Novell DirXML® Driver for JDBC*

1.6.2

www.novell.com

IMPLEMENTATION GUIDE

January 15, 2004

Novell®

Legal Notices

Novell, Inc.
1800 South Novell Place
Provo, UT 84606
U.S.A.

www.novell.com

DirXML Driver for JDBC Implementation Guide
January 15, 2004

Online Documentation: To access the online documentation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc. in the United States and other countries.
DirXML is a registered trademark of Novell, Inc. in the United States and other countries.
eDirectory is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc. in the United States and other countries.
Novell is a registered trademark of Novell, Inc. in the United States and other countries.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Contents

About This Guide

The DirXML® Driver for Java* Database Connectivity (JDBC*) provides a generic solution for synchronizing data between Novell® eDirectory™ and relational databases.

This guide provides an overview of the driver's technology as well as configuration instructions.

Additional Documentation

For documentation on using DirXML and the other DirXML drivers, see the DirXML Documentation Web site (http://www.novell.com/documentation/lg/dirxmldrivers).

Documentation Updates

For the most recent version of this document, see the DirXML Drivers Documentation Web site (http://www.novell.com/documentation/lg/dirxmldrivers/index.html).

Documentation Conventions

In this documentation, a greater-than symbol (>) is used to separate actions within a step and items within a cross-reference path.

A trademark symbol (®, ™, etc.) denotes a Novell trademark. An asterisk (*) denotes a third-party trademark.

User Comments

We want to hear your comments and suggestions about this manual and the other documentation included with Novell DirXML. To contact us, send e-mail to proddoc@novell.com.

1 Introducing the DirXML Driver for JDBC

The DirXML® Driver for Java Database Connectivity (JDBC), subsequently referred to as the driver, provides a generic solution for synchronizing data between Novell® eDirectory™ and JDBC-accessible relational databases.

The principal value of this driver resides in its generic nature. Unlike most DirXML drivers that interface with a single, well-defined application, this driver can interface with most relational databases and database-hosted applications.

Overview

In this section, you will find information on the following topics:

New Features

Driver Features

- Referential attribute support can now be disabled.

Driver Bug Fixes

- Referential attribute support can now be disabled via the "Enable Referential Support?" driver parameter. This allows this driver to be backwards compatible with the 1.0 driver.
- Objects can now be added on the publisher channel without <add-attr> child elements.
- Publication <add> events followed by <modify> events are no longer optimized out. This ensures backwards compatibility with the 1.5 driver.

DirXML 2.0 Features

DirXML 2.0 includes the following new features. For more information, refer to the *Nsure Identity Manager 2 Administration Guide* (http://www.novell.com/documentation/lg/dirxml20/admin/data/alxnk27.html).

Password Management

The new password management framework includes the following benefits:

- New Password Policies let you create rules for passwords and assign them to users, containers, or the whole eDirectory tree. You can enable Universal Password, which lets you enforce detailed criteria for passwords and allows for special characters.
- Password Synchronization 2.0 is now cross-platform, and it lets you enforce your Password Policies across connected systems. New notification templates let you automatically send messages to users about their password synchronization status.
- Using Password Policies, you can also provide Forgotten Password Self-Service and Reset Password Self-Service to your users. These new features can help you reduce help desk calls. Notification templates are also included for automatically sending forgotten password and password hint messages to users.

Policy Builder Interface and DirXML Script for Creating Policies

For the most common tasks, you can now use the new Policy Builder interface to create policies for your driver without writing XSLT code. The Policy Builder helps you set up policies using the new DirXML Script.

Role-Based Entitlements

For many drivers, Role-Based Entitlements is an option in the sample driver configuration that you can choose when importing the driver.

Role-Based Entitlements let you grant entitlements on connected systems to a group of Novell® eDirectory™ users. Using Entitlement Policies, you can streamline management of business policies and reduce the need to configure your DirXML drivers.

Novell Nsure Audit

Novell Nsure™ Audit is a centralized, cross-platform auditing service. It collects event data from multiple applications across multiple platforms and writes the data to a single, non-repudiable data store. Nsure Audit is also capable of creating filtered data stores. Based on criteria you define, Nsure Audit captures specific types of events and writes those events to secondary data stores.

Global Configuration Values

Global configuration values (GCVs) are new settings that are similar to driver parameters. Global configuration values can be specified for a driver set as well as an individual driver. If a driver does not have a value for a particular GCV, the driver inherits the value for that GCV from the driver set.

GCVs allow you to specify settings for new DirXML features such as Password Synchronization, as well as settings that are specific to the function of an individual driver configuration. Some GCVs are provided with the drivers, but you can also add your own. You can refer to these values in a policy to help you customize your driver configuration.

Driver Heartbeat

The DirXML engine now accepts driver heartbeat documents from drivers, and drivers can be configured to send them.

Driver Concepts

The following are some important terms and concepts you should know before installing and configuring the driver:

DirXML Driver for JDBC

The driver consists of three files: JDBCShim.jar, JDBCUtil.jar, and CommonDriverShim.jar. In addition to these files, you will need a third-party JDBC driver to communicate with each respective database.

Third-Party JDBC Driver

One of the numerous JDBC implementations used by the driver to communicate with a particular database. For example, classes12.zip is one of Oracle's JDBC drivers.

JDBC Driver Type

There are four types of third-party JDBC drivers:

1. A third-party JDBC driver that is partially Java and communicates indirectly with a database through an ODBC driver. Type 1 drivers serve as a JDBC-ODBC bridge. Sun provides a JDBC-ODBC bridge driver for experimental use and for situations when no other type of third-party JDBC driver is available.
2. A third-party JDBC driver that is partially Java and communicates indirectly with a database through its native client APIs.
3. A third-party JDBC driver that is pure Java and communicates indirectly with a database through a middleware server.
4. A third-party JDBC driver that is pure Java and communicates directly with a database.

Type 3 and 4 drivers are generally more stable than type 1 and 2 drivers. Type 1 and 2 drivers are generally faster than type 3 and 4 drivers. Type 2 and 3 drivers are generally more secure than type 1 and 4 drivers. If you choose to use a type 1 or type 2 driver, you must use the remote loader to ensure the integrity of the directory process.

Directory Schema

The set of object classes and attributes in the directory. For example, the eDirectory User class and Given Name attribute are part of the eDirectory schema.

Application Schema

The set of classes and attributes in an application. Because databases have no concept of classes or attributes, the driver maps eDirectory classes to views or tables and eDirectory attributes to columns.

Synchronization Schema

The database schema visible to the the driver.

Logical Database Class

The set of tables or views used to represent an eDirectory class in a database.

Database Concepts

In the following section, you will learn about important database concepts, including:

Database Schema

A database schema is a set of database objects, such as tables, views, stored procedures, and so forth that are owned by a particular database user.

Ownership is often expressed using a dot notation such as `dirxml.emp`, where `dirxml` is the name of the database user that owns the table `emp`. All of the database objects owned by `dirxml` constitute the dirxml database schema.

Data Manipulation Language

Data Manipulation Language (DML) statements are highly standardized statements that manipulate database data. DMLs are essentially the same regardless of the database you use.

The following example shows several DML statements:

```
SELECT * FROM emp;
INSERT INTO emp(lname) VALUES('Doe');
UPDATE emp SET fname = 'John' WHERE empno = 1;
```

NOTE: Examples used throughout the implementation guide are for the Oracle* database.

Data Definition Language

Data Definition Language (DDL) statements manipulate database objects such as tables, indexes, user accounts, and so forth. DDL statements are proprietary and differ substantially between databases.

The following example shows a DDL statement:

```
CREATE TABLE emp
(
     empno NUBMER(8),
     fname VARCHAR2(64),
     lname VARCHAR2(64)
);

CREATE USER dirxml IDENTIFIED BY novell;
```

Transactions

A transaction is an atomic database operation that consists of one or more statements. When a transaction is complete, all statements in the transaction are committed. When a transaction is interrupted or one of the statements in the transaction has an error, the transaction is said to roll back. When a transaction is rolled back, the database is left in the same state it was before the transaction began.

Transactions are either manual (user-defined) or automatic. Manual transactions can consist of one or more statements and must be explicitly committed. Automatic transactions consist of a single statement and are implicitly committed after each statement is executed.

Manual Transactions

Manual transactions usually contain more than one statement. DDL statements typically cannot be grouped with DML statements in a manual transaction. The following example shows a manual transaction:

```
INSERT INTO emp(lname) VALUES('Doe');
UPDATE emp SET fname = 'John' WHERE empno = 1;
COMMIT; /* explicit */
```

Automatic Transactions

Automatic transactions consist of only one statement. They are often referred to as auto-committed statements because changes are implicitly committed after each statement. When a statement runs automatically, it is autonomous of any other statement. The following example shows an automatic transaction:

```
INSERT INTO emp(lname) VALUES('Doe');
/* COMMIT; implicit */
```

Triggers

A database trigger is programmable logic that is associated with a table, which fires or executes under certain conditions. Triggers are often useful for creating side effects in a database. The following is an example of a database trigger on table `emp`.

```
CREATE TABLE emp
(
     empno NUMBER(8),
```

```
     fname VARCHAR(64),
     lname VARCHAR(64)
);

CREATE TRIGGER t_emp_insert
     AFTER INSERT ON emp
     FOR EACH ROW

BEGIN
     UPDATE emp SET fname = 'John';
END;
```

When a statement is executed against a table with triggers, a trigger will fire if the statement satisfies the conditions specified in the trigger. For example, using the above table, if the following insert statement were executed,

```
INSERT INTO emp(LNAME) VALUES('Doe')
```

Trigger `t_emp_insert` would fire after the insert statement is executed and the following update statement would also be executed:

```
UPDATE emp SET fname = 'John'
```

A trigger can typically be fired before or after the statement that triggered it. Statements that are run as part of a database trigger are typically included in the same transaction as the triggering statement. In the above example, both the insert and update statements would be committed or rolled back together.

Identity Columns/Sequences

Identity columns and sequences are used to generate unique primary key values.

An identity column is a self-incrementing column used to uniquely identify a row in a table. Identity columns values are automatically filled-in when a row is inserted into a table.

A sequence object is a counter that can be used to uniquely identify a row in a table. Unlike an identity column, a sequence object is not bound to a single table. If used by a single table, however, a sequence object can be used to achieve an equivalent result.

The following is an example of a sequence object:

```
CREATE SEQUENCE seq_empno
    START WITH 1
    INCREMENT BY 1
    NOMINVALUE
    NOMAXVALUE
    CACHE 100
    ORDER;
```

Stored Procedures/Functions

A stored procedure or function is programmable logic stored in a database. Unlike triggers, stored procedures or functions are not associated with a table. They can be invoked from almost any context.

The subscriber can use stored procedures or functions to retrieve primary key values from rows inserted into tables for the purpose of creating associations. Stored procedures or functions can also be invoked from within embedded SQL statements or triggers.

The distinction between stored procedures and functions varies by database. Typically, both can return output. How they return output is at issue. Stored procedures usually return values through parameters. Functions usually return values through a result set.

The following is an example of a stored procedure:

```
CREATE
PROCEDURE sp_empno
(
    io_empno IN OUT NUMBER,
    i_fname IN VARCHAR2
)
IS
BEGIN
    IF (io_empno IS NULL) THEN
        SELECT seq_empno.nextval INTO io_empno FROM DUAL;
END IF;
END sp_empno;
```

Data Synchronization Models

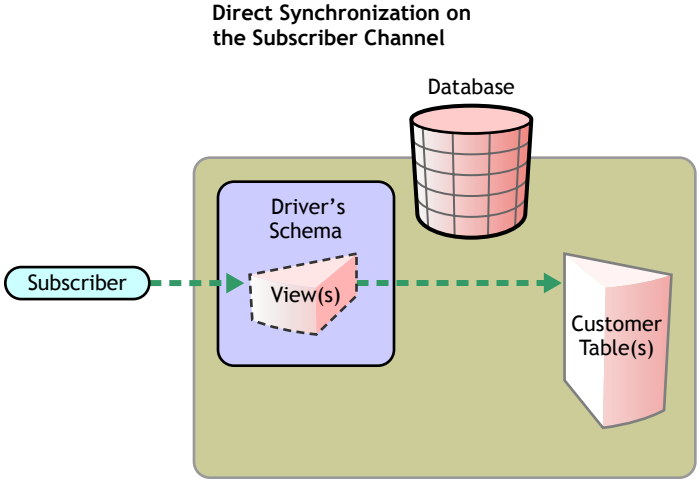
The driver supports two data synchronization models: direct and indirect. The following sections describe how direct and indirect synchronization work on both the Subscriber and Publisher channels.

Direct Synchronization

Direct synchronization uses views to synchronize information between eDirectory and a database. The following information explains how direct synchronization works on the Subscriber and Publisher channels.

In the following scenarios, you can have one or more customer tables or views.

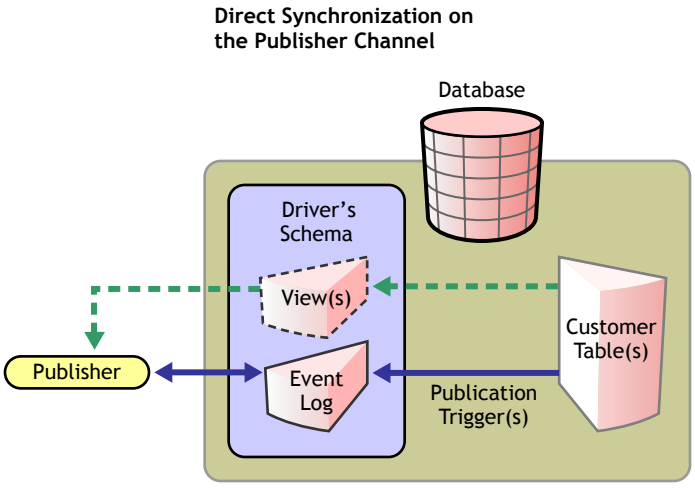
Subscriber Channel



The subscriber updates customer tables through a view in the driver's schema. A view can be used to synchronize directly with customer tables.

NOTE: Direct synchronization without a view is only possible if all columns of interest are located in the same table and if that table has an explicit primary key constraint.

Publisher Channel



When a customer table is updated, publication triggers insert rows into the event log table. The publisher then reads the inserted rows and updates eDirectory.

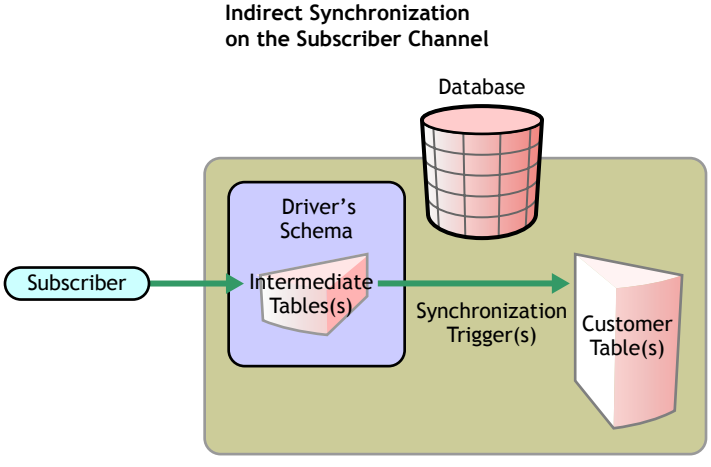
Depending on the contents of the rows read from the event log table, the publisher might need to retrieve additional information from the view before updating eDirectory. After updating eDirectory, the publisher then deletes or marks the rows as processed.

Indirect Synchronization

Indirect synchronization uses intermediate tables to synchronize information between eDirectory and a database. The following information explains how indirect synchronization works on the Subscriber and Publisher channels.

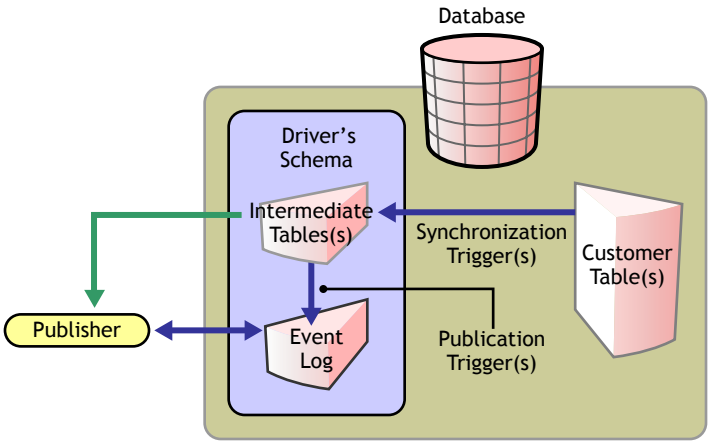
In the following scenarios, you can have one or more customer tables or intermediate tables.

Subscriber Channel



The subscriber updates the intermediate table in the driver's schema. The synchronization triggers directly update the customer tables elsewhere in the database.

Indirect Synchronization
on the Publisher Channel



When customer tables are updated, synchronization triggers update the intermediate tables. Publication triggers then insert one or more rows into the event log table. The publisher then reads the inserted rows and updates eDirectory.

Depending on the contents of the rows read from the event log table, the publisher might need to retrieve additional information from the intermediate tables before updating eDirectory. After updating eDirectory, the publisher then deletes or marks the rows as processed.

2 Understanding Driver Prerequisites

The following sections contain important information you should review before installing and configuring the driver.

Driver Prerequisites

The DirXML® Driver for JDBC requires the following:

- Novell® eDirectory™ 8.6.2 or higher
- DirXML 1.1a or higher
- ConsoleOne® 1.3.3 or higher or Novell iManager 1.5
- Java Virtual Machine (JVM*) 1.2 or higher
- A third-party JDBC driver

Supported Platforms

The driver runs on all DirXML-enabled platforms, including Windows* NT*/2000, NetWare®, Solaris*, and Linux*.

Supported Databases

The driver uses the JDBC 1.0 API to execute SQL statements and obtain metadata from a database. As such, a database must be JDBC-accessible. The following databases have been tested and are recommended for use with this product:

Database	Version
IBM* DB2 Universal Database (UDB)	7.2 or higher
Microsoft* SQL Server 2000	Service Pack 2 or higher

Database	Version
Microsoft SQL Server 7	Service Pack 4
Oracle 8i	Release 3 (8.1.7)
Oracle 9i	Release 2 (9.2.0.1) or higher
Sybase* Adaptive Server Enterprise (ASE)	12.5 or higher
MySQL*	3.23
Informix* Dynamic Server (IDS)	9.30 or higher

You can use other databases; however, they must meet the following minimum requirements:

- Support the SQL-92 entry level grammar.
- Support triggers or some auditing capability suitable for event capture and replication (publication only).

Recommended Third-Party JDBC Drivers

We recommend using type 3 or type 4 third-party JDBC drivers whenever possible. We also recommend using the latest version of these drivers. If you choose to use a type 1 or type 2 driver, you must use the remote loader to ensure the integrity of the directory process.

The following third-party drivers have been tested and are recommended for use with the DirXML Driver for JDBC:

Driver Name	Version
Oracle 8i JDBC Driver	8.1.7.1
Oracle 9i JDBC Driver	9.2.0.1 or later
BEA* Weblogic* Type 4 jDriver for Microsoft SQL Server 7/2000	5.1.0, Service Pack 11 or later
Sybase jConnect JDBC Driver	5.5 or later
Microsoft SQL Server 2000 Driver for JDBC	2.2 or later
Informix JDBC Driver	9.3 or later
MySQL Connector/J	2.0.14 or later
IBM Type 3 JDBC Driver for DB2 UDB	7.2 or later

The following third-party JDBC drivers have been tested, but are not recommended for use with this product:

- Sun Type 1 JDBC-ODBC Bridge driver (JRE 1.2)

We strongly recommend that you use the recommended third-party drivers whenever possible.

Minimum Third-Party JDBC Driver Requirements

The driver might not work with all third-party drivers. If you choose to use another third-party driver, it must meet the following requirements to work with the DirXML Driver for JDBC:

- Support required metadata methods.

  Refer to Appendix D, “java.sql.DatabaseMetaData Methods,” on page 91 for a current list of the required and optional java.sql.DatabaseMetaData method calls made by the driver. This list of requirements might be extended in future releases. As such, all java.sql.DatabaseMetaData methods should be supported. If the third-party driver does not meet these requirements, you might have to purchase a different third-party driver in the future.

- Return accurate data from select statements.
- Correctly execute the insert, update, and delete statements issued by the driver.

Refer to Appendix E, “JDBC 1.0 Methods,” on page 93 for a list of JDBC methods used by the driver. This list can be used in collaboration with third-party driver documentation to identify potential incompatibilities.

Considerations When Using Other Third-Party JDBC Drivers

- Because the driver is dependent upon third-party drivers, bugs in those drivers might cause the driver to malfunction. In order to assist you in debugging third-party drivers, the driver's trace output has been enhanced to include JDBC API-level tracing (level 5) and third-party driver tracing (level 6).
- Stored procedure or function support and connectivity (specifically reconnection) are likely points of failure.

Using The Sun JDBC-ODBC Bridge Driver

Because of the increased instability inherent in using an ODBC driver and known issues with the 1.3.*x* Java Runtime Environment (JRE) JDBC-ODBC Bridge driver, We strongly recommend using a pure Java (type 3 or 4) JDBC driver whenever possible. If you choose to use a type 1 or type 2 driver, you must use the remote loader to ensure the integrity of the directory process.

The principle disadvantage to using a type 1 JDBC bridge driver and a native ODBC driver is increased instability. Errors in the native libraries from the ODBC driver imported through the JDBC bridge driver could bring down the directory.

The driver and JDBC-ODBC Bridge driver might not work with all third-party ODBC drivers. The list of third-party JDBC driver requirements applies to ODBC drivers as well. Refer to “Recommended Third-Party JDBC Drivers” on page 22 for more information.

- If you choose to use the Microsoft ODBC driver for SQL Server (SQLSRV32.DLL), We recommend installing the latest version of Microsoft Data Access Components (MDAC). MDAC can be downloaded from free from Microsoft’s Web site. (http://www.microsoft.com/data/download.htm).
- The Bridge driver included in the JRE (Java Runtime Environment) 1.3.x contains a known defect regarding IN OUT stored procedure parameters. Calling a stored procedure with IN OUT parameters results in a memory access violation that brings down the directory. The recommended solution to this problem is to use the 1.2.x JRE with DirXML. However, doing so will reduce the performance of all drivers running on the server. DirXML supports the use of Hotspot only with the 1.3.x JRE or greater. 1.4.x JRE has not been tested with the driver.

Security

In order to ensure that a secure connection exists between the driver and a third-party driver, we recommends that you run the driver remotely.

When the driver cannot run remotely, you might want to use a type 2 or type 3 JDBC driver. These driver types often facilitate a greater degree of security through middleware servers or client APIs than other JDBC driver types.

Known Issues

This section lists the current known driver issues.

General

- Some databases, such as Sybase and DB2, have proprietary time stamp formats that cannot be parsed by the java.sql.Timestamp class.

  When synchronizing time stamp columns from these databases, time stamp values placed in the event log table should be in ODBC canonical format (i.e., yyyy-mm-dd hh:mm:ss.ffffffff). Alternatively, these values can be converted to ODBC cannonical format via style sheets. When time stamps are used as primary keys, however, time stamp values must be placed in the event log table in ODBC cannonical format. Time stamp values can be reformatted on the database using a general-purpose programming language, such as Java, or the database's native SQL programming language.

- When eDirectory Time and Timestamp syntaxes are interpreted as signed integers, they cannot store dates before Jan 1, 1902 or after Jan 1, 2038.

IBM DB2

- After applying an IBM Fixpack to your DB2 server, you should use the the updated db2java.zip file on the database server with the driver. Otherwise, you might receive connectivity errors such as "CLI0601E Invalid statement handle or statement is closed."

JDBC-ODBC Bridge

- The Bridge driver included in the JRE (Java Runtime Environment) 1.3.x contains a known defect regarding IN OUT stored procedure parameters. Calling a stored procedure with IN OUT parameters results in a memory access violation that brings down the directory. The recommended solution to this problem is to use the 1.2.2 JRE with DirXML. However, doing so will reduce the performance of all drivers running on the server. DirXML only supports the use of Hotspot with the 1.3.x JRE or greater.

Oracle

- You may experience high CPU utilitzation problems when executing embedded SQL statements unless you place a `jdbc:type` attribute on each `<jdbc:statement>` element. A general means of avoiding this problem is to set the driver parameter Handle Statement Results to **`single`**.

- The 8.1.6 version of the Oracle JDBC driver has a bug that effects publication. It incorrectly returns NULL values for some fields in the event log table when their values are actually non-NULL.

  The net effect is that the driver treats several rows as No Operations or NOOPs, in which the rows are ignored, and the driver produces an incomplete publication document. Earlier

versions might also exhibit the same problem. We recommend that you use the 8.1.7 version because it is backwards-compatible with most versions of Oracle 8.

- In order to connect to older versions of Oracle on NetWare (for example, Oracle 8.0.3), you must use the CLASS111.zip JDBC driver supplied on the Oracle installation CD.

Microsoft SQL Server

- Microsoft's ODBC driver for SQL Server returns an ambiguous type java.sql.Types.OTHER for data types NVARCHAR, NCHAR, NTEXT, and UNIQUEIDENTIFIER. The driver assumes type java.sql.Types.OTHER to be NVARCHAR, NCHAR, or NTEXT. Because of this, type UNIQUEIDENTIFIER is not supported.
- Microsoft's SQL Server 2000 Driver for JDBC issues the following error when the driver parameter driver parameter Reuse Statements is set anything other than **no**: "Can't start manual transaction mode because there are cloned."

Sybase

- In order to ensure ANSI-compliant padding and truncation behavior for binary values, binary columns (other than image) must be exactly the size of the eDirectory attribute that maps to them, constrained NOT NULL, and added to the publisher or subscriber Create rule. If they are constrained NULL, trailing zeros, which are significant to eDirectory, will be truncated. If binary columns exceed the size of their respective eDirectory attributes, extra 0s will be appended to the value.

MySQL

- TIMESTAMP columns, when updated after being initially set to 0 or NULL, are always set to the current date and time. To compensate for this behavior, we recommend you map eDirectory Time and Timestamp syntaxes to DATETIME columns.
- Publication is not supported. MySQL does not support the query used by the publisher to retrieve events from the Event Log table.

Informix

- NUMERIC or DECIMAL columns cannot be used as primary keys unless the scale (that is, the number of columns to the right of the decimal point) is explicitly set to zero when the table is created. By default, the scale is set to 255.

Limitations

The following section lists the known driver limitations.

- The driver does not support the use of delimited database identifiers.
- Direct synchronization (using views) does not support multi-valued or referential attribute synchronization.
- Informix Dynamic Server databases created with LOG MODE ANSI are not supported. Databases created with this option use delimited identifiers for user/schema names. The driver does not currently support delimited identifiers.
- Publication is not supported. MySQL does not support the query used by the publisher to retrieve events from the Event Log table.
- JDBC 2.0 data types are not supported with the exception of CLOB and BLOB.

3 Installing or Upgrading the Driver

In this section, you will find information and procedures to help you install and upgrade the driver:

Installing the Driver

The DirXML® Driver for JDBC requires Novell® eDirectory™ and database-side configuration. We recommend that you configure your database and test it independently of the driver.

You should use these instructions if no previous installation exists for the driver. After you have downloaded the CD image, complete the following instructions to install the driver:

Installing the Driver

1. Shut down eDirectory.
2. Copy JDBCShim.jar, JDBCUtil.jar, and CommonDriverShim.jar into the appropriate directory for your platform. Use the following table to determine the appropriate directory:

Platform	Directory Path
NetWare®	SYS:\SYSTEM\LIB
Solaris or Linux	/usr/lib/dirxml/classes
Windows NT/2000	NOVELL\NDS\LIB

3. Copy the appropriate third-party driver files into the same directory you specified in the previous step.

   NOTE: These third-party files are not bundled with the driver. These files should be properly licensed, if necessary, when used in a production environment.

   The following table includes third-party driver download information, by vendor.

Vendor	Database	Filename(s)	Download Instructions
Oracle	Oracle 8i	classes12.zip, nls_charset12.zip	Oracle Technology Network (http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html) You must first register for free with Oracle's Technology Network. You should download version 8.1.7.1 or later.
Oracle	Oracle 9i	classes12.zip, nls_charset12.zip	Oracle Technology Network (http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html) You must first register for free with Oracle's Technology Network. You should download version 9.2.0.1 or later.
BEA Systems	Microsoft SQL Server 7/ 2000	unspecified	BEA Download Center (http://commerce.bea.com/downloads/weblogic_server.jsp) This driver requires free registration and expires on a monthly basis. You should select the JDBC Drivers product and download MSSQLServer4 Kit, Version 7 and 2000. You will need to zip/jar the file yourself. To do this: 1. Place the weblogic\mssqlserver4v70\license\WeblogicLicense.xml file into the \weblogic\smsqlserver4v70\classes directory. 2. Zip/jar the ...\classes\weblogic directory, including its contents, and the ...\classes\WeblogicLicense.xml file. This directory and XML file must be at the root of the zipped/jarred file.
Microsoft	SQL Server 2000	msbase.jar mssqlserver.jar msutil.jar	Microsoft Downloads (http://www.microsoft.com/sql/downloads/2000/jdbc.asp) Click the appropriate download option for your platform.
IBM	DB2 Universal Database (UDB) 7.2	db2java.zip	IBM Downloads (http://www-4.ibm.com/software/data/db2/udb/downloads.html) This driver is part of the free DB2 UDB Personal Developer's Edition download. If you apply Fixpacks, ensure that you use the db2java.zip file on the patched server and not the file supplied with the original download.
Sybase	Adaptive Server Enterprise (ASE) 12.5	jconn2.jar	Sybase Downloads (http://www.sybase.com/downloads) To download this driver, select jConnect for JDBC.
MySQL	MySQL 3.23	mysql-connector-java-2.0.14-bin.jar	MySQL Downloads (http://www.mysql.com/downloads/api-jdbc.html) Select MySQL Connector/J2 or later.
Informix	Informix Dynamic Server (IDS) 9.3	ifxjdbc.jar	not available for download

Vendor	Database	Filename(s)	Download Instructions
Sun	N\A	N\A	Sun Downloads (http://java.sun.com/j2se/downloads.html) Sun's JDBC-ODBC Bridge driver is freely available as part of the Java Runtime Environment (JRE).

4. Restart eDirectory.
5. Start ConsoleOne or iManager.

Importing a Preconfigured Driver

The preconfigured drivers are the example configurations only. We recommend that you install a preconfigured configuration and run it before customizing the driver. Preconfigured driver files are provided for the following databases:

Database	1.6.1 or Earlier Filename	1.6.2 Filename
Oracle 8i, 9i	JDBCOracleDirect.xml JDBCOracleIndirect.xml	JDBCOracle.xml
Microsoft SQL Server 7/2000	JDBCMSSQLDirect.xml JDBCMSSQLIndirect.xml	JDBCMSSQL.xml
IBM Universal Database (UDB) 7.2	JDBCDB2Direct.xml JDBCDB2Indirect.xml	JDBCDB2.xml
Sybase Adaptive Server Enterprise (ASE) 12.5	JDBCSybaseDirect.xml JDBCSybaseIndirect.xml	JDBCSybase.xml
MySQL 3.23	JDBCMySQLIndirect.xml	JDBCMySQL.xml
Informix Dynamic Server (IDS) 9.3	JDBCInformixDirect.xml JDBCInformixIndirect.xml	JDBCInformix.xml

All configurations use the same conventions, regardless of database:

- String field lengths are 64 characters. Fields of this length can hold most eDirectory attributes. You may want to refine field lengths to enhance storage efficiency.
- Primary key field lengths are 8 digits.
- The `record_id` column in the `eventlog` table has the maximum numeric precision permitted by each database.
- All table, trigger, stored procedure, index, and constraint names are lowercase. This is the most commonly used case convention.
- Triggers names are prefaced with `t_`, stored procedure names are prefaced with `sp_`, index names are prefaced with `i_`, check constraints are prefaced with `chk_`, primary key constraints are prefaced with `pk_` and foreign key constraints are prefaced with `fk_`.

- Check, primary, and foreign key constraints follow this naming convention: *prefix_table-name_column-name* (for example, pk_emp_empno, fk_phone_empno, chk_eventlog_event_type)
- Triggers follow this naming convention: t_*table-name_operation* (for example, t_emp_insert)
- Indexes follow this naming convention: i_*table_name_number* (for example, i_eventlog_1)
- Identity columns and sequence objects cache 100 values.
- Usernames are the last name of a User concatenated with a primary key value (for example, John Doe's username could be Doe1).
- Initial passwords are the last name of a User (for example, John Doe's password would be `Doe`). Sybase passwords must be at least 6 characters long. When shorter than 6 characters, last names are padded with the character 'p' (for example, John Doe's password would be `Doeppp`). The padding character can be adjusted in the Subscriber Command Transformation style sheet.

Importing the Driver Configuration

The driver configuration (XML) file creates and configures the objects needed in order for driver work properly. It also includes sample rules and style sheets you can modify for your implementation.

1. In iManager, select DirXML Management > Create Driver.
2. Select a driver set.

   If you place this driver in a new driver set, you must specify a driver set name, context, and associated server.
3. Mark Import a Preconfigured Driver from the Server and select the .xml file.

   The driver configuration file is installed on the Web server when you set up iManager.
4. You will be prompted to enter a name for the driver. Enter the driver's name and click Next to continue.
5. (Optional) Click Define Security Equivalences.

   5a Click Add, then select an object with Admin rights (or any other rights that you want the driver to have).

   5b Click Apply, then click ok.
6. (Optional) Click Exclude Administrative Roles to exclude objects from replication.

   6a Click Add, then select any users you want to exclude (such as the admin user).

   6b Click Apply, then click ok.
7. Click Next to view the import summary. Verify that the configuration is correct, then click Finish with Overview.

The necessary DirXML objects have now been created. If you didn't define security equivalences or exclude administrative users during the import, you can complete these tasks by modifying the driver object's properties.

Installing Database Objects

The following information explains how to install and configure database objects (tables, triggers, indexes, and so forth) for synchronization with the default, preconfigured driver.

SQL scripts are located in the tools\sql*database* directory.

This section contains information to help you:

IMPORTANT: We recommend installing or uninstalling preconfigured drivers and database scripts as a unit. To prevent unintentional mismatching, database scripts and preconfigured drivers now contain headers with a version number, the target database name, and the database version.

For uninstallation information, refer to Chapter 7, "Uninstalling the Driver and Database Objects," on page 79.

Configuring Oracle Objects

1. From an Oracle client, such as SQL Plus, logon as user **SYSTEM**. By default, the password for SYSTEM is **MANAGER**.
2. Execute the first installation script for direct or indirect synchronization. For example:

   SQL> @c:\tools\sql\oracle\direct\INSTALL_DIRECT_1.sql

   SQL> @c:\tools\sql\oracle\indirect\INSTALL_INDIRECT_1.sql
3. Log on as user **dirxml** using **dirxml** as the password.
4. Execute the second installation script for direct or indirect synchronization. For example:

   SQL> @c:\tools\sql\oracle\direct\INSTALL_DIRECT_2.sql

   SQL> @c:\tools\sql\oracle\indirect\INSTALL_INDIRECT_2.sql

NOTE: Before executing the provided publication tests as SYSTEM, you must log in and create a new session. Otherwise, you won't be able to see the sequence objects owned by dirxml.

If the scripts execute correctly, you should see notifications that the database objects have been created. If there are errors, ensure that you are logged in as the correct user. Before re-running the installation scripts, be sure to execute the uninstallation script (for example, UNINSTALL_DIRECT.sql or UNINSTALL_INDIRECT.sql).

Troubleshooting Tips

- When generating events for publication, make sure you are logged in as someone other than the dirxml user. If you make changes as the dirxml user, your changes will not be published.
- Be sure to commit changes. Until you commit your changes, they will not be published.

Configuring Microsoft SQL Server Objects

1. Start Query Analyzer.
2. Log on to your database server as user **sa**. By default, the sa user has no password.
3. Open and execute the first script for direct or indirect synchronization. For example:

   tools\sql\mssql\direct\INSTALL_DIRECT_1.sql

   tools\sql\mssql\indirect\INSTALL_INDIRECT_1.sql
4. Log on to your database server as user **dirxml** using **dirxml** as the password.
5. Open and execute the second installation script for direct or indirect synchronization. For example:

   tools\sql\mssql\direct\INSTALL_DIRECT_2.sql

   tools\sql\mssql\indirect\INSTALL_INDIRECT_2.sql

Troubleshooting Tips

- When generating events for publication, make sure you are logged in as someone other than the dirxml user. If you make changes as the dirxml user, your changes will not be published.
- Be sure to commit changes. Until you commit your changes, they will not be published. The keyword for commit for Microsoft SQL Server is go.
- Make sure you are logged in as the correct user in the correct database when running the scripts.

Configuring IBM DB2 Objects

For DB2 Universal Database, you must manually create a database user account and database before running the provided scripts.Because the process of creating user accounts differs between operating systems, Step 1 below is OS-specific. These instructions are for a Windows NT operating environment. If you reinstall the database objects, you only need to repeat Step 6 through Step 8.

1. Create a user account for user **dirxml** using **dirxml** as the password in User Manager for Domains.
   - Remember to uncheck the User Must Change Password at Next Login check box for this account.
   - You might want to also check the Password Never Expires check box.

   NOTE: The remaining instructions are OS-independent.
2. Start the Control Center.
3. Right-click Databases > click Create Database Using Wizard.
4. Name the database **dirxml** > then click Finish.
5. Copy JDBCUtil.jar to your DB2 server.
6. Start the Command Center from the Control Center.
7. Change the name of the administrator account and password for your server before executing the first installation script.

8. Click the Script tab > open the Script menu > import and execute the script for direct or indirect synchronization. For example:

   tools\sql\db2\direct\INSTALL_DIRECT_1.sql

   tools\sql\db2\indirect\INSTALL_INDIRECT_1.sql

9. Import the second installation script for direct or indirect synchronization. For example:

   tools\sql\db2\direct\INSTALL_DIRECT_2.sql

   tools\sql\db2\indirect\INSTALL_INDIRECT_2.sql

10. Adjust the path to JDBCUtil.jar and execute the script.

Troubleshooting Tips

- When generating events for publication, make sure you are logged in as someone other than the `dirxml` user. If you make changes as the `dirxml` user, your changes will not be published.
- Make sure you commit your changes. Until you commit changes, they won't be published.

Configuring Sybase Objects

This section explains how to install database objects on Sybase Adaptive Server Enterprise (ASE).

If you haven't installed JDBC support on your Sybase server, you should complete this task. Refer to your server's installation manual for further details. If installation is required, you should execute the sql_server*.sql script to install java.sql.DatabaseMetaData support.

1. From a Sybase client, such as isql, logon as user **`sa`** and execute the first installation script for direct or indirect synchronization. By default, the `sa` account has no password. For example:

   isql -U sa -P -i tools\sql\sybase\direct\INSTALL_DIRECT_1.sql

   isql -U sa -P -i tools\sql\sybase\indirect\INSTALL_INDIRECT_1.sql

2. Log on as user **`dirxml`** using **`dirxml`** as the password and execute the second installation script for direct or indirect synchronization. For example:

   isql -U dirxml -P dirxml -i tools\sql\sybase\direct\INSTALL_DIRECT_2.sql

   isql -U dirxml -P dirxml -i tools\sql\sybase\direct\INSTALL_INDIRECT_2.sql

Troubleshooting Tips

- Make sure you commit your changes. Until you commit changes, they won't be published. The keyword for commit for Sybase is `go`.

Configuring MySQL Objects

1. From a MySQL client, such as mysql, log on as user **`root`** or another user with administrative privileges. By default, the `root` user has no password.

2. Execute the first script for indirect synchronization. For example:

   mysql> \. c:\tools\sql\mysql\indirect\INSTALL_INDIRECT_1.sql

3. Open and execute the second installation script for indirect synchronization. For example:

   mysql> \. c:\tools\sql\mysql\indirect\INSTALL_INDIRECT_2.sql

Configuring Informix Objects

For Informix Dynamic Server, you must manually create a database user account before running the provided scripts.

Because the process of creating user accounts differs between operating systems, Step 1 below is OS-specific. These instructions are for a Windows NT operating environment. If you reinstall the database objects, you only need to repeat Step 2 through Step 6.

Installation Instructions

1. Create a user account for user **dirxml** using **dirxml** as the password in User Manager for Domains.
   - Remember to uncheck the User Must Change Password at Next Login checkbox for this account.
   - You might want to also check the Password Never Expires check box.

   NOTE: The remaining instructions are OS-independent.
2. Start SQL Editor.
3. Log on to your server as user **informix**. By default, the password for informix is **informix**.
4. Open and execute the first script for direct or indirect synchronization. For example:

   tools\sql\informix\direct\INSTALL_DIRECT_1.sql

   tools\sql\informix\indirect\INSTALL_INDIRECT_1.sql
5. Log on to your database server as user **dirxml** using password **dirxml**.
6. Open and execute the second installation script for direct or indirect synchronization. For example:

   tools\sql\informix\direct\INSTALL_DIRECT_2.sql

   tools\sql\informix\indirect\INSTALL_INDIRECT_2.sql

Troubleshooting Tips

- When generating events for publication, make sure you are logged in as someone other than the dirxml user. If you make changes as the dirxml user, your changes will not be published.
- Be sure to commit changes. Until you commit your changes, they will not be published.
- Make sure you are logged in as the correct user on the correct database when running the scripts.

Upgrading the Driver

Use the following information and procedures if you are upgrading the driver from a previous version.

Upgrade Requirements

For versions prior to 1.5, refer to the DirXML Driver 1.5 for JDBC Implementation Guide (http://www.novell.com/documentation/lg/dirxmldrivers/index.html). Be sure to use the 1.6 association utility. It supersedes all previous versions.

Upgrading from 1.5 to 1.6

After you download the CD image, perform the following steps to upgrade a previous version of the driver:

1. Stop the drivers being upgraded. Select Manual for the driver's startup option.
2. Stop eDirectory.
3. Replace JDBCShim.jar, JDBCUtil.jar, and CommonDriverShim.jar.
4. Restart eDirectory.
5. (Optional) Install the preconfigured drivers.

   You should uninstall the previous preconfigured drivers and execute the database uninstall scripts before installing the new preconfigured drivers and scripts.
6. Set the driver's startup options to their previous values.
7. Restart the drivers.

Activating the Driver

DirXML and DirXML drivers must be activated within 90 days of installation, or they will shut down. At any time during the 90 days, or afterward, you can choose to activate DirXML products to a fully licensed state.

To activate your driver, you should:

- Purchase DirXML licenses
- Generate a Product Activation Request
- Submit the Product Activation Request
- Install the Product Activation Credential received from Novell

For more information about completing these tasks, refer to Activating Your DirXML Product (http://www.novell.com/documentation/lg/dirxml11a/index.html).

4 Configuring the Driver

This section explains how to set up driver configuration parameters with possible configuration values. Before you begin, make sure you have the appropriate driver files and a working knowledge of Novell® eDirectory™ and iManager.

Setting Driver Authentication Parameters

After you import the driver, you need to provide authentication information for the database.

Configuring Driver Authentication

1. In Novell iManager, click DirXML Management > Overview.
2. Locate the driver set containing the driver, then click the driver's icon.
3. From the DirXML Driver Overview, click the driver object, which will display the driver configurations. Refer to the individual driver implementation guides for more information about each driver's specific parameters.
4. Enter driver authentication information:

Parameter Name	Sample Configuration Value	Default Value	Required Field
Authentication ID	dirxml		yes
Authentication Context	jdbc:oracle:thin:@255.255.255.255:1521:ora		yes
Application Password	dirxml		yes

Authentication ID

Authentication ID is the name of the driver's database user/login account. This user must exist and be granted login/session privileges on the database or a connection cannot be established. In addition, this user must have rights to select, insert, update, and delete on tables in the synchronization schema or synchronization will fail.

Authentication Context

Authentication Context is the JDBC URL of the target database.

URL format and content are proprietary and differ between third-party drivers. There are some similarities in content, however. Each URL, whatever the format, usually includes an IP address or DNS name, port number, and a database identifier. Consult your third-party driver documentation for the exact syntax and the content requirements of your driver.

The following table lists example URLs for third-party drivers. You will need to substitute the appropriate IP address, port number, and database/data source identifiers for your database. These examples use IP address 255.255.255.255, the default port number for each database, and `dirxml` database/data source identifier.

Third-party Driver	Example JDBC URL Syntax
Oracle8i, 9i JDBC Drivers	jdbc:oracle:thin:@255.255.255.255:1521:dirxml
IBM DB2 UDB JDBC Driver	jdbc:db2://255.255.255.255/dirxml
BEA Weblogic jDriver for Microsoft SQL Server 7/2000	jdbc:weblogic:mssqlserver4:dirxml@255.255.255.255:1433
Microsoft SQL Server 2000 Driver for JDBC	jdbc:microsoft:sqlserver://255.255.255.255:1433;DatabaseName=dirxml
Sybase jConnect	jdbc:sybase:Tds:255.255.255.255:2048/dirxml
MySQL Connector/J	jdbc:mysql://255.255.255.255:3306/dirxml
Informix JDBC Driver	jdbc:informix-sqli://255.255.255.255:1526/dirxml:informixserver=server
Sun's JDBC-ODBC Bridge Driver	jdbc:odbc:dirxml

Application Password

This is the password for the database user/login account that is used by the driver. You must create a user/login account on the database and grant login/session privileges to this account or the driver will be unable to connect.

NOTE: ConsoleOne® will not show the asterisk (*) characters in the New Password fields when you reopen the driver's Properties dialog box. The password does persist, however, and doesn't need to be re-entered.

Driver Parameters

After you smightet the driver authentication parameters, you should set the driver's parameters.

Driver parameters are divided into three categories:

- Driver
- Subscriber
- Publisher

Configuring Driver Settings

1. In iManager, click DirXML Management > Overview.
2. Locate the driver set containing the driver, then click the driver's icon.
3. From the DirXML Driver Overview, click the driver object, which will display the driver configurations.

The following table lists the driver settings and sample values:

Parameter Name	Sample Configuration Value	Default	Required	Tag
Third-Party JDBC Driver Class Name	oracle.jdbc.driver.OracleDriver		yes	<jdbc-class>
Synchronize Schema	dirxml		yes	<sync-schema>
Synchronize Tables	emp		yes	<sync-tables>
Reuse Statements?	yes	yes	no	<reuse-statements>
Use Manual Transactions?	yes	(dynamically determined)	no	<use-manual-transactions>
Use Single Connection?	no	no	no	<use-single-connection>
Default Transaction Isolation Level	read committed	(same)	no	<transaction-isolation-level>
Connection Tester Class Name	com.novell.nds.dirxml.driver.jdbc.util.JDBCConnectionTester	(same)	no	<connection-tester-class>
Connection Test Statement	SELECT empno FROM dirxml.emp where -1 = 0		no	<connection-test-stmt>
Retrieve Minimal Metadata?	no	no	no	<minimal-metadata>
Handle Statement Results?	yes	yes	no	<handle-stmt-results>
Connection Initialization String	USE dirxml		no	<connection-init>
Enable Referential Support?	yes	yes	no	<enable-refs>

Third-Party JDBC Driver Class Name

Third-Party JDBC Driver Class Name is a required, case-sensitive parameter. This name refers to the fully-qualified class name of your third-party driver. The following table lists the class name for tested third-party drivers:

Third-party Driver	Value
Oracle8i, 9i JDBC drivers	oracle.jdbc.driver.OracleDriver
IBM DB2 UDB JDBC Driver	COM.ibm.db2.jdbc.net.DB2Driver

Third-party Driver	Value
BEA Weblogic jDriver for MSSQL Server 7/2000	weblogic.jdbc.mssqlserver4.Driver
Microsoft SQL Server 2000 Driver for JDBC	com.microsoft.jdbc.sqlserver.SQLServerDriver
Sybase jConnect 5.5	com.sybase.jdbc2.jdbc.SybDriver
MySQL Connector/J	org.gjt.mm.mysql.Driver
Informix JDBC driver	com.informix.jdbc.IfxDriver
Sun JDBC-ODBC driver	sun.jdbc.odbc.JdbcOdbcDriver

Synchronize Schema

Synchronize schema is a required parameter that might be case-sensitive. This parameter identifies the database schema being synchronized. A database schema is analogous to the name of the owner of the tables being synchronized. For example, if you wanted to synchronize two tables, `emp` and `phone`, each belonging to the database user `dirxml`, you would enter `dirxml` in this field. When this parameter is used, Synchronize Tables must be left empty or omitted from a driver's configuration.

Synchronize Tables

Synchronize Tables is a required parameter that might be case-sensitive. This parameter allows you to create a virtual database schema by listing the names of the logical database classes to synchronize. Logical database class names are the names of parent tables and views. It is an error to list child table names. This parameter is useful when synchronizing with databases that do not support the concept of schema or the synchronization schema contains a large number of tables of which only a few are of interest to the driver. If you synchronize two tables or views with the same names in different schemas, be sure to schema-prefix the table or view names in the schema mapping rule. The driver does not schema-prefix table or view names returned from the getSchema() operation by default. When this parameter is used, Synchronize Schema must be left empty or omitted from a driver's configuration.

Reuse Statements

Reuse Statements is a case-insensitive parameter that might be required for some third-party drivers. If you set the parameter to **`yes`**, which is the default, the driver allocates java.sql.Statement, java.sql.PreparedStatement, and java.sql.CallableStatement objects once and reuses them. When set to a **no**, the driver allocates/deallocates statement objects each time they are used. Setting this parameter to **no** will degrade driver performance.

This parameter must be set to **no** when using Microsoft's SQL Server 2000 Driver for JDBC.

To maximize driver performance, we recommend that you use the default value or omit this parameter from most driver configurations.

Use Manual Transactions

Use Manual Transactions is a case-insensitive parameter whose value is derived from database metadata at runtime. This parameter should only be used when it is necessary to override default driver behavior. For instance, for MySQL, transaction support is determined on a per table rather

than per database basis. As such, it is necessary to disable manual transaction support when synchronizing to tables without transaction support.

When set to **yes**, the driver supports the use of manual transactions. When set to **no**, each statement executed by the driver is an automatic transaction.

To ensure data integrity in the target database, we recommend that this parameter be omitted from most driver configurations.

Use Single Connection

Use single connection is a case-insensitive parameter that might be required for some third-party drivers. When set to **yes**, both the Subscriber and Publisher channels share a single connection. When set to **no**, which is the default, each channel uses a separate connection. Setting this parameter to **yes** will degrade driver performance.This parameter should only be set to **yes** when both the Subscriber and Publisher channels are in use.

To maximize driver performance, we recommend that you use the default value or omit this parameter from most driver configurations.

Default Transaction Isolation Level

Default Transaction Isolation Level is an optional, case-insensitive parameter. This parameter sets the default transaction isolation level for connections used by the driver. There are five possible values, four of which correspond to the public constants defined in the java.sql.Connection interface:

- `none`
- `read uncommitted`
- `read committed`
- `repeatable read`
- `serializable`

The default value is `read committed`. We recommend using the default transaction isolation level of `read committed`. For more information on these values, refer to Sun's Web site. (http://java.sun.com)

Because some third-party drivers do not support setting a connection's transaction isolation level to `none`, the driver also supports the additional value of `unsupported`.

Connection Test Statement

Connection Test Statement is an optional parameter that might be case-sensitive. This parameter is a quick alternative to creating a connection tester class. Often, it is sufficient to detect connection failure by sending an arbitrary SQL statement across the wire.

When present, this parameter overrides the Connection Tester Class Name parameter.

Connection Tester Class Name

Connection Tester Class Name is a case-sensitive parameter that might be required for some third-party drivers. This is the fully-qualified class name of the class used to determine connection state. This class must be public, have a public, default constructor, and implement the com.novell.nds.dirxml.driver.jdbc.db.DBConnectionTester interface.

The default value is
`com.novell.nds.dirxml.driver.jdbc.util.JDBCConnectionTester`

Microsoft's SQL driver for JDBC, set the value to:
`com.novell.nds.dirxml.driver.jdbc.db.MSSQLConnectionTester`

For Informix's JDBC driver, set the value to:
`com.novell.nds.dirxml.driver.jdbc.db.InformixConnectionTester`

For Mysql Connector/J driver, set the value to:
`com.novell.nds.dirxml.driver.jdbc.db.MySQLConnectionTester`

This parameter is ignored when a Connection Test Statement parameter value is specified.

Retrieve Minimal MetaData

Retrieve Minimal Metadata is a case-insensitive parameter that might be required for some databases. When set to **yes**, the driver calls only required metadata methods. When set to **no**, which is the default value, the driver calls required and optional metadata methods. Refer to Appendix D, "java.sql.DatabaseMetaData Methods," on page 91 for more a list of required and optional metadata methods. Optional metadata methods are required for multi-valued and referential attribute synchronization.

Setting this value to **yes** will improve the driver's startup time at the expense of driver functionality.

Handle Statement Results?

Handle statement results is an optional parameter that is case-insensitive. This parameter tells the driver how many result sets can be generated by an arbitrary SQL statement. There are three possible values:

- `none`
- `single`
- `multiple`

The default value is `multiple`. For backwards compatibility reasons, `yes` equates to `multipleno`; equates to `none`.

For Microsoft's ODBC driver, Oracle's JDBC drivers, or Informix's JDBC driver, you should set this parameter to `single`. For other third-party drivers, we recommend that you use the default value or omit this parameter from most driver configurations.

Connection Initialization String

Connection initialization string is an optional parameter that might be case-sensitive. The connection initialization string is used to set properties on connections used by the driver. Multiple statement values must be delimited by a semicolon. This parameter is useful for adjusting ANSI-compatibility standards and database context.

Enable Referential Support?

Enable referential support is an optional parameter that is case-insensitive. This parameter tells the driver to interpret foreign key constraints that refer to parent tables of other database classes as referential attributes. Referential attributes are typically used to denote containment (e.g., group membership). When set to **yes**, which is the default, the driver interprets said columns as referential. When set to **no**, the driver interprets said columns as non-referential. The purpose of this parameter is to ensure backwards compatibility with the 1.0 version of this driver. For 1.0 compatibility, this parameter should be set to **no**.

Subscriber Settings

The following table lists the subscriber settings and sample values.

Parameter Name	Sample Configuration Value	Default Value	Required	Tag
Disable	yes	no	no	<disable>
Primary Key Generation	emp("sp_empno(empno,fname)")		no	<key-gen>
Key Generation Timing	after	before	no	<key-gen-timing>
Check Update Counts?	yes	yes	no	<check-update-count>

Disable

Disable is an optional, case-insensitive parameter. When set to `yes`, the subscriber channel does not process events; instead it returns warnings. When set to `no`, which is the default, the subscriber processes events.

Primary Key Generation

Primary key generation is an optional, complex parameter that might be case-sensitive. Database identifiers used in this value must not be delimited.

When processing `<add>` events, the subscriber uses primary key values to create associations. This parameter specifies how the subscriber obtains the primary key values necessary to construct association values. There are three possibilities:

1. The necessary primary key values are already present in the XML event.
2. The subscriber needs to generate the necessary primary key values.
3. The subscriber should obtain the necessary primary key values by calling a user-defined stored procedure or function in the database.

Method 1: By default, the driver assumes primary key values are already present in the XML event. If this is the case, no values need to be generated. This is desirable when an eDirectory attribute, such as GUID, is explicitly schema-mapped to a table or view's primary key column.

The syntax for Method 1 is: *logical-database-classname*(none)

For example:

```
emp(none)
```

```
view_emp(none)
```

Method 2: It is often desirable in a testing environment to have the subscriber generate primary key values before a stored procedure or function is available. This method can also be used against databases that do not support stored procedures or functions. For any numeric column types, the driver uses a simple (MAX+1) function to generate primary key values. In the case of string column types, the driver generates a random alpha character sequence. Other data types are not supported.

The syntax for Method 2 is: *logical-database-classname*(driver)

For example:

```
emp(driver)
```

```
view_emp(driver)
```

Method 3: Primary key values are obtained from a user-defined stored procedure or function.

The syntax for stored procedures is: *logical-database-classname("stored-procedure-signature")*, where *stored-procedure-signature = procedure-name(column-name, . . .)*.

For example:

```
emp("sp_empno(empno, fname)")
```

```
view_emp("sp_empno(pk_empno, fname)")
```

The syntax for functions is: *logical-database-classname("? = function-signature")*, where *function-signature = function-name(column-name, . . .)*.

For example:

```
emp("? = sp_empno(empno, fname)")
```

```
view_emp("? = sp_empno(pk_empno, fname)")
```

This notation maps a parent table or view to a user-defined stored procedure or function. The column names are those of the logical database class that should be passed to the stored procedure or function. Parameter order, number, and data type must correspond to the order, number, and data type of the parameters expected by the procedure or function. For stored procedures, primary key columns must be passed as IN OUT parameters. Non-key columns must be passed as IN parameters.

Some Additional Considerations Regarding Primary Key Generation

- When using Method 1, GUID rather than CN should be schema-mapped to a primary key column.
- When using Method 3, primary key columns should not be schema-mapped or included in the subscriber or publisher filters.
- When synchronizing multiple classes, a primary key generation method should be declared for each logical database class. Multiple values can be space-delimited or comma-delimited.

Key Generation Timing

Key Generation Timing is a case-insensitive parameter that is required for most databases when Primary Key Generation Methods 2 and 3 are used.

There are two legal values:

- `before`
- `after`

The default value is `before`.

Primary Key Generation Method 1: This parameter is ignored.

Primary Key Generation Method 2: When set to **`before`**, the subscriber executes a select statement before a row is inserted into a parent table or a view. When set to **`after`**, the subscriber executes a select statement after a row is inserted into a parent table or a view.

Primary Key Generation Method 3: When set to **`before`**, procedures or functions declared in the Primary Key Generation parameter are called before a row is inserted into a parent table or a view. When set to **`after`**, procedures or functions are called after a row is inserted into a parent table or a view.

For all databases except Oracle, this parameter should be set to `after`. For Oracle, the default value should be used or the parameter omitted.

Check Update Counts?

Check Update Counts is an optional, case-insensitive parameter. When set to a `yes`, which is the default, update counts are checked to ensure that when rows in a table or view are inserted, updated, or deleted, it actually happened. If this parameter is set to `yes` and rows are not updated, an error is issued. When set to `no`, update counts are not checked. This parameter should be set to `no` when statements are redefined in before-trigger logic on a table or instead-of-trigger logic on a view.

When synchronizing to Microsoft SQL Server, you should use the default value since errors in trigger logic (that might roll back a transaction) are not propagated back to the subscriber.

Publisher Settings

The following table lists the publisher settings with default and sample values for the configuration:

Parameter Name	Sample Configuration Value	Default Value	Required	Tag
Disable	yes	no	no	<disable>
Log Table Name	eventlog		yes	<log-table>
Polling Interval (seconds)	1-604800 (1 week)	10	no	<polling-interval>
Reconnect Interval (seconds)	1-3600 (1 hour)	30	no	<reconnect-interval>
Optimize Updates	yes	no	no	<optimize-update>
Delete from Log	yes	yes	no	<delete-from-log>
Allow Loopback?	yes	no	no	<check-update-count>

Disable

Disable is an optional, case-insensitive parameter that specifies whether the publisher channel should open a connection to the database and poll the event log table for events. When set to `yes`, the publisher does not establish a connection to the database nor does it poll the event log table. When set to `no`, which is the default, the publisher connects to the database and polls the event log table.

Log Table Name

Log table name is a required parameter that might be case-sensitive. This parameter specifies the name of the table where database events are stored for publication. This value must not be delimited.

Polling Interval

Polling interval is an optional, case-insensitive parameter that specifies how often, in seconds, the publisher should poll the event log table for events. The default value is ten seconds.

We recommend setting this value to no less than ten seconds.

Reconnect Interval

Reconnect interval is an optional, case-insensitive parameter that specifies how often, in seconds, the publisher should attempt to reconnect to the target database. The default value is thirty seconds.

We recommend setting this value to no less than ten seconds.

Optimize Updates

This optional, case-insensitive parameter specifies whether the publisher channel should ignore type 2 events that contain the same old and new values. Equality is determined in a case-sensitive string comparison operation. NULL values are considered equal. This option is useful if publication triggers are not optimized. When set to `yes`, type 2 events are optimized. When set to `no`, which is the default, type 2 events are unoptimized.

Delete from Log

This optional, case-insensitive parameter specifies whether the publisher should delete processed records from the event log table. When set to `no`, the publisher does not delete processed rows from the table. Instead, the publisher sets the `status` field to 'S' for success. This setting is useful for debugging purposes. When set to `yes`, which is the default value, processed rows are deleted. This is the proper setting for a production environment. Rows that are processed with errors remain in the event log table independent of this value.

This parameter should be set to `no` only for debugging purposes. Publication performance is degraded when this parameter is set to `yes`. If an auditing mechanism is desired in a production environment, rows inserted into the event log table for publication should also be written to a mirror table.

Trace Levels

In order to see debugging output from the driver, you need to add a DirXML-DriverTraceLevel attribute value from 1 to 6 on the driver set containing the driver. This attribute is commonly confused with the DirXML-XSL TraceLevel attribute. For more information on driver set trace levels, refer to the DirXML Administration Guide (http://www.novell.com/documentation).

The driver supports the following six trace levels:

1. Minimal
2. Database properties
3. Connection status, SQL statements, event log records
4. Verbose

5. JDBC API (methods, arguments, returned values, etc.)
6. Third-party driver

Levels 5-6 are particularly useful for debugging third-party drivers.

Configuring Third-Party JDBC Drivers

The following guidelines will assist you in configuring third-party drivers. For specific configuration instructions, refer to your third-party driver's documentation.

- Use the latest version of the driver available.
- When configuring an ODBC data source, be careful not to override any driver authentication parameters (for example, username and password settings).
- Third-party driver behavior might be configurable. In many cases, incompatibility issues can be resolved by adjusting the driver's configuration properties.
- When dealing with international characters, it is often necessary to explicitly specify the character encoding used by the database to third-party drivers by appending a property string to the end of the driver's JDBC URL. Properties usually consist of a property keyword and character encoding value (for example, jdbc:odbc:mssql;charSet=Big5). The property keyword might vary between third-party drivers.

  The possible character encoding values are defined by Sun. Refer to Sun's Supported Encoding Web site (http://java.sun.com/products/jdk/1.1/docs/guide/intl/encoding.doc.html) for more information.

The following table lists the recommended settings for maximum driver compatibility. These settings are useful when using an untested third-party driver.

Parameter Name	Value
Synchronize Tables	*table-list*
Reuse Statements?	no
Use Manual Transactions?	no
Use Single Connection	yes
Default Transaction Isolation Level	unsupported
Retrieve Minimal Metadata?	yes
Handle Statement Results?	single

5 Advanced Driver Configuration

After installing a sample preconfiguration and database script, you will need to customize the driver for specialized use. This section contains important conceptual information, sample configurations, and so forth to help you configure the driver.

Schema Mapping

The following table shows a high-level view of how the driver maps Novell® eDirectory™ objects to database objects.

eDirectory Object	Database Object
Tree	Schema
Class	Table/View
Attribute	Column
Association	Primary Key

Logical Database Classes

A logical database class is the set of tables or views used to represent an eDirectory class in a database. A logical database class can consist of a single view or one parent table and zero or more child tables. The name of a logical database class is the name of the parent table or view.

Indirect Synchronization

In an indirect synchronization model, the driver maps the following:

eDirectory Object	Database Object
Classes	Tables
Attributes	Columns

eDirectory Object	Database Object
1 Class	1 Parent table and 0 or more child tables
Single-valued attribute	Parent table column
Multi-valued attribute	Parent table column or Child table column (preferred)

Mapping eDirectory Classes to Logical Database Classes

In the following example, the logical database class emp consists of one parent table emp and one child table phone. Logical class emp is mapped to the eDirectory class User.

```
CREATE TABLE dirxml.emp
(
    empno NUMERIC(8) NOT NULL,
    fname VARCHAR(64),
    lname VARCHAR(64),
    pwdminlen NUMERIC(4),

    CONSTRAINT pk_emp_empno PRIMARY KEY(empno)
);

CREATE TABLE dirxml.phone
(
    empno NUMERIC(8) NOT NULL,
    phone VARCHAR(64) NOT NULL,

    CONSTRAINT fk_phone_empno FOREIGN KEY(empno) REFERENCES
    emp(empno)
);
```

```
<rule name="MappingRule">
     <attr-name-map>
           <class-name>
                 <nds-name>User</nds-name>
                 <app-name>emp</app-name>
           </class-name>
           <attr-name class-name="User">
                 <nds-name>Given Name</nds-name>
                  <app-name>fname</app-name>
           </attr-name>
           <attr-name class-name="User">
                 <nds-name>Surname</nds-name>
                  <app-name>lname</app-name>
           </attr-name>
          <attr-name class-name="User">
               <nds-name>Password Minimum Length</nds-name>
               <app-name>pwdminlen</app-name>
          </attr-name>
           <attr-name class-name="User">
                 <nds-name>Telephone Number</nds-name>
                  <app-name>phone.phoneno</app-name>
```

```
            </attr-name>
        </attr-name-map>
</rule>
```

Parent Tables

Parent tables are tables with an explicit primary key constraint that contains one or more columns. In a parent table, an explicit primary key constraint is required so that the driver knows which fields to include in an association value.

```
CREATE TABLE dirxml.emp
(
    empno NUMERIC(8) NOT NULL,
    ...
    CONSTRAINT pk_emp_empno PRIMARY KEY(empno)
);
```

The following table contains sample data for `dirxml.emp`.

empno	fname	lname
1	John	Doe

The resulting association for this row would be:

`empno=1,table=emp,schema=dirxml`

NOTE: The case of database identifiers in association values is determined dynamically at runtime from database metadata.

Parent Table Columns

Parent table columns can contain only one value. As such, they are ideal for mapping single-valued eDirectory attributes. For example, mapping the single-valued eDirectory attribute Password Minimum Length to the single-valued parent table column `pwdminlen`.

Parent table columns are implicitly prefixed with the name of the parent table. It is not necessary to explicitly table-prefix parent table columns. For example, `emp.fname` is equivalent to `fname` for schema mapping purposes.

```
<rule name="MappingRule">
   <attr-name-map>
      <class-name>
         <nds-name>User</nds-name>
         <app-name>emp</app-name>
      </class-name>
      <attr-name class-name="User">
         <nds-name>Given Name</nds-name>
          <app-name>fname</app-name>
      </attr-name>
   </attr-name-map>
</rule>
```

Large binary and string data types should be typically mapped to parent table columns. In order to map to a child table column, a data type must be comparable in an SQL statements. Large data types usually cannot be compared in SQL statements.

Large binary and string data types can be mapped to child table columns if <remove-value> events on these types are transformed in style sheets into a <remove-all-values> element followed by a series of <add-value> elements, one for each value.

Child Tables

A child table is a table that has a foreign key constraint on its parent table's primary key, linking the two tables together. The columns that comprise the child table's foreign key must have the same name as the columns in the parent table's primary key. This common column name is used by the publisher to identify all rows in the event log table pertaining to a single logical database class.

The following example shows the relationship between parent table emp and child table phone. Note the use of the same column name empno in each table.

```
CREATE TABLE dirxml.emp
(
    empno NUMERIC(8) NOT NULL,
    ...
    CONSTRAINT pk_emp_empno PRIMARY KEY(empno)
);

CREATE TABLE dirxml.phone
(
    empno NUMERIC(8) NOT NULL,
    phoneno VARCHAR(64) NOT NULL,

    CONSTRAINT fk_phone_empno FOREIGN KEY(empno) REFERENCES
    emp(empno)
);
```

The constrained column in a child table identifies the parent table. In the above example, the constrained column in child table phone is empno. The only purpose of this column is to relate tables phone and emp. Because constrained columns do not contain any useful information, they should be omitted from publication triggers and the schema mapping rule.

The unconstrained column is the column of interest. It represents a single, multi-valued attribute. In the above example, the unconstrained column is phoneno. Because unconstrained columns can hold multiple values, they are ideal for mapping multi-valued eDirectory attributes. For example, mapping the multi-valued eDirectory attribute Telephone Number to phone.phoneno.

All columns in a child table should be constrained NOT NULL.

NOTE: Each multi-valued, eDirectory attribute must be mapped to a different child table column.

The following table contains sample data for dirxml.phone.

empno	phoneno
1	111-1111
1	222-2222

When mapping a multi-valued eDirectory attribute to a child table column, the child column name must be explicitly prefixed with the child table name (for example, phone.phoneno). Otherwise, the driver will implicitly interpret phoneno as emp.phoneno, not phone.phoneno.

```
<rule name="MappingRule">
   <attr-name-map>
      <class-name>
         <nds-name>User</nds-name>
         <app-name>emp</app-name>
      </class-name>
      <attr-name class-name="User">
         <nds-name>Telephone Number</nds-name>
          <app-name>phone.phoneno</app-name>
      </attr-name>
   </attr-name-map>
</rule>
```

Referential Attributes

Referential containment can be represented in the database through the use of foreign key constraints. Referential attributes are columns within a logical database class that refer to the primary key columns of parent tables of other logical database classes.

Single-Valued Referential Attributes

Two parent tables can be related through a single parent table column. This column must have a foreign key constraint pointing to the other parent table's primary key. The following example relates a single parent table `user` to itself.

```
CREATE TABLE user
(
    idu NUMBER(8) NOT NULL,
    manager NUMBER(8),

    CONSTRAINT pk_user_idu PRIMARY KEY(idu),
    CONSTRAINT fk_user_idu FOREIGN KEY(manager)REFERENCES
    user(idu)
);
```

```
<rule name="Mapping Rule">
    <attr-name-map>
        <class-name>
            <nds-name>User</nds-name>
            <app-name>user</app-name>
        </class-name>
        <attr-name class-name="User">
            <nds-name>manager</nds-name>
            <app-name>manager</app-name>
        </attr-name>
    </attr-name-map>
</rule>
```

Single-valued, referential colums must be nullable.

Multi-valued Referential Attributes

Two parent tables can be related through a common child table. This child table must have a foreign key constraint pointing to each parent table's primary key. The following example relates two parent tables `user` and `group` through a common child table `member`.

```
CREATE TABLE user
(
```

```
    idu NUMBER(8) NOT NULL,
    lname VARCHAR(64) NOT NULL,

    CONSTRAINT pk_user_idu PRIMARY KEY(idu)
);

CREATE TABLE group
(
    idg NUMBER(8) NOT NULL,

    CONSTRAINT pk_group_idg PRIMARY KEY(idg)
);

CREATE TABLE member
(
    idg NUMBER(8)NOT NULL,
    idu NUMBER(8)NOT NULL,

    CONSTRAINT fk_member_idg FOREIGN KEY(idg) REFERENCES
    group(idg),
    CONSTRAINT fk_member_idu FOREIGN KEY(idu) REFERENCES
    user(idu)
);
```

```
<rule name="Mapping Rule">
    <attr-name-map>
        <class-name>
            <nds-name>User</nds-name>
            <app-name>user</app-name>
        </class-name>
        <attr-name class-name="User">
            <nds-name>Surname</nds-name>
            <app-name>lname</app-name>
        </attr-name>
        <class-name>
            <nds-name>Group</nds-name>
            <app-name>group</app-name>
        </class-name>
        <attr-name class-name="Group">
            <nds-name>Member</nds-name>
            <app-name>member.idu</app-name>
        </attr-name>
    </attr-name-map>
</rule>
```

The first constrained column in a child table determines ownership. In the above example, `member` is considered to be part of class `group`. `member` is said to be a proper child of `group`. The second constrained column in a child table is the multi-valued referential attribute. Both columns must be constrained NOT NULL.

In the following example, the order of the constrained columns has been reversed so `member` is part of class `user`. To more accurately reflect the relationship, `member` has been renamed to `member_of`.

```
CREATE TABLE user
(
    idu NUMBER(8) NOT NULL,
    lname VARCHAR(64) NOT NULL,

    CONSTRAINT pk_user_idu PRIMARY KEY(idu)
);
```

```
CREATE TABLE group
(
    idg NUMBER(8) NOT NULL,

    CONSTRAINT pk_group_idg PRIMARY KEY(idg)
);

CREATE TABLE member_of
(
    idu NUMBER(8)NOT NULL,
    idg NUMBER(8)NOT NULL,

    CONSTRAINT fk_member_idg FOREIGN KEY(idg) REFERENCES
    group(idg),
    CONSTRAINT fk_member_idu FOREIGN KEY(idu) REFERENCES
    user(idu)
);
```

```
<rule name="Mapping Rule">
    <attr-name-map>
        <class-name>
            <nds-name>User</nds-name>
            <app-name>user</app-name>
        </class-name>
        <attr-name class-name="User">
            <nds-name>Surname</nds-name>
            <app-name>lname</app-name>
        </attr-name>
        <attr-name class-name="User">
            <nds-name>Group Membership</nds-name>
            <app-name>member_of.idg</app-name>
        </attr-name>
        <class-name>
            <nds-name>Group</nds-name>
            <app-name>group</app-name>
        </class-name>
    </attr-name-map>
</rule>
```

In databases where position is meaningless, order is determined by lexographical comparison.

In general, it is only necessary to synchronize multi-valued, referential attributes as part of one class or the other, not both. If you wanted to synchronize referential attributes for both classes, it would be necessary to construct two child tables, one for each class. For example, if you wanted to synchronize Group Membership and Member, you would need two child tables: `member_of` and `member`.

In practice, when synchronizing User and Group objects, we recommend that you synchronize the Group Membership attribute of User instead of the Member attribute of Group. When synchronizing Member, events are generated for unassociated Users added to associated Groups. When synchronizing Group Membership, events are only generated for associated Users added to associated Groups.

Direct Synchronization

In a direct synchronization model, the driver maps the following:

eDirectory Object	Database Object
Classes	Views
Attributes	View Columns
1 Class	View
Single-valued attribute	View Column
Multi-valued attribute	View Column

A view is a logical table. Unlike parent or child tables, they do not physically exist in the database. As such, views cannot have primary key/foreign key constraints. In order to identify to the driver which fields to use when constructing association values, one or more view columns must be prefixed with pk_ (case-insensitive).

NOTE: Views must be constructed in such a way that the pk_ prefixed view columns uniquely identify a single row.

The update capabilities of views vary widely between databases. Most databases allow views to be updated under certain conditions. If views are strictly read-only, then they cannot be used for subscription. Microsoft SQL Server 2000 and Oracle 8i and 9i allow update logic to be defined on views in instead-of-triggers, which allows a view to join multiple tables and still be updateable.

```
CREATE TABLE dirxml.emp
(
    empno NUMERIC(8) NOT NULL UNIQUE,
    fname VARCHAR(64),
    lname VARCHAR(64),
    pwdminlen NUMERIC(4),
    phoneno VARCHAR(64)
);

CREATE VIEW dirxml.view_emp
(pk_empno, fname, lname, pwdminlen, phoneno)
AS
SELECT empno, fname, lname, pwdminlen, phoneno FROM dirxml.emp;
```

```
<rule name="MappingRule">
     <attr-name-map>
           <class-name>
              <nds-name>User</nds-name>
              <app-name>view_emp</app-name>
          </class-name>
          <attr-name class-name="User">
              <nds-name>Given Name</nds-name>
              <app-name>fname</app-name>
          </attr-name>
          <attr-name class-name="User">
              <nds-name>Surname</nds-name>
              <app-name>lname</app-name>
          </attr-name>
          <attr-name class-name="User">
              <nds-name>Password Minimum Length</nds-name>
```

```
            <app-name>pwdminlen</app-name>
        </attr-name>
        <attr-name class-name="User">
            <nds-name>Telephone Number</nds-name>
            <app-name>phoneno</app-name>
        </attr-name>
    </attr-name-map>
</rule>
```

Synchronizing Primary Key Columns

When the database is the authoritative source of primary key columns, they should generally be omitted from the publisher and subscriber filters, the schema mapping rule, and publication triggers.

When eDirectory is the authoritative source of primary key columns, they should be included in the subscriber filter and schema mapping rule and omitted from the publisher filter and publication triggers. Also, GUID rather than CN is recommended for use as a primary key. CN is multi-valued and can change. GUID is single-valued and static.

Synchronizing Multiple Classes

When synchronizing multiple eDirectory classes, it is necessary to synchronize each class to a different parent table or view. Each logical database class must have a unique primary key column name. This common column name is used by the publisher to identify all rows in the event log table pertaining to a single logical database class. For example, logical database classes `user` and `group` each have a unique primary key column name.

```
CREATE TABLE user
(
    idu NUMBER(8) NOT NULL,
    lname VARCHAR(64) NOT NULL,

    CONSTRAINT pk_user_idu PRIMARY KEY(idu)
);

CREATE TABLE group
(
    idg NUMBER(8) NOT NULL,

    CONSTRAINT pk_group_idg PRIMARY KEY(idg)
);
```

Mapping Multi-Valued Attributes to Single-Valued Database Fields

By default, the driver assumes that all eDirectory attributes mapped to parent table columns or view columns are single-valued. Because the driver is unaware of the eDirectory schema, it has no way of knowing whether an eDirectory attribute is single-valued or multi-valued. Accordingly, multi-valued and single-valued attribute mappings are handled identically.

The driver implements the Most Recently Touched (MRT) algorithm with regard to single-valued parent table or view columns. An MRT algorithm ensures that the most recently added attribute value or most recently deleted attribute value will be stored in the database. The algorithm is adequate if the attribute in question is single-valued, and has some undesirable consequences if the attribute is multi-valued.

When a value is deleted from a multi-valued attribute, the database field it is mapped to will be set to NULL and will remain NULL until another value is added. Several solutions to this undesirable behavior are outlined below.

- The preferred solution is to extend the eDirectory schema so that only single-valued attributes are mapping to parent table or view columns.
- For indirect synchronization, map each multi-valued attribute to its own child table.
- For both direct or indirect synchronization, use style sheets to delimit multiple values before inserting them into a parent table or view column.
- Implement a first or last value per replica policy in style sheets using methods provided in the com.novell.nds.dirxml.driver.jdbc.util.MappingPolicy class. Under a first-value-per-replica (FPR) policy, the first attribute value on the DirXML replica is always synchronized. Under a last-value-per-replica (LPR) policy, the last attribute value on a replica is always synchronized. All of the preconfigured drivers demonstrate a first-value-per-replica policy. They map the multi-valued eDirectory attributes Given name, Surname, and Facsimile Telephone Number to the single-valued columns `fname`, `lname`, and `faxno` respectively.

Event Mapping

The following table summarizes how the Subscriber maps XML events to SQL statements:

XML Event	SQL Equivalent
<add>	1 or more insert statements; 0 or 1 select statements; 0 or 1 stored procedure or function calls
<modify>	0 or 1 update statements; 0 or more insert statements; 0 or more delete statements
<delete>	1 or more delete statements; 0 or more update statements
<query>	1 or more select statements
<move> or <rename>	0 statements

Add Events

Add events map to one insert statement for the parent table or view and zero or more insert statements for each child table. For Primary Key Generation Method 2, one select statement is executed. For Primary Key Generation Methods 3, one stored procedure or function call is executed.

Modify Events

Modify events map to zero or one update statements for the parent table or view and zero or more insert and delete statements for each child table.

Delete Events

Delete events map to one delete statement for the parent table or view, and zero or one update statement for each single-valued, referential, parent table column.

Delete events map zero or more delete statements for each multi-valued, referential, child table column.

Query Events

Query events map to one select statement for the parent table or view and zero or one select statement for each child table.

Move and Rename Events

Move and rename events are No Operations or NOOPs. They are always mapped to zero statements.

The Event Log Table

The event log table is where publication events are stored. This section discusses the structure and limitations of the event log table.

You can customize the name of the event log table and its columns to avoid conflicts with reserved database words. The order, number, and data types of its columns, however, must remain constant. In databases where position is meaningless, order is determined by lexicographical comparison.

Event Log Columns

- `record_id`

  The `record_id` column is used to uniquely identify rows in the event log table. This column must contain sequential, ascending, positive, unique integer values.

- `status`

  The `status` column indicates the state of a given row. The possible values are:

  - 'N' = new
  - 'U' = unknown
  - 'S' = success
  - 'W' = warning
  - 'F' = fatal
  - 'E' = error

  All rows inserted into the event log table must have a `status` value of 'N' in order to be processed. The remainder of the status characters are used solely by the publisher. All other characters are reserved for future use.

  NOTE: Status values are case-sensitive.

- `event_type`

  Values in this column must be between 1 and 8. Event types fall into two major categories: per-field (1-3, 7-8) and per-row (4-6). Per-field events are more granular and than per-row events, but they require more space in the event log table. Per-row events are less granular and require less space in the event log table. Per-field event types can be thought of as per-attribute. Per-row event types can be thought of as per-object.

  Event types can also be grouped into two additional categories: query-back (5-8) and non-query-back (1-4). Query-back events are useful when synchronizing large binary and string data types.

In general, a combination of event types from each category yields the best time, space, and complexity trade-offs.

The following values are used to classify event types. All other numbers are reserved for future use.

- 1 = insert field
- 2 = update field
- 3 = update field (remove-all-values)
- 4 = delete row
- 5 = insert row (query-back)
- 6 = update row (query-back)
- 7 = insert field (query-back)
- 8 = update field (query-back)

- `event_time`

  Reserved for future use. This value must not be NULL.

- `perpetrator`

  The user who instigated the event. A NULL value is interpreted as a user other than the driver user. As such, records with `perpetrator` = NULL or !*driver's username* are published. Records with `perpetrator` = *driver's username* are not published unless the publisher parameter Allow Loopback is set to `yes`.

- `table_name`

  The name of the table or view where the event occurred.

- `table_key`

  Values for this column must be formatted exactly the same in all triggers of a logical database class. For example,

  primary key column name = value + primary key column name = value . . .

  - For indirect preconfigured drivers, for example, the value for this column would be `empno=1`.
  - For direct preconfigured drivers, for example, the value for this column would be `pk_empno=1`.

  NOTE: Primary key values placed in the `table_key` field should be delimited (that is, double-quoted) if they contain the following characters:

  , ; ' + = \ " < >

  Differences in padding or formatting might result in out-of-order event processing. For performance reasons, you should remove any unnecessary white space from numeric values. (For example, "empno=1" is preferred over "empno= 1")

- `column_name`

  The name of the column that was changed. The column is used only by per-field (1-3, 7-8) event types. Even though this column is used only for per-field event types, it must always be present in the event log table. If it is missing, the publisher will shut down the driver.

- `old_value`

The field's old value.The old value column name is used only by per-field, non-query-back event types (1-3). Even though this column is only used for these event types, it must always be present in the event log table. If it is missing, the publisher will cause the driver to shut down.

- `new_value`

  The field's new value. The new value column name is used only by per-field, non-query-back event types (1-3). Even though this column is only used for these event types, it must always be present in the event log table. If it is missing, the publisher will cause the driver to shut down.

Event Types

This section describes in greater detail the different event types and how they are interpreted by the publisher.

The table below shows the basic correlation between publication event types and the XML generated by the publisher.

Event Type	Resulting XML
insert	<add>
update	<modify>
delete	<delete>

The example below illustrates the XML generated by the publisher for events logged on table `emp` for each possible event type.

```
CREATE TABLE dirxml.emp
(
    empno NUMERIC(8) NOT NULL,
    fname VARCHAR2(64),
    photo LONGRAW,

    CONSTRAINT pk_emp_empno PRIMARY KEY(empno)
);
```

The table below shows the initial contents of `emp` after a new row has been inserted:

empno	fname	lname	photo
1	Jack	Frost	0xAAAA

The table below shows the current contents of `emp` after the row has been updated:

empno	fname	lname	photo
1	John	Doe	0xBBBB

1. Insert Field

The table below shows the contents of the event log table after a new row is inserted into table `emp`. The value for column `photo` has been Base64-encoded. The Base64-encoded equivalent of 0xAAAA is qqo=.

event_type	table	table_key	column_name	old_value	new_value
1	emp	empno=1	fname	NULL	Jack
1	emp	empno=1	lname	NULL	Frost
1	emp	empno=1	photo	NULL	qqo=

The XML generated by the Publisher would be:

```
<add class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <add-attr attr-name="fname">
        <value type="string">Jack</value>
    </add-attr>
    <add-attr attr-name="lname">
        <value type="string">Frost</value>
    </add-attr>
    <add-attr attr-name="photo">
        <value type="octet">qqo=</value>
    </add-attr>
</add>
```

2. Update Field

   The table below shows the contents of the event log table after the row in table `emp` has been updated. The values for column `photo` has been Base64-encoded. The Base64-encoded equivalent of 0xBBBB is u7s=.

event_type	table	table_key	column_name	old_value	new_value
2	emp	empno=1	fname	Jack	John
2	emp	empno=1	lname	Frost	Doe
2	emp	empno=1	photo	qqo=	u7s=

The XML generated by the Publisher would be:

```
<modify class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <modify-attr attr-name="fname">
        <remove-value>
            <value type="string">Jack</value>
        </remove-value>
        <add-value>
            <value type="string">John</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="lname">
        <remove-value>
            <value type="string">Frost</value>
```

```
        </remove-value>
        <add-value>
            <value type="string">Doe</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="photo">
        <remove-value>
            <value type="octet">qqo=</value>
        </remove-value>
        <add-value>
            <value type="octet">u7s=</value>
        </add-value>
    </modify-attr>
</modify>
```

3. Update Field (Remove-All-Values)

   The table below shows the contents of the event log table after the row in table `emp` has been updated. The value for column `photo` has been Base64-encoded.

event_type	table	table_key	column_name	old_value	new_value
3	emp	empno=1	fname	Jack	John
3	emp	empno=1	lname	Frost	Doe
3	emp	empno=1	photo	qqo=	u7s=

   The XML generated by the Publisher would be:

```
<modify class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <modify-attr attr-name="fname">
        <remove-all-values/>
        <add-value>
            <value type="string">John</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="lname">
        <remove-all-values/>
        <add-value>
            <value type="string">Doe</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="photo">
        <remove-all-values/>
        <add-value>
            <value type="octet">u7s=</value>
        </add-value>
    </modify-attr>
</modify>
```

4. Delete Row

   The table below shows the contents of the event log table after the row in table `emp` has been deleted.

event_type	table	table_key	column_name	old_value	new_value
4	emp	empno=1	NULL	NULL	NULL

The XML generated by the Publisher would be:

```
<delete class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
</delete>
```

5. Insert Row (Query-Back)

   The table below shows the contents of the event log table after a new row is inserted into table emp.

event_type	table	table_key	column_name	old_value	new_value
5	emp	empno=1	NULL	NULL	NULL

   The XML generated by the Publisher is listed below. Note that the values reflect the current contents of table emp, not the initial contents.

```
<add class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <add-attr attr-name="fname">
        <value type="string">John</value>
    </add-attr>
    <add-attr attr-name="lname">
        <value type="string">Doe</value>
    </add-attr>
    <add-attr attr-name="photo">
        <value type="octet">u7s=</value>
    </add-attr>
</add>
```

6. Update Row (Query-Back)

   The table below shows the contents of the event log table after the row in table emp has been updated.

event_type	table	table_key	column_name	old_value	new_value
6	emp	empno=1	NULL	NULL	NULL

   The XML generated by the Publisher is listed below. Note that the values reflect the current contents of table emp, not the initial contents.

```
<modify class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <modify-attr attr-name="fname">
        <remove-all-values/>
        <add-value>
            <value type="string">John</value>
        </add-value>
```

```
    </modify-attr>
    <modify-attr attr-name="lname">
        <remove-all-values/>
        <add-value>
            <value type="string">Doe</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="photo">
        <remove-all-values/>
        <add-value>
            <value type="octet">u7s=</value>
        </add-value>
    </modify-attr>
</modify>
```

7. Insert Field (Query-Back)

   The table below shows the contents of the event log table after a new row is inserted into table emp. Old and new values are omitted because they are not used.

event_type	table	table_key	column_name	old_value	new_value
7	emp	empno=1	fname	NULL	NULL
7	emp	empno=1	lname	NULL	NULL
7	emp	empno=1	photo	NULL	NULL

   The XML generated by the Publisher is listed below. Note that the values reflect the current contents of table emp, not the initial contents.

```
<add class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <add-attr attr-name="fname">
        <value type="string">John</value>
    </add-attr>
    <add-attr attr-name="lname">
        <value type="string">Doe</value>
    </add-attr>
    <add-attr attr-name="photo">
        <value type="octet">u7s=</value>
    </add-attr>
</add>
```

8. Update Field (Query-Back)

   The table below shows the contents of the event log table after the row in table emp has been updated. Old and new values are omitted since they are not used.

event_type	table	table_key	column_name	old_value	new_value
8	emp	empno=1	fname	NULL	NULL
8	emp	empno=1	lname	NULL	NULL
8	emp	empno=1	photo	NULL	NULL

The XML generated by the Publisher is listed below. Note that the values reflect the current contents of table emp, not the inital contents.

```
<modify class-name="emp">
    <association>empno=1,table=emp,schema=dirxml
    </association>
    <modify-attr attr-name="fname">
        <remove-all-values/>
       <add-value>
            <value type="string">John</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="lname">
        <remove-all-values/>
       <add-value>
            <value type="string">Doe</value>
        </add-value>
    </modify-attr>
    <modify-attr attr-name="photo">
        <remove-all-values/>
       <add-value>
            <value type="octet">u7s=</value>
        </add-value>
    </modify-attr>
</modify>
```

Using Structured Query Language in XML Events

The following section includes information that will help you include Structured Query Language (SQL) in XML events.

All examples reference table emp below. The primary key generation method used to obtain primary key values is irrelevant for purposes of the examples in this section.

```
CREATE TABLE emp
(
    empno NUMERIC(8) NOT NULL,
    fname VARCHAR2(64),
    lanem VARCHAR2(64),

    CONSTRAINT pk_emp_empno PRIMARY KEY(empno)
);
```

NOTE: The namespace prefix jdbc used throughout this section is implicitly bound to the namespace urn:dirxml:jdbc when referenced outside of an XML document.

Introduction

You can use embedded SQL in XML events. In the same way that you can install database triggers on a table and cause side effects in a database, embedded SQL in XML events acts as a virtual trigger with similar capabilities.

SQL is embedded in XML events through the <jdbc:statement> and <jdbc:sql> elements. The <jdbc:statement> element can contain one or more <jdbc:sql> elements.

The following XML example shows an embedded SQL statement.

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp">
```

```
          <add-attr name="lname">
               <value>Doe</value>
          </add-attr>
     </add>
     <jdbc:statement>
          <jdbc:sql> UPDATE dirxml.emp SET fname = 'John'
          </jdbc:sql>
     </jdbc:statement>
</input>
```

Because the subscriber resolves <add> events to one or more insert statements, the above XML would resolve to:

```
INSERT INTO dirxml.emp(lname)VALUES('Doe');
UPDATE dirxml.emp SET fname = 'John';
```

IMPORTANT: You should use namespace-prefixed elements and attributes to embed SQL (otherwise, the driver will not recognize them). In the above example, the namespace is `urn:dirxml:jdbc`. The prefix is the identifier to the right of the `xmlns` identifier. In the above example, the prefix is `jdbc`. In practice, the prefix can be whatever you want it to be as long as is is bound to the correct namespace.

Variable Substitution

Rather than require you to parse field values from an association, the subscriber supports variable substitution in embedded SQL statements. For example:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
    <modify class-name="emp">
        <association>empno=1,table=emp,schema=dirxml
        </association>
        <modify-attr name="lname">
            <add-value>
                <value>DoeRaeMe</value>
            </add-value>
        </modify-attr>
    </modify>
    <jdbc:statement>
           <jdbc:sql>UPDATE dirml.emp SET fname = 'John' WHERE
           empno = {$empno}</jdbc:sql>
    </jdbc:statement>
</input>
```

Variable placeholders must adhere to the XSLT attribute value template syntax: {$*field-name*} and the association element must precede the <jdbc:statement> element in the XML document or must be present as a child of the <jdbc:statement> element. The *field-name* must refer to one of the naming RDN attribute names in the association value. In the above example, there is only one naming attribute, empno.

An <add> event is the only event where an association element is not required to proceed embedded SQL statements with variable substitution because the association has not been created yet. Additionally, any embedded SQL statements using variable substitution must follow, not proceed, the <add> event. For example:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp">
          <add-attr name="lname">
               <value>Doe</value>
          </add-attr>
     </add>
     <jdbc:statement>
```

```
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'John' WHERE
          empno = {$empno}</jdbc:sql>
     </jdbc:statement>
</input>
```

In order prevent tracing of sensitive information, you can use `{$$password}` to refer to the contents of a `<password>` element within the same document.

```
<input xmlns:jdbc="urn:dirxml:jdbc">
   <add class-name="emp">
      <add-attr name="lname">
         <value>Doe</value>
      </add-attr>
      <password>Doe{$empno}</password>
   </add>
   <jdbc:statement>
      <jdbc:sql>CREATE USER Doe IDENTIFIED BY
      {$$password}</jdbc:sql>
   </jdbc:statement>
</input>
```

Statement Placement

In the same way that database triggers can fire before or after a triggering statement, embedded SQL can be positioned before or after the triggering XML event. The following examples show how you could embed SQL before or after an XML event.

Before Trigger

```
<input xmlns:jdbc"urn:dirxml:jdbc">
      <jdbc:statement>
          <association>empno=1,table=emp,schema=dirxml
          </association>
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'John' WHERE
           empno = ${empno}</JDBC:SQL>
      </jdbc:statement>
     <modify class-name="emp">
          <association>empno=1,table=emp,schema=dirxml
          </association>
          <modify-attr name="lname">
               <remove-all-values/>
               <add-value>
                   <value>Doe</value>
               </add-value>
          </modify-attr>
     </modify>
</input>
```

The above XML resolves to:

```
UPDATE dirxml.emp SET fname = 'John' WHERE empno = 1;
UPDATE dirxml.emp SET lname = 'Doe' WHERE empno = 1;
```

After Trigger

```
<input xmlns:jdbc"urn:dirxml:jdbc">
     <modify class-name="emp">
          <association>empno=1,table=emp,schema=dirxml
          </association>
          <modify-attr name="lname">
               <remove-all-values/>
               <add-value>
                   <value>Doe</value>
               </add-value>
          </modify-attr>
     </modify>
     <jdbc:statement>
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'John' WHERE
           empno = {$empno}</jdbc:sql>
     </jdbc:statement>
</input>
```

The above XML resolves to:

```
UPDATE dirxml.emp SET lname = 'Doe' WHERE empno = 1;
UPDATE dirxml.emp SET fname = 'John' WHERE empno = 1;
```

Manual vs. Automatic Transactions

You can manually group embedded SQL and XML events using these two custom attributes:

- jdbc:transaction-type
- jdbc:transaction-id

jdbc:transaction-type

This attribute has two values: `manual` and `auto`. By default, most XML events of interest are set to the manual transaction type. The manual setting enables XML events to resolve to more than one SQL statement.

Embedded SQL events are set to auto transaction type by default because some SQL statements cannot be included in a manual transaction

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp" jdbc:transaction-type="auto">
           <add-attr name="lname">
                 <value>Doe</value>
           </add-attr>
     </add>
     <jdbc:statement>
           <jdbc:sql>UPDATE dirxml.emp SET fname = 'John' WHERE
           empno = {$empno}</jdbc:sql>
     </jdbc:statement>
</input>
```

The above XML resolves to:

```
INSERT INTO dirxml.emp(lname) VALUES('Doe');
/* COMMIT; implicit */

UPDATE dirxml.emp SET fname = 'John' WHERE empno = 1;
/* COMMIT; implicit */
```

jdbc:transaction-id

This attribute is ignored by the subscriber unless the element's `jdbc:transaction-type` attribute value defaults to or is explicitly set to **`manual`**. The following XML shows an example of a manual transaction:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp" jdbc:transaction-id="0">
          <add-attr name="lname">
               <value>Doe</value>
          </add-attr>
     </add>
     <jdbc:statement jdbc:transaction-type="manual"
      jdbc:transaction-id="0">
           <jdbc:sql>UPDATE dirxml.emp SET fname = 'John' WHERE
           empno = {$empno}</jdbc:sql>
     </jdbc:statement>
</input>
```

The above XML code resolves to:

```
INSERT INTO dirxml.emp(lname) VALUES('Doe');
UPDATE dirxml.emp SET fname = 'John' WHERE empno = 1;
COMMIT; /* explicit */
```

Transaction Isolation Level

In addition to grouping statements, transactions are used to preserve the integrity of data in a database. Transactions can lock data in order to prevent concurrent access or modification. How locks are set is determined by the isolation level of a transaction. Usually, the default isolation level used by the driver is sufficient and should not be altered.

The custom attribute `jdbc:isolation-level` allows you to adjust the isolation transaction level should the need ever arise. There are five possible values defined in the java.sql.Connection interface:

- `none`
- `read uncommitted`
- `read committed`
- `repeatable read`
- `serializable`

The driver's default transaction isolation level is `read committed`. In the case of a manual transaction, the `jdbc:isolation-level` attribute should be placed on the first element in the transaction. This attribute is ignored on subsequent elements. For example:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp" jdbc:transaction-id="0"
      jdbc:isolation-level="serializable">
           <add-attr name="lname">
                <value>Doe</value>
           </add-attr>
      </add>
      <jdbc:statement jdbc:transaction-type="manual"
       jdbc:transaction-id="0">
           <jdbc:sql>UPDATE dirxml.emp SET fname = 'John'
```

```
            WHERE empno = {$empno}</jdbc:sql>
      </jdbc:statement>
</input>
```

The above XML resolves to:

```
INSERT INTO dirxml.emp(lname) VALUES('Doe');
UPDATE dirxml.emp SET fname = 'John' WHERE empno = 1;
COMMIT; /* explicit */
```

Statement Type

The driver executes embedded SQL statements, but it doesn't understand them. The JDBC interface defines several methods for executing different types of SQL statements. The following table contains these methods:

Statement Type	Method of Execution
SELECT	Statement.executeQuery(String)
INSERT	Statement.executeUpdate(String)
UPDATE	Statement.executeUpdate(String)
DELETE	Statement.executeUpdate(String)
CALL or EXECUTE	Statement.execute(String)
Any of the above statements	

The simplest solution is to map all SQL statements to the execute() method. By default, this is the method the driver uses. Some third-party drivers, particularly Oracle's JDBC driver, incorrectly implement the methods used to determine the number of results generated by the execute() method. As a result, the driver can get caught in an infinite loop leading to high CPU utilization. To circumvent this problem, the `jdbc:type` attribute can be used on any `<jdbc:statement>` element to map the SQL statements contained therein to the executeQuery() or executeUpdate() methods instead of the default execute() method.

The `jdbc:type` attribute has two values: `update` and `query`. The value should be set to `update` for insert, update, or delete statements and `query` for select statements. In the absence of this attribute, the driver maps all SQL statements to the execute() method. If placed on any element other than `<jdbc:statement>`, this attribute is ignored.

We recommend that you place the `jdbc:type="query"` attribute value on all select statements, and the `jdbc:type="update"` attribute value on all insert, update, and delete statements.

The following XML shows an example of the `jdbc:type` attribute:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp">
          <add-attr name="lname">
               <value>Doe</value>
          </add-attr>
     </add>
     <jdbc:statement jdbc:type="update">
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'John'
           WHERE empno = {$empno}</jdbc:sql>
     </jdbc:statement>
</input>
```

SQL Queries

In order to fully support the query capabilities of a database and avoid the difficulty of translating native SQL queries into an XML format, the driver supports native SQL query processing. Select statements can be embedded in XML documents in exactly the same way as any other SQL statement.

For example, if we assumed the contents of table `emp` were:

empno	fname	lname
1	John	Doe

The XML document below would result in an output document containing a single result set.

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <jdbc:statement jdbc:type="query">
          <jdbc:sql>SELECT * FROM dirxml.emp</jdbc:sql>
     </jdbc:statement>
</input>

<output xmlns:jdbc="urn:dirxml:jdbc">
     <jdbc:result-set jdbc:number-of-rows="1">
          <jdbc:row jdbc:number="1">
               <jdbc:column jdbc:name="empno"
                            jdbc:position="1"
                            jdbc:type="java.sql.Types.DECIMAL
                     <jdbc:value>l</jdbc:value>
                </jdbc:column>
                <jdbc:column jdbc:name="fname"
                             jdbc:position="2"
                             jdbc:type="java.sql.Types.VARCHAR>
                     <jdbc:value>John</jdbc:value>
                 </jdbc:column>
                 <jdbc:column jdbc:name="lname"
                           jdbc:position="3"
                             jdbc:type="java.sql.Types.VARCHAR>
                      <jdbc:value>Doe</jdbc:value>
                  </jdbc:column>
           </jdbc:row>
      </jdbc:result-set>
      <status level="success"/>
</output>
```

SQL queries always produce a single `<jdbc:result-set>` element whether or not the result set contains any rows. If the result set is empty, the `jdbc:number-of-rows` attribute will be set to zero.

More than one query can be embedded in a document. SQL queries do not require that the tables being references are known to the driver; XML queries do.

Data Definition Language (DDL) Statements

It is generally not possible to run a Data Definition Language (DDL) statement in a database trigger because most databases do not allow mixed DML and DDL transactions. While virtual triggers do not overcome this transactional limitation, they do allow DDL statements to be executed as a side effect of an XML event. For example:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp">
          <add-attr name="lname">
               <value>Doe</value>
          </add-attr>
     </add>
     <jdbc:statement>
          <jdbc:sql>CREATE USER dirxml IDENTIFIED BY novell
          </jdbc:sql>
     </jdbc:statement>
</input>
```

The above XML resolves to:

```
INSERT INTO dirxml.emp(lname) VALUES('Doe');
/* COMMIT;  implicit */

CREATE USER dirxml IDENTIFIED BY novell;
/* COMMIT;  implicit */
```

Using the `jdbc:transaction-id` and `jdbc:transaction-type` attributes to group DML and DDL statements into a single transaction would result in the transaction being rolled back on most databases. Because DDL statements are generally executed as separate transactions, it is possible that the insert statement in the example above might succeed and the create user statement might roll back. It is not possible, however, that the insert statement fail and the create user statement succeed. The driver stops executing chained transactions at the point where the first transaction is rolled back.

Logical Operations

Because it is not generally possible to mix DML and DDL statements in a single transaction, a single event can consist of one or more transactions. The `jdbc:op-id` and `jdbc:op-type` can be used to group multiple transactions together into a single logical operation. When so grouped, all members of the operation are handled as a single unit with regard to status. If one member errors, all members return the same status level. Similarly, all members share the same status type.

```
<input xmlns:jdbc="urn:dirxml:jdbc">
     <add class-name="emp" jdbc:op-id="0"
          jdbc:op-type="password-set-operation">
          <add-attr name="lname">
               <value>Doe</value>
          </add-attr>
          <password>Doe{$empno}</password>
     </add>
     <jdbc:statement jdbc:op-id="0">
          <jdbc:sql>CREATE USER Doe IDENTIFIED BY {$$password}
          </jdbc:sql>
     </jdbc:statement>
</input>
```

The `jdbc:op-type` attribute is ignored on all elements except the first element in the operation.

Best Practices

For performance reasons, it is better to call a single stored procedure that contains multiple statements than to embed multiple SQL statements in an XML document. For example:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
      <add class-name="emp">
           <add-attr name="lname">
                <value>Doe</value>
           </add-attr>
      </add>
      <jdbc:statement>
           <jdbc:sql>CALL PROCEDURE set_fname('John', 'Joe',
           'Jimmy')</jdbc:sql>
      </jdbc:statement>
</input>
```

Is preferred to:

```
<input xmlns:jdbc="urn:dirxml:jdbc">
      <add class-name="emp">
           <add-attr name="lname">
                <value>Doe</value>
          </add-attr>
     </add>
     <jdbc:statement>
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'John'
           WHERE empno = {$empno}</jdbc:sql>
     </jdbc:statement>
     <jdbc:statement>
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'Joe'
           WHERE empno = {$empno}</jdbc:sql>
     </jdbc:statement>
     <jdbc:statement>
          <jdbc:sql>UPDATE dirxml.emp SET fname = 'Jimmy'
           WHERE empno = {$empno}</jdbc:sql>
     </jdbc:statement>
</input>
```

6 Using the JDBC Association Utility

This section contains information on using the JDBC association utility. The utility is designed to normalize associations of objects associated under the 1.0 or later drivers. It also provides several other features designed to simplify driver administration.

This version of the utility is backwards compatible with all versions of the JDBC driver back to version 1.0 and supersedes all previous versions of the utility.

Understanding the Utility

This utility supports seven independent operations:

1. List objects associated with a driver (default)
2. List objects with multiple associations to a driver
3. List objects with invalid associations to a driver

   An association is invalid if:

   - It is malformed. (For example, the association is missing the schema RDN, missing the table RDN, or the schema keyword is misspelled.)
   - It contains database identifiers that do not map to identifiers in the target database. (For example, an association includes a mapping to a table that does not exist.)
   - It maps to no row or multiple rows. An association is broken if it doesn't map to a row. Also, associations aren't unique if they map to more than one row.
4. List objects that need to be normalized
   - A normalized association is valid, correctly ordered, and uses the correct case. Normal case is uppercase for case-insensitive databases and mixed case for case-sensitive databases.
5. Normalize object associations listed by the previous operation
6. List object associations to be modified
   - Allows for global replacement of schema, table, and column names based on search criteria.
7. Modify object associations listed by the previous operation

The following table lists the operations and whether they are read-only or write.

Operation	Read-Only vs. Write
1. List objects associated with a driver	Read-only
2. List objects with multiple associations to a driver	Read-only

Operation	Read-Only vs. Write
3. List objects with invalid associations to a driver	Read-only
4. List objects that need to be normalized	Read-only
5. Normalize object associations listed by the previous operation	Write
6. List object associations to be modified	Read-only
7. Modify object associations listed by the previous operation	Write

Before You Begin

Modifying associations can potentially cause problems. If associations are corrupted, DirXML ceases to function, so you should use write operations only when necessary. To avoid unintentional association corruption, this utility creates an undo ldiff file for all write operations.

You should review the following cautions before using the utility:

- This utility, like the driver, assumes database identifiers are undelimited (unquoted and contain no special characters).
- It is extremely important that all object associations related to a driver be updated together.
  - In order to see all of the objects associated with a particular driver, this utility should be run on the Novell® eDirectory™ server where the driver is run or where the driver is being remoted from.
  - All of the objects associated with a particular driver must be contained by the LDAP search base.

    NOTE: To ensure complete containment, we recommend that you use your tree's root container as the search base.
- Make sure the JDBC URL of target database supplied to this utility is the same as the one used by the driver. Pointing this utility at a case-insensitive database when the database is actually case-sensitive might result in associations being normalized to the wrong case.
- Because this utility is run locally, it uses an unsecured connection, so the eDirectory LDAP server must be temporarily configured to accept clear text passwords. Depending upon the third-party JDBC driver you are using, the database connection established by this utility might be insecure.

  NOTE: We recommend changing the shim's authentication password on the database after running this utility.

Using the Utility

This utility must be run once for each instance of the driver installed on the target server.

A properties file is provided for each supported database and can be found in tools\sql*database*\properties.txt file.

NOTE: For more information on how to run the utility from the command line, refer to run.bat in the tools\util directory of the download image.

1 Stop the driver.

2. Identify and remove extraneous associations (operations 2 and 3).

   No object associated by the JDBC driver should have multiple associations. Extraneous associations must be removed manually on a per object basis. Operation 3 might help you identify which of the multiple associations is actually valid. After this is known, the extraneous associations can probably be discarded.

3. Identify and fix invalid associations (operation 3 and possibly operations 6 and 7).

   As a general rule of thumb, if the problem is isolated, edit each invalid association manually. If the problem is repetitive and affects a large number of associations, consider using operations 6 and 7. This utility can replace bad identifiers on a global basis, but cannot insert or remove them where they do not already exist.

4. Normalize associations (operations 4 and 5).

Editing Associations

This utility requires two parameters (*oldRDN* and *newRDN*) for operations 6 and 7. This section explains how to use these parameters.

The first value is the search criterion, the second is the replacement value. The wildcard character * can be used under certain scenarios to generalize the search criterion or replacement value.

Three types of search and replace operations are possible:

1. Schema name replacement

   Wildcards are supported on the right side only. For example,

   - Replace schema `old` with schema `new`

     oldRDN: **`schema=old`**

     newRDN: **`schema=new`**

2. Table name replacement

   Wildcards are not supported. For example,

   - Replace table `old` with table `new`:

     oldRDN: **`table=old`**

     newRDN: **`table=new`**

3. Column name replacement

   Wildcards are required on the right side, but they aren't supported on the left side. For example,

   - Replace column `old` with column `new`:

     oldRDN: **`old=*`**

     newRDN: **`new=*`**

7 Uninstalling the Driver and Database Objects

In this section, you will learn how to uninstall a driver and its corresponding database objects.

Uninstalling the Driver

This section provides information about uninstalling the driver.

When deleting Novell® eDirectory™ objects, you must delete all child objects before you can delete a parent object. For example, you must delete all rules and style sheets on the Publisher channel before you can delete the Publisher object. Similarly, you must delete both the Publisher and Subscriber objects before you can delete the Driver object.

To remove a driver object from eDirectory:

1. In Novell iManager, click DirXML Management > Overview.
2. From Overview, locate the driver set where the driver exists, then click Delete Driver.
3. Click the Driver you want to delete, then click ok.

Uninstalling Database Objects

This section provides information and procedures about uninstalling the database objects.

This section contains information to help you:

- "Uninstalling Oracle Objects" on page 79
- "Uninstalling Microsoft SQL Server Objects" on page 80
- "Uninstalling IBM DB2 UDB Objects" on page 80
- "Uninstalling Sybase Objects" on page 80
- "Uninstalling MySQL Objects" on page 80
- "Uninstall Informix Objects" on page 80

IMPORTANT: We recommend installing and uninstalling preconfigured drivers and database scripts as a unit. To prevent unintentional mismatching, database scripts and preconfigured drivers now contain headers with a version number, the target database name, and the database version.

Uninstalling Oracle Objects

1. From an Oracle client, such as SQL Plus, log in in as user **SYSTEM**. By default, the `SYSTEM` user password is **MANAGER**.
2. Execute the uninstallation script for direct or indirect synchronization. For example:

   SQL> @c:\tools\sql\oracle\direct\UNINSTALL_DIRECT.sql

   SQL> @c:\tools\sql\oracle\indirect\UNINSTALL_INDIRECT.sql

Uninstalling Microsoft SQL Server Objects

1. Start Query Analyzer.
2. Log on to your sever as user **sa**. By default, the sa user has no password.
3. Open and execute the uninstallation script for direct or indirect synchronization. For example:

   tools\sql\mssql\direct\UNINSTALL_DIRECT.sql

   tools\sql\mssql\indirect\UNINSTALL_INDIRECT.sql

Uninstalling IBM DB2 UDB Objects

1. Start Command Center.
2. Click the Script tab > open the Script menu > import the uninstallation script for direct or indirect synchronization. For example:

   tools\sql\db2\direct\UNINSTALL_DIRECT.sql

   tools\sql\db2\indirect\UNINSTALL_INDIRECT.sql
3. Change the name of the administrator account and password for your server before executing the uninstallation script.
4. Execute the script.

NOTE: The uninstall script does not destroy the dirxml database or dirxml OS user account.

Uninstalling Sybase Objects

1. From a Sybase client, such as isql, log on as user **sa** and execute the uninstallation script for direct or indirect synchronization. By default, the sa user has no password. For example:

   ```
   isql -U sa -P -i
   c:\tools\sql\sybase\direct\UNINSTALL_DIRECT.sql
   ```

   ```
   isql -U sa -P -i
   c:\tools\sql\sybase\indirect\UNINSTALL_INDIRECT.sql
   ```

Uninstalling MySQL Objects

1. From a MySQL client, such as mysql, log on as user **root** and execute the uninstallation script for indirect synchronization. By default, the root user has no password. For example:

   mysql> \. c:\tools\sql\oracle\indirect\UNINSTALL_INDIRECT.sql

Uninstall Informix Objects

1. Start SQL Editor.
2. Logon to your server as user **informix**. By default, the informix user password is **informix**.
3. Execute the uninstallation script for direct or indirect synchronization. For example:

   tools\sql\informix\direct\UNINSTALL_DIRECT.sql

   tools\sql\informix\indirect\UNINSTALL_INDIRECT.sql

NOTE: The uninstall script does not destroy the dirxml OS user account.

A Best Practices

The following section lists important best practices for using the driver. You can find additional information in Chapter 4, “Configuring the Driver,” on page 37 and Chapter 5, “Advanced Driver Configuration,” on page 49.

- For direct synchronization, you must prefix one or more view column names with `pk_` (case-insensitive).
- For indirect synchronization, ensure that all tables comprising a logical database class have the same primary and foreign key column names.
- For both direct and indirect synchronization, ensure that you use different primary key and foreign key column names between logical database classes.
- Primary key values placed in the `table_key` field should be delimited (that is, double-quoted) if they contain the following characters:

  , ; ’ + = \ " < >

  This is usually only an issue if the primary key column has a binary type. An example is provided in the tools\sql\example\pbx directory.
- When eDirectory is the authoritative source of primary key values, GUID rather than CN is recommended for use as a primary key. Unlike CN, GUID is single-valued and does not change.
- Foreign key columns should always be omitted from publication triggers.
- DSTrace should not be used in a production environment.
- Do not include primary key columns in publication triggers if they are static (that is, they do not change.)

- We recommend that you place the `jdbc:type="query"` attribute value on all embedded select statements, and the `jdbc:type="update"` attribute value on all embedded insert, update, and delete statements.
- For performance and security reasons, you should run the driver remotely whenever possible.

B Common Questions

The following section contains answers to some common questions you might encounter as you install or configure the driver. These include:

Why Can't the Driver See My Tables or Views?

The driver is capable only of synchronizing tables that have explicit primary key constraints. Explicit constraints are used by the driver to determine which fields should be utilized when constructing associations. As such, the driver ignores any unconstrained tables.
If you are trying to synchronize with tables that lack explicit constraints, you will need to either add them or synchronize to intermediate tables with the required constraints. The latter is the preferred solution.

To be seen by the driver, a view must contain at least one column name prefixed with `pk_` (case-insensitive).

How Do I Synchronize Tables Located in Multiple Schemas?

You'll need to either alias the tables into this driver's schema, synchronize to intermediate tables in the driver's schema and move the data across schema boundaries, use a view, or create a virtual schema via the new Synchronize Tables driver parameter.

Why Isn’t the Driver Processing Records in the Event Log?

There are several explanations for this behavior. First, you should check the `perpetrator` field of the rows in question and make sure the value is set to something other than the driver’s user name. The driver only checks the `perpetrator` field if the publisher Allow Loopback parameter is set to `no`. The driver prevents event loopback by ignoring all records where the `perpetrator` field value is equal to the driver’s username.

You should also ensure that the record's `status` field is set to 'N' (new). Records with `status` fields set to something other than 'N' will not be processed. Also, make sure to explicitly commit changes. Changes are only tentative until you commit them.

Can the Driver Manage Database User Accounts?

Yes, database accounts can be managed using embedded SQL. For more information, refer to “Using Structured Query Language in XML Events” on page 66.

Can the Driver Synchronize Large Binary and String Data Types?

Yes. Large binary and string data types can be subscribed and published. Large binary and string data types can be published using query-back event types.

Why is Publication so Slow?

If the event log table contains a large number of rows, it should be indexed. Example indexes are provided in all database installation scripts. The statements used by the driver to maintain the event log can be viewed using trace level 3. Examples are also provided in the tools\sql\example\STATEMENTS.sql file.

Indexes in the installation scripts can be further refined to enhance publication performance. Placing indexes in a different tablespace or physical disk than the event log will also enhance publication performance.

Also, the Delete From Log publication parameter should be set to **no** in a production environment.

Can the Driver Synchronize Multiple Classes?

Yes. However, primary key column names must be unique between logical database classes. For example, if *class1* is mapped to *table1* with primary key column name *key1* and *class2* is mapped to *table2* with primary key column name *key2*, then the name of *key1* cannot equal *key2*. This requirement can always be satisfied if intermediate tables or views are used.

Why Must Foreign Key Column and Primary Key Columns Have the Same Name?

Within each logical database class, primary key and foreign key column names must match. Between logical database classes, they must differ. This common name is used by the publisher to identify all records in the event log table pertaining to a single, logical database object even if the object spans multiple tables.

Does the Driver Support SSL Encryption?

No. How the driver communicates with a given database is dependent upon the third-party driver being used. Some third-party drivers support SSL sockets while others do not. Even if SSL is supported, there is no standardized way of enabling SLL encryption between third-party drivers. The general solution for this problem is to remotely run the driver and your third-party driver which allows the driver and your third-party driver to run locally on the database server. All data traveling across the network between the engine and the driver will be SLL encrypted.

Another possibility is to use a type 3 or type 2 third-party JDBC driver. Database middleware and client APIs usually provide some sort of secure connectivity.

How Do I Map Multi-Valued Attributes to Single-Valued Database Fields?

For detailed information on how to map multi-valued attributes to single-valued database fields, refer to "Mapping Multi-Valued Attributes to Single-Valued Database Fields" on page 57.

Why is the Driver Synchronizing Garbage Strings?

The database and the third-party driver are probably using incompatible character encoding. This can be remedied by adjusting the character encoding used by your third-party driver.

For more information, refer to the Character Encoding Values (http://java.sun.com/products/jdk/1.1/docs/guide/intl/encoding.doc.html) defined by Sun.

C Supported Data Types

The driver is capable of synchronizing JDBC 1.0 string, numeric, time, and binary data types. How JDBC data types map to a database's native data types is database-dependent. The following list includes the supported java.sql types:

- Numeric Types
  - java.sql.Types.BIGINT
  - java.sql.Types.BIT
  - java.sql.Types.DECIMAL
  - java.sql.Types.DOUBLE
  - java.sql.Types.NUMERIC
  - java.sql.Types.REAL
  - java.sql.Types.FLOAT
  - java.sql.Types.INTEGER
  - java.sql.Types.SMALLINT
  - java.sql.Types.TINYINT
- String Types
  - java.sql.Types.CHAR
  - java.sql.Types.LONGCHAR
  - java.sql.Types.VARCHAR
- Time Types
  - java.sql.Types.DATE
  - java.sql.Types.TIME
  - java.sql.Types.TIMESTAMP
- Binary Types
  - java.sql.Types.BINARY
  - java.sql.Types.VARBINARY
  - java.sql.Types.LONGVARBINARY

D java.sql.DatabaseMetaData Methods

This section lists the required and optional java.sql.DatabaseMetaData methods currently used by the driver. For more information on these methods, refer to Sun's Web Site on Interface MetaData (http://java.sun.com/products/jdk/1.2/docs/api).

Required methods:

- java.sql.ResultSet getColumns(java.lang.String catalog, java.lang.String schemaPattern, java.lang.String tableNamePattern, java.lang.String columnNamePattern)
- java.sql.ResultSet getPrimaryKeys(java.lang.String catalog, java.lang.String schema, java.lang.String table)
- java.sql.ResultSet getTables(java.lang.String catalog, java.lang.String schemaPattern, java.lang.String tableNamePattern, java.lang.String[] types)
- boolean storesLowerCaseIdentifiers()
- boolean storesMixedCaseIdentifiers()
- boolean storesUpperCaseIdentifiers()

Optional methods:

- boolean dataDefinitionCausesTransactionCommit()
- boolean dataDefinitionIgnoredInTransactions()
- java.sql.ResultSet getExportedKeys(java.lang.String catalog, java.lang.String schema, java.lang.String table)
- int getMaxConnections()
- int getMaxColumnsInSelect()
- int getMaxStatements()
- int getMaxStatementLength()
- java.sql.ResultSet getTableTypes()
- java.lang.String getUserName()
- boolean supportsDataDefinitionAndDataManiuplationTransactions()
- boolean supportsDataManipulationTransactionsOnly()
- boolean supportsSchemasInDataManipulation()
- boolean supportsSchemasInProcedureCalls()
- boolean supportsTransactions()
- boolean supportsMultipleTransactions()
- boolean supportsTransactionIsolationLevel(int level)

E JDBC 1.0 Methods

This section lists the JDBC 1.0 methods (other than DatabaseMetaData methods) used by the driver. Methods are organized by class. Often, third-party driver vendors list defects or known issues by method. This section can be used in collaboration with third-party driver documentation to troubleshoot or anticipate potential interoperability problems.

- java.sql.DriverManager

  java.sql.Connection getConnection(java.lang.String url, java.lang.String user, java.lang.String password)

- java.sql.PreparedStatement

  void clearParameters()
  void setNull(int parameterIndex, int sqlType)
  void setString(int parameterIndex, java.sql.String x)
  void setBoolean(int parameterIndex, boolean x)
  void setBigDecimal(int parameterIndex, java.math.BigDecimal x)
  void setLong(int parameterIndex, long x)
  void setDouble(int parameterIndex, double x)
  void setInt(int parameterIndex, int x)
  void setFloat(int parameterIndex, float x)
  void setShort(int parameterIndex, short x)
  void setByte(int parameterIndex, byte x)
  void setTimestamp(int parameterIndex, java.sql.Timestamp x)
  void setTime(int parameterIndex, java.sql.Time x)
  void setDate(int parameterIndex, java.sql.Date x)
  void setBytes(int parameterIndex, bytes[] x)

- java.sql.Statement

  void clearWarnings()
  void close()
  boolean execute(String sql)
  java.sql.ResultSet executeQuery(String sql)
  int executeUpdate(String sql)
  boolean getMoreResults()
  int getUpdateCount()
  java.sql.ResultSet getResultSet()

- java.sql.CallableStatement

  void registerOutParameter(int parameterIndex, int sqlType)

- java.sql.Connection

  void close()
  void commit()
  void rollback()
  int getTransactionIsolation()
  void setAutoCommit(boolean autoCommit)
  java.sql.PreparedStatement prepareStatement(String sql)
  java.sql.CallableStatement prepareCall(String sql)
  java.sql.Statement createStatement()

- java.sql.ResultSet

  void close()
  boolean next()
  java.lang.String getString(int columnIndex)
  java.lang.String getString(java.lang.String columnName)
  java.math.BigDecimal getBigDecimal(int columnIndex, int scale)
  long getLong(int columnIndex)
  double getDouble(int columnIndex)
  int getInt(int columnIndex)
  float getFloat(int columnIndex)
  short getShort(int columnIndex)
  byte getByte(int columnIndex)
  boolean getBoolean(int columnIndex)
  byte[] getBytes(int columnIndex)
  byte[] getBytes(java.lang.String columnName)
  java.sql.Timestamp getTimestamp(int columnIndex)
  java.sql.Time getTime(int columnIndex)
  java.sql.Date getDate(int columnIndex)
  java.io.InputStream getBinaryStream(String columnName)