

Daffodil DB

JDBC Reference Guide

Version 4.1

March 2005

Copyright © Daffodil Software Limited
SCO 42, 3rd Floor
Old Judicial Complex, Civil lines
Gurgaon - 122001
Haryana, India.
www.daffodildb.com

All rights reserved. Daffodil DB™ is a registered trademark of Daffodil Software Limited. Java™ is a registered trademark of Sun Microsystems, Inc. All other brand and product names are trademarks of their respective companies.

Table of Contents

Preface	5
Purpose.....	5
Target Audience.....	6
Related Documentation	7
Daffodil DB and JDBC 3.0	8
Daffodil DB JDBC Driver.....	8
JDBC 3.0 features supported by Daffodil DB.....	9
Compatibility	10
Fundamentals of Daffodil DB JDBC Driver.....	11
How to access Daffodil DB through JDBC?.....	11
Import JDBC classes.....	11
Register/Load JDBC Driver.....	11
Establishing a connection	12
Examples	12
Establishing a Connection with Read-Only Mode	13
Minimal Program.....	14
Different ways for loading Daffodil DB JDBC Driver	15
Different ways to connect to Daffodil DB	16
Working with multiple connections in Daffodil DB.....	17
Database Connection URL Attributes	18
Creating and Dropping a Database	20
Database-level Encryption*.....	21
Multiple Data File Support.....	22
Standard JDBC Features	23
Datatypes.....	23
Prepared Statements	24
Callable Statements.....	25
Batch Operations	26
ResultSet	27
DatabaseMetaData	28
Transaction Isolation Level	28
Advanced JDBC Features	29
Savepoint.....	29

RowSet*	30
Connection Pooling	31
Updatable ResultSet	32
Partial fetching in ResultSet	32
Retrieving Parameter Metadata	33
Daffodil DB XA Support*	34
Distributed Transaction Concepts	35
Classes related to Resource Managers	37
Controlling JDBC Session properties	38
Avoiding Deadlocks	40
End Note	41
Sign Up for Support	42
We Need Feedback!	43
License Notice	44

* Features that are not supported in One\$DB

Preface

Purpose

Daffodil DB JDBC Reference Guide explains how to use Daffodil DB and JDBC (Java Database Connectivity) technology to develop applications. It provides instructions for using JDBC within Daffodil DB Embedded edition as well as Networked edition. It walks through the basic Daffodil DB and JDBC concepts like JDBC 3.0 features supported by Daffodil DB, how to create and access Daffodil DB databases through JDBC API, Daffodil DB support for JDBC and JTA, and how to use Daffodil DB in a Distributed Transaction Processing environment.

For more comprehensive and up-to-date information on JDBC and JTA, please refer the following resources:

- <http://java.sun.com/products/jdbc>
- <http://java.sun.com/products/jta>
- <http://developer.java.sun.com/developer/Books/JDBCTutorial/>
- <http://java.sun.com/products/jdbc/download.html#corespec30>

Target Audience

This guide is intended to act as a ready reference tool for software developers building applications using Daffodil DB through JDBC interface. This guide assumes that you are familiar with the following concepts:

- Basis JDBC (Java Database Connectivity) Concepts.
- Basic SQL (Standard Query Language) Concepts.
- Fundamentals of Relational Database Concepts.
- Basic Java Programming Language.
- Familiar with your OS and Server/Client concepts.

It is also assumed that the reader has already gone through [Getting Started with Daffodil DB Guide](#).

Related Documentation

Daffodil DB Getting Started Guide	Designed to help new and intermediate Daffodil DB users navigate and perform common tasks like How to start and stop Daffodil DB, Understanding key variables used by Daffodil DB, User documentation bundled with Daffodil DB. Also briefly describes Daffodil DB Editions and Tools
Daffodil DB System Guide	Describes the architecture of Daffodil DB and provides the information that the user might need to keep Daffodil DB running with high performance and reliability in a server framework or a multi-user application server. Also describes the standards on which Daffodil DB had been built, transaction capabilities and some of the unique features supported by Daffodil DB.
Daffodil DB SQL Reference Guide	Covers all the SQL-99 features supported by Daffodil DB. This ready reference tool describes in detail the syntax and semantics of SQL language statements and elements for Daffodil DB. It explains how to use SQL with Daffodil DB and how to perform various database operations on Daffodil DB such as creating tables or indexes, managing transactions and sessions, Daffodil DB security features etc.
Daffodil DB Tools Guide	Explains how to use Daffodil DB Browser with Embedded as well as Server versions of Daffodil DB. Describes how to perform various database operations on Daffodil DB using Daffodil DB Browser such as creating a database, creating database objects, manipulating data, creating triggers etc.
Daffodil DB PHP Extension Guide	Explains how to talk to Daffodil DB through PHP interface. Describes in detail about the supported PHP functions, with which a user can easily perform various tasks like connecting/disconnecting to a database, executing queries, distributed transaction processing, manipulating the ResultSet etc.

Daffodil DB and JDBC 3.0

Daffodil DB JDBC driver implements standard JDBC 3.0 (Java Database Connectivity) interface defined by Sun Microsystems. The core JDBC Application Program Interface (API) consists of a set of call level interfaces found in the [java.sql](#) and [javax.sql](#) packages. The JDBC API is used by Java applications to access and manipulate the data stored in a database by invoking SQL commands.

Daffodil DB JDBC Driver

Daffodil DB JDBC driver supports JDBC 3.0 specifications and had been **certified for J2EE applications by Sun Microsystems Inc.**

The Daffodil DB JDBC driver is a native protocol for all Java drivers (Type #4 among the categories defined by Sun Microsystems). Type 4 drivers are direct-to-database, pure Java drivers ("thin" driver). Type 4 drivers takes JDBC calls and translates them into the network protocol (proprietary protocol) used directly by any Database Management System (DBMS). Thus, client machines or application servers can make direct calls to the DBMS server. Type 4 drivers provide faster performance and direct access to DBMS features.

In the *Embedded Edition* of Daffodil DB (when invoked from an application running on the same JVM as Daffodil DB), the JDBC driver supports connections to a Daffodil DB database in the local mode. Network transport is not required to access the database. In *Networked (Client/Server) mode*, the client application dispatches JDBC requests to the Daffodil DB server over a network.

Applications running in embedded mode use a different driver from that used by applications running in the client/server mode.

JDBC 3.0 features supported by Daffodil DB

API	Description
java.sql.BatchUpdateException	Provides information about errors occurring during batch operations.
java.sql.Blob	Provides access to and manipulation of Binary Large Object data (BLOB).
java.sql.CallableStatement	Provides access to and manipulation of Stored Procedures.
java.sql.Clob	Provides access to and manipulation of Character Large Object data(CLOB)
java.sql.Connection	Constructs and manages the connection to the database and also provides metadata information about the database
java.sql.DatabaseMetaData	Provides metadata about the database.
java.sql.Driver	Provides information about and the JDBC driver and manage it.
java.sql.ParameterMetaData	Provides the number, type and properties of parameters to prepared statements.
java.sql.PreparedStatement	Manages dynamic SQL statements.
java.sql. ResultSet	Encapsulates a set of rows.
java.sql.ResultSetMetaData	Provides metadata information about the ResultSet.
java.sql.Statement	Manages static SQL statements.
javax.sql.ConnectionPoolDataSource	Supports caching and reusing of physical connections, which improves application performance and scalability.
javax.sql.DataSource	Provides access to JDBC drivers and manages data sources.
javax.sql.PooledConnection	Represents physical connection to a DataSource.
javax.sql.RowSet	Encapsulates a set of rows that have been retrieved from a tabular DataSource.

<code>javax.sql.RowSetMetaData</code>	Manages row set of metadata information.
<code>javax.sql.XAConnection</code>	An XA Connection object is a Pooled Connection object that can participate in a distributed transaction.
<code>javax.sql.XADataSource</code>	Provides connections that can participate in a distributed transaction.

Compatibility

Platforms

Daffodil DB JDBC driver had been tested and certified on all platforms including Windows and Linux that support JDK 1.3 or higher.

JDBC Version

The Daffodil DB JDBC driver is compatible with JDBC version 3.0

JDK Version

The Daffodil DB JDBC driver requires JDK version 1.3 or higher.

Fundamentals of Daffodil DB JDBC Driver

This section describes the most fundamental JDBC features supported by Daffodil DB JDBC driver. It explains how to connect to *Daffodil DB Embedded edition* or *Daffodil DB Networked edition*.

How to access Daffodil DB through JDBC?

To access Daffodil DB database, the application must perform the following tasks:

- Import JDBC classes.
- Register/Load Daffodil DB JDBC driver.
- Establish connection to the database through the driver.

Import JDBC classes

To import JDBC classes into the application, you have to insert the following import statements at the beginning of your program:

```
import java.sql.*;    //import the JDBC classes
```

Register/Load JDBC Driver

Use the following method calls to register Daffodil DB JDBC driver with the JDBC Driver Manager:

For Daffodil DB Embedded Edition

```
Class.forName ("in.co.daffodil.db.jdbc.DaffodilDBDriver");
```

For Daffodil DB Server Edition

```
Class.forName ("in.co.daffodil.db.rmi.RmiDaffodilDBDriver");
```

After loading the Daffodil DB JDBC driver, Daffodil DB is available for taking connection.

Note: - The Daffodil DB driver classes must be included in the application's classpath.

For Embedded mode you need to include ***DaffodilDB_Embedded.jar*** and ***DaffodilDB_Common.jar***

For Network mode you need to include ***DaffodilDB_Client.jar*** in your application's classpath.

Establishing a connection

Connect to the database using JDBC Driver Manager's [getConnection \(\)](#) method. The following method calls creates a database named “**School**” if it does not exist and connects to it:

Examples

// For Daffodil DB Embedded Edition

1.

```
Connection con = DriverManager.getConnection
("jdbc:daffodilDB_embedded:School;create=true","","");
```
- OR
2.

```
String driver = "in.co.daffodil.db.jdbc.DaffodilDBDriver";
String url= "jdbc:daffodilDB_embedded:School;create=true";
Class.forName(driver);
Connection con = DriverManager.getConnection(url,
"daffodil", "daffodil");
```

// For Daffodil DB Server Edition

```
Connection con = DriverManager.getConnection
("jdbc:daffodilDB://<hostname>:<port>/School;create=true","","");
```

If one of the loaded drivers recognizes the JDBC URL supplied to the method [DriverManager.getConnection \(\)](#), that driver will establish a connection to the Daffodil DB Server specified in the JDBC URL.

The connection returned by the method [DriverManager.getConnection \(\)](#) is an open connection that can be used to create JDBC statements, which passes SQL statements given by a user to the Daffodil DB server.

Establishing a Connection with Read-Only Mode

For establishing a connection with read-only mode, you have to specify `readonly=true` in the connection URL. Connect to the database using JDBC Driver Manager's [getConnection \(\)](#) method. This method throws an exception if database does not exist on the specified path. So database must exist to get connection in read-only mode.

Example: (for Embedded mode)

```
String driver = "in.co.daffodil.db.jdbc.DaffodilDBDriver";
String url = "jdbc:daffodilDB_embedded:School;path =
             c:/DaffodilDB3_4/databases;readonly=true";

Class.forName(driver); // Register the driver

Connection con = DriverManager.getConnection(url, "daffodil", "daffodil");
// Connect to database
```

If one of the loaded drivers recognize the JDBC URL supplied to the method [DriverManager.getConnection \(\)](#), that driver will establish a connection to the Daffodil DB Server specified in the JDBC URL.

The connection returned by the method [DriverManager.getConnection\(\)](#) is an open connection that can be used to create JDBC statements, which passes SQL statements given by a user to the Daffodil DB server.

In read-only mode user can not perform any operation related to Data Definition Language and Data Manipulation Language such as [Create table Table_Name\(Column_Name Data_Type\)](#) OR [insert into Table_Name values\(1\)](#). A user can perform only Data Query and Control Language operations such as [Select * from Table_Name](#).

Minimal Program

The following sample class connects to an embedded Daffodil DB database “**School**”:

```
//import jdbc classes
import java.sql.*;
public class sample {
    public static void main (String[] args) {
        try {
            //Register the driver
            Class.forName ("in.co.daffodil.db.jdbc.DaffodilDBDriver");
            System.out.println ("Daffodil DB started!");
            //Connect to the database
            Connection con = DriverManager.getConnection
                ("jdbc:daffodilDB_embedded:School;create=true","","");
        } catch (Throwable e) {
            System.out.println ("exception thrown:"+e);
        }
    }
}
```

Different ways for loading Daffodil DB JDBC Driver

JDBC driver class for Daffodil DB embedded: [in.co.daffodil.db.jdbc.DaffodilDBDriver](#)

JDBC driver class for Daffodil DB server: [in.co.daffodil.db.rmi.RmiDaffodilDBDriver](#)

Daffodil DB JDBC driver (embedded or server) can be loaded through one of the following ways.

Note: - The examples mentioned here are for the embedded edition.

- **`Class.forName("in.co.daffodil.db.jdbc.DaffodilDBDriver");`**

This is the one recommended by JavaSoft. However, some JVMs may not load the class when it is accessed. They can delay it until instances of the class are created.

- **`Class.forName("in.co.daffodil.db.jdbc.DaffodilDBDriver").newInstance()`**
- **`new in.co.daffodil.db.jdbc.DaffodilDBDriver()`**

Same as `Class.forName("in.co.daffodil.db.jdbc.DaffodilDBDriver").newInstance()`, except that it requires the class to be specified when the code is compiled.

- **`Class c = in.co.daffodil.db.jdbc.DaffodilDBDriver.class`**

Same as `Class.forName("in.co.daffodil.db.jdbc.DaffodilDBDriver").newInstance()`, except that it requires the class to be specified when the code is compiled.

Different ways to connect to Daffodil DB

Daffodil DB server and Daffodil DB database can be accessed by specifying the connection attributes in the Daffodil DB database connection URL. The general rule for specifying the connection attributes are:

- ❑ **For Embedded edition:** "jdbc:daffodilDB_embedded:<database name>;property1=value1;property2=value2;propertyn=valuen","<user name>","<password>"
- ❑ **For Server edition:** jdbc:daffodilDB://<hostname>:<port>/<database name>;property1=value1;property2=value2;propertyn=valuen","<user name>","<password>"

Here property1, property2 .. propertyn should be one of the properties supported by Daffodil DB. If such a property, which is not supported by Daffodil DB is passed, then it is ignored by Daffodil DB.

User name and password can also be passed as name-value pairs (properties) in the connection URL.

For example:

// For Daffodil DB Embedded Edition

```
Connection con = DriverManager.getConnection  
("jdbc:daffodilDB_embedded:School;create=true","user1","password1");
```

// For Daffodil DB Server Edition

```
Connection con = DriverManager.getConnection  
("jdbc:daffodilDB://oneOfTheMachines:3456/School;create=true","user1","  
password1");
```

The database name passed in the connection URL is searched in the following folders in the order given below (for both embedded and server editions):

- ❑ **daffodilDB_home** (system property)
- ❑ **daffodilDB_home** (environment variable)
- ❑ **<user.home>/DaffodilDB** (system property)

Daffodil DB does not support passing of an absolute or relative path of the database in the connection URL. It is mandatory for the database to be present in the **DAFFODILDB_HOME** folder.

For more information on **DAFFODILDB_HOME**, please refer to "[Getting Started with Daffodil DB Guide](#)".

Working with multiple connections in Daffodil DB

A Connection object represents connection with a database. Within the scope of one connection, you can access only a single Daffodil DB database. A single application can have one or more connections to Daffodil DB, either to a single database or to many different databases, which can span to different Daffodil DB Instances.

The following example (for Embedded) describes a scenario in which an application establishes three separate connections to two different databases in the current system.

```
Connection conn = DriverManager.getConnection
("jdbc:daffodilDB_embedded:Sapling");
System.out.println ("Connected to database Sapling ");
conn.setAutoCommit (false);

Connection conn2 = DriverManager.getConnection
("jdbc:daffodilDB_embedded:newDB;create=true");
System.out.println ("Created AND connected to newDB");
conn2.setAutoCommit (false);

Connection conn3 = DriverManager.getConnection
("jdbc:daffodilDB_embedded:newDB");
System.out.println ("Got second connection to newDB");
conn3.setAutoCommit (false);
```

Database Connection URL Attributes

An application in an embedded environment uses a different database connection URL format from the one used by applications in a client/server environment. Pairs of attributes and values should be specified as part of Daffodil DB database connection URL. The examples in this section use the database connection URL for use in an **embedded environment**. Same attributes and values can also be specified in case Daffodil DB is used as a database server edition.

Daffodil DB recognizes the following as valid attributes:

- ❑ **create:** The value for this attribute is Boolean (either true or false) which specifies whether Daffodil DB should create a new database with the name passed, in case database already does not exist. Default value for this parameter is *false*. For creating a new database, you have to specify *create = true*.
- ❑ **user:** The value for this attribute specifies the name of the user connecting to the database. There is no default value for this parameter.
- ❑ **password:** The value for this attribute specifies password of a user connecting to the database. There is no default value for this parameter.
- ❑ **ENCRYPTIONSUPPORT:** The value for this attribute is Boolean (either true or false) which specifies whether to store data in an encrypted form or not. Default value for this parameter is *false*. If set to *true*, the data will be stored in an encrypted form.
- ❑ **ENCRYPTIONALGO:** The value for this attribute is the name of an encryption algorithm according to which user wants to encrypt the database.
- ❑ **ENCRYPTIONKEY:** User may set encryption key of his choice.
- ❑ **MULTIFILESUPPORT:** The value for this attribute is Boolean (either true or false). If set to true, multiple data file support will be enabled. Default value is *false*.
- ❑ **INITIALFILESIZE:** This value for this attribute specifies the initial size of database in MB, if in case a new database is created. Default value for this parameter is 5m.
- ❑ **INCREMENTFACTOR:** This is an integer which specifies the factor by which the database size had to increase after the space allocated to the database had been taken up or creates subsequent file if MULTIFILESUPPORT is set to true . This is expressed in terms of percentage of the current size of the database. Default value for this parameter is 20%. Valid values for this parameter are from 1 to 100.
- ❑ **verbose:** The string passed in this parameter will display at the server side traces, when server runs under verbose mode. It is helpful for the debugging purpose. It is extremely helpful in the scenario where database is accessed by multiple connections or multiple users.
- ❑ **readonly:** This attribute is a Boolean parameter which specifies the mode in which connection is to be made with Daffodil DB. When it is set to true, the connection will be in read-only mode. In read-only mode, a database must exist.

Default value for this parameter is *false*.

If any other attribute is passed with the connection URL, it is simply ignored (**all parameters being case sensitive**)

Example code snippet

// For Daffodil DB Embedded Edition only

```
Connection con = DriverManager.getConnection  
("jdbc:daffodilDB_embedded:School;readonly=false;create=true;  
INITIALFILESIZE=20;INCREMENTFACTOR=50","user1","password1");
```

You can also set the attributes listed above by passing a Properties object along with a database connection URL to *DriverManager.getConnection ()* when obtaining a connection.

Specifying Attributes in Properties Object

Instead of specifying attributes on the database connection URL, you can specify attributes as properties in the Properties object that can be passed as a second argument to the *DriverManager.getConnection ()* method.

For example, to set the “create” attribute to “true” (in case of **embedded environment**), the following steps needs to be performed:

```
Class.forName ("in.co.daffodil.db.jdbc.DaffodilDBDriver").newInstance ();  
Properties p = new Properties ();  
p.put ("create", "true");  
Connection conn = DriverManager.getConnection  
("jdbc:daffodilDB_embedded:databaseName", p);
```

Creating and Dropping a Database

A new database can be created by supplying database name with the database connection URL and specifying *create=true*. Daffodil DB creates new database inside a new subdirectory under **DAFFODILDB_HOME** with the same name as that of the new database.

For example:

// For Daffodil DB Embedded Edition

```
Connection con = DriverManager.getConnection  
("jdbc:daffodilDB_embedded:School;create=true","user1","password1");
```

// For Daffodil DB Server Edition

```
Connection con = DriverManager.getConnection  
("jdbc:daffodilDB://oneOfTheMachines:3456/School;create=true","user1","  
password1");
```

The above written sample code will create a new database with the name “**School**”, if it does not exist already, and provides a connection to the new database. If database with the name “School” already exists then it will simply provide a connection to the existing database.

Database can be dropped using drop database statement.

(Note: - The database can't be dropped if more than one connection is open.)

For more information on **drop database statement**, please refer “[Daffodil DB SQL Reference Guide](#)”.

Database-level Encryption*

User may encrypt the database to avoid any kind of malicious access to data stored on the disk. If user needs to encrypt most or all of the data in a database, then it is required to encrypt data at the database level. Daffodil DB allows its user to use the encryption system of one's choice. The interface is designed universally to allow usage of industry standard encryption algorithms like DES, AES, BLOWFISH, DES3, TEA, IDEA, and TWOFISH. User is required to specify the encryption key, which is a block of bytes that the encryption algorithm will use as a secret key. The key length is algorithm-specific; popular algorithms use keys of 64,128, or 256 bits in length. User needs to set `ENCRYPTIONSUPPORT` property to `true` at the time of creation of database and may choose the desired algorithm for encrypting the database.

Example:

```
String url = "jdbc:daffodilDB_embedded:STUDENTDB;create=true";
String driver = "in.co.daffodil.db.jdbc.DaffodilDBDriver";
Properties prop = new Properties();
prop.setProperty("user","daisy");
prop.setProperty("password","daisy");
prop.setProperty("create","true");
prop.setProperty("ENCRYPTIONSUPPORT","true");
prop.setProperty("ENCPTIONALGO","tea");
prop.setProperty("ENCRYPTIONKEY","daisy");
Class.forName(driver);
java.sql.Connection con = DriverManager.getConnection(url,prop);
```

In the above mentioned example a database named **STUDENTDB** is created which will be stored in an encrypted form using *TEA* encryption algorithm and encryption key is *daisy*.

* Features that are not supported in One\$DB

Multiple Data File Support

To provide support for multiple data file, you need to set the following properties:

```
props.put("user", "userName");  
props.put("password", "password");  
props.put("create", "true");  
props.put("MULTIFILESUPPORT", "true");  
props.put("INITIALFILESIZE", "500m");  
props.put("INCREMENTFACTOR", "100");
```

Standard JDBC Features

This section summarizes how Daffodil DB JDBC driver utilizes the standard features of JDBC.

Datatypes

Daffodil DB JDBC driver supports the SQL data types required by JDBC 3.0. The following table shows the mapping between JDBC 3.0 data types and Daffodil DB data types.

JDBC Data Type	Daffodil DB Data Type
CHAR	CHAR / CHARACTER
VARCHAR	VARCHAR / CHARVARYING / CHARACTERVARYING
LONGVARCHAR	LONGVARCHAR
NUMERIC	NUMERIC
DECIMAL	DEC / DECIMAL
BIT	BIT
BOOLEAN	BOOLEAN
TINYINT	TINYINT
SMALLINT	SMALLINT
INTEGER	INT / INTEGER
BIGINT	BIGINT LONG
REAL	REAL
FLOAT	FLOAT
DOUBLE	DOUBLE PRECISION
BINARY	BINARY
VARBINARY	VARBINARY
LONGVARBINARY	LONGVARBINARY

DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
CLOB	CLOB / CHARLARGEOBJECT / CHARACTERLARGEOBJECT
BLOB	BLOB
DATALINK	VARCHAR
JAVA_OBJECT	BLOB

Prepared Statements

Daffodil DB JDBC driver provides support for prepared statements which can improve the performance of the applications relative to static JDBC statements. Unlike a static JDBC statement, dynamic or prepared statements are only compiled once, regardless of the number of times they are used. A dynamic JDBC statement is used when you need multiple executions of a particular SQL statement that has varying values associated with it.

Because Prepared Statement objects are precompiled, their execution can be faster than those of Statement objects. Hence, if an SQL statement is required to execute many times, it would be better to use prepared statements.

The following code fragment, where `con` is a Connection object, creates a Prepared Statement object containing an SQL update statement with one parameter:

```
PreparedStatement pstmt1 = con.prepareStatement(  
    "UPDATE Teacher SET salary = salary + 1000 WHERE EmployeeId = ?");
```

The object `pstmt1` now contains the statement "UPDATE Teacher SET salary = salary + 1000 WHERE EmployeeId = ?".

Before a Prepared Statement object is executed, the value of each '?' parameter must be set. This is done by calling a `setXXX ()` method, where XXX is an appropriate type for the parameter.

For example, if the parameter is of the type, String in the Java Programming Language, the method to use is `setString ()`. The first argument to the `setXXX ()` method is the ordinal position of the parameter to be set, with numbering starting at 1. The second argument is the value to which the parameter is to be set.

For example:

```
pstmt1.setString(1, "Emp001");
```

There are three methods `execute ()`, `executeQuery ()`, and `executeUpdate ()` to execute the statement depending upon the type of statement, which is set into the Prepared Statement object.

Callable Statements

Daffodil DB JDBC Driver implements Callable Statement interface (which extends Prepared Statement) with methods for executing and retrieving results from stored procedures.

Daffodil JDBC Driver supports IN, OUT and INOUT parameters for callable statements. A callable statement may take n number of input parameters and return n number of output parameters. As with Statement and Prepared Statement objects, Callable Statement objects are created by Connection objects.

For example:

The following Database stored procedure “Student_Marks_InOut_Proc” retrieves marks of a student for the StudentId passed

```
CREATE PROCEDURE Student_Marks_InOut_Proc  
  ( INOUT INOUT_PARAM INTEGER)  
SPECIFIC Student_Marks_InOut_Proc as  
BEGIN  
    SELECT Marks into INOUT_PARAM from MarksRecord  
    WHERE StudentId=INOUT_PARAM;  
END;
```

The SQL statement for calling a stored procedure would be:

```
CALL Student_Marks_InOut_Proc (5)
```

The following code calls the above stored procedure using a callable statement:

// Create SQL to invoke a stored procedure

```
String SQL_USE_PROC = "{ call Student_Marks_InOut_Proc(?) }";
```

// Create a callable statement with one binding parameters

```
m_callStmt = m_conn.prepareCall(SQL_USE_PROC);  
m_callStmt.setInt(1, 24);  
m_callStmt.executeQuery();
```

// Close the callable statement

```
m_callStmt.close();
```

Batch Operations

Daffodil DB JDBC driver supports JDBC SQL batches. You can either use batch multiple SQL statements using a [java.sql.Statement](#) object, or batch multiple calls of a single SQL statement using a [java.sql.PreparedStatement](#) object.

Auto-commit should be disabled while using batch updates.

The following code, in which con is an active connection, illustrates a batch operation with auto-commit turned off:

```
con.setAutoCommit (false);
PreparedStatement updateTeacherSalary = con.prepareStatement (
    "UPDATE TEACHER SET salary = salary + 1000 WHERE EMPLOYEEID =?");
updateTeacherSalary.setInt (1, 5);
updateTeacherSalary.addBatch ();
updateTeacherSalary.setInt (1, 6);
updateTeacherSalary.addBatch ();
int [ ] updateCounts = ps.executeBatch ();
con.commit ();
con.setAutoCommit (true);
```

If auto-commit is not set to false before executing the batch operation, all the operations in the batch would be operated in auto-commit mode i.e. commit will be performed after every statement.

ResultSet

Daffodil DB JDBC driver provides all the types of ResultSet according to JDBC 3.0 specification and are listed below:

- ☐ TYPE_FORWARD_ONLY
- ☐ TYPE_SCROLL_INSENSITIVE
- ☐ TYPE_SCROLL_SENSITIVE

The default ResultSet type is ***TYPE_FORWARD_ONLY***.

Daffodil DB supports both the concurrency levels specified by JDBC specification:

- ☐ CONCUR_READ_ONLY
- ☐ CONCUR_UPDATABLE

DatabaseMetaData

The DatabaseMetaData interface is implemented by Daffodil DB JDBC driver to provide information about Daffodil DB database engine to applications using Daffodil DB. It is used primarily by application servers and tools. The Daffodil DB JDBC driver supports all the standard JDBC DatabaseMetaData methods.

Example:

```
DaffodilDBDatabaseMetaData DBM = (DaffodilDBDatabaseMetaData)  
con.getMetaData();
```

Transaction Isolation Level

Daffodil DB provides five transaction isolation levels. A connection determines its own isolation level. So JDBC provides an application with a way to specify a level of transaction isolation. JDBC defines the following isolation levels:

- ☐ TRANSACTION_READ_UNCOMMITTED
- ☐ TRANSACTION_READ_COMMITTED
- ☐ TRANSACTION_REPEATABLE_READ
- ☐ TRANSACTION_SERIALIZABLE

In Daffodil DB, the following java.sql.Connection isolation levels are available:

- ☐ TRANSACTION_SERIALIZABLE
- ☐ TRANSACTION_REPEATABLE_READ
- ☐ TRANSACTION_READ_COMMITTED
- ☐ TRANSACTION_READ_UNCOMMITTED
- ☐ SESSION_SERIALIZABLE

For more information on isolation levels, please refer “[Daffodil DB System Guide](#)”.

Advanced JDBC Features

This section summarizes how Daffodil DB JDBC driver utilizes some of the advanced features of JDBC 3.0.

Savepoint

Daffodil DB JDBC Driver provides support for Savepoint which allow programmers to create subunits of transactions that can be managed independently. This introduces transaction flexibility and allows programmers to create if/then/else situations, which removes the "all or nothingness" of transactions. Once savepoint has been set, transaction can be rolled back to that savepoint without affecting the work done prior to the savepoint.

Example:

```
Int rowcount = stmt.executeUpdate ("update tx_ledger set counter =  
counter + 1");  
Savepoint sv1 = conn.setSavepoint ("svpoint1");  
rowcount = stmt.executeUpdate ("delete from student");  
conn.rollback (sv1);  
conn.commit ();
```

RowSet*

JDBC RowSet provides a disconnected, serializable, scrollable container for tabular data. A RowSet object can be thought of as a disconnected set of rows those are cached outside the DataSource. An important intended use of the RowSet is as a container for tabular data that can be exchanged amid different components of a distributed application, such as Enterprise JavaBeans components. Data contained in a RowSet may be updated and then resynchronized with the underlying tabular DataSource.

A [javax.sql.RowSet](#) object encapsulates a set of rows retrieved from a tabular DataSource. Because the RowSet interface includes an event notification mechanism and supports get and set properties, every RowSet object is a JavaBeans component. This means, for example, that a RowSet can be used as a JavaBeans component in visual JavaBeans development environment. As a result, a RowSet instance can be created and configured at design time, and its methods can be executed during run time.

Daffodil DB JDBC Driver supports both connected and disconnected types of RowSet.

To use RowSet object in a program, user needs to import [in.co.daffodil.db.rowset](#) package.

Example:

Following properties need to be set and the command that is set using *setCommand* property will be executed through execute property of RowSet.

```
rowset.setUsername("daffodil");  
rowset.setPassword("daffodil");  
rowset.setUrl("jdbc:daffodilDB_embedded:school");  
rowset.setCommand("select * from Post");  
rowset.execute();
```

* Features that are not supported in One\$DB

Connection Pooling

In the basic DataSource implementation, there is a one-to-one correspondence between the client's Connection object and the physical database connection. When the Connection object is closed, the physical connection is dropped as well.

The overhead of opening, initializing, and closing of the physical connection is incurred for each client session. Connection pool resolves this problem by maintaining a cache of physical database connections those can be reused across client sessions. Connection pooling greatly improves performance and scalability, particularly in a three-tier environment where multiple clients can share a smaller number of physical database connections. The application server provides its clients with an implementation of the DataSource interface that makes connection pooling transparent to the client. As a result, the client gets better performance and scalability while using the same JNDI and DataSource APIs as before.

Daffodil DB implements the following JDBC extensions to provide connection pooling support:

- ☐ javax.sql.ConnectionPoolDataSource
- ☐ javax.sql.PooledConnection

ConnectionPooling Implementation Class

[**in.co.daffodil.db.jdbc.DBPooledConnection**](#)

This is a class which provides implementation of ConnectionPooling

Example:

```
DBPooledConnection connPool =  
(DBPooledConnection)ds.getPooledConnection()
```

Updatable ResultSet

Daffodil DB JDBC Driver (with necessary support from Daffodil DB) supports updatable ResultSet for simple select queries and for all isolation levels. It also supports updatable ResultSet for simple queries having all types of Predicates and Aliases.

Example:

```
Statement1 = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
ResultSet.CONCUR_UPDATABLE);
```

Partial fetching in ResultSet

Daffodil DB JDBC Driver supports partial fetching in ResultSet. By knowing the number of rows of data to retrieve or fetch from a database, [Statement.setFetchSize](#) (for each ResultSet produced by the statement) or [ResultSet.setFetchSize](#) can be used to configure this parameter to minimize network traffic and improve database performance.

Retrieving Parameter Metadata

This capability allows programmers to dynamically describe the data types associated with parameters in SQL Statements. The ability to do this makes parameter handling even more consistent and efficient.

Example:

```
PreparedStatement ps = conn.prepareStatement ("{call  
MODIFY_TEACHER_SALARY (?,?)});  
ParameterMetaData pmd = ps.getParameterMetaData ();  
int colType = pmd.getParameterType (1);
```

where colType is a constant that is used to identify the JDBC Data Types. Entries of all such JDBC data types are stored in java.sql.Types.

Daffodil DB XA Support*

Daffodil DB is a J2EE-conformant component in a distributed J2EE system. As such, Daffodil DB is just a part of a larger system that includes, among other things, a JNDI server, a connection pool module, a transaction manager, a resource manager, and user applications. Within this system, Daffodil DB can serve as the **resource manager**. The term resource manager is often used while discussing distributed transactions. Resource manager is an entity that manages data or some other resource.

Daffodil DB distributed transaction support allows more than one database or connection to participate in the same transaction. Distributed transactions use XA DataSource entries rather than DataSource or ConnectionPooled DataSource entries. Such entries can participate in distributed transactions using the methods provided in the XA DataSource implementation.

In order to qualify as a resource manager in a J2EE system, J2EE requires the following basic areas of support:

- ☐ JNDI support
- ☐ Connection Pooling
- ☐ XA Support

You can use Daffodil DB in a Distributed Transaction Processing (DTP) environment to write Enterprise JavaBeans that are transactional across multiple Daffodil DB Servers. Workgroup environments, such as J2EE and J2SE where the data extends across multiple databases can benefit using Daffodil DB, because Daffodil DB JDBC driver supports the 2-phase commit protocol used by the Java Transactional API (JTA).

According to the X/Open's Distributed Transaction Processing (DTP) Model, a DTP environment specifies those application programs, which can use the Resource Manager and a Transaction Manager to access multiple data sources through one global transaction. In the JDBC, distributed transaction functionality is built on top of connection pooling functionality.

A distributed transaction system typically relies on an external transaction manager such as an application server that implements standard Java Transaction API functionality to coordinate the individual transactions. XA functionality is usually hidden from a client application, being implemented in a middle-tier environment such as an application server.

_ * Features that are not supported in One\$DB.

Distributed Transaction Concepts

Applications participating in global transactions that use connections cannot use normal connection instances like COMMIT, AutoCommit, or ROLLBACK functionality, because all COMMIT or ROLLBACK operations in a global transaction must be coordinated among all resource managers. Any attempt to use the *commit ()* or *rollback () method* or enabling the auto-commit of a connection instance would result in SQL exception. When you use XA functionality, the transaction manager uses XA resource instances to prepare and coordinate each transaction branch and then to commit or rollback all transaction branches appropriately.

XA functionality includes the following key components:

XA DataSource

These are extensions of connection pool DataSource and other DataSource, and are similar in concept as well as functionality. There will be one XA DataSource instance for each resource manager (database) that will be used in the distributed transaction. User needs to create XA DataSource instances in their middle-tier software. XA DataSource produces XA connections.

XA Connections

These are extensions of pooled connections. An XA connection encapsulates a physical database connection and individual connection instances are temporary handles to these physical connections. An XA connection instance from an XA DataSource instance can be obtained (using a get method) in your middle-tier software. Multiple XA connection instances from a single XA DataSource instance can be obtained if the distributed transaction involves multiple physical connections in the same database.

XA Resources

A transaction manager coordinating the transaction branches of a distributed transaction uses the XA resources. You will get one XA resource instance from each XA connection instance (using a get method), typically in your middle-tier software. There is a one-to-one correlation between XA connection instances and XA resource instances. Each XA resource instance has the functionality to start, end, prepare, commit, or roll back the operations of the transaction branch running in the session with which the XA resource instance is associated. The transaction manager uses XA resource instances to coordinate all the transaction branches that constitute a distributed transaction. Each XA resource instance provides the following functionality, invoked by the transaction manager:

- It associates and disassociates distributed transactions with the transaction branch operating in the XA connection instance that produces this XA resource instance. This is done by using transaction IDs.
- It performs two-phase commit functionality of a distributed transaction to ensure that changes are not committed in one transaction branch before there is an assurance that the changes will succeed in all transaction branches.

Distributed Transaction IDs

These are used to identify transaction branches. Each ID includes a transaction branch ID component and a distributed transaction ID component. This is how a branch is

associated with a distributed transaction.

All XA resource instances associated with a given distributed transaction will have a transaction ID that includes the same distributed transaction ID component.

Each transaction branch is assigned a unique transaction ID, which includes the following information:

- ☐ format identifier (4 bytes)
- ☐ global transaction identifier (64 bytes)
- ☐ branch qualifier (64 bytes)

A format identifier specifies a Java transaction manager. The 64-byte global transaction identifier value will be identical in the transaction IDs of all transaction branches belonging to the same distributed transaction. The overall transaction ID, however, is unique for every transaction branch. An XA transaction ID instance is an instance of a class that implements the standard [javax.transaction.xa.Xid](#) interface, which is a Java mapping of the X/Open transaction identifier Xid structure.

Classes related to Resource Managers

DataSource Factory Class

[in.co.daffodil.db.rmi.DataSourceFactory](#)

This is a class for Daffodil DB DataSource. System administrators can obtain DataSource (for remote access) objects from this class.

DataSource Implementation Class

[in.co.daffodil.db.jdbc.DBDataSource](#)

This is a class which provides implementation of DataSource in embedded mode.

[in.co.daffodil.db.rmi.RmiDaffodilDBDataSource](#)

This class provides the implementation of DataSource in network (rmi) mode.

XADataSource Interface Implementation

[in.co.daffodil.db.rmi.RmiDaffodilDBDataSource](#)

This class provides the functionalities of XA DataSource. The [javax.sql.XADataSource](#) interface outlines standard functionality of XA DataSource, which are factories for XA connections.

Xid Interface Implementation

[in.co.daffodil.db.jdbc.DBXID](#)

List of Properties required to set in DataSource\XADataSource

CreateDatabase	true/false
DatabaseHome	Path where database is to be created (required only for embedded mode)
DatabaseName	<name>
User	<user>
Password	<password>

Controlling JDBC Session properties

A JDBC session is defined as a set of database transactions performed using the same JDBC Connection object. From the JDBC driver perspective, a client session is represented by a JDBC Connection object.

A user can set following properties for a session:

Isolation Level

If a connection does not specify its isolation level, it inherits the default isolation level for Daffodil DB system. The default value for that property is `READ_COMMITTED`. When set to `READ_COMMITTED`, the connection inherits the `TRANSACTION_READ_COMMITTED` isolation level. When set to `SERIALIZABLE`, the connection inherits the `TRANSACTION_SERIALIZABLE` isolation level.

To override the inherited default, use [*java.sql.Connection.setTransactionIsolation \(\)*](#) method.

Daffodil DB supports following isolation levels for transactions:

- ☐ `TRANSACTION_SERIALIZABLE`
- ☐ `TRANSACTION_REPEATABLE_READ`
- ☐ `TRANSACTION_READ_COMMITTED`
- ☐ `TRANSACTION_READ_UNCOMMITTED`
- ☐ `SESSION_SERIALIZABLE`

Transaction Access Mode

User can change the access mode of a transaction to either read or write. In a read mode user can only read data from the database but cannot perform DML or DDL operations on the database. In write mode, the user can perform all kinds of operations on the database, based on rights given to the user.

For example:

[*Connection.setReadOnly \(Boolean val\);*](#)

If `val` is set to `true`, the transaction is set to read-only mode. If set to `false`, it is set to read-write mode.

Constraint Mode

If a database constraint had been defined as deferrable (a deferrable constraint can be immediate or deferred), then mode of the constraint can be converted from immediate to deferred and vice-versa during the course of the session. Every deferrable constraint has a default constraint mode. When you change the mode of a deferrable constraint, it is changed only for that particular session.

If constraint mode is deferred, then the constraint checking takes place when a transaction commits, or the constraint mode is explicitly changed to immediate by using the [*"Set Constraints name immediate"*](#) statement.

If constraint mode is immediate, then constraint checking effectively takes place for each SQL statement executed on the database.

The constraint mode can be changed by using *Statement.execute ()* method.

For example:

`statement1.execute ("set constraints constraint_name deferred")`

Avoiding Deadlocks

In a database, a deadlock is a situation in which two or more transactions are waiting for one another to give up their locks. For example, Transaction A may hold a lock on some rows in the **Accounts** table and needs to update some rows in the **Orders** table to finish. Transaction B holds locks on the same rows in the **Orders** table but needs to update the rows in the **Accounts** table held by Transaction A. Transaction A cannot complete its transaction because of the lock on **Orders** by Transaction B. Transaction B cannot complete its transaction because of the lock on **Accounts** by Transaction A. All activities come to a halt and remain standstill forever unless one of the transactions give up the locks it has acquired.

Using TRANSACTION_READ_COMMITTED isolation level (the default isolation level) is likely to avoid deadlocks. However, deadlocks are still possible.

But Daffodil DB application developers can avoid deadlocks by using **consistent application logic**. For example, transactions that access **Accounts** and **Orders** should always access the tables in the same order. So, in the scenario described above, Transaction B simply waits for transaction A to release the lock on **Orders**, before it begins. When transaction A releases the lock on **Orders**, Transaction B can proceed freely.

When a transaction waits for more than a specific amount of time to obtain a lock, either due to a deadlock situation, or due to a long running transaction, holding locks on the table/record required by the first transaction, it times out or aborts with appropriate exception code and message.

Limitations: - Deadlocks are detected only within a single Daffodil DB database. Deadlocks across multiple databases are not detected. Non-database deadlocks caused by Java synchronization primitives are also not detected by Daffodil DB.

End Note

Although this manual reflects the most current information possible, you should read the ***Daffodil DB Release Notes*** from time to time for latest information and updates on Daffodil DB.

Release Notes are available at: <http://www.daffodildb.com/daffodil-release-notes.html>

Sign Up for Support

If you have started working with Daffodil DB, please remember to sign up for the benefits you are entitled to as a Daffodil DB customer.

For free support, be a part of our online developer community at [Daffodil Developer Forum](#)

For buying support packages, please visit: <http://www.daffodildb.com/support-overview.html>

For more information regarding support, write to us at: support@daffodildb.com

We Need Feedback!

If you spot a typographical error in the *Daffodil DB JDBC Reference Guide*, or if you have thought of a way to make this manual better, we would love to hear from you!

Please submit a report in Bugzilla <http://www.daffodildb.com/bugzilla/index.cgi> OR write to us at: feedback@daffodildb.com

License Notice

© 2004, Daffodil Software Limited
All Rights Reserved.

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is for informational purpose only, and is liable to change without prior notice. Daffodil Software Limited assumes no responsibility or liability whatsoever for any errors or inaccuracies that may appear in this documentation. No part of this product or any other product of Daffodil Software Limited or related documentation may be stored, transmitted, reproduced or used in any other manner in any form by any means without prior written authorization from Daffodil Software Limited.