reuse `PreparedStatement` objects in much the same way connection objects can be reused in a connection-pooling environment.

To enable statement pooling set the [`maxStatements`] property of either the `DriverManager` class or basic `DataSource` implementation to a value greater than zero (0).

Note these JDBC/MP driver-implementation details when using and troubleshooting statement pooling:

- The properties passed through the `Properties` parameter have precedence over the command-line properties.
- Statement pooling is completely transparent to the application. Pooled statements are not associated with a specific connection but are pooled across the application.
- When using statement pooling, the application should explicitly close the prepared statement using the `PreparedStatement.close` method. `PreparedStatement` objects that are not in scope are not reused unless the application explicitly closes them.
- When an application requests a `PreparedStatement` object, the pool tries to find a proper object in the free pool. For JDBC/MP, some of the factors that determine whether an object is proper are the SQL string, catalog, `resultSet` type, and `resultSet` concurrency. If a suitable proper exists, the pool returns the object to the application. If the pool does not find a proper object, the JDBC/MP driver creates a new `PreparedStatement` object. An application can assure that a `PreparedStatement` object is returned to the free pool by calling either of these:

  ```
  PreparedStatement.close

  Connection.close
  ```
- When the number of statements in the pool reaches the `maxStatements` limit and a statement is requested that is not found in the free pool, the pool removes the least-recently used statement from the free pool before a creating a new `PreparedStatement` object . The statement-pooling component assumes that any SQL/MP control statements in effect at the time of use or reuse are the same as those in effect at the time of compilation. However, if the SQL/MP control statements are not the same, reuse of a `PreparedStatement` object might result in unexpected behavior. The SQL/MP engine 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 on a table was eliminated by a DDL or SQL utility operation.

Although these conditions only cause the `PreparedsStatement` to be recompiled once, avoid them so you can gain maximum use of statement pooling. For detailed description on automatic recompilation, see the SQL/MP Programming Manual for C.

For more information on statement pooling, see the Sun Microsystems JDBC 3.0 Specification.

JDBC/MP Connection-Pooling and Statement-Pooling Properties

The JDBC/MP driver uses properties to determine the connection-pooling and statement-pooling implementation for a user application. These properties are:

- `maxStatements` Property
- `minPoolSize` Property
- `maxPoolSize` Property
- `poolLogging` Property
- `TransactionMode` Property
- `jdbcPropfile` Property

For information on using these features, see Connection Pooling in a Basic DataSource Implementation and Statement Pooling.

`maxStatements` Property

Sets the total number of `PreparedStatement` objects that the connection pool should cache. Statements are pooled across the application, not just within a connection. Specify the `maxStatements` property as:

`int`

The integer can be 0 through n, where n is the maximum integer value. 0 disables statement pooling. The default is 0.

`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 n, where n is the maximum integer value and less than `maxPoolSize`. The default is 0. The value determines connection-pool use:

- When the number of physical connections in the free pool reaches the `minPoolSize` value, the JDBC/MP driver closes subsequent connections by physically closing the connections—not by adding them to the free pool.
- Zero (0) means that `minPoolSize` is equal to `maxPoolSize`.
- If `maxPoolSize` is -1, the value of `minPoolSize` is ignored.

`maxPoolSize` Property

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

`int`

The integer can be -1, 0 through n, where n is the maximum integer value and greater than `minPoolSize`. The default is 0. The value determines connection-pool use:

- 0 indicates no maximum pool size.
- -1 indicates connection pooling is disabled.

`poolLogging` Property

Indicates whether logging information about the state of the connection or statement pool should be written to the `logWriter`. Specify the `poolLogging` property as:

`boolean`

The value can be `true` or `false`. True indicates that the information should be written; false indicates that no information should be written.

This property is effective only if calling the `DataSource.getLogWriter` method does not return null.

`TransactionMode` Property

Sets the total transaction mode. Specify the `TransactionMode` property as:

string

The value can be either `INTERNAL` or `EXTERNAL`. The default is `EXTERNAL`. The value indicates transaction mode:

> `INTERNAL` indicates that you want the JDBC/MP driver to manage transactions.
>
> `EXTERNAL` indicates that you want the application to manage transactions by using JTA or the `com.tandem.tmf.Current` class. For information on the `com.tandem.tmf.Current` class, see the `com.tandem.tmf` package in the NonStop Server for Java API and Reference.

`jdbcPropfile` Property

Sets the name of the Properties file that is used to pass alias names for SQL/MP tables. Specify the `jdbcPropfile` property as:

`string`

The value can be any valid identifier. The default is null.

Troubleshooting

This subsection explains these subjects:

- Tracing
- `No suitable driver` error
- Data truncation
- Dangling statements

Tracing

Through the NonStop Server for Java, the JDBC trace facility enables you to trace the entry point of all the JDBC methods called from the Java applications. For information on how to use this facility, see the NonStop Server for Java Programmer's Reference.

`No suitable driver` Error

The error message `No suitable driver` indicates that the Java Virtual Machine (JVM) could not load the JDBC driver that the program needs. For instructions on loading JDBC drivers, see Loading a Driver.

Data Truncation

Data truncation occurs without warning. To prevent data truncation, use Java data types that are large enough to hold the SQL data items that are to be stored in them. (See Compatible Java and SQL/MP Data Types.)

Dangling Statements

SQL/MP drivers track all statements within a connection, but when an exception occurs, Java's **garbage collection** feature might prevent a driver from immediately closing an open statement. A statement that is left dangling inside a connection could cause unexpected results.

To avoid dangling statements:

- Create statements outside `try-catch` blocks.
- Close statements when they are no longer needed.

In this example, if `executeQuery()` throws an exception, the statement inside the `try-catch` block might be left dangling:

```
try {
        Statement stmt = conn.createStatement();
        ResultSet rs   = stmt.executeQuery("drop table usertable");
}
catch (SQLException sqlex) {
        // error handling
}
```

This example is like the preceding example except that it declares the statement outside the `try-catch` block. If `executeQuery()` throws an exception, the statement is not left dangling.

```
Statement stmt;
ResultSet rs;
```

```
try {
        stmt = conn.createStatement();
        ResultSet rs  = stmt.executeQuery("drop table usertable");
}
catch (SQLException sqlex) {
        // error handling
}

finally {
        stmt.close();
}
```

Sample SQL/MP Program

This Java program connects itself to a NonStop SQL/MP database, sends a simple `select` statement to the database, and processes the results:

```
import java.sql.*;

class sqlmpExample {
   // Load SQL/MP driver the first time, so you do not have to specify
   // -Djava.drivers=com.tandem.sqlmp.SQLMPDriver when running java.

   static {
        try {
               Class.forName("com.tandem.sqlmp.SQLMPDriver");
        } catch (Exception ex) {}
   }

   public static void main(String argv[]) throws SQLException {

     String sqlmpURL = "jdbc:sqlmp:";

     // Use a valid table name from an existing database in the select statement.

     String sqlmpQuery = "select * from $VOL1.MYVOL.MYTAB";
     Statement stmt = null;
     Connection sqlmpConn = null;
     ResultSet res = null;

     try {
         // Try to connect to the SQL/MP database.
         sqlmpConn = DriverManager.getConnection(sqlmpURL);

         // Create an SQL statement object to submit SQL statements.
         stmt = sqlmpConn.createStatement();

         // Submit a query and create a ResultSet object.
         res = stmt.executeQuery(sqlmpQuery);

         // Display the results of the query.
         while(res.next()) {

             // Use get* methods appropriate to the columns in your table
```

```
            System.out.println("Col1 = " + res.getInt(1));
            System.out.println("Col2 = " + res.getString(2));

            System.out.println("Col1 = " + res.getInt(1));
            System.out.println("Col2 = " + res.getString(2));
            }
        res.close();
        stmt.close();
        sqlmpConn.close();
     }
     catch (SQLException sqlex) {
        System.out.println(sqlex);
        // other error handling
        res.close();
        stmt.close();
        sqlmpConn.close();
        }
    }
}
```

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

JDBC/MP Compliance

The JDBC Driver for SQL/MP (JDBC/MP driver) conforms where applicable to the JDBC 3.0 API.

JDBC API Call Exception Summary

These methods in the `java.sql` package throw an `SQLException` with the message SQL/MP: *method-name* is an unsupported method:

Method	Comments

`CallableStatement.getArray(int parameterIndex)`
`CallableStatement.BigDecimal(int parameterIndex)`
`CallableStatement.getBlob(int parameterIndex)`
`CallableStatement.getBoolean(int parameterIndex)`
`CallableStatement.getByte(int parameterIndex)`
`CallableStatement.getBytes(int parameterIndex)`
`CallableStatement.getClob(int parameterIndex)`
`CallableStatement.getDate(int parameterIndex)`
`CallableStatement.getDouble(int parameterIndex)`
`CallableStatement.getFloat(int parameterIndex)`
`CallableStatement.getInt(int parameterIndex, Map map)`
`CallableStatement.getLong(int parameterIndex)`
`CallableStatement.getObject(int parameterIndex, Map map)`
`CallableStatement.getObject(String parameterName, Map map)`
`CallableStatement.getRef(int parameterIndex)`
`CallableStatement.getShort(int parameterIndex)`
`CallableStatement.getString(int parameterIndex)`
`CallableStatement.getTime(int parameterIndex)`
`CallableStatement.getTimestamp(int parameterIndex)`
`CallableStatement.registerOutParameter(int parameterIndex, int sqlType)`
	The particular `CallableStatement` method is not supported.

CallableStatement.registerOutParameter(int parameterIndex, int scale) CallableStatement.wasNull()	
Connection.createStatement(int resultSetType, int resultSetConcurrency, int holdability) Connection.prepareCall(String sql, int resultSetType, int resultSetConcurrency, int holdability) Connection.setHoldability(int holdability) Connection.setReadOnly(boolean parameter) Connection.setTypeMap(java.util.Map map)	The particular `Connection` methods are not supported.
PreparedStatement.setURL(int parameterIndex, URL x)	The particular `PreparedStatement` method is not supported.
ResultSet.getArray(int i) ResultSet.getObject(int columnIndex, Map map) ResultSet.getObject(String columnName, Map map) ResultSet.getRef(int columnIndex) ResultSet.getRef(String columnName) ResultSet.getURL(int i) ResultSet.getURL(String x) ResultSet.refreshRow() ResultSet.updateArray(int columnIndex, java.sql.Array array) ResultSet.updateArray(String columnName, java.sql.Array array) ResultSet.updateArray(int columnIndex, java.sql.Array array) ResultSet.updateRef(int columnIndex, java.sql.Ref ref) ResultSet.updateRef(String columnName, java.sql.Ref ref)	The particular `ResultSet` methods are not supported.

These 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) Connection.prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int holdability)	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)`
	Automatically generated keys are not supported.

These methods in the `java.sql` package throw an `SQLException` with the message "SQL/MP: *method-name* Data type not supported":

Method	Comments

`PreparedStatement.setArray(int i, Array x)`
`PreparedStatement.setBytes(int ColumnIndex, bytes[] x)`
`PreparedStatement.setRef(int i, Ref x)`
	The particular data type is not supported.

Compliance Information

The JDBC/MP driver is not considered JDBC compliant for these reasons:

- `CallableStatement` is not supported because NonStop SQL/MP does not support stored procedures. The `CallableStatement` class exists, but every method in the class throws an "unsupported method" exception.
- Positioned update and delete operations are not supported because NonStop SQL/MP does not support these operations from dynamic SQL (the underlying method the JDBC Driver for SQL/MP uses for accessing a database). However, positioned update and delete operations can be performed from scrollable, insensitive result sets.
- Multithreading is not completely supported because calling both `executeQuery()` and `executeUpdates()` causes the entire JVM (not just the thread executing the SQL operation) to block.
- Column aliasing is not supported because NonStop SQL/MP does not support column aliasing.
- The ODBC minimum SQL grammar is not supported. Therefore, the JDBC Driver for SQL/MP does not support these:
  - Numeric functions, string functions, system functions, or the convert function.
  - Escape syntax for time or date literals, LIKE escape characters, or escape characters for outer joins. Because NonStop SQL/MP does not support stored procedures, escape syntax is not supported for stored procedures.
- The ANSI92 entry-level SQL grammar is not supported because NonStop SQL/MP does not conform to Entry Level SQL as described in the **ANSI** and **ISO/IEC** standards for that topic.

The JDBC specification identifies several areas that are optional and need not be supported if the database management system does not provide support. The JDBC Driver for SQL/MP does not support these optional interfaces:

- SQL3 data types, which include `Array`, `Ref`, and `Struct`. NonStop SQL/MP does not support those data types.
- Custom type-maps used to map between SQL user-defined data types and Java classes. These type-maps include `SQLData`, `SQLInput`, and `SQLOutput`, which are used in conjunction with custom type-maps. NonStop SQL/MP does not support user-defined data types.
- Any methods that use `java.util.Map`, such as `Connection.setTypeMap()`. These methods are not supported because their only function is to provide custom type-maps, which are not supported.
- `XAConnection` and `SADataSource` classes, which are used to support distributed transactions. None of the classes or interfaces in `javax.transaction.xa` are supported.

All other areas of the JDBC Driver for SQL/MP are compliant with the JDBC specifications, including such optional features as BLOBs, CLOBs, and DataSource support.

JDBC Driver for SQL/MP Support for JDBC 3.0

This subsection provides specific information about how the JDBC Driver for SQL/MP supports the JDBC 3.0 API in the areas of:

- Result Sets
- Metadata Support
- Batch Updates
- BLOBs and CLOBs
- JDBC 2.0 Standard Extensions

Result Set Support

These paragraphs describe JDBC/MP's Result Set support for the areas of scrolling and concurrency:

Scrolling

JDBC identifies three types of result sets: forward-only, scroll-insensitive, and scroll-sensitive. Both scroll-related result set types support scrolling, but they differ in their ability to make changes in the database visible when the result set is open. A scroll-insensitive result set provides a static view of the underlying data it contains (the data is fixed when the result set is created). A scroll-sensitive result set is sensitive to underlying changes that are made in the data when the result set is open.

The JDBC Driver for SQL/MP supports scroll-insensitive result sets. JDBC/MP supports scroll-insensitive result sets by storing the entire result set in memory. For this reason, if you select a large scroll-insensitive result set, performance can be affected. In some instances, the JDBC Driver for SQL/MP might choose an alternative result set type when the result set is created.

Concurrency

An application can choose from two different concurrency types for a result set: read-only and updatable. A read-only result set does not allow updates of its contents. An updatable result set allows rows to be updated, deleted, and inserted.

With scrollable result sets, the JDBC Driver for SQL/MP might sometimes need to choose an alternative concurrency type for a result set when the result set is created. Generally, queries that meet these criteria produce an updatable result set:

- The query references only a single table in the database.
- The query does not contain any join operations.
- The query selects the primary key of the table it references.
- The query selects all the nonnullable columns in the underlying table.
- The query selects all columns that do not have a default value.

For the type forward-only result set, although concurrency type updatable is specified, JDBC Driver for SQL/MP switches the concurrency to type read-only.

Metadata Support

This table lists the `java.SQL.DatabaseMetaData` methods for data that the SQL/MP database does not contain; therefore, NULL values are returned.

Method	Comments

`java.sql.DatabaseMetaData.getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)`	These columns are added, but the values are set to NULL because SQL/MP does not support the column type for types: SCOPE_CATLOG, SCOPE_SCHEMA, SCOPE_TABLE, and SOURCE_DATA_TYPE.
`java.sql.DatabaseMetaData.getSchemas()`	Not applicable.
`java.sql.DatabaseMetaData.getTables(String catalog, String schemaPattern, String[] types)`	These columns are added to the column data, but the values are set to NULL because NonStop SQL/MP does not support the column type for types: TYPE_CAT, TYPE_SCHEMA, TYPE_NAME, SELF_REFERENCING_COL_NAME, and REF_GENERATION.
`java.sql.DatabaseMetaData.UDTs(String catalog, String schemaPattern, String tableNamePattern, int[] types)`	Not applicable.

Batch Update Support

The batch update facility enables a Statement object to submit a set of update commands together as a single unit, or batch, to the underlying database management system. The JDBC 2.0 documentation states that batch updates result in a performance improvement. Because NonStop SQL/MP does not support batch updates, it is unlikely that a performance improvement can occur if your application uses the batch update feature. Nevertheless, the batch update capability is available.

BLOBs and CLOBs

A BLOB represents a Binary Large OBject. A CLOB represents a Character Large OBject. BLOBs are typically used to store images (such as GIFs) or serialized objects in the database. CLOBs are typically used to store large character data (such as an XML file) in the database.

Because NonStop SQL/MP does not support either the `Clob` or the `Blob` data type, JDBC provides an alternative method that stores the contents of a BLOB or CLOB in an Enscribe file and stores the name of that file in the column associated with the BLOB or CLOB. From the point of view of the JDBC user, a BLOB or CLOB is any Java object that extends from the `java.lang.Object` class. BLOBs and CLOBs must be serializable objects and must implement the `java.io.Serializable` interface. This enables the JDBC driver to:

- Serialize a `BLOB` or `CLOB` object into an array of bytes to be stored in a BLOB/CLOB file.
- Deserialize the contents of the BLOB/CLOB file to re-create the original object.

Because the JDBC driver creates and reads BLOBs and CLOBs in the same manner, the rest of this description refers only to BLOBs. You must define a CLOB in exactly the same manner as a BLOB and use the BLOB syntax.

SQL/MP Table Definition

To use the `Blob` data type, you must define a column in the associated SQL/MP table to store the name of the Enscribe file associated with that BLOB. The data type of the column must be either `CHARACTER` or `VARCHAR` and must be long enough to store the Enscribe file name (at least 36 characters).

Creating a BLOB

A client JDBC application that needs to store a BLOB in an SQL/MP database table or retrieve a BLOB from an SQL/MP database table provides a `java.util.Properties` object to the JDBC driver during connection time. This Properties object provides the driver with all the information it needs to create or read the BLOB. An additional required property, called a `BlobMapFile` property, specifies a unique file name in the `$volume.subvolume.filename` format. This file stores a list of all the BLOB files created by a particular application. This file is later used by the Cleanup Utility to remove obsolete BLOB files.

`BlobInfo` Properties

`BlobInfo` properties provide information about all table columns that are used to store BLOBs. Each `BlobInfo` property consists of a key and value pair. The key is used to identify a unique `BlobInfo` property. The value specifies a particular BLOB column in a table, the catalog for that table, and file locations where BLOB files should be created.

The Properties object should contain one `'BlobInfo'` property for every table column that contains BLOBs. A `BlobInfo` property is identified by a key of type `'BlobInfoN'`, where `N` represents a unique number used to identify that property.

Format of a `BlobInfo` Property

The format of a `BlobInfo` property is:

```
BlobInfoN, sqlmp_table_name, column_name, sqlmp_catalog_of_Blob_table, $vol.subvol [, ... ]
```

where

`N`

is a unique number used to identify the `BlobInfo` property.

`sqlmp_table_name`

is the name of the BLOB table.

`column_name`

is the name of the column to be associated with the BLOB. This column should be defined to SQL/MP as `CHAR` or `VARCHAR 36`.

`sqlmp_catalog`

is the name of the catalog associated with the BLOB table.

`$vol.subvol [, ... ]`

is a list of one or more locations in the Guardian file system where BLOB files can be stored. You must have write access to these volume and subvolume locations.

`BlobMapFile` Property

A `BlobMapFile` property is used to specify a unique file name in the `$volume.subvolume.filename` format. If the specified file does not already exist, the JDBC driver creates a key-sequenced Enscribe file using the specified name. This file is used to store a list of all BLOB files created by the driver for that particular application. The BLOB map file is secured for use only by its owner. A `BlobMapFile` property is identified by a `'BlobMapFile'` key. This property is required for BLOB support. The `BlobMapFile` property is used for later cleanup of obsolete BLOB files.

Format of a `BlobMapFile` Property

The format of a BlobMapFile property is:

```
BlobFileMap,$volume.subvolume.filename
```

where

`BlobFileMap`

is the key to the property.

`$volume.subvolume.filename`

is a file name in Guardian format.

Example Using BlobInfo Properties and the BlobMapFile Property

This example illustrates how to create `BlobInfo` properties. Two `BlobInfo` properties are created by this example.

The column `PICTURE` in table `$VOL1.MYVOL.PRODUCT` is used to contain BLOBs. The catalog for this table is `$VOL.CTLG`. BLOB files associated with this column should be created in `$VOL1.SUBVOL1` and `$VOL2.SUBVOL2`. The column `EMPPIC` in table `$VOL2.MYVOL.EMPLOYEE` is used to contain BLOBs. The catalog for this table is `$VOL.CTLG`. BLOB files associated with this column should be created in `$VOL2.SUBVOL2`.

A `BlobMapFile` property has also been defined for this example. The file `$VOL2.MYVOL.BLOBMAP` is created by the driver to store a list of all the BLOB files created by the driver for this application.

```
java.util.Properties prop = new java.util.Properties( );
// add BlobInfo Properties
prop.put("BlobInfo1", ","$VOL1.MYVOL.PRODUCT, PICTURE, $VOL.CTLG, $VOL1.SUBVOL1, \
> $VOL2.SUBVOL2");
prop.put("BlobInfo2", ","$VOL2.MYVOL.EMPLOYEE, EMPPIC, $VOL.CTLG, $VOL2.SUBVOL2");
");//add BlobMapFile Property
prop.put("BlobMapFile", ","$VOL2.MYVOL.BLOBMAP");
");// add other properties to java.util.Properties object.....
```

The BLOB table name, column name, and catalog name are required by JDBC. At least one file location per `BlobInfo` property is required to provide BLOB support. If more than one file location is provided, the JDBC driver searches through the list of file locations when creating BLOB files.

Note: The table name and catalog name must be provided in the `$volume.subvolume` format. Aliases for table names or catalog names cannot be substituted for BLOB properties.

The Properties object can also contain other connection-related properties and their values, such as logon ID, password, alias, and so on. The Properties object can also be created by loading the properties from a file. In either case, the format for all BLOB properties must be the same.

Note: After a Properties object has been specified while creating a database connection, the connection is aware of only those BLOB properties specified during connection creation. Any changes or updates made to the Properties object after the connection has been granted are invisible to that connection.

The JDBC drivers use the specified Properties object to ascertain which columns will be used for storing BLOBs for every table the application will use. The JDBC drivers use the BLOB properties to decide if the object to be stored or retrieved is a BLOB. The BLOB properties are also used to decide the location where a BLOB file should be created for a particular BLOB column in a table.

The JDBC drivers also provide some preliminary error checking on the BLOB properties. The error checking ensures that all information required in a `BlobInfo` property has been provided, and that all file names and file locations are in the correct `$volume.subvolume` format.

After a connection to the database has been granted to an application, the application can then use JDBC to access and use BLOBs as it would any other NonStop SQL/MP supported data type. Client applications can use JDBC to:

- Write a BLOB to a database table
- Retrieve a BLOB from a database table
- Update a BLOB stored in a database table
- Delete a row containing a BLOB from a database table

Writing a BLOB to the Database

A JDBC application can write a BLOB to the database using the `setObject( )` method in the `PreparedStatement` interface. Using the SQL/MP JDBC Driver, a BLOB can be written to a database table using any of these methods supported by `SQLMPPreparedStatement`:

```
public void setObject(int parameterIndex, Object x, int targetSqLType, int scale)
public void setObject(int parameterIndex, Object x, int targetSqLType)
```

```
public void setObject(int parameterIndex, Object x)
```

where:

`parameterIndex`
: is the parameter to be set.

`x`
: is any serializable Java object.

`targetSqlType`
: should be `java.sql.Types.OTHER`

`scale`
: is ignored for BLOBs.

Errors encountered during the writing of a BLOB to a BLOB file generate an `SQLException` in this format:

```
SQLMP: error message. See File System error ## for further details.
```

where:

`error message`
: is the error encountered while performing this operation.

`error ##`
: is the file system error returned while writing the BLOB to a BLOB file.

Note: Writing a BLOB to a database runs as a waited operation, so writing a BLOB within a thread blocks that thread.

Example of Writing a BLOB to the Database

This example assumes that a `java.util.Properties` object with `BLOB` properties has been created as described in the previous example and a connection granted using this Properties object. The sample table `$VOL2.MYVOL.EMPLOYEE` contains two columns:

`EMPNAME`
: stores an employee's name

`EMPPIC`
: stores an employee's picture.

```
java.lang.String insertString = new String("INSERT INTO $VOL2.MYVOL.EMPLOYEE VALUES
(?,?)");

// create a PreparedStatement object for the connection
PreparedStatement psmt = con.PreparedStatement(insertString);

// set values to be inserted including a BLOB and insert a new row in the table.
psmt.setString(1, "Mickey Mouse");
psmt.setObject(2, MickyMousePic);  //MickyMousePic is a serializable Java object.
psmt.executeUpdate( );
```

Reading a BLOB From the Database

A JDBC application can retrieve a BLOB object from the database using the `getObject( )` method in the `ResultSet` interface. A BLOB can also be retrieved as a byte array using the `ResultSet` interface. Using the SQLMP JDBC Driver, a BLOB can be read from a database table using these methods supported by `SQLMPResultSet`.

```
public Object getObject(int columnIndex)
public Object getObject(String columnName)
public byte[ ] getBytes(int columnIndex)
public byte[ ] getBytes(String columnName)
```

where:

`columnName`
: is the name of the BLOB column.

`columnIndex`
: is the index of the BLOB column.

Errors encountered during the retrieval of a BLOB from its BLOB file generate an `SQLException` with this message:

```
SQLMP: error message. See File System error ## for further details.
```

where:

`error message`
: is the error encountered when performing this operation.

`error ##`
: is the file system error returned when reading the BLOB from a BLOB file.

Note: Retrieving a BLOB from a database runs as a waited operation, so reading a BLOB within a thread blocks that thread.

Example of Retrieving a BLOB From the Database

This example assumes that a `java.util.Properties` object with BLOB properties has been created as described in the previous example and a connection granted using this Properties object. The sample table `$VOL2.MYVOL.EMPLOYEE` contains two columns:

`EMPNAME`
: stores an employee's name.

`EMPPIC`
: stores an employee's picture.

```
java.lang.String selectString = new String("SELECT * FROM $VOL2.MYVOL.EMPLOYEE");

// create a Statement object for the connection
 stmt = conn.createStatement(    );

// get a ResultSet with the contents of the table.
ResultSet results = stmt.executeQuery(selectString);
//retrieve content of first row
String name = results.getString(1);
Object blob = results.getObject(2);
```

Note: The BLOB can also be retrieved as an array of bytes instead of retrieving as an object:

```
byte[ ] blob_contents = results.getBytes(2);
```

Removing Obsolete BLOB Files

A Cleanup utility is provided to periodically remove the obsolete BLOB files from the file system. BLOB files become obsolete because of one of these operations:

- An application updates a BLOB column. The driver creates a new BLOB file to store the new `BLOB` object. The driver then writes the name of the new BLOB file in the table. The BLOB file containing the original BLOB now becomes obsolete.
- The application deletes a row containing a BLOB from the table. The BLOB file name is removed from the table and is not referenced by any other tables. The BLOB file is now obsolete.

The Cleanup utility uses a BLOB map file. The JDBC driver adds an entry for every BLOB file created by an application to this BLOB map file. During cleanup, the JDBC driver deletes all obsolete BLOB files from the file system and their corresponding entries from the BLOB map file.

The Cleanup utility can be run only by the owner of the BLOB map file. The owner of the application must run the Cleanup

utility to remove any obsolete BLOB files that have been created for the application from the file system. To maintain the integrity of the application's database tables, run this utility only when none of the affected tables containing BLOBs are in use. The Cleanup utility is a Java application, run it from the command line in this way:

```
java com.tandem.sqlmp.SQLMPBlobCleanup  [errfile]
```

where:

`com.tandem.sqlmp.SQLMPBlobCleanup`
: is the Cleanup utility

`Blob map file`
: is the name of the BLOB map file specified by the application

`errfile`
: errors encountered during cleanup are logged to this file

The `Blob map file` parameter is required to successfully run the Cleanup utility. As an option, an error file can be provided to log any errors encountered during cleanup.

> **Note:** The Guardian name of the BLOB map file (in the `$volume.subvolume.filename` format) must be provided to the Cleanup utility. The `Blob map file` name must be enclosed in single quotes so that it is not interpreted by the OSS shell. For example:
>
> ```
> java com.tandem.sqlmp.SQLMPBlobCleanup '$myvol.blobs.mapfile' err
> ```

JDBC 2.0 Standard Extensions

The enhancements to the JDBC 2.0 Standard Extension API include these:

- JNDI for naming databases
- Distributed transaction support

JNDI for Naming Databases

JNDI (Java Naming and Directory Interface) enables an application to specify a logical name that JNDI associates with a particular data source. Using JNDI solves these problems.

For JDBC 1.0, using the JDBC driver manager was the only way to connect to a database. With the JDBC 1.0 approach, the JDBC driver that creates a database connection must first be registered with the JDBC driver manager. However, the JDBC driver class name often identifies a driver vendor whose code loads a driver specific to that vendor's product. The code, therefore, is not portable. In addition, an application needs to specify a JDBC URL when connecting to a database by means of the driver manager. This URL might not only be specific to a particular vendor's JDBC product, but for databases other than SQL/MP, the URL might also be specific to a particular computer name and port.

A JDBC data source object is a Java programming language object that implements the `DataSource` interface. A data source object is a factory for JDBC connections. Like other interfaces in the JDBC API, implementations of `DataSource` must be supplied by JDBC driver vendors.

JDBC Data Source Properties

The JDBC 2.0 API specifies these standard names for data source properties:

- `DatabaseName`
- `DataSourceName`
- `Description`
- `Network Protocol`
- `Password`
- `RoleName`
- `ServerName`
- `UserName`

All these data source properties are of type `String`, and all are optional except the `Description` property, which is required to provide a description of the data source. The NonStop Server for Java includes the property `TransactionMode`, also of type `String`. Acceptable values for that property are `INTERNAL` or `EXTERNAL`. If you specify `INTERNAL`, JDBC manages (begins, ends, rolls back) transactions within the connection. If you specify `EXTERNAL`, you are responsible for beginning and ending transactions.

Distributed Transactions

For an application, the NonStop Server for Java JDBC driver supports distributed transactions in a standard way, as described in the *JDBC 3.0 Specification*. The JDBC API assumes that distributed transaction boundaries are controlled either by a middle-tier server or another API, such as the user transaction portion of the Java Transaction API. The NonStop Server for Java does not support the middleware (those implementing resource managers) pieces of distributed transactions. Instead, the NonStop Server for Java leaves the implementation of such functionality to the middleware vendor.

If you are using distributed transactions, you must use the transaction-aware driver, `sqlmptx`. For more information see Autocommit Mode and the Transaction Aware Driver, and Nonautocommit Mode and the Transaction-Aware Driver.

Floating Point Support

JDBC/MP driver and NonStop Server for Java use the **IEEE** floating-point format and SQL/MP stores values in TNS floating-point format. Therefore, JDBC/MP driver converts any FLOAT (32-bit) number or DOUBLE (64-bit) number read through SQL/MP to IEEE floating-point format and converts any FLOAT or DOUBLE number to TNS floating-point format to store in the SQL/MP database.

For a 32-bit floating point number, a TNS floating-point number has a larger exponent than an IEEE floating-point number. This difference causes the TNS floating-point format being able to represent a larger range of numbers, but with less precision than the IEEE floating-point format. Converting a TNS 32-bit floating-point number to an IEEE 32-bit floating-point number has the biggest potential for error.

A TNS 64-bit floating-point number has a smaller exponent than the IEEE floating-point number. This difference results in the IEEE floating point format being able to represent a larger range of numbers, but with less precision than the TNS floating-point format.

An exception is thrown anytime one of these conditions occurs when writing data to the database or reading data from the database:

- Overflow
- Underflow
- Not a Number
- Infinity

An SQLWarning can occur indicating a "loss of precision" in the conversion from a TNS floating-point number to an IEEE floating-point number or from an IEEE floating-point number to a TNS floating-point number.

For the range of floating-point values and double-precision values and considerations for handling them in your application programs, see the *NonStop Server for Java Programmer's Reference*.

HP Extensions

All the properties in the basic `DataSource` implementation are specific to the JDBC/MP driver. For information on these properties, see Basic DataSource Ojbect Properties.

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

JDBC Trace Facility

The JDBC trace facility is designed to trace the entry point of all the JDBC methods called from the Java applications. The JDBC trace facility can be enabled in any of these ways in which a JDBC connection to a database can be obtained:

- Tracing Using the DriverManager Class
- Tracing Using the DataSource Implementation
- Tracing by Loading the Trace Driver Within the Program
- Tracing Using a Wrapper Data Source

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 these 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 is used to obtain the connection. This table shows the URL and the corresponding JDBC driver used to obtain the JDBC connection:

URL	JDBC Driver
`jdbc:sqlmp:`	JDBC/MP driver
`jdbc:sqlmptx:`	JDBC/MP driver

Java applications should turn on tracing using the `DriverManager.setLogWriter` method, for example by using this 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).

This 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 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.

Output Format

The format of the trace output is:

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

where

`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

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

JDBC Trace Facility Demonstration Programs

The JDBC driver for SQL/MP provides a demonstration program in its installation directory. This program is described in a readme file in the directory. By default, the location is:

JDBC/MP driver

    install-dir/jdbcMp/T1227V30/demo/jdbcTrace

where *install-dir* NSJ4 installation location.

These programs demonstrate tracing by creating a wrapper around the driver-specific data source to be traced.

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

Glossary

A B C D E F G H I J L N O P R S T U V W

A

abstract class

In Java, a class designed only as a parent from which subclasses can be derived, which is not itself appropriate 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

(1) A software program written for or by a user for a specific purpose
(2) 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

BLOB

A data type used to represent Binary Large OBjects. These are typically used to store, in Enscribe files, images or serialized objects in the database. The internal structure and content of a `Blob` object are immaterial to the NonStop Server for Java.

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 VM loads this code, it either interprets it or just-in-time compiles it into native RISC code.

C

C language

A widely used, general-purpose programming language developed by Dennis Ritchie of Bell Labs in the late 1960s. C is the primary language used to develop programs in UNIX environments.

C++ language

A derivative of the C language that has been adapted for use in developing object-oriented programs.

catalog

In NonStop SQL/MP and NonStop 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). You can set the class path explicitly or with the `CLASSPATH` environment variable.

CLOB

A data type used to represent Character Large OBjects These are typically used to store,

in Enscribe files, images or serialized objects in the database. The internal structure and content of a `Clob` object are immaterial to the NonStop Server for Java.

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.

conformant

A Java implementation is conformant if it passes all the tests for J2SE.

connection pooling

A framework for pooling JDBC connections.

Core API

The minimal set of APIs that developers can assume is present on all delivered implementations of the Java Platform when the final API is released by Sun Microsystems, Inc. The NonStop Server for Java conforms to every Core API in the JDK. Core API is also called Java Core Classes and Java Foundation Classes.

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 SQL/MP. The NonStop Server for Java has these driver implementations: JDBC Driver for SQL/MP (JDBC/MP) and JDBC Driver for SQL/MX (JDBC/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 HP 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

garbage collection

The process that reclaims dynamically allocated storage during program execution. The

term usually refers to automatic periodic storage reclamation by the garbage collector (part of the run-time system), as opposed to explicit code to free specific blocks of memory.

`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

HP JDBC Driver for SQL/MP (JDBC/MP)

The product that provides access to NonStop SQL/MP and conforms to the JDBC API.

HP JDBC Driver for SQL/MX (JDBC/MX)

The product that provides access to NonStop SQL/MX and conforms to the JDBC API.

HP NonStop operating system

The operating system for NonStop systems.

HP NonStop ODBC Server

The HP implementation of ODBC for NonStop systems.

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 Technical Library

The browser-based interface to NonStop computing technical information.

HP NonStop Transaction Management Facility (TMF)

An HP product that provides transaction protection, database consistency, and database

recovery. SQL statements issued through a JDBC driver against a NonStop SQL database 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

(1) 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.
(2) 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 a Java Virtual Machine 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).

`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 runs Java bytecode. Also, the command to run the Java interpreter. The Java command invokes the Java Run-Time.

Java 2 Platform Standard Edition (J2SE)

The core Java technology platform, which provides a complete environment for applications development on desktops and servers and for deployment in embedded environments. For more information, see the Sun Microsystems JDK 5.0 Documentation.

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).

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 Foundation Classes

See Core API.

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 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 Run-time

See J2SE Runtime Environment.

Java Virtual Machine

The process that loads, links, verifies, and interprets Java bytecode. The NonStop Server for Java 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).

JDK

See Java Development Kit (JDK).

JNDI

See Java Naming and Directory Interface (JNDI).

JNI

See Java Native Interface (JNI).

`jre`

The Java run-time environment, which runs Java bytecode. Also, the command to run the Java run-time environment.

JVM

See Java Virtual Machine.

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.

N

native

A non-Java routine (written in a language such as C or C++) that is called by a Java class.

NonStop Technical Library (NTL)

The browser-based interface to NonStop computing technical information. NTL replaces HP Total Information Manager (TIM).

node

(1) An addressable device attached to a computer network.
(2) A hypertext document.

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

RDF

See Remote Duplicate Database Facility (RDF).

Remote Duplicate Database Facility (RDF)

The HP software product that does these:

- 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

(1) An implementation of a system used as a stand-alone system or as a node in an Expand network.

(2) 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.

(3) 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.

SQLCI

SQL/MP Conversational Interface.

SQLMPConnectionPoolDataSource

The JDBC/MP driver class that manages connection pools for an application server.

SQLMPDataSource

The JDBC/MP driver class that manages connections.

SQL/MP

See HP NonStop SQL/MP.

SQL/MX

See HP NonStop SQL/MX.

statement pooling

A framework for pooling `PreparedStatement` objects.

Stored procedure in Java (SPJ)

A stored procedure whose body is a static Java method.

stub

(1) 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.
(2) 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)

TNS/E

The hardware platform based on the Intel® Itanium® architecture and the HP NonStop operating system and software that are specific to that platform. All code is PIC (position independent code).

TNS/R

The hardware platform based on the MIPS" architecture and the HP NonStop operating system and software that are specific to that platform. Code may be PIC (position independent code) or non-PIC.

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.

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 Machineand 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, in addition to 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).

A B C D E F G H I J L N O P R S T U W

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

Index

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

abbreviations used in this document
accessing NonStop SQL databases
aliases for database objects
    creating and using
    when you can use
    which SQL statements support
API packages
architecture
audience for this document
autocommit mode
    standard driver
    standard driver, nonautocommit mode
    transaction-aware driver
    transaction-aware driver, nonautocommit mode

B

batch updates, JDBC 2.0
BLOB (Binary Large OBject)
    `BlobInfo` properties
    `BlobInfo` property format
    `BlobMapFile` property
    `BlobMapFile` property format
    creating
    defining a SQL/MP table
    property examples
    reading from a database
    removing obsolete files
    retrieving from a database, example

writing to a database
writing to a database, example

C

catalogs
See database objects
CLOB (Character Large Object)
columns, setting to null
compliance
JDBC API call exception summary
JDBC driver noncompliance
JDBC driver support for JDBC 3.0
components of JDBC/MP
connection pooling and application servers
ConnectionPoolDataSource object properties
guidelines
registering with name service
connection pooling properties for JDBC/MP
connection pooling using basic DataSource interface
DataSource object properties
determining if pool is large enough
requesting pooling
using an SQLMPDataSource
connecting programs to databases
conventions, notation
conversion, data type

D

dangling statements
data source for tracing
data source properties, JDBC
data truncation
data type conversion
data types compatible with Java data types
database engines, SQL/MP
database objects
aliases, creating and using
aliases, when you can use

aliases, which SQL statements support
names for
setting columns to null
databases, connecting to
example, standard driver
example, transaction aware driver
`DataSource` object properties
HP extensions for
DataSource implementation, enabling tracing
distributed transactions, support for
Demonstration programs
of JDBC trace facility
document structure
driver class
adding
loading
specifying
driver, JDBC
noncompliance
support for JDBC 3.0 API
`DriverManager` class
driver list for `getConnection`
statement pooling with
`DriverManager` class, enabling tracing
drivers
architecture
loading
See also loading drivers
requirements for
standard, `sqlmp`
transaction-aware, `sqlmptx`
types of
verifying installation

E

extensions, standard API for JDBC 2.0
distributed transactions
JDBC data source properties
JNDI

extensions, HP

F

files, properties of
floating point support

G

`getConnection` method

H

HP extensions

I

indexes
    See database objects
implementation, enabling tracing
InitialPoolSize property
installation, verifying

J

Java data types, converting to
JDBC data source properties
JDBC driver
JDBC trace facility
    output format
    tracing by loading the trace driver within the program
    tracing using a wrapper data source
    tracing using the `DataSource` implementation
    tracing using the `DriverManager` class
JDBC driver types
JDBC `driverManager`
JDBC 3.0 API, support for
    batch updates
    BLOBs and CLOBs

See also BLOB (Binary Large Object)
call exception summary
metadata
result sets
standard extensions
JDBC/MP, components of
requirements for
standard driver
transaction-aware driver
troubleshooting
`jdbcPropfile` property
basic DataSource object properties
specifying properties with
use with connection pooling
JNDI for naming databases

L

loading drivers
adding the driver class
loading the driver class
specifying the driver class

M

`MaxIdleTime` property, application server
`maxPoolSize` property, application server
`maxPoolSize` property, JDBC/MP
`maxStatements` property, application server
`maxStatements` property, JDBC/MP
methods, tracing
metadata support
`minPoolSize` property, application server
`minPoolSize` property, JDBC/MP

N

name service, registering with
`No suitable driver` error

nonautocommit mode, SQL/MP
  standard driver, autocommit mode
  standard driver
  transaction-aware driver, autocommit mode
  transaction-aware driver
Notation conventions

P

passing SQL/MP statements to databases
pooling
  See connection pooling entries and statement pooling
poolLogging property, JDBC/MP
printing this document
properties
  basic DataSource object
  connection pooling with the DriverManager class
  HP extensions for
  specifying using put() method
  specifying in a properties file
  standard ConnectionPoolDataSource
  statement pooling
properties file
  loading programmatically
  loading using a system property
PropertyCycle property
put() calls for alias creation

R

related reading
  NonStop Server for Java documents
  NonStop system computing documents
  Sun Microsystems documents
requirements for drivers
result sets, JDBC 2.0

S

sample programs
- SQL/MP, connecting and using
- SQL/MP program

SQL/MP, accessing databases
sqlmp driver
sqlmptx
standard extension API for JDBC 2.0
statements
- dangling
- passing to databases

statement pooling
- how to use
- properties for

structure of this document

T

tables, setting columns to null
TNS floating point
transaction-aware driver, sqlmptx
Tracing
- See JDBC trace facility

tracing
TransactionMode property, JDBC/MP
transactions
- autocommit mode, standard driver
- autocommit mode, transaction-aware driver
- distributed, support for
- managing with TMF
- nonautocommit mode, standard driver
- nonautocommit mode, transaction-aware driver

troubleshooting
- dangling statements
- data truncation
- No suitable driver error
- statement pooling
- tracing

truncation of data

U

updates, batch

V

verifying driver installation

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

List of Examples

Section Title	Example Title
Accessing SQL Databases with SQL/MP	• Standard Driver Example • Transaction-Aware Driver Example • Passing SQL/MP Statements to a Database • Specifying Properties Using the `put()` Method • Specifying Properties in a Properties File • Registering the `ConnectionPoolDataSource` and `DataSource` Objects with a JNDI-Based Naming Service • Determining if the Pool is Large Enough • How a Dangling Statement Can Be Created • How a Dangling Statement Can Be Avoided • Sample SQL/MP Program
JDBC/MP Compliance	• `BlobInfo` and `BlobMapFile` Properties Example

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

List of Figures

Section Title	Figure Title
Introduction to JDBC/MP	• Architecture of the JDBC Driver for SQL/MP
Accessing SQL Databases with SQL/MP	• Loading a JDBC Driver Class Into the JVM

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.

List of Tables

Section Title	Table Title(s)
About This Manual	• Document Sections
Introduction to JDBC/MP Driver	• JDBC Driver Types
Accessing SQL Databases with SQL/MP	• Compatible Java and SQL/MP Data Types • Default SQL/MP Database Object Names • Standard ConnectionPoolDataSource Object Properties
JDBC/MP Compliance	• JDBC API Call Exception Summary • Metadata Support
JDBC Trace Facility	• Tracing Using the DriverManager Class • Tracing Using the DataSource Implementation

© 2006, Hewlett-Packard Development Company L.P. All rights reserved.