Java DB Server and Administration Guide

Version 10.10

Derby Document build:
April 7, 2014, 11:54:23 AM (EDT)

Contents

Copyright

Related information

License

License

The Apache License, Version 2.0

that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.  See the License for the specific language governing
permissions and limitations under the License.
```

Relationship between Java DB and Apache Derby

Java DB is a relational database management system that is based on the Java programming language and SQL. Java DB is the Oracle release of the Apache Derby project, the Apache Software Foundation's (ASF) open source relational database project.

The Java DB product includes Derby without any modification whatsoever to the underlying source code.

Because Java DB and Derby have the same functionality, the Java DB documentation refers to the core functionality as Derby.

The Java DB 10.10 documentation is based on the Derby 10.10 documentation. References to "Derby" in the Java DB documentation should be understood as synonyms for "Java DB."

Oracle has made changes to the Apache Derby documentation. This manual is identical to the *Derby Server and Administration Guide*, with the following exceptions:

- Oracle has added this topic, "Relationship between Java DB and Apache Derby".
- In the titles of manuals, "Derby" has been changed to "Java DB".
- Path names that refer to locations within a Derby installation have been changed to refer to locations within a Java Development Kit (JDK) installation as needed. Specifically, demos and samples are in a different location within the JDK from their location in a Derby installation.

About this guide

For general information about the Derby documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with Java DB*.

For more information about Derby, visit the Derby website at http://db.apache.org/derby. The website provides pointers to the Derby Wiki and other resources, such as the derby-users mailing list, where you can ask questions about issues not covered in the documentation.

Purpose of this guide

This guide explains how to use Derby in a multiple-client environment. It also provides information that a server administrator might need to keep Derby running with a high level of performance and reliability in a server framework or in a multiple-client application server environment.

When running in embedded mode, Derby databases typically do not need any administration.

To connect multiple clients with Derby, you can embed Derby in a server framework that you choose, or you can use the Derby Network Server. This guide describes these options.

Audience

This guide has two parts, each for a different audience.

The first part of this guide is intended for developers of client/server and multiple-client applications. The second part of this guide is intended for administrators.

How this guide is organized

This guide includes the following two parts.

Part One: Derby Server Guide

- *Derby in a multi-user environment*

  Describes the different options for embedding Derby in a server framework and explains the Network Server option.
- *Using the Network Server with preexisting Derby applications*

  Describes how to change existing Derby applications to work with the Network Server.
- *Managing the Derby Network Server*

  Describes how to use shell scripts, the command line, and the Network Server API to manage the Network Server.
- *Managing the Derby Network Server remotely by using the servlet interface*

  Describes how to use the servlet interface to manage the Network Server. (The servlet interface should be used for testing purposes only, not in production.)
- *Derby Network Server advanced topics*

  Describes advanced topics for Derby Network Server users.
- *Derby Network Server sample programs*

Describes several Derby Network Server sample programs for Network Server users.

Part Two: Derby Administration Guide

- *Maintaining database integrity*

  Describes how to prevent Derby database corruption.
- *Checking database consistency*

  Describes how to check the consistency of Derby databases.
- *Backing up and restoring databases*

  Describes how to back up and restore a database.
- *Importing and exporting data*

  Describes how to import and export large amounts of data between files and database tables.
- *Replicating databases*

  Describes how to replicate databases.
- *Logging on a separate device*

  Describes how to put a database's log on a separate device, which can improve the performance of large databases.
- *Obtaining locking information*

  Describes how to get detailed information about locking status.
- *Reclaiming unused space*

  Describes how to identify and reclaim unused space in tables and related indexes.

Part One: Derby Server Guide

This part of the guide explains the Derby Network Server and other server frameworks.

Derby in a multi-user environment

This section describes how to use Derby in a multi-user (or "server") environment.

Derby in a server framework

In a sense, Derby is always an embedded product. You can embed it in an application in which users access the database from a single Java Virtual Machine (JVM), or you can embed it in a server framework (an application that allows users from different JVMs to connect to Derby simultaneously).

When Derby is embedded in an application, the local JDBC driver calls the local Derby database.

When Derby is embedded in a server framework, the server framework's connectivity software provides data to multiple client JDBC applications over a network or the Internet.

For local or remote multi-user connectivity (multiple users who access Derby from different JVMs), use the Derby Network Server. If you require features that are not included in the Network Server, you can embed the basic Derby product in another server framework.

Connectivity configurations

There are several ways to embed Derby in a server framework.

Use the Network Server
: This is the easiest way to provide connectivity to multiple users who are accessing Derby databases from different JVMs. The Derby Network Server provides this kind of connectivity to Derby databases within a single system or over a network.

Purchase another server framework
: You can use Derby within many server frameworks, such as IBM WebSphere Application Server.

Write your own framework
: Derby's flexibility allows other configurations as well. For example, rather than embedding Derby in a server that communicates with a client that uses JDBC, you can embed Derby within a servlet in a web server that communicates with a browser using HTTP.

Multiple-client features available in Derby

Derby contains some features that are useful for developing multi-user applications.

Row-level locking:

To support multi-user access, Derby uses row-level locking.

However, you can configure Derby to use table-level locking in environments that have few concurrent transactions (for example, a read-only database). Table-level locking is preferable if there are few or no writes to the server, while row-level locking is essential for good performance if many clients write to the server concurrently. The Derby optimizer tunes lock choice for queries automatically.

Multiple concurrency levels:

Derby supports the SERIALIZABLE (RR), REPEATABLE (RS), READ COMMITTED (CS), and READ UNCOMMITTED (UR) isolation levels.

CS
CS (the default isolation level) provides the best balance between concurrency and consistency in multiple-client environments.

RS
RS provides less consistency than RR but allows more concurrency.

RR
RR provides greatest consistency.

UR
UR provides maximum concurrency, if uncommitted values are allowed in the query. It is typically used if approximate results are acceptable.

See "Types and Scope of Locks in Derby Systems" in the *Java DB Developer's Guide* for more information.

Multi-connection and multi-threading:

Derby allows multiple simultaneous connections to a database, even in embedded mode.

Derby is also fully multi-threaded, and you can have multiple threads active at the same time. However, JDBC semantics impose some limitations on multi-threading. See the *Java DB Developer's Guide* for more information.

Administrative tools:

Derby provides some tools and features to assist database administrators, including the following.

- Consistency checker
- Online backup
- Procedures for importing and exporting data
- Database replication
- The ability to put a database's log on a separate device
- Locking information monitoring
- Reclaiming unused space

These tools and features are discussed in Part Two of this guide. See the sections in that part for more information.

The Derby Network Server

The Derby Network Server provides multi-user connectivity to Derby databases within a single system or over a network.

The Network Server uses the standard Distributed Relational Database Architecture (DRDA) protocol to receive and reply to queries from clients. Databases are accessed through the Derby Network Server by using the Derby Network Client driver.

The Network Server is a solution for multiple JVMs that connect to the database, unlike the embedded scenario where only one JVM runs as part of the system. When Derby is embedded in a single-JVM application, the embedded JDBC driver calls the local Derby database. When Derby is embedded in a server framework, the server framework's connectivity software provides data to multiple client JDBC applications over a network or the Internet.

To run the Derby Network Server, you need to install the following files:

- On the server side, install `derby.jar` and `derbynet.jar`.
- On the client side, install `derbyclient.jar`.

There are several ways to manage the Derby Network Server, including:

- Through the command line
- By using `.bat` and `.ksh` scripts
- With your own Java program (written using the Network Server API)
- By setting Network Server properties

Using the Network Server with preexisting Derby applications explains how to change existing Java applications that currently run against Derby in embedded mode to run against the Derby Network Server.

Managing the Derby Network Server explains how to manage the Network Server by using the command line, including starting and stopping it.

Derby Network Server advanced topics contains advanced topics for Derby Network Server users.

Because of the differences in JDBC drivers that are used, you might encounter differences in functionality when running Derby in the Network Server framework as opposed to running it embedded in a user application. Refer to Using the Network Server with preexisting Derby applications for a complete list of the differences between embedded and Network Server configurations.

Embedded servers

Because Derby is written in the Java programming language, you have great flexibility in how you choose to configure your deployment.

For example, you can run Derby, the JDBC server framework, and another application in the same JVM as a single process.

How to start an embedded server from an application

In one thread, the embedding application starts the local JDBC driver for its own access.

```
/*
    If you are running on JDK 6 or higher, you do not
    need to invoke Class.forName(). In that environment, the
    EmbeddedDriver loads automatically.
*/
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sample");
```

In another thread, the same application starts the server framework to allow remote access. Starting the server framework from within the application allows both the server and the application to run in the same JVM.

Embedded server example

You can start the Network Server in another thread automatically when Derby starts by setting the `derby.drda.startNetworkServer` property, or you can start it by using a program.

See Setting Network Server properties for details on this property.

The following example shows how to start the Network Server by using a program:

```
import org.apache.derby.drda.NetworkServerControl;
import java.net.InetAddress;
NetworkServerControl server = new NetworkServerControl
 (InetAddress.getByName("localhost"),1527);
server.start(null);
```

The program that starts the Network Server can access the database by using either the embedded driver or the Network Client driver. The server framework's attempt to

boot the local JDBC driver is ignored because it has already been booted within the application's JVM. The server framework simply accesses the instance of Derby that is already booted. There is no conflict between the application and the server framework.

The remote client can then connect through the Derby client driver:

```
String nsURL="jdbc:derby://localhost:1527/sample";
java.util.Properties props = new java.util.Properties();
props.setProperty("user","usr");
props.setProperty("password","pwd");

/*
    If you are running on JDK 6 or higher, you do not
    need to invoke Class.forName(). In that environment, the
    ClientDriver loads automatically.
*/
Class.forName("org.apache.derby.jdbc.ClientDriver");
Connection conn = DriverManager.getConnection(nsURL, props);

/*interact with Derby*/
Statement s = conn.createStatement();

ResultSet rs = s.executeQuery(
"SELECT * FROM HotelBookings");
```

About this guide and the Network Server documentation

This guide assumes that you are familiar with Derby features and tuning.

Before reading this guide, you should first learn about basic Derby functionality by reading the *Java DB Developer's Guide*. Also, because multi-user environments typically have performance and tuning issues, you should read *Tuning Java DB*.

Using the Network Server with preexisting Derby applications

You must modify Java applications that currently run against Derby in embedded mode so that they work with the Derby Network Server.

The topics in this section discuss these changes.

The Network Server and Java Virtual Machines (JVMs)

The Derby Network Server is compatible with Java Platform, Standard Edition (Java SE) 5 and above.

Installing required jar files and adding them to the classpath

To use the Network Server and network client driver, add the following jar file to your server classpath.

- `derbyrun.jar`

Adding this file to your classpath has the effect of including all of the Derby classes in your classpath. These classes are in the following jar files, which you can also add to your classpath separately:

- `derbynet.jar`

  This jar file contains the Network Server code. It must be in your classpath to start the Network Server.
- `derby.jar`

This jar file contains the Derby database engine code. It must be in the classpath in order for the Network Server to access Derby databases. `derby.jar` is included in the `Class-Path` attribute of `derbynet.jar`'s manifest file. If you have `derbynet.jar` in the classpath and `derby.jar` is in the same directory as `derbynet.jar`, it is not necessary to include `derby.jar` explicitly.

- `derbyclient.jar`

  This jar file contains the Derby Network Client JDBC driver that is necessary for communication with the Network Server. It must be in the classpath of the application on the client side in order to access Derby databases over a network.

All of the jar files are in the `$DERBY_HOME/lib` directory.

Derby provides script files for setting the classpath to work with the Network Server. The scripts are located in the `$DERBY_HOME/bin` directory.

- `setNetworkClientCP.bat` (Windows)
- `setNetworkClientCP` (UNIX)
- `setNetworkServerCP.bat` (Windows)
- `setNetworkServerCP` (UNIX)

See Managing the Derby Network Server and *Getting Started with Java DB* for more information on setting the classpath.

Starting the Network Server

To start the Network Server, you can invoke a script, a jar file, or a class.

> Important: Always shut down the Network Server properly after use, because failure to do so might result in unpredictable side effects, such as blocked ports on the server.

You are strongly urged to enable user authentication and user authorization when you run a Network Server. For details on how to configure user authentication, see "Working with user authentication" in the *Java DB Developer's Guide*. For information on user authorization, see "Users and authorization identifiers" and "User authorizations" in the *Java DB Developer's Guide*. You are also urged to install a Java security manager with a customized security policy. For details on how to do this, see Customizing the Network Server's security policy.

If you are running Java SE 7 or later, and if you start the Derby Network Server from the command line as described here, access to databases and to other Derby files is by default restricted to the operating system account that started the Network Server. It is possible to override this default behavior. For more information, see Controlling database file access.

You can start the Network Server in any of the following ways:

- If you are relatively new to the Java programming language, follow the instructions in "Setting up your environment" in *Getting Started with Java DB* to set the `DERBY_HOME` and `JAVA_HOME` environment variables and to add `DERBY_HOME/bin` to your path. Then use the `startNetworkServer.bat` script to start the Network Server on Windows machines and the `startNetworkServer` script to start the Network Server on UNIX systems. These scripts are located in `$DERBY_HOME/bin`, where `$DERBY_HOME` is the directory where you installed Derby.

  You can run NetworkServerControl commands only from the host that started the Network Server. The following table shows the sequence of commands.

  Table 1. Commands to run the `startNetworkServer` command

Operating System	Command

Windows	`set DERBY_HOME=C:\derby`
	`set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_24`
	`set PATH=%DERBY_HOME%\bin;%PATH%`
	`startNetworkServer`

UNIX (Korn Shell)	`export DERBY_HOME=/opt/derby`
	`export JAVA_HOME=/usr/j2se`
	`export PATH="$DERBY_HOME/bin:$PATH"`
	`startNetworkServer`

- If you are a regular Java user but are new to Derby, set the `DERBY_HOME` environment variable, then use a `java` command to invoke the `derbyrun.jar` or `derbynet.jar` file, as shown in the following table.

  Table 2. Commands to invoke the Derby jar files

Operating System	Command

Windows	`set DERBY_HOME=C:\derby`
	`java -jar %DERBY_HOME%\lib\derbyrun.jar server start`
	or
	`java -jar %DERBY_HOME%\lib\derbynet.jar start`

UNIX (Korn Shell)	`export DERBY_HOME=/opt/derby`
	`java -jar $DERBY_HOME/lib/derbyrun.jar server start`
	or
	`java -jar $DERBY_HOME/lib/derbynet.jar start`

  To see the command syntax, invoke `derbyrun.jar server` or `derbynet.jar` with no arguments.
- If you are familiar with both the Java programming language and Derby, you have already set `DERBY_HOME`. Set your classpath to include the Derby jar files. Then use a `java` command to invoke the `NetworkServerControl` class directly, as shown in the following table.

  Table 3. Commands to invoke the `NetworkServerControl` class

Operating System	Command
Windows	`%DERBY_HOME%\bin\setNetworkServerCP` `java org.apache.derby.drda.NetworkServerControl start`
UNIX (Korn Shell)	`$DERBY_HOME/bin/setNetworkServerCP` `java org.apache.derby.drda.NetworkServerControl start`

The default system directory is the directory in which Derby was started. (See the *Java DB Developer's Guide* for more information about the default system directory.)

You can specify a different host or port number when you start the Network Server by specifying an option to the command.

- Specify a port number other than the default (1527) by using the `-p` *`portnumber`* option, as shown in the following example:

```
java org.apache.derby.drda.NetworkServerControl start -p 1368
```

- Specify a specific interface (host name or IP address) to listen on other than the default (`localhost`) by using the `-h` option, as shown in the following example:

  ```
  $DERBY_HOME/bin/startNetworkServer -h myhost -p 1368
  ```

  where *myhost* is the host name or IP address.

  Remember: Before using the `-h` option, you should run under the Java security manager with a customized security policy and you should enable user authentication.

By default, the Network Server will listen to requests only on the loopback address, which means that it will only accept connections from the local host.

Starting the Network Server from a Java application

Always shut down the Network Server properly after use, because failure to do so might result in unpredictable side effects, such as blocked ports on the server.

There are two ways to start the Network Server from a Java application.

- You can include the following line in the `derby.properties` file:

  ```
  derby.drda.startNetworkServer=true
  ```

  This starts the server on the default port, 1527, listening on `localhost` (all interfaces).

  To specify a different port or a specific interface in the `derby.properties` file, include the following lines, respectively:

  ```
  derby.drda.portNumber=1110
  derby.drda.host=myhost
  ```

  You can also specify the `startNetworkServer` and `portNumber` properties by using a Java command:

  ```
  java -Dderby.drda.startNetworkServer=true \
  -Dderby.drda.portNumber=1110 \
  -Dderby.drda.host=myhost yourApp
  ```

- You can use the NetworkServerControl API to start the Network Server from a separate thread within a Java application:

  ```
  NetworkServerControl server = new NetworkServerControl();
  server.start (null);
  ```

Starting the Network Server on IPv6/IPv4 dual stack Windows machines

Add the following JVM properties to the command when you start the server on an IPv6/IPv4 dual stack Windows machine.

```
-Djava.net.preferIPv4Stack=false
-Djava.net.preferIPv6Addresses=true
```

Shutting down the Network Server

To shut down the Network Server, you can invoke a script, a jar file, or a class.

The scripts to shut down the Network Server are located in the `$DERBY_HOME/bin` directory.

> Important: If user authentication is enabled, you must specify a valid Derby user name and password; if the user authentication check fails, you'll see an authentication error and the running server remains intact. Note that Derby does not yet restrict the

shutdown privilege to specific users: the server can be shut down by any user on the server machine who presents valid credentials.

- To shut down the Network Server by using the scripts provided for Windows systems, use:

```
stopNetworkServer.bat [-h hostname] [-p portnumber] [-user username]
 [-password password]
```

- To shut down the Network Server by using the scripts provided for UNIX systems, use:

```
stopNetworkServer [-h hostname] [-p portnumber] [-user username]
 [-password password]
```

Shutting down by using the command line

From the command line, you can shut down the Network Server by invoking a jar file or a class.

You must provide user credential arguments to shut down a server running with user authentication.

- To shut down the Network Server by invoking a jar file from the `$DERBY_HOME/lib` directory, use:

```
java -jar derbyrun.jar server shutdown [-h <hostname>] [-p
 <portnumber>] [-user <username>] [-password <password>]
```

  or

```
java -jar derbynet.jar shutdown [-h <hostname>] [-p <portnumber>]
 [-user <username>] [-password <password>]
```

- To shut down the Network Server by invoking a class, use:

```
java org.apache.derby.drda.NetworkServerControl shutdown [-h
 <hostname>] [-p <portnumber>] [-user <username>] [-password
 <password>]
```

Shutting down by using the API

You can use the NetworkServerControl API to shut down the Network Server from within a Java application.

The name of the method that you use to shut down the Network Server is `shutdown()`.

For example, the following command shuts down the Network Server running on the current machine using the default port number (1527):

```
NetworkServerControl server = new NetworkServerControl();
server.shutdown();
```

To shut down a server running with user authentication, you need to use a NetworkServerControl instance created with user credentials:

```
NetworkServerControl server = new NetworkServerControl(username,
 password);
server.shutdown();
```

Obtaining system information

You can obtain information about the Network Server, such as version and current property values, Java information, and Derby database server information, by using the `sysinfo` utility.

The `sysinfo` utility is available from scripts, the command line, and the NetworkServerControl API.

The following scripts are located in the `$DERBY_HOME/bin` directory. Before running these scripts, make sure that the Derby Network Server is started.

- Run the following script to obtain information about the Network Server on a Windows system:

  ```
  NetworkServerControl.bat sysinfo [-h hostname][-p portnumber]
  ```
- Run the following script to obtain information about the Network Server on a UNIX system:

  ```
  NetworkServerControl sysinfo [-h hostname] [-p portnumber]
  ```

For more information on the `sysinfo` utility, see the *Java DB Tools and Utilities Guide*.

You can also use Java Management Extensions (JMX) technology to obtain system information. For details, see Using Java Management Extensions (JMX) technology.

Obtaining system information by using the command line

To run `sysinfo` from the command line, use a command like one of the following while the Network Server is running.

```
java -jar $DERBY_HOME/lib/derbyrun.jar server sysinfo
  [-h hostname][-p portnumber]
```

```
java org.apache.derby.drda.NetworkServerControl sysinfo
  [-h hostname][-p portnumber]
```

Administrative commands such as `sysinfo` can only execute on the host where the server was started, even if the server was started with the `-h` option.

Obtaining system information by using the API

The `getSysinfo` method produces the same information as the `sysinfo` command.

The signature for this method is

```
String getSysinfo();
```

For example:

```
NetworkServerControl serverControl = new NetworkServerControl();
String myinfo = serverControl.getSysinfo();
```

The `getSysinfo` method returns information about the Network Server that is running on the current machine on the default port number (1527).

Obtaining Network Server runtime information:

Use the `runtimeinfo` command or the `getRuntimeInfo` method to get memory usage and current session information about the Network Server, including user, database, and prepared statement information.

- To run `runtimeinfo` from the command line:

  ```
  java org.apache.derby.drda.NetworkServerControl runtimeinfo
     [-h <hostname>][<-p portnumber>]
  ```
- The `getRuntimeInfo` method returns the same information as the `runtimeinfo` command. The signature for the `getRuntimeInfo` method is

  ```
  String getRuntimeInfo()
  ```

  For example:

  ```
  NetworkServerControl serverControl = new NetworkServerControl();
  String myinfo = serverControl.getRuntimeInfo();
  ```

Obtaining Network Server properties by using the getCurrentProperties method:

The `getCurrentProperties` method is a Java method that you can use to obtain information about the Network Server.

It returns a `Properties` object with the value of all the Network Server properties as they are currently set.

The signature of this method is:

```
Properties getCurrentProperties()
```

For example:

```
NetworkServerControl server = new NetworkServerControl();
Properties p = server.getCurrentProperties();
p.list(System.out);
System.out.println(p.getProperty("derby.drda.host"));
```

As shown in the previous example, you can look up the current properties and then work with individual properties if needed by using various APIs on the `Properties` class. You can also print out all the properties by using the `Properties.list` method.

Accessing the Network Server by using the network client driver

When connecting to the Network Server, your application needs to load a driver and connection URL that is specific to the Network Server. In addition, you must specify a user name and password if you are using authentication.

The driver that you need to access the Network Server is:

```
org.apache.derby.jdbc.ClientDriver
```

The syntax of the URL that is required to access the Network Server is:

```
jdbc:derby://server[:port]/
databaseName[;URL-attribute=value [;...]]
```

where the *`URL-attribute`* is either a Derby embedded or network client attribute.

To access an in-memory database using the Network Server, the syntax is:

```
jdbc:derby://server[:port]/memory:
databaseName[;URL-attribute=value [;...]]
```

For more information, see "Using in-memory databases" in the *Java DB Developer's Guide*.

For both driver and DataSource access, the database name (including path), user, password, and other attribute values must consist of characters that can be converted to UTF-8. The total byte length of the database name plus attributes when converted to UTF-8 must not exceed 255 bytes; keep in mind that in UTF-8, a character may occupy from 1 to 4 bytes. You may be able to work around this restriction for long paths or paths that include multibyte characters by setting the `derby.system.home` system property when starting the Network Server and accessing the database with a relative path that is shorter and does not include multibyte characters.

The following table shows standard JDBC DataSource properties.

Table 4. Standard JDBC DataSource properties

Property	Type	Description	URL Attribute	Notes
databaseName	String	The name of the database. This property is required.	None	This property is also available using EmbeddedDataSource.
dataSourceName	String	The data source name.	None	This property is also available using EmbeddedDataSource.
description	String	A description of the data source.	None	This property is also available using EmbeddedDataSource.
user	String	The user's account name.	user	Default is APP. This property is also available using EmbeddedDataSource.
password	String	The user's database password.	password	This property is also available using EmbeddedDataSource.
serverName	String	The host name or TCP/IP address where the server is listening for requests.	None	Default is localhost.
portNumber	Intege	The port number where the server is listening for requests.	None	Default is 1527.

The following table shows client-specific JDBC DataSource properties.

Table 5. Client-specific DataSource properties

Property	Type	Description	URL Attribute	Notes
traceFile	String	The filename for tracing output. Setting this property turns on tracing. See Network client tracing.	traceFile	None
traceDirectory	String	The directory for the tracing output. Each connection will send output to a separate file. Setting this property turns	traceDirectory	None

Property	Type	Description	URL Attribute	Notes
		on tracing. See Network client tracing.		
traceLevel	Intege	The level of client tracing if traceFile or t is set.	traceLevel	The default is TRACE_ALL.
traceFileAppend	Boolea	Value is true if tracing output should append to the existing trace file.	traceFileAppend	The default is false.
securityMechani	Intege	The security mechanism. See Network client security.	securityMechanis	The default is USER_ONLY_SEC
retrieveMessage	Boolea	Retrieve message text from the server. A stored procedure is called to retrieve the message text with each SQLException and might start a new unit of work. Set this property to false if you do not want the performance impact or when starting new units of work.	retrieveMessageT	The default is true.
ssl	String	The SSL mode for the client connection. See Network encryption and authentication with SSL/TLS.	ssl	The default is off.

The following table shows server-specific JDBC DataSource properties.

Table 6. Server-specific DataSource properties

Property	Type	Description	URL Attribute	Notes
connectionAttr	String	Set to the list of Derby embedded	Various	This property is also available using EmbeddedDataSource.

Property	Type	Description	URL Attribute	Notes
		connection attributes separated by semicolons.		See the *Java DB Reference Manual* for more information about the various connection attributes.
`createDatabase`	`String`	If set to `create`, create the database specified with the `databaseName` property.	`create`	This property is also available using `EmbeddedDataSource`. See the *Java DB Reference Manual* for more information. Similar to setting `connectionAttribute` to `create=true`. Only `create` is allowed; other values equate to `null`. The result of conflicting settings of `createDatabase`, `shutdownDatabase`, and `connectionAttributes` is undefined.
`shutdownDataba`	`String`	If set to `shutdown`, shut down the database specified with the `databaseName` property.	`shutdown`	This property is also available using `EmbeddedDataSource`. See the *Java DB Reference Manual* for more information. Similar to setting `connectionAttribut` to `shutdown=true`. Only `shutdown` is allowed; other values equate to `null`. The result of conflicting settings of `createDatabase`, `shutdownDatabase`, and `connectionAttributes` is undefined. If authentication **and** SQL authorization are both enabled, database shutdown is restricted to the database owner.

Note: The `setAttributesAsPassword` property, which is available for the embedded DataSource, is not available for the client DataSource.

Network client security

The Derby Network Client allows you to select a security mechanism by specifying a value for the `securityMechanism` property.

You can set the `securityMechanism` property in one of the following ways:

- When you are using the `DriverManager` interface, set `securityMechanism` in a `java.util.Properties` object before you invoke the form of the `getConnection` method, which includes the `java.util.Properties` parameter.
- When you are using the `DataSource` interface to create and deploy your own DataSource objects, invoke the `DataSource.setSecurityMechanism` method after you create a DataSource object.

The following table lists the security mechanisms that the Derby Network Client supports, and the corresponding property value to specify to obtain this security mechanism. The default security mechanism is the user id only if no password is set. If the password is set, the default security mechanism is both the user id and password. The default user is APP if no other user is specified.

Table 7. Security mechanisms supported by the Derby Network Client

Security Mechan	securityMechanism Property Value	Comments
User id and password	`ClientDataSource.CLEAR_TEXT_PASSWORD_` (0x03)	Default if password is set
User id only	`ClientDataSource.USER_ONLY_SECURITY` (0x04)	Default if password is not set
Encrypted user id and encrypted password	`ClientDataSource.ENCRYPTED_USER_AND_P` (0x09)	Encryption requires a JCE implementation that supports the Diffie-Hellman algorithm with a public prime of 256 bits.

Derby provides three ClientDataSource implementations. You can use the `org.apache.derby.jdbc.ClientDataSource` class on Java SE 5 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 3 or JDBC 4.0 methods. You can use the `org.apache.derby.jdbc.ClientDataSource40` class on Java SE 6 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 4.1 or JDBC 4.2 methods. You must use the `org.apache.derby.jdbc.BasicClientDataSource40` class on Java SE 8 Compact Profile 2 or 3.

Network client tracing

The Derby Network client provides a tracing facility to collect JDBC trace information and view protocol flows.

There are various ways to obtain trace output. However, the easiest way to obtain trace output is to use the `traceFile=`*`path`* attribute on the URL in `ij`. The following example shows all tracing going to the file `trace.out` from an `ij` session.

```
ij>connect 'jdbc:derby://localhost:1527/mydb;
create=true;traceFile=trace.out;user=user1;password=secret4me';
```

To append trace information to the specified file, use the `traceFileAppend=true` URL attribute in addition to `traceFile=`*`path`*.

For more information, see "traceFile=path attribute" and "traceFileAppend=true attribute" in the *Java DB Reference Manual*.

Implementing ClientDataSource tracing

You can use one of three methods to collect tracing data while obtaining connections from the `ClientDataSource`:

- Use the `setLogWriter(java.io.PrintWriter)` method of `ClientDataSource` and set the `PrintWriter` to a non-null value.
- Use the `setTraceFile(String filename)` method of `ClientDataSource`.
- Use the `setTraceDirectory(String dirname)` method of `ClientDataSource` to trace each connection flow in its own file for programs that have multiple connections.

Derby provides three `ClientDataSource` implementations. You can use the `org.apache.derby.jdbc.ClientDataSource` class on Java SE 5 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 3 or JDBC 4.0 methods. You can use the `org.apache.derby.jdbc.ClientDataSource40` class on Java SE 6 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 4.1 or JDBC 4.2 methods. You must use the `org.apache.derby.jdbc.BasicClientDataSource40`class on Java SE 8 Compact Profile 2 or 3.

Implementing DriverManager tracing

Use one of the following two options to enable and collect tracing information while obtaining connections using the `DriverManager`:

- Use the `setLogWriter(java.io.PrintWriter)` method of `DriverManager` and set the `PrintWriter` to a non null-value.
- Use the `traceFile=`*`path`* or `traceDirectory=`*`path`* URL attributes to set these properties prior to creating the connection with the `DriverManager.getConnection()` method. For more information, see "traceFile=path attribute" and "traceDirectory=path attribute" in the *Java DB Reference Manual*.

Changing the default trace level

The default trace level is `ClientDataSource.TRACE_ALL`. You can choose the tracing level by calling the `setTraceLevel(int level)` method or by setting the `traceLevel=`*`value`* URL attribute:

```
String url = "jdbc:derby://samplehost.example.com:1528/mydb" +
 ";traceFile=/u/user1/trace.out" +
 ";traceLevel=" +
 org.apache.derby.jdbc.ClientDataSource.TRACE_PROTOCOL_FLOWS;
DriverManager.getConnection(url,"user1","secret4me");
```

The following table shows the tracing levels you can set.

Table 8. Available tracing levels and values

Trace Level	Value
org.apache.derby.jdbc.ClientDataSource.TRACE_NONE	0x0
org.apache.derby.jdbc.ClientDataSource.TRACE_CONNECTION_CALLS	0x1
org.apache.derby.jdbc.ClientDataSource.TRACE_STATEMENT_CALLS	0x2
org.apache.derby.jdbc.ClientDataSource.TRACE_RESULT_SET_CALLS	0x4
org.apache.derby.jdbc.ClientDataSource.TRACE _DRIVER_CONFIGURATION	0x10
org.apache.derby.jdbc.ClientDataSource.TRACE_CONNECTS	0x20
org.apache.derby.jdbc.ClientDataSource.TRACE_PROTOCOL_FLOWS	0x40
org.apache.derby.jdbc.ClientDataSource.TRACE _RESULT_SET_META_DATA	0x80

Trace Level	Value
org.apache.derby.jdbc.ClientDataSource.TRACE_PARAMETER_META_DATA	0x100
org.apache.derby.jdbc.ClientDataSource.TRACE_DIAGNOSTICS	0x200
org.apache.derby.jdbc.ClientDataSource.TRACE_XA_CALLS	0x800
org.apache.derby.jdbc.ClientDataSource.TRACE_ALL	0xFFFFFFFF

To specify more than one trace level, use one of the following techniques:

- Use bitwise OR operators (|) with two or more trace values. For example, to trace PROTOCOL flows and connection calls, specify this value for `traceLevel`:

  ```
  TRACE_PROTOCOL_FLOWS | TRACE_CONNECTION_CALLS
  ```
- Use a bitwise complement operator (~) with a trace value to specify all except a certain trace. For example, to trace everything except PROTOCOL flows, specify this value for `traceLevel`:

  ```
  ~TRACE_PROTOCOL_FLOWS
  ```

For more information, see "traceLevel=value attribute" in the *Java DB Reference Manual*.

Network client driver examples

The following examples specify the user and password URL attributes.

To enable user authentication, you must either use NATIVE authentication or explicitly set the property `derby.connection.requireAuthentication` to `true`. Otherwise, Derby does not require a user name and password. For details on how to enable user authentication, see "Working with user authentication" in the *Java DB Developer's Guide*.

For a multi-user product, you would typically specify authentication for the system in the `derby.properties` file for your server, since it is in a trusted environment. The following property setting specifies NATIVE authentication:

```
derby.authentication.provider=NATIVE:myCredentialsDB:LOCAL
```

> Important: It is strongly recommended that production systems rely on NATIVE authentication, an external directory service such as LDAP, or a user-defined class for authentication. It is also strongly recommended that production systems protect network connections with SSL/TLS.

Example 1

The following example connects to the default server name localhost on the default port, 1527, and to the database sample.

```
jdbc:derby://localhost:1527/sample;user=judy;password=no12see
```

Example 2

The following example specifies both Derby and Network Client driver attributes:

```
jdbc:derby://localhost:1527/sample;create=true;user=judy;password=no12see
```

Example 3

This example connects to the default server name `localhost` on the default port, 1527, and includes the path in the database name portion of the URL.

```
jdbc:derby://localhost:1527/c:/my-db-dir/my-db-name;user=judy;
password=no12see
```

For a programming example that shows how to connect to the server using NATIVE authentication, see "NATIVE authentication and SQL authorization example" in the *Java DB Developer's Guide*.

Accessing the Network Server by using a DataSource object

The Network Server supports a set of Derby Network Client driver `DataSource` classes.

You can use the `org.apache.derby.jdbc.ClientDataSource` and `org.apache.derby.jdbc.ClientConnectionPoolDataSource` classes on Java SE 5 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 3 or JDBC 4.0 methods. You can use the `org.apache.derby.jdbc.ClientDataSource40` and `org.apache.derby.jdbc.ClientConnectionPoolDataSource40` classes on Java SE 6 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 4.1 or JDBC 4.2 methods. You must use the `org.apache.derby.jdbc.BasicClientDataSource40` and `org.apache.derby.jdbc.BasicClientConnectionPoolDataSource40` classes on Java SE 8 Compact Profile 2 or 3.

If your client is running on the Java SE 6 platform or higher, all connection objects returned from the `DataSource` will be JDBC 4 connection objects, whether or not you are using a `DataSource` whose name ends in "40".

Using statement caching

Derby supports JDBC statement caching, which can improve the performance of applications that use `PreparedStatement` or `CallableStatement` objects. Statement caching avoids the performance penalty incurred by going over the network from the client to the server to prepare a statement that has already been prepared on the same connection.

To use statement caching, you must use an `org.apache.derby.jdbc.ClientConnectionPoolDataSource`, `org.apache.derby.jdbc.ClientConnectionPoolDataSource40`, or `org.apache.derby.jdbc.BasicClientConnectionPoolDataSource40` object. After you instantiate this object, perform these steps:

1. Specify the desired size of your statement cache by calling the `setMaxStatements` method on the `DataSource` object, specifying an argument greater than zero.
2. Call the `getPooledConnection` method on the `DataSource` object to obtain a `javax.sql.PooledConnection` object (a physical connection).
3. Call the `javax.sql.PooledConnection.getConnection` method to obtain a `java.sql.Connection` object (a logical connection).

After you obtain a connection, use either prepared statements or callable statements to interact with the database. Close each statement to return it to the cache after you finish using it. The statements you create are held in the cache on the client side and reused when needed.

See Statement caching example for a code example.

Use of the JDBC statement cache makes each physical connection use more memory. The amount depends on how many statements the connection is allowed to cache and how many statements are actually cached.

If you enable JDBC statement caching, error handling changes slightly. Some errors that previously appeared when the `prepareStatement` method was executed may now appear during statement execution. For example, suppose you query a table using a

prepared statement that is then cached. If the table is deleted, the prepared statement that queries the table is not invalidated. If the query is prepared again on the same connection, the cached object is fetched from the cache, and the `prepareStatement` call seems to have succeeded, although the statement has not actually been prepared. When the prepared statement is executed, the error is detected on the server side, and the client is notified.

DataSource access examples

These examples use `org.apache.derby.jdbc.ClientDataSource` and `org.apache.derby.jdbc.ClientConnectionPoolDataSource` to access the Network Server.

The following example uses `org.apache.derby.jdbc.ClientDataSource` to access the Network Server:

```
org.apache.derby.jdbc.ClientDataSource ds =
   new org.apache.derby.jdbc.ClientDataSource();
ds.setDatabaseName("mydb");
ds.setCreateDatabase("create");
ds.setUser("user");
ds.setPassword("mypass");

// The host on which Network Server is running
ds.setServerName("localhost");

// The port on which Network Server is listening
ds.setPortNumber(1527);

Connection conn = ds.getConnection();
```

Statement caching example

The following example uses `org.apache.derby.jdbc.ClientConnectionPoolDataSource` to access the Network Server and use JDBC statement caching:

```
org.apache.derby.jdbc.ClientConnectionPoolDataSource cpds =
   new ClientConnectionPoolDataSource();

// Set the number of statements the cache is allowed to cache.
// Any number greater than zero will enable the cache.
cpds.setMaxStatements(20);

// Set other DataSource properties
cpds.setDatabaseName("mydb");
cpds.setCreateDatabase("create");
cpds.setUser("user");
cpds.setPassword("mypass");
cpds.setServerName("localhost");
cpds.setPortNumber(1527);

// This physical connection will have JDBC statement caching enabled.
javax.sql.PooledConnection pc = cpds.getPooledConnection();

// Create a logical connection.
java.sql.Connection con = pc.getConnection();

// Interact with the database.
java.sql.PreparedStatement ps = con.prepareStatement(
   "select * from myTable where id = ?");
...
ps.close(); // Inserts or returns statement to the cache
...
con.close();

// The next logical connection can gain from using the cache.
```

```
con = pc.getConnection();

// This prepare causes a statement to be fetched from the local cache.
PreparedStatement ps = con.prepareStatement(
   "select * from myTable where id = ?");
...

// To dispose of the cache, close the connection.
pc.close();
```

XA and the Network Server

Both the Derby embedded driver and the Network Server provide XA support. The Network Server provides DRDA level 7 support. DRDA clients that support XAMGR, such as the Derby network client, can send XA requests to the Network Server.

Using XA with the network client driver

You can access XA support for the Network Server by using the network client driver's XA DataSource interface.

You can use the `org.apache.derby.jdbc.ClientXADataSource` class on Java SE 5 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 3 or JDBC 4.0 methods. You can use the `org.apache.derby.jdbc.ClientXADataSource40` class on Java SE 6 and above (except Java SE 8 Compact Profiles), in applications that call JDBC 4.1 or JDBC 4.2 methods. You must use the `org.apache.derby.jdbc.BasicClientXADataSource40` class on Java SE 8 Compact Profile 2 or 3.

If your client is running on the Java SE 6 platform or higher, all connection objects returned from the DataSource will be JDBC 4 connection objects, whether or not you are using a DataSource whose name ends in "40".

The following example illustrates how to obtain an XA connection with the network client driver:

```
import org.apache.derby.jdbc.ClientXADataSource;
import javax.sql.XAConnection;
...

XAConnection xaConnection = null;
Connection conn = null;

ClientXADataSource ds = new ClientXADataSource();

ds.setDatabaseName ("sample");
ds.setCreateDatabase("create");

ds.setServerName("localhost");
ds.setPortNumber(1527);

xaConnection = ds.getXAConnection("auser", "shhhh");

conn = xaConnection.getConnection();
```

Using the Derby tools with the Network Server

The Derby tools `ij` and `dblook` work in embedded mode and client/server mode.

Using the Derby ij tool with the Network Server

To use the `ij` tool with the network client driver, follow these steps.

1. Start `ij` in one of the following ways. For details, see "Starting ij" in the *Java DB Tools and Utilities Guide*.
   a. Use a script.

      Run the `ij.bat` script on Windows systems and the `ij` script on UNIX systems. These scripts are located in the `$DERBY_HOME/bin` directory.
   b. Run the `ij` tool using the `$DERBY_HOME/lib/derbyrun.jar` file.

      ```
      java -jar derbyrun.jar ij
      ```
   c. Run the `ij` tool by specifying the class name.

      ```
      java org.apache.derby.tools.ij
      ```
2. Connect by specifying the URL:

   ```
   ij> CONNECT 'jdbc:derby://localhost:1527/sample'
    USER 'judy' PASSWORD 'no12see';
   ```

   See Network client driver examples for additional URL examples.

Using the Derby dblook tool with the Network Server

To use the `dblook` tool with the network client driver, follow these steps.

1. Make sure the Network Server is running. See Starting the Network Server for more information.
2. Include the necessary Derby and network client driver connection attributes as part of the database URL, as in the following example:

   ```
   java org.apache.derby.tools.dblook -d
    'jdbc:derby://localhost:1527/sample;
   user=user1;password=secret4me;'
   ```

   For details on using the `dblook` tool, see the *Java DB Tools and Utilities Guide*.

Differences between running Derby in embedded mode and using the Network Server

This section describes the differences between running Derby in embedded mode and using the Network Server.

Note: There may be undocumented differences that have not yet been identified.

Differences between the embedded client and the network client driver

The following are known differences that exist between the Derby embedded driver and the network client driver.

There may be undocumented differences that have not yet been identified. Some differences with the network client may be changed in future releases to match the embedded driver functionality.

- Error messages and `SQLStates` can differ between the network client and embedded driver.
- Treatment of error situations encountered during batch processing with `java.sql.Statement`, `java.sql.PreparedStatement`, and `java.sql.CallableStatement` is different. With the embedded driver, processing stops when an error is encountered; with the network client driver, processing continues, but an appropriate value as defined in the `java.sql.Statement` API is returned in the resulting update count array.
- To use an encrypted user id and password, you need to have the IBM's Java Cryptography Extension (JCE) Version 1.2.1 or later.

Updatable result sets

In Derby, the functionality of updatable result sets in a server environment and in an embedded environment are similar, with the exception of the following differences.

- The embedded driver allows for statement name changes when there is an open result set on the statement object. This is not supported in a server environment.
- Use of the `updateBytes` method on the CHAR, VARCHAR, and LONG VARCHAR datatypes is supported in an embedded environment, but is not supported in a server environment.

User authentication differences

When you run Derby in embedded mode or when you use the Derby Network Server, you can enable or disable server-side user authentication. However, when you use the Network Server, the default security mechanism (`CLEAR_TEXT_PASSWORD_SECURITY`) requires that you supply both the user name and password.

In addition to the default user name and password security mechanism, `org.apache.derby.jdbc.ClientDataSource.CLEAR_TEXT_PASSWORD_SECURITY`, Derby Network Server supports the following security properties:

- UserID (`org.apache.derby.jdbc.ClientDataSource.USER_ONLY_SECURITY`)

  When you use this mechanism, you must specify only the `user` property. All other mechanisms require you to specify both the user name and the password.
- Encrypted UserID and encrypted password (`org.apache.derby.jdbc.ClientDataSource.EN`

  When you use this mechanism, both password and user id are encrypted.

The user name that is specified upon connection is the default schema for the connection, if a schema with that name exists. See the *Java DB Developer's Guide* for more information on schema and user names.

If you specify any other security mechanism, you will receive an exception.

To change the default, you can specify another security mechanism either as a property or on the URL (using the `securityMechanism=value` attribute) when you make the connection. For details, see Network client security and "securityMechanism=value attribute" in the *Java DB Reference Manual*.

Whether the security mechanism you specify for the client actually takes effect depends upon the setting of the `derby.drda.securityMechanism` property for the Network Server. If the `derby.drda.securityMechanism` property is set, the Network Server accepts only connections that use the security mechanism specified by the property setting. If the `derby.drda.securityMechanism` property is not set, clients can use any valid security mechanism. For details, see derby.drda.securityMechanism property.

Security mechanism options when user authentication is enabled on the Network Server:

When user authentication is enabled in Derby, you can use either of the following security mechanisms.

- Clear text user name and password security, the default
- Encrypted user name and password security

Security mechanism options when user authentication is disabled on the Network Server:

When user authentication is turned off in Derby, you can use any of the security mechanism options.

You must provide a user and password for all security mechanisms except `USER_ONLY_SECURITY`. However, because user authentication is disabled in the Derby server, the user name and password that you supply do not have to be among those recognized as valid by Derby.

Enabling the encrypted user ID and password security mechanism:

To use the encrypted user ID and password security mechanism, you need a Java environment with a JCE (Java Cryptography Extension) which supports the Diffie-Hellman algorithm with a public prime of 256 bits.

The Java Platform, Standard Edition (Java SE) requires a public prime of 512 bits or more.

To use the encrypted user id and password security mechanism during JDBC connection using the network client, specify the `securityMechanism` in the connection property.
Note: If an encrypted database is booted in the Network Server, users can connect to the database without giving the `bootPassword`. The first connection to the database must provide the `bootPassword`, but all subsequent connections do not need to supply it. To remove access from the encrypted database, use the `shutdown=true` option to shut down the database.

Differences in JDBC methods

A few JDBC methods behave differently with the embedded driver from the way they behave with the client driver.

These methods are as follows:

```
Connection.prepareStatement(String sql, String[] columnNames)
Connection.prepareStatement(String sql, int[] columnIndexes)

Statement.execute(String sql, String[] columnNames)
Statement.execute(String sql, int[] columnIndexes)
Statement.executeUpdate(String sql, String[] columnNames)
Statement.executeUpdate(String sql, int[] columnIndexes)
```

The differences in behavior are described in "Autogenerated keys" in the *Java DB Reference Manual*.

Differences using the Connection.setReadOnly method

In embedded mode, when the `Connection.setReadOnly` method has `true` as the parameter, the connection is marked as a read-only connection. When you use the Network Server, the `Connection.setReadOnly(true)` method is ignored, and the connection is not marked as a read-only connection.

Setting port numbers

By default, the Derby Network Server listens on TCP/IP port number 1527. If you want to use a different port number, you can specify it on the command line when starting the Network Server.

For example:

```
java org.apache.derby.drda.NetworkServerControl start -p 1088
```

1. However, it is better to specify the port numbers by using any of the following methods:
   - Change the `startNetworkServer.bat` or `startNetworkServer.ksh` script
   - Use the `derby.drda.portNumber` property in `derby.properties`

See Starting the Network Server for more information.

Managing the Derby Network Server

The Derby Network Server can be run in either of the following configurations.

- As a stand-alone server, in which case it is an independent Java process embedding the Derby database engine
- As an embedded server, in which case it is embedded within another Java application, and both the Network Server framework and the Derby database engine are loaded by the Java application

You can use Java Management Extensions (JMX) technology to monitor and manage Derby and the Network Server. For information on how to do this, see Using Java Management Extensions (JMX) technology.

You can manage the Network Server by using shell scripts, the command line, or the Network Server API.

Overview of Derby Network Server management

You can start the Derby Network Server by using the command line or by using the Derby Network Server API.

Derby provides scripts for you to use to start the server from the command line. Before starting the server, you will probably set certain Derby and Network Server properties.

Using the NetworkServerControl API

You need to create an instance of the `NetworkServerControl` class if you are using the API.

There are four constructors for this class.

Note: Before enabling connections from other systems, ensure that you are running under a security manager.

- `NetworkServerControl()`

  This constructor creates an instance that listens either on the default port (1527) or the port that is set by the `derby.drda.portNumber` property. It will also listen on the host set by the `derby.drda.host` property or the loopback address if the property is not set. This is the default constructor; it does not allow remote connections. It is equivalent to calling `NetworkServerControl(InetAddress.getByName("localhost"),1527)` if no properties are set.
- `NetworkServerControl(InetAddress address, int portNumber)`

  This constructor creates an instance that listens on the specified `portNumber` on the specified address. The `InetAddress` will be passed to `ServerSocket`. NULL is an invalid address value. The following examples show how you might allow the Network Server to accept connections from other hosts:

  ```
  // accepts connections from other hosts on an IPv4 system
  NetworkServerControl serverControl =
    new NetworkServerControl(InetAddress.getByName("0.0.0.0"),1527);
  ```

  ```
  // accepts connections from other hosts on an IPv6 system
  NetworkServerControl serverControl =
    new NetworkServerControl(InetAddress.getByName("::"),1527);
  ```
- `NetworkServerControl(String userName, String password)`

  If a network server should run with user authentication, certain operations like `NetworkServerControl.shutdown()` require that you provide user credentials. This constructor creates an instance with user credentials, which are then used for operations that require them. In all other aspects, this constructor behaves like `NetworkServerControl()`.

- `NetworkServerControl(InetAddress address, int portNumber, String userName, String password)`

  This constructor creates an instance with user credentials, which are then used for operations that require them. In all other aspects, this constructor behaves like `NetworkServerControl(InetAddress address, int portNumber)`.

Setting Network Server properties

You can specify Network Server properties in the following ways.

- On the command line
- In the `.bat` or `.ksh` files (load the properties by executing `java -D`)
- In the `derby.properties` file

Properties specified on the command line or in the `.bat` or `.ksh` files take precedence over the properties in the `derby.properties` file. Arguments included in commands that are issued on the command line take precedence over property values.

derby.drda.host property

Causes the Network Server to listen on a specific network interface.

This property allows multiple instances of Network Server to run on a single machine, each using its own unique host:port combination. The host needs to be set to enable remote connections.

By default, the Network Server will listen only on the loopback address. If the property is set to `0.0.0.0`, the Network Server will listen on all interfaces.

Ensure that you are running under a security manager and that user authorization is enabled before you enable remote connections with this property.

Syntax

```
derby.drda.host=hostName
```

Default

If no host name is specified, the Network Server listens on the loopback address of the current machine (localhost).

Example

```
derby.drda.host=myhost
```

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.keepAlive property

Indicates whether SO_KEEPALIVE is enabled on sockets.

The keepAlive mechanism is used to detect when clients disconnect unexpectedly. A *keepalive probe* is sent to the client if a long time (by default, more than two hours) passes with no other data being sent or received. The `derby.drda.keepAlive` property is used to detect and clean up connections for clients on powered-off machines or clients that have disconnected unexpectedly.

If the property is set to `false`, Derby will not attempt to clean up disconnected clients. The keepAlive mechanism might be disabled if clients need to resume work without reconnecting even after being disconnected from the network for some time. To disable keepAlive probes on Network Server connections, set this property to `false`.

Syntax

```
derby.drda.keepAlive={true|false}
```

Default

True.

Example

```
derby.drdra.keepAlive=false
```

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.logConnections property

Indicates whether to log connections.

This property also controls the logging of the connection number. Connection number tracing, if enabled, goes to both the `derby.log` file and the Network Server console.

Syntax

```
derby.drda.logConnections={true|false}
```

Default

False.

Example

```
derby.drda.logConnections=true
```

Static or dynamic

Dynamic. You can change system values by using commands after the Network Server has been started.

derby.drda.maxThreads property

Sets the maximum number of connection threads that the Network Server will allocate.

If all of the connection threads are currently being used and the Network Server has already allocated the maximum number of threads, the threads will be shared by using the `derby.drda.timeSlice` property to determine when sessions will be swapped.

Syntax

```
derby.drda.maxThreads=numThreads
```

Default

0 (zero).

Example

```
derby.drda.maxThreads=50
```

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.minThreads property

Sets the minimum number of connection threads that the Network Server will allocate.

By default, connection threads are allocated as needed.

Syntax

```
derby.drda.minThreads=numThreads
```

Default

0 (zero).

Example

```
derby.drda.minThreads=10
```

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.portNumber property

Indicates the port number to use.

Syntax

```
derby.drda.portNumber=portNumber
```

Default

If no port number is specified, 1527 is the default.

Example

```
derby.drda.portNumber=1110
```

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.securityMechanism property

Restricts the client connections based on the security mechanism.

If the `derby.drda.securityMechanism` property is set to a valid mechanism, the Network Server accepts only connections which use that security mechanism. No other types of connections are accepted. If the `derby.drda.securityMechanism` property is not set, the Network Server accepts any connection which uses a valid security mechanism.

Syntax

```
derby.drda.securityMechanism={
    USER_ONLY_SECURITY |
    CLEAR_TEXT_PASSWORD_SECURITY |
    ENCRYPTED_USER_AND_PASSWORD_SECURITY
}
```

Default

None.

Example

```
derby.drda.securityMechanism=USER_ONLY_SECURITY
```

The server that runs with this setting accepts only client connections with the `USER_ONLY_SECURITY` value.

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.sslMode property

Indicates whether the client connection is encrypted or not, and whether certificate-based peer authentication is enabled.

Syntax

```
derby.drda.sslMode={
    off |
    basic |
    peerAuthentication
}
```

Default

`off`.

Example

```
derby.drda.sslMode=basic
```

The server that runs with this setting accepts client connections encrypted with SSL.

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.startNetworkServer property

Indicates whether the Network Server will start automatically when you start Derby.

Use the `derby.drda.startNetworkServer` property to simplify embedding the Network Server in your Java application. When you set this property to true, the Network Server will automatically start when you start Derby (in this context, Derby will start when the embedded driver is loaded). Only one Network Server can be started in a JVM.

Note: If you start the Network Server with this property set to true, the Network Server will stop when your application ends or when you stop it by other means (for example, by using the Java API or the command line interface, or by shutting down the Derby system), whichever comes first.

Syntax

```
derby.drda.startNetworkServer={true|false}
```

Default

False.

Example

```
derby.drda.startNetworkServer=true
```

Static or dynamic

Static. You must shut down the Network Server and restart Derby for this change to take effect.

derby.drda.streamOutBufferSize property

Configures the size of the buffer for streaming blob/clob data from server to client.

If the configured size is 0 or less, the buffer is not placed.

Note: This configuration is used when optimizing streaming blob/clob from server to client.

If there were found many small packets, of which sizes are much lower than maximum size of packet possible in the network, it will improve performance of streaming to setting this configuration.

Recommended value of this configuration is maximum packet size possible in the network minus appropriate size for header.

Syntax

```
derby.drda.streamOutBufferSize=sizeOfBuffer
```

Default

0 (zero).

Example

```
derby.drda.streamOutBufferSize=1024
```

Static or dynamic

Dynamic. You can change system values by using commands after the Network Server has been started.

derby.drda.timeSlice property

Sets the number of milliseconds that each connection will use before yielding to another connection.

This property is relevant only if the `derby.drda.maxThreads` property is set to a value greater than zero.

Syntax

```
derby.drda.timeSlice=milliseconds
```

Default

0 (zero).

Example

```
derby.drda.timeSlice=2000
```

Static or dynamic

Static. You must restart the Network Server for changes to take effect.

derby.drda.traceAll property

Turns tracing on for all sessions.

Syntax

```
derby.drda.traceAll={true|false}
```

Default

False.

Example

```
derby.drda.traceAll=true
```

Static or dynamic

Dynamic. You can change system values by using commands after the Network Server has been started.

derby.drda.traceDirectory property

Indicates the location of tracing files.

Security Considerations

The Network Server will attempt to create the trace directory (and any parent directories) if they do not exist. This will require that the Java security policy for `derbynet.jar`

permits verification of the existence of the named trace directory and all necessary parent directories. For each directory created, the policy must allow

```
permission java.io.FilePermission "directory", "read,write";
```

and for the trace directory itself, the policy must allow

```
permission java.io.FilePermission "tracedirectory${/}-", "write";
```

See Customizing the Network Server's security policy for information about customizing the Network Server's security policy.

Syntax

```
derby.drda.traceDirectory=traceFileDirectory
```

Default

If the `derby.system.home` property has been set, it is the default. Otherwise, the default is the current directory.

Example

```
derby.drda.traceDirectory=c:/Derby/trace
```

Static or dynamic

Dynamic. You can change system values by using commands after the Network Server has been started.

Verifying startup

To verify that the Derby Network Server is currently running, use the `ping` command.

You can use the `ping` command in the following ways:

- You can use the script `NetworkServerControl.bat` for Windows systems or `NetworkServerControl.ksh` for UNIX systems with the `ping` command. For example:

  ```
  NetworkServerControl ping [-h <hostname>;] [-p <portnumber>]
  ```
- You can use the NetworkServerControl `ping` command:

  ```
  java org.apache.derby.drda.NetworkServerControl
     ping [-h <hostname>] [-p <portnumber>]
  ```
- You can use the NetworkServerControl API to verify startup from within a Java application:

  ```
  ping();
  ```

The following example uses a method to verify startup. It will try to verify for the specified number of seconds:

```
private static boolean isServerStarted(NetworkServerControl server, int
 ntries) {
    for (int i = 1; i <= ntries; i ++) {
        try {
            Thread.sleep(500);
            server.ping();
            return true;
        } catch (Exception e) {
            if (i == ntries) {
                return false;
            }
        }
    }
```

```
    return false;
}
```

Using Java Management Extensions (JMX) technology

Derby includes a set of MBeans (Managed Beans) and their attributes and operations, providing monitoring and management capabilities.

Before using the Derby MBeans, you should have a basic understanding of JMX technology. A good source of information is the "Monitoring and Management for the Java Platform" web page at http://docs.oracle.com/javase/7/docs/technotes/guides/management/ .

The Derby MBeans instrument one or more parts of a running Derby system. This instrumentation gives you real-time access to Derby-specific information and features from a host of your choice, if you configure your Java Virtual Machine (JVM) and the Derby security features to enable this access.

The Derby JMX features are automatically available when Derby is started in a JVM that supports the platform MBean server. Java SE 5 and subsequent releases all support JMX technology.

You start Derby by loading the Derby embedded driver. If you are using the Derby Network Server, the embedded driver is automatically loaded in the server JVM when the server is started.

You may access the Derby MBeans by using an existing JMX client utility such as JConsole, or programmatically by writing your own Java code that uses JMX.

Introduction to the Derby MBeans

Derby provides the MBeans described in this section.

The public API documentation for each Derby MBean describes its features in detail.

VersionMBean

`VersionMBean` exposes version information about the running Derby system jar file.

- Interface: `org.apache.derby.mbeans.VersionMBean`
- Implementation: `org.apache.derby.iapi.services.info.Version` (not in the public API)
- ObjectName: `org.apache.derby:type=Version,system=<sysID>,jar=derby.jar` (monitors `derby.jar`, the Derby engine), or `org.apache.derby:type=Version,system=<sysID>,jar=derbynet.jar` (monitors `derbynet.jar`, the server)
- Instruments: `org.apache.derby.iapi.services.info.ProductVersionHolder`

JDBCMBean

`JDBCMBean` exposes information about the JDBC driver.

- Interface: `org.apache.derby.mbeans.JDBCMBean`
- Implementation: `org.apache.derby.jdbc.JDBC` (not in the public API)
- ObjectName: `org.apache.derby:type=JDBC,system=<sysID>`
- Instruments: `org.apache.derby.jdbc.InternalDriver` and `org.apache.derby.iapi.services.info.JVMInfo`

ManagementMBean

`ManagementMBean` manages the state of the Derby MBeans (registered or not).

- Interface `org.apache.derby.mbeans.ManagementMBean`
- Implementation: `org.apache.derby.mbeans.Management` (part of the public API; may be registered by JMX clients)
- Extended by:
  `org.apache.derby.iapi.services.jmx.ManagementService` (interface; not in the public API), with the following implementations:
  - `org.apache.derby.impl.services.jmx.JMXManagementService` (not public)
  - `org.apache.derby.impl.services.jmxnone.NoManagementService` (not in the public API; empty implementation for environments without the required JMX support)
- ObjectName: `org.apache.derby:type=Management,system=<sysID>` when registered by Derby
- Instruments:
  `org.apache.derby.impl.services.jmx.JMXManagementService`

NetworkServerMBean

`NetworkServerMBean` monitors and manages a running instance of the Network Server.

- Interface: `org.apache.derby.mbeans.drda.NetworkServerMBean`
- Implementation: `org.apache.derby.impl.drda.NetworkServerMBeanImpl` (not in the public API)
- ObjectName: `org.apache.derby:type=NetworkServer,system=<sysID>`
- Instruments: `org.apache.derby.impl.drda.NetworkServerControlImpl`

Enabling and disabling JMX

You can use JMX management and monitoring both locally and remotely.

The term local means *on the same host (machine)* and *running as the same user*. For example, this means that local JMX access is possible only if the JVM you want to access is running on the same host and as the same user as the user who is running a JMX client such as JConsole (or a different user with sufficient file system permissions). In order to allow other users to access the JVM, or to allow access from other hosts, remote JMX must be enabled.

Local JMX access

If you are using a Java SE 6 or later JVM, local JMX management and monitoring are most likely enabled by default.

Some JVMs, for example Java SE 5 JVMs, do not enable local JMX management by default. Refer to the documentation for your JVM for details. A common way to enable local JMX access on these JVMs is to include the `-Dcom.sun.management.jmxremote` option on the command line when you start the JVM.

Remote JMX access

Remote JMX management and monitoring is a powerful Java feature, allowing you to monitor a specific JVM from a remote location. Enabling remote JMX requires explicit actions by the JVM administrator, since it may involve exposing sensitive information about your system.

The most common way to enable remote JMX access to your JVM is to specify a TCP/IP port number and some basic security settings when you start the JVM. The security settings commonly include authentication and SSL (Secure

Socket Layer). Derby attempts to use the JVM's built-in platform MBean server. For a list of current command line options (system properties) and their meanings, refer to the table in the *Java SE Monitoring and Management Guide* at http://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html#gdeum .

The following topics describe ways to enable and disable remote JMX access.

Enabling remote JMX with no authentication or SSL

The following simple example starts the Derby Network Server on the command line with *insecure* remote JMX management and monitoring enabled, using an Oracle JDK 6 or later JVM.

Password authentication over SSL is enabled by default, but here these security features are disabled, to keep the example simple.

> Important: It is not recommended to disable authentication or SSL in production environments.

```
java -Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-jar $DERBY_HOME/lib/derbyrun.jar server start
```

When you start the Network Server from the command line, it automatically installs a security manager using a basic default security policy, unless you specify the `-noSecurityManager` option. You may need to customize this policy to make it suit your needs. See Fine-grained authorization using a security policy for details.

Enabling remote JMX with password authentication only

Some JVMs include built-in support for JMX password authentication.

For example, with Oracle Java Development Kit (JDK) 6 or later, authentication is enabled by default, and it is possible to specify a properties file that contains usernames and passwords. The properties file syntax is the same as for other Java properties files.

Note: When authentication is enabled and a Java Security Manager is installed, additional permissions may need to be granted to users in the security policy used. See Fine-grained authorization using a security policy for details.

For example, you could create a password file called `jmxremote.password`:

```
## Defining two "roles", each with its own password
monitorRole  derbym
controlRole  derby
```

The security of the password file relies on your file system's access control mechanisms. The file must be readable by the owner only. Also, you may need to change the permissions on the password file to be readable only by the user who starts the server. To do this on Windows (NTFS), use a command like the following:

```
cacls jmxremote.password /P username:R
```

Note: FAT file systems do not support this feature.

The following example starts the Network Server on the command line with built-in JMX password authentication enabled. SSL is disabled, meaning that JMX information, including user names and passwords most likely will be transferred unprotected on the computer network. The command line appears on multiple lines to improve readability, but you would enter it as a single `java` command.

> Important: It is not recommended to disable SSL in production environments.

```
java -Dcom.sun.management.jmxremote.port=9999
```

```
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.password.file=jmxremote.password
-jar lib/derbyrun.jar server start
```

Enabling remote JMX with password authentication and SSL

This example shows how to start the Network Server as follows.

- Using Oracle JDK 6 or later
- Using a Java security manager and a custom policy file, `jmx.policy`
- Allowing connections from remote hosts (that is, on all IPv4 network interfaces) by specifying `-h 0.0.0.0`
- Using password authentication, as described in Enabling remote JMX with password authentication only, using the `jmxremote.password` file
- Using SSL (Secure Socket Layer) for the following:
  - Authenticating clients
  - Encrypting all JMX-related network communication
  - Protecting the RMI registry used by the MBean server

This level of protection may or may not be adequate for you, but it is more secure than the previous examples.

The command line appears on multiple lines to improve readability, but you would enter it as a single `java` command.

```
java -Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.password.file=jmxremote.password
-Djavax.net.ssl.keyStore=/home/user/.keystore
-Djavax.net.ssl.keyStorePassword=myKeyStorePassword
-Dcom.sun.management.jmxremote.ssl.need.client.auth=true
-Djavax.net.ssl.trustStore=/home/user/.truststore
-Djavax.net.ssl.trustStorePassword=myTrustStorePassword
-Dcom.sun.management.jmxremote.registry.ssl=true
-Djava.security.manager
-Djava.security.policy=jmx.policy
-jar lib/derbyrun.jar server start -h 0.0.0.0
```

Note: When password authentication is enabled and a Java Security Manager is installed, a number of JMX-related permissions need to be granted to trusted users in the security policy used. See Fine-grained authorization using a security policy for details.

In the example above, system properties specify the keystore containing the server's key pair, the keystore password, the truststore containing the client certificates, and the truststore password. Setting up SSL keystores and truststores is partly described in Key and certificate handling. Other topics in the section Network encryption and authentication with SSL/TLS provide information on protecting database network traffic using SSL.

When you configure SSL as described above, the following requirements apply:

- The password of the private key must be the same as the password of the keystore.
- If the keystore contains more than one key pair, the key pair you want to use must be listed first among all the keys in the keystore. Otherwise, you (or the clients) may see an exception with a message like the following:

  ```
  unable to find valid certification path to requested target
  ```

The system property `com.sun.management.jmxremote.ssl.need.client.auth=true` specifies that clients must use SSL to authenticate themselves. This property, as well as the truststore properties, may be removed if you do not want to authenticate clients using SSL. However, there may be security risks associated with using password authentication only.

The system property `com.sun.management.jmxremote.registry.ssl=true` was new in JDK 6 and aims at resolving security issues with the RMI registry used in relation with JMX. This property must be used in conjunction with `com.sun.management.jmxremote.ssl.need.client.auth=true` in order to fully secure the RMI registry.

If you use a Java SE 5 JDK, clients must provide an additional entry in the environment map passed to the `JMXConnector` when enabling SSL protection of the registry:

```
env.put("com.sun.jndi.rmi.factory.socket", new
 SslRMIClientSocketFactory());
```

See Connecting to the MBean Server for details.

Clients must also specify and use proper keystores and/or truststores (the truststores must contain the server's SSL certificate).

For more information about the system properties used above and potential security risks, see "Monitoring and Management Using JMX Technology" at http://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html .

Simple authorization using an access file

Some JVMs support a simple access file system for controlling JMX access.

An access file is formatted the same way as password files (described in Enabling remote JMX with password authentication only), and associates roles with an access level. Valid access levels are `readonly` and `readwrite`:

- The `readonly` level only allows the JMX client to read an MBean's attributes and receive notifications.
- The `readwrite` level also allows setting attributes, invoking operations, and creating and removing MBeans.

To use an access file for JMX authorization, specify the name of the access file using a system property upon JVM startup:

```
-Dcom.sun.management.jmxremote.access.file=jmxremote.access
```

The contents of such an access file may look like this:

```
monitorRole   readonly
controlRole   readwrite
```

For more information, see "Monitoring and Management Using JMX Technology" at http://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html .

Fine-grained authorization using a security policy

When you start the Network Server from the command line, it installs a security manager and a basic security policy by default.

This policy includes the required permissions to allow JMX users to access the Derby MBeans if JMX user authentication is disabled. If JMX user authentication is enabled, you may need to grant additional permissions to specific users (JMXPrincipals).

The `NetworkServerMBean`'s `ping` operation requires the `derbynet.jar` file to be granted an additional permission that is not included in the default security policy:

```
// If the server is listening on the loopback interface only (default)
permission java.net.SocketPermission "localhost", "connect,resolve";

// If the server's network interface setting (-h or derby.drda.host) is
//   non-default
// Note: Allows outbound connections to any host!
permission java.net.SocketPermission "*", "connect,resolve";
```

If you are using a custom security policy, refer to the public API documentation for the Derby MBeans and to the Derby security policy file template (`$DERBY_HOME/demo/templates/server.policy`) for details about the permissions you may need to set to allow or restrict specific JMX access. This recommendation also applies if you are running Derby embedded with a security manager installed.

See Running the Network Server under the security manager for more information about security policy files.

Some example permissions are included in the following code. These permissions are not necessarily suitable for any particular application or environment; some customization is probably needed. Only permissions relating to the Derby JMX features have been included in the code. Additional permissions are needed for use of Derby.

```
//
// permissions for the user/principal "controlRole", for all codebases:
//
grant principal javax.management.remote.JMXPrincipal "controlRole" {

  // Derby system permissions (what is the user allowed to do?)
  //  See API docs for SystemPermission and the specific MBeans for
  //  details.
  permission org.apache.derby.security.SystemPermission "jmx", "control";
  permission org.apache.derby.security.SystemPermission "engine",
      "monitor";
  permission org.apache.derby.security.SystemPermission "server",
      "monitor,control";

  // MBean permissions (which mbeans and associated actions should be
  //  allowed for this user?)
  //  Target name format is: className#member[objectName], where
  //   objectName is: domain:keyProperties
  //  Asterisk (*) means "all". See MBeanPermission API docs for details.
  permission javax.management.MBeanPermission
      "org.apache.derby.mbeans.*#*[org.apache.derby:*]", "getAttribute";
  permission javax.management.MBeanPermission
      "org.apache.derby.mbeans.JDBCMBean#acceptsURL[org.apache.derby:*]",
      "invoke";
  permission javax.management.MBeanPermission
"org.apache.derby.mbeans.drda.NetworkServerMBean#ping[org.apache.derby:*]",
      "invoke";
  permission javax.management.MBeanPermission
      "org.apache.derby.mbeans.ManagementMBean#*[org.apache.derby:*]",
      "invoke";

  // Extra permissions for application controlled ManagementMBean:
  //   Not needed if you do not intend to create/register your own
  //   Derby Management MBean.
  //   Wildcards (*) allow all domains, key properties and MBean members.
  //   You may want to be more specific here.
  permission javax.management.MBeanPermission
      "org.apache.derby.mbeans.Management#-[*:*]",
      "instantiate,registerMBean,unregisterMBean";
  permission javax.management.MBeanPermission
      "org.apache.derby.mbeans.Management#*[*:*]", "invoke";

  //
  // jconsole:
  //  - most of these permissions are needed to let JConsole query the
  //    MBean server and display information about Derby's mbeans as well
  //    as some default platform MBeans/MXBeans.
  //  - if you don't use JConsole, but query the MBean server from your
  //    JMX client app, some of these permissions may be needed.
  permission javax.management.MBeanPermission
      "org.apache.derby.mbeans.*#-[org.apache.derby:*]",
      "getMBeanInfo,queryNames,isInstanceOf";
  permission javax.management.MBeanPermission
      "sun.management.*#-[java.*:*]",
```

```
      "getMBeanInfo,isInstanceOf,queryNames";
  permission javax.management.MBeanPermission
      "sun.management.*#*[java.*:*]", "getAttribute,invoke";
  permission javax.management.MBeanPermission
      "sun.management.*#-[com.sun.management*:*]",
      "getMBeanInfo,isInstanceOf,queryNames";
  permission javax.management.MBeanPermission
      "com.sun.management.*#-[java.*:*]",
      "getMBeanInfo,isInstanceOf,queryNames";
  permission javax.management.MBeanPermission
      "com.sun.management.*#*[java.*:*]", "getAttribute,invoke";
  permission javax.management.MBeanPermission "java.*#-[java.*:*]",
      "getMBeanInfo,isInstanceOf,queryNames";
  permission javax.management.MBeanPermission
"javax.management.MBeanServerDelegate#-
[JMImplementation:type=MBeanServerDelegate]",
      "getMBeanInfo,isInstanceOf,queryNames,addNotificationListener";
  permission java.net.SocketPermission "*", "resolve";
  permission java.util.PropertyPermission "java.class.path", "read";
  permission java.util.PropertyPermission "java.library.path", "read";
  permission java.lang.management.ManagementPermission "monitor";
  // end jconsole
};


grant codeBase "${derby.install.url}derby.jar"
{
  // Allows Derby to create an MBeanServer:
  //
  permission javax.management.MBeanServerPermission "createMBeanServer";

  // Allows access to Derby's built-in MBeans, within the domain
  //  org.apache.derby. Derby must be allowed to register and unregister
  //  these MBeans.
  // It is possible to allow access only to specific MBeans, attributes,
  //  or operations. To fine-tune this permission, see the API doc of
  //  javax.management.MBeanPermission or the JMX Instrumentation and
  //  Agent Specification.
  //
  permission javax.management.MBeanPermission
      "org.apache.derby.*#[org.apache.derby:*]",
      "registerMBean,unregisterMBean";

  // Trusts Derby code to be a source of MBeans and to register these in
  // the MBean server.
  //
  permission javax.management.MBeanTrustPermission "register";

  // Gives permission for JMX to be used against Derby.
  // If JMX user authentication is being used, a whole set of
  //  fine-grained permissions needs to be granted to allow specific
  //  users access to MBeans and actions they perform (see JMXPrincipal
  //  permissions above).
  // Needed to allow access to all actions related to MBeans in the
  // org.apache.derby.mbeans package.
  //
  permission org.apache.derby.security.SystemPermission "jmx", "control";
  permission org.apache.derby.security.SystemPermission "engine",
      "monitor";
  permission org.apache.derby.security.SystemPermission "server",
      "monitor";

  // add additonal derby.jar related permissions here...
};


grant codeBase "${derby.install.url}derbynet.jar"
{
  // Accept connections from any host (only localhost access is required
  //  for JMX).
```

```
  //
  permission java.net.SocketPermission "*", "accept";

  // For outbound MBean operations such as NetworkServerMBean's ping:
  // The wildcard "*" is to allow pings to both localhost and any other
  //  server host.
  //
  permission java.net.SocketPermission "*", "connect,resolve";

  // Gives permission for JMX to be used against Derby.
  // If JMX user authentication is being used, a whole set of
  //  fine-grained permissions need to be granted to allow specific users
  //  access to MBeans and actions they perform (see JMXPrincipal
  //  permissions above).
  // Needed to allow access to all actions related to the
  //  NetworkServerMBean.
  //
  permission org.apache.derby.security.SystemPermission "server",
      "control,monitor";

  // add additonal derbynet.jar related permissions here...
```

In the example above, the system property `derby.install.url` is used to tell the security manager/policy implementation where to find the codebases `derby.jar` and `derbynet.jar`. Using a property provides flexibility; however, you may avoid the use of such a property by specifying the full codebase URLs directly in the policy file. The value of this property may be specified on the command line, as shown below:

```
-Dderby.install.url=file:/home/user/derby/10.9.1/lib/
```

or

```
-Dderby.install.url=file:/C:/derby/10.9.1/lib/
```

For more information about policy files, granting permissions, and property expansion, see "Default Policy Implementation and Policy File Syntax" at http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html and "Policy File Creation and Management" at http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyGuide.html.

Debugging permission issues

Dealing with security managers, policy files and permissions is not always easy. Sometimes an action you want to perform fails due to some security or permission issue that you do not understand. The following tip may help.

When you start the JVM that is being protected by the security manager, add a `java.security.debug` flag to see detailed output related to security policy and permission usage. For a list of valid options, use the following command:

```
java -Djava.security.debug=help
```

For example, you could use the following option when you start the Network Server from the command line:

```
-Djava.security.debug=access:failure
```

This option will print information to the console that allows you to learn specifically which permissions are granted and which are missing when a failure occurs. Due to the amount of output generated when you set the debug flag, it may be wise to store the output in a file and search through it afterwards.

For example, to find out details about a missing permission, search for the text "access denied" in the output, and you will see something like the following:

```
access: access denied
    (org.apache.derby.security.SystemPermission engine monitor)
java.lang.Exception: Stack trace
   at java.lang.Thread.dumpStack(Thread.java:1158)
   ...
   at org.apache.derby.iapi.services.info.Version.getVersionString
       (Unknown Source)
...
```

The above example output shows that the `derby.jar` code base was missing the following permission as the JMX client was accessing the `VersionString` attribute of the `VersionMBean` for `derby.jar`:

```
org.apache.derby.security.SystemPermission "engine", "monitor";
```

Disabling access to MBeans

You may wish to disable or restrict access to MBeans in security-conscious environments. You can do this using either of the following techniques.

The first technique is to use the `stopManagement()` method of `ManagementMBean`. This method unregisters all of the Derby MBeans except `ManagementMBean` itself, so it does not turn access off completely.

The second technique is to run the Network Server with a custom security policy that does not grant `derby.jar` the permissions needed to register MBeans. For example, you can modify the Network Server's basic policy (see Basic Network Server security policy) by commenting out this section:

```
// Allows access to Derby's built-in MBeans, within the domain
// org.apache.derby.
// Derby must be allowed to register and unregister these MBeans.
// It is possible to allow access only to specific MBeans, attributes or
// operations. To fine tune this permission, see the javadoc of
// javax.management.MBeanPermission or the JMX Instrumentation and Agent
// Specification.
//
permission javax.management.MBeanPermission
    "org.apache.derby.*#[org.apache.derby:*]",
    "registerMBean,unregisterMBean";
```

If the permission to register MBeans is not granted to `derby.jar`, Derby will silently skip starting the management service at boot time.

Using JConsole to access the Derby MBeans

JConsole is a graphical JMX-compliant tool that is available in recent versions of the Oracle JDKs. JConsole enables you to monitor and manage Java applications and virtual machines on a local or remote machine.

You may use JConsole from JDK 6 or later even if you are running Derby using an earlier version of the JDK (or just the JRE). (You could also use JConsole from JDK 5 if you are running Derby using JDK 6 or later.) It is recommended that you use the newest version possible. More information about JConsole is available in the OpenJDK project at http://openjdk.java.net/tools/svc/jconsole/index.html.

Starting JConsole and connecting to Derby

In the Oracle JDK, the JConsole binary is available in `JDK_HOME/bin`, where JDK_HOME is the directory in which the JDK is installed. To start JConsole, use the `jconsole` command, as in the following example on a UNIX system:

```
/usr/local/java/jdk1.7.0/bin/jconsole
```

If you did not disable SSL when booting the managed JVM, you probably have to provide a truststore containing the server's SSL certificate to be able to establish JMX connections. If SSL client authentication is enabled, a keystore must be configured as well (see Enabling remote JMX with password authentication and SSL for details). The following example shows how to start JConsole with SSL client and server authentication:

```
jconsole -J-Djavax.net.ssl.trustStore=/home/user/.truststoreForClient
-J-Djavax.net.ssl.trustStorePassword=myTruststorePassword
-J-Djavax.net.ssl.keyStore=/home/user/.keystoreForClient
-J-Djavax.net.ssl.keyStorePassword=myKeyStorePassword
```

A graphical user interface (GUI) appears. For additional startup options, refer to the JConsole documentation. Once the GUI starts, you are presented with a list of the JVMs that are accessible on the local host. Locate the JVM that is running Derby and connect to it.

To connect to a JVM on a remote host, you will need to supply the host name and port number, or a JMX service URL, instead.

If you cannot find the Derby JVM running on the local host, make sure you are running JConsole as the same user as the Derby JVM, or as a different user with sufficient file system permissions. If you are using Java SE 5, make sure you have enabled JMX. When you use Java SE 6 or later, local JMX access is enabled by default.

Accessing MBeans

Once you have connected to a JVM via JConsole, the JVM's MBeans should be available on a separate tab in the internal JConsole window. Under the domain `org.apache.derby` you should see a list of MBeans. Browse the MBeans and their attributes and operations by navigating the hierarchy presented.

Another useful JConsole feature is that you can view dynamic data represented as JMX attributes in graph form. To view these graphs, double-click an attribute value that is a number.

Using custom Java code to access the Derby MBeans

In addition to using a tool like JConsole, you can also access the Derby MBeans from a Java application.

How to do this may depend on how you configure the JVM that is running Derby, how you configure user authentication and authorization, or the host(s) from which you want to access the MBeans.

This section has some example code to help you get started. You will find the JMX classes you need in the packages `javax.management` and `javax.management.remote`.

You do not need any Derby libraries in the JMX client application's classpath (unless MBean proxies are used).

Connecting to the MBean Server

Derby will attempt to register its MBeans with the platform MBean server of the JVM running the Derby system (embedded or Network Server). The following examples assume that you have configured the Derby JVM to enable remote JMX, which means that you have set a port number (`com.sun.management.jmxremote.port`) to be used by the JMX Server Connector.

The examples below assume that the port configured for remote JMX is 9999, that the host name of the host running Derby is `example.com`, and that this host is reachable from the client host. (This host name is fictitious, and is used for example purposes only.)

The following example code shows how to connect to the MBean Server when JMX security has been disabled:

```
JMXServiceURL url = new JMXServiceURL(
        "service:jmx:rmi:///jndi/rmi://example.com:9999/jmxrmi");
JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
MBeanServerConnection mbeanServerConn =
    jmxc.getMBeanServerConnection();
```

The following code shows how to connect to the MBean server when JMX password authentication is enabled (the default):

```
JMXServiceURL url = new JMXServiceURL(
        "service:jmx:rmi:///jndi/rmi://example.com:9999/jmxrmi");
// Assuming the following JMX credentials:
//  username=controlRole, password=derby
String[] credentials = new String[] { "controlRole" , "derby" };
HashMap<String,Object> env = new HashMap<String,Object>();
// Set credentials (jmx.remote.credentials,
//  see JMX Remote API 1.0 spec section 3.4)
env.put(JMXConnector.CREDENTIALS, credentials);
// if the server's RMI registry is protected with SSL/TLS (JDK 6)
//  (com.sun.management.jmxremote.registry.ssl=true), the following
//  entry must be included:
//env.put("com.sun.jndi.rmi.factory.socket",
//    new SslRMIClientSocketFactory());  // uncomment if needed

// Connect to the server
JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
MBeanServerConnection mbeanServerConn =
    jmxc.getMBeanServerConnection();
```

Note: Not specifying `SslRMIClientSocketFactory` when required may result in the error message `java.rmi.ConnectIOException: non-JRMP server at remote endpoint`.

Creating a ManagementMBean

The only Derby MBean that can be created by a JMX client is the `ManagementMBean`. This MBean is useful for controlling Derby management (for example, enabling and disabling management or MBeans), and to obtain information such as the system identifier (which may be needed to specify MBeans later).

If you create such an MBean from your application, and if Derby has already registered a `ManagementMBean` instance, the new MBean cannot have the same object name as the `ManagementMBean` already registered with the server. It is therefore recommended to use a different object name domain (that is, different from `example.com`) and/or a different `type` key property value (different from `Management`).

The following example code shows how to create and register a new `ManagementMBean` with the MBean server:

```
ObjectName mgmtObjName = new ObjectName("com.example.app",
                "type", "DerbyManagement");
try {
    ObjectInstance mgmtObj =
        mbeanServerConn.createMBean("example.com.mbeans.Management",
            mgmtObjName);
} catch (InstanceAlreadyExistsException e) {
    // A management MBean with this object name already exists!
}
```

Activating Derby management

Derby attempts to activate its JMX management service by default, so it will usually be active unless you explicitly deactivate it, providing that Derby has permissions to perform

the activation. If Derby management is not active, you will not be able to access any MBeans except the `ManagementMBean`.

By accessing the `ManagementActive` attribute of the `ManagementMBean`, you can check whether the Derby JMX management service is active or not. The following example code performs this check and activates the Derby management service if it is not already active:

```
// assuming we already have a reference to the
//  ManagementMBean's object name
Boolean active = (Boolean)
    mbeanServerConn.getAttribute(mgmtObjName, "ManagementActive");
if (!active.booleanValue()) {
    // start management
    mbeanServerConn.invoke(mgmtObjName, "startManagement",
        new Object[0], new String[0]);
}
```

Obtaining the system identifier

The system identifier is a unique `String` that distinguishes one running Derby system from another. All MBeans that are instantiated by Derby include the system identifier in their object names.

One way to access an MBean is to fully specify its object name when contacting the MBean server. For this, you need to know the current system identifier. (Alternative ways to access MBeans include querying the MBean server for all MBeans, or for MBeans whose object names match a specific pattern.)

The following example shows how to obtain the system identifier by accessing a `ManagementMBean`:

```
// assuming we already have a reference to the
//  ManagementMBean's object name
String systemID = (String) mbeanServerConn.getAttribute(mgmtObjName,
    "SystemIdentifier");
```

The following example shows how to obtain the system identifier from a Derby MBean's object name:

```
// assuming we already have a reference to the ObjectName
//  of an MBean registered by Derby, for example the
//  Derby-registered ManagementMBean
String systemID = derbyMgmtObjectName.getKeyProperty("system");
```

Accessing a specific Derby-registered MBean

In the previous examples, you have already seen how to read a single MBean attribute, and how to invoke an MBean operation. In order to do this, you usually need a reference to the MBean's `ObjectName`.

If you consult the public API documentation for the Derby MBeans and obtain the system identifier of the Derby system you are accessing through JMX, you have all the information you need to be able to instantiate a `javax.management.ObjectName` object directly, by fully specifying its `String` representation (see the `ObjectName` API documentation for details).

The following example code shows how to obtain a reference to the `VersionMBean` for `derby.jar`:

```
// Assuming we already know the system identifier
// (see examples above), systemID.
// A list of key properties is available is each MBean's Javadoc API.
Hashtable<String, String> keyProps = new Hashtable<String, String>();
keyProps.put("type", "Version");
```

```
    keyProps.put("jar", "derby.jar");
    keyProps.put("system", systemID);
    // MBeans registered by Derby always belong to the
    //  "org.apache.derby" domain
    ObjectName versionObjectName =
        new ObjectName("org.apache.derby", keyProps);

    // we can now use the object name to read an attribute
    String versionString =
        (String) mbeanServerConn.getAttribute(versionObjectName,
            "VersionString");
    System.out.println("VersionString: " + versionString);
```

The output would look something like this:

```
VersionString: 10.9.1.1 - (1305115)
```

Troubleshooting JMX connection issues

If you experience problems connecting remotely to an MBean server using JMX, it may be helpful to obtain some tracing information.

For details on connecting remotely to an MBean server, see Using JConsole to access the Derby MBeans and Using custom Java code to access the Derby MBeans.

The JMX implementation in the Oracle JDK uses the `java.util.logging` API to log JMX traces. For example, in order to trace SSL connection issues, set the system property `java.util.logging.config.file` as shown in the following:

```
java -Djava.util.logging.config.file=logging.properties MyJmxClient
```

With JConsole, a separate logging window will appear if you specify the following option when you start JConsole (see Using JConsole to access the Derby MBeans), as long as the `logging.properties` file is found:

```
-J-Djava.util.logging.config.file=logging.properties
```

The `logging.properties` file should specify log handlers and logging levels, as in the following example:

```
handlers = java.util.logging.ConsoleHandler
.level = INFO

java.util.logging.ConsoleHandler.level=FINEST
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

 // Level FINEST is suitable for diagnosing SSL-related JMX remote
 //  connection issues.
javax.management.level=FINEST
javax.management.remote.level=FINEST
```

The blog entry https://blogs.oracle.com/jmxetc/entry/troubleshooting_connection_problems_in_jconsole provides additional hints and tips.

Managing the Derby Network Server remotely by using the servlet interface

You can use the servlet interface to manage the Network Server remotely. To use the servlet interface, the servlet must be registered with an Application Server, and `derby.system.home` must be known to the Application Server.

> Important: The servlet interface is suitable only for testing purposes. It should not be used in production.

A web application archive (WAR) file for the Derby Network Server, `derby.war`, is available in `$DERBY_HOME/lib`. This file registers the Network Server's servlet at the relative path `/derbynet`. See the documentation for your Application Server for instructions on how to install it.

For example, if `derby.war` is installed in WebSphere Application Server with a context root of `derby`, the URL of the server is:

```
http://<server>[:port]/derby/derbynet
```

Notes:

- A servlet engine is not part of the Network Server.
- When the Network Server is started by the servlet interface, shutting down the Application Server also shuts the Network Server down, since both run in the same JVM.

The servlet takes the following optional configuration parameters:

host
Specifies the host name to be used by the Network Server. See the Security Considerations section below.

portNumber
Specifies the port number to be used by the Network Server.

startNetworkServerOnInit
Specifies that the Network Server is to be started when the servlet is initialized.

tracingDirectory
Specifies the location for trace files. If the tracing directory is not specified, the traces are placed in `derby.system.home`.

Security Considerations

For general security considerations for the Network Server, see Network Server security.

The `host` parameter allows configuration of the host name that will be used for the listening socket for network connections. By default, the Network Server will listen to requests only on the loopback address, which means that it will only accept connections from the local host. Changing this value could expose the server to external connections, which raises security concerns, so before using the `host` parameter, you should run under the Java security manager and enable user authentication.

This section describes the servlet pages.

Start-up page

Use the start-up page to start the server.

In addition to starting the Network Server, you can use the startup page to perform the following actions:

- Turn logging on when the server is started.
- Turn tracing on for all sessions when the server is started.

Running page

If the Network Server is running (whether it was started by initializing the servlet or in some other manner), the running page is displayed.

The running page indicates whether logging is on or off, whether tracing is on or off, and if tracing is on, indicates for which session.

You can use the running page to stop the server and turn logging and tracing on or off. The following options are available from the running page:

- Start or stop logging.
- Start or stop tracing all sessions.
- Specify session to trace. (If you choose this option, the Trace session page is displayed.)
- Change tracing directory. (If you choose this option, the Trace directory page is displayed.)
- Specify threading parameters for the Network Server. (If you choose this option, the Thread parameters page is displayed.)
- Stop the Network Server.

Trace session page

If on the running page you choose to specify a session to trace, this page is displayed. You must enter the Session ID.

You are given the option to turn tracing on or off or return to the previous menu. When you click the Trace On/Off button, information indicating the current tracing state is displayed.

Trace directory page

This page is displayed if the you choose to change the tracing directory on the Running page. You must enter the Trace Directory.

You can either set a tracing directory, or you can return to the previous menu. Additional information is displayed that indicates the current tracing directory when you click the Set Directory button.

Set Network Server parameters

The first page is displayed if the thread parameter button is clicked. Use this page to set the new parameters.

Enter the following information:

- New maximum number of threads
- New thread time slice

If either the maximum threads or time slice parameter is left blank, that value is left unchanged from the current setting.

Click Set Network Server parameters to display the updated values for the maximum threads and the time slice parameters.

Derby Network Server advanced topics

This section discusses several advanced topics for users of the Derby Network Server.

Network Server security

By default, the Derby Network Server listens only on the localhost. Clients must use the localhost host name to connect.

By default, clients cannot access the Network Server from another host.
To enable connections from other hosts, set the `derby.drda.host`

property, or start the Network Server with the `-h` option in the `java org.apache.derby.drda.NetworkServerControl start` command.

In the following example, the server will listen only on the localhost, and clients cannot access the server from another host:

```
java org.apache.derby.drda.NetworkServerControl start
```

In the following example, the server runs on the host machine `sampleserver.example.com` and also listens for clients from other hosts. Clients must specify the server in the URL or DataSource as `sampleserver.example.com`:

```
java org.apache.derby.drda.NetworkServerControl start
   -h sampleserver.example.com
```

To start the Network Server so that it will listen on all interfaces, start with an IP address of `0.0.0.0`, as shown in the following example:

```
java org.apache.derby.drda.NetworkServerControl start -h 0.0.0.0
```

A server that is started with the `-h 0.0.0.0` option will listen to client requests that originate from both `localhost` and from other machines on the network.

However, administrative commands (for example, `org.apache.derby.drda.NetworkServerControl shutdown`) can run only on the host where the server was started, even if the server was started with the `-h` option.

Controlling database file access

When Derby creates new files, the visibility of the new file (that is, which users can access it) is normally determined by the JVM environment and the file location only (that is, by the umask setting on UNIX and Linux systems and by the default file permissions on Windows NTFS).

On Java SE 7 or later, Derby may further restrict the file permissions to the operating system account that started the Java process (that is, to the minimum access needed for operation). This means that other operating system accounts will have no access to directories or files created by Derby. This behavior can be helpful in enhancing default security for database files.

The exact behavior is determined by two factors: how the Derby engine is started, and the presence or absence and specified value of the property `derby.storage.useDefaultFilePermissions`.

The two tables that follow show how file access works with Java SE 6 and with Java SE 7 and later JVMs. In both tables,

- "Environment" means that access is controlled entirely by the JVM environment and the file location only (that is, by the umask setting on UNIX and Linux systems and by the default file permissions on Windows NTFS).
- "Restricted" means that Derby restricts access to the operating system account that started the JVM.

The following table shows how file access works on Java SE 6 systems.

Table 9. File access on Java SE 6 systems

Property Setting	Server Started from Command Line	Server Started Programmatically or Embedded
Not applicable	Environment	Environment

The following table shows how file access works on Java SE 7 and later systems with various settings of the `derby.storage.useDefaultFilePermissions` property.

Table 10. File access on Java SE 7 and later systems

Property Setting	Server Started from Command Line	Server Started Programmatically or Embedded
No property specified	Restricted	Environment
Property set to true	Environment	Environment
Property set to false	Restricted	Restricted

For more information, see "derby.storage.useDefaultFilePermissions" in the *Java DB Reference Manual*.

Running the Network Server under the security manager

By default, the Network Server boots with a Basic security policy. You are encouraged to customize this policy to fit the security needs of your application and its runtime environment.

You may also run the Network Server without a security manager, although this is not recommended.

Basic Network Server security policy

If you boot the Network Server without specifying a security manager, the Network Server will install a default Java security manager that enforces a Basic policy.

This happens if you boot the Network Server as your VM's entry point, using a command like the following:

```
java org.apache.derby.drda.NetworkServerControl start ...
```

You should run your Network Server with user authentication and user authorization enabled. For details on how to enable user authentication, see "Working with user authentication" in the *Java DB Developer's Guide*. For information on user authorization, see "Users and authorization identifiers" and "User authorizations" in the *Java DB Developer's Guide*.

Some of your application code may run as procedures and functions which you have declared using the CREATE PROCEDURE and CREATE FUNCTION statements. You will need to add privileged blocks to your declared procedures and functions if they perform sensitive operations such as file and network i/o, classloading, system property reading, etc.

If for some reason you do not want to run your client/server application under a security manager, you may override the Network Server's impulse to install a default policy. For details, see Running the Network Server without a security policy.

Note that the Network Server attempts to install a security manager only if you boot the server as the entry point of your VM. The Network Server will not attempt to install a security manager if you start the server from your application using the programmatic API described in Starting the Network Server from a Java application.

You will find a template security policy in the Derby distribution at `demo/templates/server.policy`. Most likely, you will want to customize this policy. For example, probably you will want to restrict the server's liberal file i/o permissions which let the server backup/restore and export/import to or from any location in the local file system. For details on how to customize the Template policy, see Customizing the Network Server's security policy. The following example is a copy of the Basic policy:

```
// This template policy file gives examples of how to configure the
// permissions needed to run a Derby network server with the Java
// Security manager.
//
grant codeBase "${derby.install.url}derby.jar"
{
  // These permissions are needed for everyday, embedded Derby usage.
  //
  permission java.lang.RuntimePermission "createClassLoader";
  permission java.util.PropertyPermission "derby.*", "read";
  permission java.util.PropertyPermission "user.dir", "read";

  // The next two properties are used to determine if the VM is 32 or 64
  // bit.
  //
  permission java.util.PropertyPermission "sun.arch.data.model", "read";
  permission java.util.PropertyPermission "os.arch", "read";

  permission java.io.FilePermission "${derby.system.home}","read";
  permission java.io.FilePermission "${derby.system.home}${/}-",
      "read,write,delete";

  // This permission lets a DBA reload the policy file while the server
  // is still running. The policy file is reloaded by invoking the
  // SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY() system procedure.
  //
  permission java.security.SecurityPermission "getPolicy";

  // This permission lets you backup and restore databases
  // to and from arbitrary locations in your file system.
  //
  // This permission also lets you import/export data to and from
  // arbitrary locations in your file system.
  //
  // You may want to restrict this access to specific directories.
  //
  permission java.io.FilePermission "<<ALL FILES>>",
      "read,write,delete";

  // Permissions needed for JMX based management and monitoring.
  //
  // Allows this code to create an MBeanServer:
  //
  permission javax.management.MBeanServerPermission "createMBeanServer";
  //
  // Allows access to Derby's built-in MBeans, within the domain
  // org.apache.derby.
  // Derby must be allowed to register and unregister these MBeans.
  // It is possible to allow access only to specific MBeans, attributes
  // or operations. To fine tune this permission, see the javadoc of
  // javax.management.MBeanPermission or the JMX Instrumentation and
  // Agent Specification.
  //
  permission javax.management.MBeanPermission
      "org.apache.derby.*#[org.apache.derby:*]",
      "registerMBean,unregisterMBean";
```

```
  //
  // Trusts Derby code to be a source of MBeans and to register these in
  // the MBean server.
  //
  permission javax.management.MBeanTrustPermission "register";

  // getProtectionDomain is an optional permission needed for printing
  // classpath information to derby.log
  //
  permission java.lang.RuntimePermission "getProtectionDomain";

  // The following permission must be granted for
  // Connection.abort(Executor) to work. Note that this permission
  // must also be granted to outer (application) code domains.
  //
  permission java.sql.SQLPermission "callAbort";

  // Needed by file permissions restriction system:
  //
  permission java.lang.RuntimePermission "accessUserInformation";
  permission java.lang.RuntimePermission "getFileStoreAttributes";
};

grant codeBase "${derby.install.url}derbynet.jar"
{
  // This permission lets the Network Server manage connections from
  // clients.

  // Accept connections from any host. Derby is listening to the host
  // interface specified via the -h option to "NetworkServerControl
  // start" on the command line, via the address parameter to the
  // org.apache.derby.drda.NetworkServerControl constructor in the API
  // or via the property derby.drda.host; the default is localhost.
  // You may want to restrict allowed hosts, e.g. to hosts in a specific
  // subdomain, e.g. "*.example.com".
  //
  permission java.net.SocketPermission "*", "accept";

  // Allow the server to listen to the socket on the default port (1527).
  // If you have specified another port number with the -p option to
  // "NetworkServerControl start" on the command line, or with the
  // portNumber parameter to the NetworkServerControl constructor in the
  // API, or with the property derby.drda.portNumber, you should change
  // the port number in the permission statement accordingly.
  //
  permission java.net.SocketPermission "localhost:1527", "listen";

  // Needed for server tracing.
  //
  permission java.io.FilePermission "${derby.drda.traceDirectory}${/}-",
      "read,write,delete";

  // Needed by file permissions restriction system:
  //
  permission java.lang.RuntimePermission "accessUserInformation";
  permission java.lang.RuntimePermission "getFileStoreAttributes";
  permission java.util.PropertyPermission
      "derby.__serverStartedFromCmdLine", "read, write";

  // JMX: Uncomment this permission to allow the ping operation of the
  //      NetworkServerMBean to connect to the Network Server.
  //
  //permission java.net.SocketPermission "*", "connect,resolve";

  // Needed by sysinfo. The file permission is needed to
  // check the existence of jars on the classpath. You can
  // limit this permission to just the locations which hold
  // your jar files.
  //
  // In this template file, this block of permissions is granted
```

```
  // to derbynet.jar under the assumption that derbynet.jar is
  // the first jar file in your classpath which contains the
  // sysinfo classes. If that is not the case, then you will want
  // to grant this block of permissions to the first jar file
  // in your classpath which contains the sysinfo classes.
  // Those classes are bundled into the following Derby
  // jar files:
  //
  //    derbynet.jar
  //    derby.jar
  //    derbyclient.jar
  //    derbytools.jar
  //
  permission java.util.PropertyPermission "user.*", "read";
  permission java.util.PropertyPermission "java.home", "read";
  permission java.util.PropertyPermission "java.class.path", "read";
  permission java.util.PropertyPermission "java.runtime.version", "read";
  permission java.util.PropertyPermission "java.fullversion", "read";
  permission java.lang.RuntimePermission "getProtectionDomain";
  permission java.io.FilePermission "<<ALL FILES>>", "read";
  permission java.io.FilePermission "java.runtime.version", "read";
  permission java.io.FilePermission "java.fullversion", "read";
};
```

Customizing the Network Server's security policy

You will probably want to customize the Network Server's Basic security policy.

The Network Server's Basic security policy is documented in Basic Network Server security policy.

For example, you might want to restrict the server's liberal file I/O permissions, which let the server backup/restore and export/import to or from any location in the local file system. Customizing the security policy is simple:

1. A template policy lives in the Derby distribution at `demo/templates/server.policy`. Copy the file from this location to your own file, say `myCustomized.policy`. All of the following edits take place in your custom file.
2. Replace the `${derby.install.url}` variable with the location of the Derby jars in your local file system.
3. Replace the `${derby.system.home}` variable with the location of your Derby system directory. Alternatively, rather than replacing this variable, you can simply set the value of the `derby.system.home` system property when you boot the server.
4. Replace the `${derby.drda.traceDirectory}` variable with the location of your server trace file if you plan to use tracing.
5. Grant `java.net.SocketPermission` to `derby.jar` if you are using LDAP authentication, so that the Derby code is allowed to contact the LDAP server to perform the authentication.
6. You may want to restrict the socket permission for `derbynet.jar`, which by default accepts connections from any host (`"*"`). Note that the special wildcard address `"0.0.0.0"` is not understood by `SocketPermission`, even though Derby accepts this wildcard as a valid value for accepting connections on all network interfaces (IPv4).
7. Refine the file permissions needed by backup/restore, import/export, and the loading of application jars.

The following is a sample customized policy file:

```
grant codeBase "file:/usr/local/share/sw/derby/lib/derby.jar"
{
  // These permissions are needed for everyday, embedded Derby usage.
  //
```

```
permission java.lang.RuntimePermission "createClassLoader";
permission java.util.PropertyPermission "derby.*", "read";
permission java.util.PropertyPermission "user.dir", "read";

// The next two properties are used to determine if the VM is 32 or
// 64 bit.
//
permission java.util.PropertyPermission "sun.arch.data.model", "read";
permission java.util.PropertyPermission "os.arch", "read";

// Customized to actual location of derby.system.home:
//
permission java.io.FilePermission
    "/usr/local/shoppingCartApp/databases", "read";
permission java.io.FilePermission
    "/usr/local/shoppingCartApp/databases/-", "read,write,delete";

// This permission lets a DBA reload the policy file while the server
// is still running. The policy file is reloaded by invoking the
// SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY() system procedure.
//
permission java.security.SecurityPermission "getPolicy";

// This permission lets you backup and restore databases
// to and from a selected branch of the local file system:
//
permission java.io.FilePermission
  "/usr/local/shoppingCartApp/backups/-", "read,write,delete";

// This permission lets you import data from
// a selected branch of the local file system:
//
permission java.io.FilePermission
    "/usr/local/shoppingCartApp/imports/-", "read";

// This permission lets you export data to
// a selected branch of the local file system:
//
permission java.io.FilePermission
    "/usr/local/shoppingCartApp/exports/-", "write";

// This permission lets you load your databases with jar files of
// application code
//
permission java.io.FilePermission "/usr/local/shoppingCartApp/lib/*",
    "read";

// LDAP server on localhost using default LDAP port 389:
//
permission java.net.SocketPermission "localhost:389",
    "connect,resolve";

// Permissions needed for JMX based management and monitoring.
// Uncomment the following MBeanServerPermission, MBeanPermission and
// MBeanTrustPermission if you need JMX monitoring. Consider the
// security implications before you open up for JMX
// monitoring.
//
// Allows this code to create an MBeanServer:
//
// permission javax.management.MBeanServerPermission
//     "createMBeanServer";
//
// Allows access to Derby's built-in MBeans, within the domain
// org.apache.derby. Derby must be allowed to register and unregister
// these MBeans. It is possible to allow access only to specific
// MBeans, attributes or operations. To fine-tune this permission, see
// the API documentation for javax.management.MBeanPermission or the
// JMX Instrumentation and Agent Specification:
//
```

```
  // permission javax.management.MBeanPermission
  //     "org.apache.derby.*#[org.apache.derby:*]",
  //     "registerMBean, unregisterMBean";
  //
  // Trusts Derby code to be a source of MBeans and to register these
  // in the MBean server:
  //
  // permission javax.management.MBeanTrustPermission "register";

  // getProtectionDomain is an optional permission needed for printing
  // classpath information to derby.log. Consider if this could be a
  // security risk before enabling it.
  //
  // permission java.lang.RuntimePermission "getProtectionDomain";

  // The following permission must be granted for
  // Connection.abort(Executor) to work. Note that this permission must
  // also be granted to outer (application) code domains.
  // Uncomment this permission if you plan to use Connection.abort.
  //
  // permission java.sql.SQLPermission "callAbort";

  // Needed by file permissions restriction system (see the
  // documentation for derby.storage.useDefaultFilePermissions in the
  // Reference Manual). Consider restricting the database file-level
  // permissions for security.
  //
  permission java.lang.RuntimePermission "accessUserInformation";
  permission java.lang.RuntimePermission "getFileStoreAttributes";
};

grant codeBase "file:/usr/local/share/sw/derby/lib/derbynet.jar"
{
  // This permission lets the Network Server manage connections from
  // clients originating from the localhost, on any port. Consider the
  // security implications before you open up database connections
  // from other hosts.
  //
  permission java.net.SocketPermission "localhost:0-", "accept,listen";

  // Needed for server tracing.
  //
  permission java.io.FilePermission
      "/usr/local/shoppingCartApp/trace/-", "read,write,delete";

  // Needed by file permissions restriction system:
  //
  permission java.lang.RuntimePermission "accessUserInformation";
  permission java.lang.RuntimePermission "getFileStoreAttributes";

  // Mandatory if running with a customized policy file.
  //
  permission java.util.PropertyPermission
      "derby.__serverStartedFromCmdLine", "read, write";

  //
  // JMX: Uncomment this permission to allow the ping operation of the
  //      NetworkServerMBean to connect to the Network Server.
  //
  // permission java.net.SocketPermission "*", "connect,resolve";

  // Needed by the sysinfo tool only. You may want to remove the
  // block of permissions below if you don't plan to use it. The file
  // permission is needed to check the existence of jars on the
  // classpath. You can limit this permission to just the locations
  // which hold your jar files.
  //
  // This block of permissions is granted to derbynet.jar under the
  // assumption that derbynet.jar is the first jar file in your
  // classpath which contains the sysinfo classes. If that is not the
```

```
  // case, then you will want to grant this block of permissions to
  // the first jar file in your classpath which contains the sysinfo
  // classes.  Those classes are bundled into the following Derby jar
  // files:
  //
  //    derbynet.jar
  //    derby.jar
  //    derbyclient.jar
  //    derbytools.jar
  //
  permission java.util.PropertyPermission "user.*", "read";
  permission java.util.PropertyPermission "java.home", "read";
  permission java.util.PropertyPermission "java.class.path", "read";
  permission java.util.PropertyPermission "java.runtime.version", "read";
  permission java.util.PropertyPermission "java.fullversion", "read";
  permission java.lang.RuntimePermission "getProtectionDomain";
  permission java.io.FilePermission
      "/usr/local/shoppingCartApp/jars/-", "read";
  permission java.io.FilePer mission "java.runtime.version", "read";
  permission java.io.FilePermission "java.fullversion", "read";
};
```

After customizing the Basic policy, you may bring up the Network Server as follows:

```
java -Djava.security.manager \
-Djava.security.policy=/usr/local/shoppingCartApp/lib/myCustomized.policy
 \
org.apache.derby.drda.NetworkServerControl start -h localhost
```

Running the Network Server without a security policy

You may override the Network Server's impulse to install a security manager if, for some reason, you need to run your application outside Java's security protections.

CAUTION: You incur a severe security risk by opening up the server to all clients without limiting access via user authentication and a security policy.

Use the `-noSecurityManager` option to force the Network Server to come up without a security manager. For example:

```
java org.apache.derby.drda.NetworkServerControl start \
   -h localhost -noSecurityManager
```

Running the Network Server with user authentication

By default, the Network Server boots with user authentication disabled. However, it is strongly recommended that you run your Network Server with user authentication enabled.

For details on how to enable user authentication, see "Working with user authentication" in the *Java DB Developer's Guide*.

Network encryption and authentication with SSL/TLS

By default, all Derby network traffic is unencrypted, with the exception of user names and user passwords, which may be encrypted separately.

See Network client security for more information.

There is also no network layer access control mechanism. For deployment scenarios where these are possible security issues, the Derby Network Server supports network security with Secure Socket Layer/Transport Layer Security (SSL/TLS).

With SSL/TLS, the client/server communication protocol is encrypted, and both the client and the server may, independently of each other, require certificate-based authentication of the other part.

It is assumed that the reader is somewhat familiar with SSL, key pairs, and certificates. This documentation is also based on the Java Development Kit (JDK) and its `keytool` application.

For the remainder of this section, the term *SSL* is used for SSL/TLS, and the term *peer* is used for the other part of the communication (the server's *peer* is the client and vice versa).

SSL for Derby (both for client and for server) operates in three possible modes:

off
: The default, no SSL encryption

basic
: SSL encryption, no peer authentication

peerAuthentication
: SSL encryption and peer authentication

Peer authentication may be set on the server, on the client, or on both. Peer authentication means that the other side of the SSL connection is authenticated based on a trusted certificate installed locally.

Alternatively, a Certification Authority (CA) certificate may be installed locally and the peer has a certificate signed by that authority. How to achieve this is not described in this document. Consult your Java environment documentation for details on this.

Attention: *If a plaintext client tries to communicate with an SSL server, or if an SSL client tries to communicate with a plaintext server, the plaintext side of the communication will see the SSL communication as noise and report protocol errors.*

Key and certificate handling

For SSL operation, the server always needs a key pair. If the server runs in peer authentication mode (the server authenticates the clients), each client needs its own key pair. In general, if one end of the communication wants to authenticate its partner, the first end needs to install a certificate generated by the partner.

The key pair is located in a file which is called a *key store*, and the JDK's SSL provider needs the system properties `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword` to access the keystore.

The certificates of trusted parties are installed in a file called a *trust store*. The JDK's SSL provider needs the system properties `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` to access the trust store.

Key pair generation

Key pairs are generated with `keytool -genkey`. The simplest way to generate a key pair is to do the following:

```
keytool -genkey alias -keystore keystore
```

`keytool` will prompt for needed information, such as identity details and passwords.

Consult the JDK documentation for more information on `keytool`.

Certificate generation

Certificates are generated with `keytool -export` as follows:

```
keytool -export -alias alias -keystore keystore \
        -rfc -file certificate-file
```

The certificate file may then be distributed to the relevant parties.

Certificate installation

Installation of a certificate in a trust store is done with `keytool -import` as follows:

```
keytool -import -alias alias -file certificate-file \
        -keystore truststore
```

Examples

Generate the server key pair:

```
>keytool -genkey -alias myDerbyServer -keystore serverKeyStore.key
```

Generate a server certificate:

```
keytool -export -alias myDerbyServer -keystore serverKeyStore.key \
        -rfc -file myServer.cert
```

Generate a client key pair:

```
keytool -genkey -alias aDerbyClient -keystore clientKeyStore.key
```

Generate a client certficate:

```
keytool -export -alias aDerbyClient -keystore clientKeyStore.key \
        -rfc -file aClient.cert
```

Install a client certificate in the server's trust store:

```
keytool -import -alias aDerbyClient -file aClient.cert
        -keystore serverTrustStore.key
```

Install the server certificate in a client's trust store:

```
keytool -import -alias myDerbyServer -file myServer.cert
        -keystore clientTrustStore.key
```

Starting the server with SSL/TLS

For server SSL/TLS, a server key pair needs to be generated. If the server is going to do client authentication, the client certificates need to be installed in the trust store.

These operations are described in Key and certificate handling.

SSL at the server side is activated with the property `derby.drda.sslMode` (default off) or the `-ssl` option for the server start command.

Starting the server with basic SSL encryption

When the SSL mode is set to `basic`, the server will only accept SSL encrypted connections.

The properties `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword` need to be set with the proper values.

Example

```
java -Djavax.net.ssl.keyStore=serverKeyStore.key \
     -Djavax.net.ssl.keyStorePassword=qwerty \
     -jar derbyrun.jar server start -ssl basic
```

Starting a server which authenticates clients

When the server's SSL mode is set to `peerAuthentication`, the server authenticates its clients' identity in addition to encrypting network traffic. In this situation, the server's *trust store* must contain a certificate for each client which will connect.

The `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` need to be set in addition to the properties above.

See Running the client with SSL/TLS for client settings when the server does client authentication.

Example

```
java -Djavax.net.ssl.keyStore=serverKeyStore.key \
     -Djavax.net.ssl.keyStorePassword=qwerty \
     -Djavax.net.ssl.trustStore=serverTrustStore.key \
     -Djavax.net.ssl.trustStorePassword=qwerty \
     -jar derbyrun.jar server start -ssl peerAuthentication
```

Running the client with SSL/TLS

Basic SSL encryption on the client is enabled either by the URL attribute `ssl`, the property `ssl`, or the datasource attribute `ssl` set to `basic`.

Example

```
Connection c =
   getConnection("jdbc:derby://myhost:1527/db;ssl=basic");
```

Running a client which authenticates the server

If the client wants to authenticate the server, then the client's *trust store* must contain the server's certificate. See Key and certificate handling.

Client SSL with server authentication is enabled by the URL attribute `ssl` or the property `ssl` set to `peerAuthentication`. In addition, the system properties `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` need to be set.

Example

```
System.setProperty("javax.net.ssl.trustStore","clientTrustStore.key");
   System.setProperty("javax.net.ssl.trustStorePassword","qwerty");
   Connection c =

getConnection("jdbc:derby://myhost:1527/db;ssl=peerAuthentication");
```

Running the client when the server does client authentication

If the server does client authentication, the client will need a key pair and a client certificate which is installed in the server's *trust store*. See Key and certificate handling.

The client needs to set `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword`.

Example

```
   System.setProperty("javax.net.ssl.keyStore","clientKeyStore.key");
   System.setProperty("javax.net.ssl.keyStorePassword","qwerty");
   Connection c =
      getConnection("jdbc:derby://myhost:1527/db;ssl=basic");
```

Running the client when both parties do peer authentication

This is a combination of the two last variants.

Example

```
   System.setProperty("javax.net.ssl.keyStore","clientKeyStore.key");
   System.setProperty("javax.net.ssl.keyStorePassword","qwerty");

System.setProperty("javax.net.ssl.trustStore","clientTrustStore.key");
```

```
    System.setProperty("javax.net.ssl.trustStorePassword","qwerty");
    Connection c =

 getConnection("jdbc:derby://myhost:1527/db;ssl=peerAuthentication");
```

Other server commands

The other server commands (`shutdown`, `ping`, `sysinfo`, `runtimeinfo`, `logconnections`, `maxthreads`, `timeslice`, `trace`, and `tracedirectory`) are implemented as clients, and they behave exactly as clients with regards to SSL.

See Running the client with SSL/TLS for more information.

The SSL mode is set with the property `derby.drda.sslMode` or the server command option `-ssl`.

Examples

The following command will shut down an SSL-enabled server:

```
java -jar derbyrun.jar server shutdown -ssl basic
```

Similarly, if you have `peerAuthentication` on both sides, use the following command:

```
java -Djavax.net.ssl.keyStore=clientKeyStore.key \
     -Djavax.net.ssl.keyStorePassword=qwerty \
     -Djavax.net.ssl.trustStore=clientTrustStore.key \
     -Djavax.net.ssl.trustStorePassword=qwerty \
     -jar derbyrun.jar server shutdown -ssl peerAuthentication
```

Configuring the Network Server to handle connections

You can configure the Network Server to use a specific number of threads to handle connections. You can change the configuration on the command line.

The minimum number of threads is the number of threads that are started when the Network Server is booted. This value is specified as a property, `derby.drda.minThreads=`*`min`*. The maximum number of threads is the maximum number of threads that will be used for connections. If more connections are active than there are threads available, the extra connections must wait until the next thread becomes available. Threads can become available after a specified time, which is checked only when a thread has finished processing a communication.

- You can change the maximum number of threads by using the following command (all on one line):

  ```
  java org.apache.derby.drda.NetworkServerControl maxthreads max
    [-h hostname] [-p portnumber
  ```

  You can also use the `derby.drda.maxThreads` property to assign the maximum value. A *max* value of 0 means that there is no maximum and a new thread will be generated for a connection if there are no current threads available. This is the default. The *max* and *min* values are stored as integers, so the theoretical maximum is 2147483647 (the maximum size of an integer). But the practical maximum is determined by the machine configuration.
- To change the time that a thread should work on one session's request and check if there are waiting sessions, use the following command (all on one line):

  ```
  java org.apache.derby.drda.NetworkServerControl
    timeslice milliseconds [-h hostname] [-p portnumber]
  ```

  You can also use the `derby.drda.timeSlice` property to set this value. A value of 0 milliseconds indicates that the thread will not give up working on the session until the session ends. A value of -1 milliseconds indicates to use the default. The

default value is 0. The maximum number of milliseconds that can be specified is 2147483647 (the maximum size of an integer).

For more information on these properties, see derby.drda.minThreads property, derby.drda.maxThreads property, and derby.drda.timeSlice property.

Controlling logging by using the log file

The Network Server uses the `derby.log` file to log problems that it encounters. It also logs connections when the property `derby.drda.logConnections` is set to `true`.

See derby.drda.logConnections property for information on this property.

The `derby.log` file is created when the Derby server is started. The Network Server then records the time and version. If a log file exists, it is overwritten, unless the property `derby.infolog.append` is set to `true`.See "derby.infolog.append" in the *Java DB Reference Manual* for information on this property.

When the Network Server is logging connections, it also logs the Connection Number; this log message is written both to the `derby.log` file and to the Network Server console.

- To turn on connection logging, use the following command (all on one line):

```
java org.apache.derby.drda.NetworkServerControl
  logconnections on [-h hostname] [-p portnumber]
```

- To turn off connection logging, use the following command (all on one line):

```
java org.apache.derby.drda.NetworkServerControl
  logconnections off [-h hostname][-p portnumber]
```

See the *Java DB Developer's Guide* for more information about the `derby.log` file.

Controlling tracing by using the trace facility

Use the trace facility only if you are working with technical support and they require tracing information.

Turning on the trace facility

Follow these steps to turn on the trace facility.

1. Turn on tracing for all sessions by specifying the following property:

   ```
   derby.drda.traceAll=true
   ```

   See derby.drda.traceAll property for information on this property.

   Alternatively, while the Network Server is running, you can use the following command (all on one line) to turn on the trace facility:

   ```
   java org.apache.derby.drda.NetworkServerControl
     trace on [-s connection-number] [-h hostname][-p portnumber]
   ```

   If you specify a *connection-number*, tracing will be turned on only for that connection.
2. Set the location of the tracing files by specifying the following property:

   ```
   derby.drda.traceDirectory=directory-for-tracing-files
   ```

   See derby.drda.traceDirectory property for information on this property.

   Alternatively, while the Network Server is running, use the following command (all on one line) to set the trace directory:

```
java org.apache.derby.drda.NetworkServerControl traceDirectory
  directory-for-tracing-files [-h hostname] [-p portnumber]
```

You need to specify only the directory where the tracing files will reside. The names of the tracing files are determined by the system. If you do not set a trace directory, the tracing files will be placed in `derby.system.home`.

The Network Server will attempt to create the trace directory (and any parent directories) if they do not exist. This will require that the Java security policy for `derbynet.jar` permits verification of the existence of the named trace directory and all necessary parent directories. For each directory created, the policy must allow

```
permission java.io.FilePermission "directory", "read,write";
```

For the trace directory itself, the policy must allow

```
permission java.io.FilePermission "tracedirectory${/}-", "write";
```

See Customizing the Network Server's security policy for information about customizing the Network Server's security policy.

Turning off the trace facility

Enter the following command (all on one line) to turn off tracing.

```
java org.apache.derby.drda.NetworkServerControl trace off
  [-s connection number] [-h hostname] [-p portnumber]
```

The tracing files are named `Server`*X*`.trace`, where *X* is a connection number.

Derby Network Server sample programs

Derby provides several sample programs for Network Server users.

The NsSample sample program

The `NsSample` demonstration program is a simple JDBC application that interacts with the Network Server.

The `NsSample` program performs the following tasks:

- Starts the Network Server.
- Checks that the Network Server is running.
- Loads the Network Client driver. (Note that this step is not necessary if you are running the client on JDK 1.6 or higher. In that environment, the network client driver loads automatically.)
- Creates the `NsSampledb` database if it has not already been created.
- Checks to see if the schema is already created, and if not, creates the schema, which includes the SAMPLETBL table and corresponding indexes.
- Connects to the database.
- Loads the schema by inserting data.
- Starts client threads to perform database related operations.
- Has each of the clients perform DML operations (select, insert, delete, update) using JDBC calls. For example, one client thread establishes an embedded connection to perform database operations, while another client thread establishes a client connection to the Network Server to perform database operations.
- Waits for the client threads to finish the tasks.
- Shuts down the Network Server at the end of the demonstration.

The sample program files, both source and compiled class files, are located in the `%JAVA_HOME%\demo\db\programs\nserverdemo\` directory if you installed the Demos and Samples for the JDK. See "Installing Java DB" in *Getting Started with Java DB* for details.

The source files are as follows:

- `NsSample.java`

  This is the entry point into the sample program. The program starts up two client threads. The first client establishes an embedded connection to perform database operations, and the second client establishes a client connection to the Network Server to perform database operations.

  You can change the following constants to modify the sample program:

  NUM_ROWS
  The number of rows that must be initially loaded into the schema.
  ITERATIONS
  The number of iterations for which each client thread does database related work.
  NUM_CLIENT_THREADS
  The number of clients that you want to run the program against.
  NETWORKSERVER_PORT
  The port on which the Network Server is running.
- `NsSampleClientThread.java`
  This file contains two Java classes:
  - The `NsSampleClientThread` class extends `Thread` and instantiates a `NsSampleWork` instance.
  - The `NsSampleWork` class contains everything that is required to perform DML operations using JDBC calls. The `doWork` method in the `NsSampleWork` class represents all the work done as part of this sample program.
- `NetworkServerUtil.java`

  This file contains helper methods to start the Network Server and to shut down the server.

The compiled class files for the `NsSample` program are:

- `NsSample.class`
- `NsSampleClientThread.class`
- `NsSampleWork.class`
- `NetworkServerUtil.class`

Running the NsSample sample program

To run the `NsSample` program, follow these steps.

1. Open a command prompt and change directories to the `%JAVA_HOME%\demo\db\programs\nserverdemo` directory.
2. Set the CLASSPATH to the current directory ("."), and also include the following jar files in order to use the Network Server and the network client driver:

   `derbynet.jar`
   The Network Server jar file. It must be in your CLASSPATH to use any of the Network Server functions.
   `derbyclient.jar`
   This jar file must be in your CLASSPATH to use the Network Client driver.
   `derby.jar`
   The Derby database engine jar file.
   `derbytools.jar`
   The Derby tools jar file.
3. Test the CLASSPATH settings by running the following Java command:

```
java org.apache.derby.tools.sysinfo
```

This command shows the Derby jar files that are in the classpath as well as their respective versions.

4. After you set up your environment correctly, run the `NsSample` program from the same directory:

```
java nserverdemo.NsSample
```

If the program runs successfully, you will receive output similar to the following:

```
Derby Network Server created
Server is ready to accept connections on port 1621.
Connection number: 1.
[NsSample] Derby Network Server started.
[NsSample] Sample Derby Network Server program demo starting.
Please wait .....................
Connection number: 2.
[NsSampleWork] Begin creating table - SAMPLETBL and necessary
 indexes.
[NsSampleClientThread] Thread id - 1; started.
[NsSampleWork] Thread id - 1; requests database connection,
 dbUrl =jdbc:derby:NSSampledb;
[NsSampleClientThread] Thread id - 2; started.
[NsSampleWork] Thread id - 2; requests database connection,
 dbUrl =jdbc:derby://localhost:1621/
 NSSampledb;deferPrepares=true;
Connection number: 3.
[NsSampleWork] Thread id - 1 selected 1 row [313,Derby36
 ,1.7686243E23,9620]
[NsSampleWork] Thread id - 1 selected 1 row [313,Derby36
 ,1.7686243E23,9620]
[NsSampleWork] Thread id - 1; deleted 1 row with t_key = 9620
[NsSampleWork] Thread id - 1 selected 1 row [700,Derby34
 ,8.7620301E9,9547]
[NsSampleWork] Thread id - 1 selected 1 row [700,Derby34
 ,8.7620301E9,9547]
[NsSampleWork] Thread id - 2 selected 1 row [700,Derby34
 ,8.7620301E9,9547]
[NsSampleWork] Thread id - 2 selected 1 row [700,Derby34
 ,8.7620301E9,9547]
[NsSampleWork] Thread id - 1; inserted 1 row.
[NsSampleWork] Thread id - 1 selected 1 row [52,Derby34
 ,8.7620301E9,9547]
[NsSampleWork] Thread id - 2; updated 1 row with t_key = 9547
[NsSampleWork] Thread id - 1; deleted 1 row with t_key = 9547
[NsSampleWork] Thread id - 2 selected 1 row [617,Derby31
 ,773.83636,9321]
[NsSampleWork] Thread id - 2 selected 1 row [617,Derby31
 ,773.83636,9321]
[NsSampleWork] Thread id - 2 selected 1 row [617,Derby31
 ,773.83636,9321]
[NsSampleWork] Thread id - 2 selected 1 row [617,Derby31
 ,773.83636,9321]
[NsSampleWork] Thread id - 1; inserted 1 row.
[NsSampleWork] Thread id - 2; deleted 1 row with t_key = 9321
[NsSampleWork] Thread id - 1; deleted 1 row with t_key = 8707
[NsSampleWork] Thread id - 1; closed connection to the database.
[NsSampleClientThread] Thread id - 1; finished all tasks.
[NsSampleWork] Thread id - 2; deleted 1 row with t_key = 8490
[NsSampleWork] Thread id - 2; closed connection to the database.
[NsSampleClientThread] Thread id - 2; finished all tasks.
[NsSample] Shutting down Network Server.
Connection number: 4.
Shutdown successful.
```

Running the `NsSample` program also creates the following new directory and file: **`NSSampledb`**

This directory makes up the `NSSampledb` database.

`derby.log`
This log file contains Derby progress and error messages.

Network Server sample programs for embedded and client connections

This Derby Network Server sample program demonstrates how to obtain an embedded connection and client connections to the same database by using the Network Server. This program shows how to use either the `DriverManager` or a DataSource to obtain client connections.

For a database to be consistent, only one JVM can access it at a time. The embedded driver is loaded when the Network Server is started. The JVM that starts the Network Server can obtain an embedded connection to the same database that the Network Server is accessing to serve clients from other JVMs. This solution provides the performance benefits of the embedded driver and also allows client connections from other JVMs to connect to the same database.

Overview of the SimpleNetworkServerSample program

The `SimpleNetworkServerSample` program starts the Derby Network Server, as well as the embedded driver, and waits for clients to connect.

The program performs the following tasks.

1. Starts the Derby Network Server by using a property and also loads the embedded driver
2. Determines if the Network Server is running
3. Creates the `NSSimpleDB` database if it is not already created
4. Obtains an embedded database connection
5. Tests the database connection by executing a sample query
6. Allows client connections to connect to the server until you decide to stop the server and exit the program
7. Closes the connection
8. Shuts down the Network Server before exiting the program

To run the sample program, use the following files in the `%JAVA_HOME%\demo\db\programs\nserverdemo\` directory:

- The source file: `SimpleNetworkServerSample.java`
- The compiled class file: `SimpleNetworkServerSample.class`

You must have installed the Demos and Samples for the JDK. See "Installing Java DB" in *Getting Started with Java DB* for details.

Running the SimpleNetworkServerSample program

To run the Derby Network Server sample program, follow these steps.

1. Open a command prompt and change directories to the `%JAVA_HOME%\demo\db\programs\nserverdemo` directory.
2. Set the classpath to include the current directory (".") and the following jar files:

   `derbynet.jar`
   The Network Server jar file. It must be in your CLASSPATH because you start the Network Server in this program.
   `derby.jar`
   The database engine jar file.
   `derbytools.jar`
   The Derby tools jar file.
3. Test the CLASSPATH settings by running the following Java command:

   ```
   java org.apache.derby.tools.sysinfo
   ```

This command displays the Derby jar files that are in the classpath.

4. After you set up your environment correctly, run the `SimpleNetworkServerSample` program from the same directory:

```
java SimpleNetworkServerSample
```

If the program runs successfully, you will receive output that is similar to that shown in the following example:

```
Starting Network Server
Testing if Network Server is up and running!
Derby Network Server now running
Got an embedded connection.
Testing embedded connection by executing a sample query
number of rows in sys.systables = 16
While my app is busy with embedded work, ij might connect like this:

        $ java -Dij.user=me -Dij.password=pw -Dij.protocol=
   jdbc:derby:\\localhost:1527\ org.apache.derby.tools.ij
        ij> connect 'NSSimpleDB';

Clients can continue to connect:
Press [Enter] to stop Server
```

Running the `SimpleNetworkServerSample` program also creates the following new directory and file:

`NSSimpleDB`
This directory makes up the `NSSimpleDB` database.

`derby.log`
This log file contains Derby progress and error messages.

Connecting a client to the Network Server with the SimpleNetworkClientSample program

The `SimpleNetworkClientSample` program is a client program that interacts with the Derby Network Server from another JVM.

The program performs the following tasks:

1. Loads the network client driver. (Note that this step is not necessary if you are running the client on JDK 1.6 or higher. In that environment, the network client driver loads automatically.)
2. Obtains a client connection by using the `DriverManager`.
3. Obtains a client connection by using a DataSource.
4. Tests the database connections by running a sample query.
5. Closes the connections and then exits the program.

To run the sample program, use the following files in the `%JAVA_HOME%\demo\db\programs\nserverdemo\` directory:

- The source file: `SimpleNetworkClientSample.java`
- The compiled class file: `SimpleNetworkClientSample.class`

You must have installed the Demos and Samples for the JDK. See "Installing Java DB" in *Getting Started with Java DB* for details.

Running the SimpleNetworkClientSample program

To connect to the Network Server that has been started with the `SimpleNetworkServerSample` program, follow these steps.

1. Open a command prompt and change directories to the `%JAVA_HOME%\demo\db\programs\nserverdemo` directory.
2. Set the classpath to include the following jar files:
    - The current directory (".")
    - `derbyclient.jar`

3. After you set up your environment correctly, run the `SimpleNetworkServerSample` program from the same directory:

```
java SimpleNetworkClientSample
```

If the program runs successfully, you will receive output similar to that shown in the following example:

```
Starting Sample client program
Got a client connection via the DriverManager.
connection from datasource;
Got a client connection via a DataSource.
Testing the connection obtained via DriverManager by executing a
 sample query
number of rows in sys.systables = 16
Testing the connection obtained via a DataSource by executing a
 sample query
number of rows in sys.systables = 16
Goodbye!
```

Part Two: Derby Administration Guide

This section of the guide is divided into several administrative tasks.

Maintaining database integrity

One of the most important responsibilities of a database administrator is to maintain the integrity of the database and prevent it from becoming corrupted.

Derby must be able to sync to disk. Some machine, disk, or operating system settings can prevent a proper sync and cause unrecoverable database corruption in the event of a power failure, system crash, or software crash. To avoid database corruption, you can do the following:

- Do not touch any files or directories in the database directory, including the `log` and `seg0` directories and the `service.properties` file. Editing, adding, or deleting files in this directory may cause data corruption and leave the database in a non-recoverable state.
- Do not enable disk write caching on the hard drive that holds the database. Disable write caching if it is turned on (it is enabled by default on many Windows systems). Disk write caching can increase operating system performance. However, it can also result in the loss of information if a power failure, equipment failure, or software failure occurs. Consult your operating system support documentation for information on how to disable disk write caching.
- Run Derby on a local drive rather than on an NFS mounted, SMB mounted, or other network mounted disk.
- Disable any other settings or options that might prevent a proper sync to disk when Derby is writing its transaction logs or other data.

Many corruption issues can arise from improper backups or restores. Back up your database in a way that prevents it from becoming corrupted:

- Always make sure the database is shut down or frozen before using operating system commands to back it up.
- Always back up the database to a fresh location rather than overwriting any existing data.

After you perform a backup, check the consistency of the database. See Checking database consistency for details.

See Backing up and restoring databases for more information.

Checking database consistency

After you perform a backup, or if you experience hardware or operating system failure, you can use the `SYSCS_UTIL.SYSCS_CHECK_TABLE` system function to verify that the database is still consistent.

It is recommended that you run `SYSCS_UTIL.SYSCS_CHECK_TABLE` on all the tables in a database offline after you back it up. Do not discard the previous backup until you have verified the consistency of the current one. Otherwise, check consistency only if there are indications that such a check is needed, because a consistency check can take a long time on a large database.

See the *Java DB Reference Manual* for details about this system function.

The SYSCS_CHECK_TABLE function

The SYSCS_UTIL.SYSCS_CHECK_TABLE function checks the consistency of a Derby table.

In particular, the SYSCS_UTIL.SYSCS_CHECK_TABLE function verifies the following conditions:

- Base tables are internally consistent
- Base tables and all associated indexes contain the same number of rows
- The values and row locations in each index match those of the base table
- All BTREE indexes are internally consistent

You run this function in an SQL statement, as follows:

```
VALUES SYSCS_UTIL.SYSCS_CHECK_TABLE(SchemaName, TableName)
```

where *SchemaName* and *TableName* are expressions that evaluate to a string data type. If you created a schema or table name as a non-delimited identifier, you must present their names in all upper case. For example:

```
VALUES SYSCS_UTIL.SYSCS_CHECK_TABLE('APP', 'CITIES')
```

The SYSCS_UTIL.SYSCS_CHECK_TABLE function returns a SMALLINT. If the table is consistent (or if you run SYSCS_UTIL.SYSCS_CHECK_TABLE on a view), SYSCS_UTIL.SYSCS_CHECK_TABLE returns a non-zero value. Otherwise, the function throws an exception on the first inconsistency that it finds.

For a consistent table, the following result is displayed:

```
1
-----
1

1 row selected
```

Sample SYSCS_CHECK_TABLE error messages

This section provides examples of error messages that the SYSCS_UTIL.SYSCS_CHECK_TABLE function can return.

If the row counts of the base table and an index differ, error message X0Y55 is issued:

```
ERROR X0Y55: The number of rows in the base table does not match
the number of rows in at least 1 of the indexes on the table. Index
'T1_I' on table 'APP.T1' has 4 rows, but the base table has 5 rows.
The suggested corrective action is to recreate the index.
```

If the index refers to a row that does not exist in the base table, error message X0X62 is issued:

```
ERROR X0X62: Inconsistency found between table 'APP.T1' and index
'T1_I'. Error when trying to retrieve row location '(1,6)' from the
table. The full index key,including the row location, is '{ 1, (1,6) }'.
The suggested corrective action is to recreate the index.
```

If a key column value differs between the base table and the index, error message X0X61 is issued:

```
ERROR X0X61: The values for column 'C10' in index 'T1_C10' and
table 'APP.T1' do not match for row location (1,7). The value in the
index is '2 2   ', while the value in the base table is 'NULL'. The full
index key, including the row location, is '{ 2 2       , (1,7) }'. The
suggested corrective action is to recreate the index.
```

Sample SYSCS_CHECK_TABLE queries

This section provides examples that illustrate how to use the SYSCS_UTIL.SYSCS_CHECK_TABLE function in queries.

To check the consistency of a single table, run a query that is similar to the one shown in the following example:

```
VALUES SYSCS_UTIL.SYSCS_CHECK_TABLE('APP', 'FLIGHTS')
```

To check the consistency of all of the tables in a schema, stopping at the first failure, run a query that is similar to the one shown in the following example:

```
SELECT tablename, SYSCS_UTIL.SYSCS_CHECK_TABLE(
    'SAMP', tablename)
FROM sys.sysschemas s, sys.systables t
WHERE s.schemaname = 'SAMP' AND s.schemaid = t.schemaid
```

To check the consistency of an entire database, stopping at the first failure, run a query that is similar to the one shown in the following example::

```
SELECT schemaname, tablename,
SYSCS_UTIL.SYSCS_CHECK_TABLE(schemaname, tablename)
FROM sys.sysschemas s, sys.systables t
WHERE s.schemaid = t.schemaid
```

Backing up and restoring databases

Derby provides a way to back up a database while it is either offline or online. You can also restore a full backup from a specified location.

To back up a database, you can do any of the following:

- Shut down the database and use operating system commands to copy it to a backup location, as described in Offline backups.
- Leave the database running and call one of four system backup procedures to copy it to a backup location, as described in Using the backup procedures to perform an online backup.
- Leave the database running, but call a system procedure to freeze the database, use operating system commands to copy it to a backup location, then call a system procedure to unfreeze the database, as described in Using operating system commands with the freeze and unfreeze system procedures to perform an online backup.

To restore a database from a backup copy, you must use one of three connection URL attributes:

- `restoreFrom=`*`path`*, described in Restoring a database from a backup copy
- `createFrom=`*`path`*, described in Creating a database from a backup copy
- `rollForwardRecoveryFrom=`*`path`*, described in Roll-forward recovery

Backing up a database

You can back up a database either offline (when it is shut down) or online (when it is running).

After you back up a database, make sure the backup copy is not corrupt. To do this, run the SYSCS_UTIL.SYSCS_CHECK_TABLE system function on all the tables in the backup copy. Do not discard the previous backup until you have verified the consistency of the current one. See Checking database consistency for more information.

The topics in this section describe how to back up a database.

Offline backups

To perform an offline backup of a database, use operating system commands to copy the database directory.

> Important: You must shut down the database before you perform an offline backup.

For example, on Windows systems, the following operating system command backs up a (closed) database that is named `sample` and that is located in `d:\mydatabases` by copying it to the directory `c:\mybackups\2012-04-01`:

```
xcopy d:\mydatabases\sample c:\mybackups\2012-04-01\sample /s /i
```

If you are not using Windows, substitute the appropriate operating system command for copying a directory and all contents to a new location.

Note: On Windows systems, do not attempt to update a database while it is being backed up in this way. Attempting to update a database during an offline backup will generate a `java.io.IOException`. Using online backups prevents this from occurring.

For large systems, shutting down the database might not be convenient. To back up a database without having to shut it down, you can use an online backup.

After you back up a database, make sure the backup copy is not corrupt. To do this, run the `SYSCS_UTIL.SYSCS_CHECK_TABLE` system function on all the tables in the backup copy. Do not discard the previous backup until you have verified the consistency of the current one. See Checking database consistency for more information.

Online backups

Use online backups to back up a database while it is running, without blocking transactions.

You can perform online backups by using several types of backup procedures or by using operating system commands with the freeze and unfreeze system procedures.

Using the backup procedures to perform an online backup:

Use the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE` procedure or one of the other system backup procedures to perform an online backup of a database to a specified location.

The backup procedures are as follows:

- `SYSCS_UTIL.SYSCS_BACKUP_DATABASE`
- `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT`
- `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE`
- `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT`

Use the
`SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE`
or
`SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT`
procedure if you want to make it possible to perform a roll-forward recovery of a damaged database. See Roll-forward recovery for details.

The `NOWAIT` versions of the procedures do not wait for transactions in progress with unlogged operations to complete before proceeding with the backup; instead, they return an error immediately.

See the *Java DB Reference Manual* for details about these system procedures.

All four of these system procedures take a string argument that represents the location in which to back up the database. Typically, you provide the full path to the backup directory. (Relative paths are interpreted as relative to the current directory, not to the `derby.system.home` directory.)

For example, to specify a backup location of `c:/mybackups/2012-04-01` for a database that is currently open, use the following statement (forward slashes are used as path separators in SQL commands):

```
CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE('c:/mybackups/2012-04-01')
```

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE` or `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT` procedure puts the database into a state in which it can be safely copied. The procedure then copies the entire original database directory (including data files, online transaction log files, and jar files) to the specified backup directory. Files that are not within the original database directory (for example, `derby.properties`) are *not* copied. With the exception of a few cases mentioned in Unlogged Operations, the procedure does not block concurrent transactions at any time.

A backup made with the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE` or `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT` procedure is not a full copy of the database, but depends on the log files created in the database since the backup. An attempt to access the backup directly will invalidate the backup. The result could include a corrupted database, missing data, errors during a subsequent attempt at restoring the database, or database corruption errors encountered only once the restored database is being used. The only supported way to access this kind of backup is to restore the database as documented in Roll-forward recovery.

The following example shows how to back up a database to a directory with a name that reflects the current date:

```
public static void backUpDatabase(Connection conn)
        throws SQLException {
    // Get today's date as a string:
    java.text.SimpleDateFormat todaysDate =
        new java.text.SimpleDateFormat("yyyy-MM-dd");
    String backupdirectory = "c:/mybackups/" +
        todaysDate.format((java.util.Calendar.getInstance()).getTime());

    CallableStatement cs =
        conn.prepareCall("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE(?)");
    cs.setString(1, backupdirectory);
    cs.execute();
    cs.close();
    System.out.println("backed up database to " + backupdirectory);
}
```

For a database that was backed up on 2012-04-01, the previous commands copy the current database to a directory of the same name in `c:/mybackups/2012-04-01`.

Uncommitted transactions do not appear in the backed-up database.

Note: Do not back up different databases with the same name to the same backup directory. If a database of the same name already exists in the backup directory, it is assumed to be an older version and is overwritten.

Unlogged Operations

For some operations, Derby does not log because it can keep the database consistent without logging the data.

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE` procedure will issue an error if there are any unlogged operations in the same transaction as the backup procedure.

If any unlogged operations are in progress in other transactions in the system when the backup starts, this procedure will block until those transactions are complete before performing the backup.

Derby automatically converts unlogged operations to logged mode if they are started while the backup is in progress (except operations that maintain application jar files in the database). Procedures to install, replace, and remove jar files in a database are blocked while the backup is in progress.

If you do not want backup to block until unlogged operations in other transactions are complete, use the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT` procedure. This procedure issues an error immediately at the start of the backup if there are any transactions in progress with unlogged operations, instead of waiting for those transactions to complete.

Unlogged operations include:

- Index creation.

  Only CREATE INDEX is logged, not all the data inserts into the index. The reason inserts into the index are not logged is that if there is a failure, it will just drop the index.

  If you create an index when the backup is in progress, it will be slower, because it has to be logged.

  Foreign keys and primary keys create backing indexes. Adding those keys to an existing table with data will also run slower.
- Importing to an empty table or replacing all the data in a table.

  In this case also, data inserts into the table are not logged. Internally, Derby creates a new table for the import, changes the catalogs to point to the new table, and drops the original table when the import completes.

  If you perform such an import operation when backup is in progress, it will be slower because data is logged.

Using operating system commands with the freeze and unfreeze system procedures to perform an online backup:

Typically, these procedures are used to speed up the copy operation involved in an online backup.

In this scenario, Derby does not perform the copy operation for you. You use the `SYSCS_UTIL.SYSCS_FREEZE_DATABASE` procedure to lock the database, and then you explicitly copy the database directory by using operating system commands.

For example, because the UNIX `tar` command uses operating system file-copying routines, and the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE` procedure uses Java I/O calls with additional internal synchronization that allow updates during the backup, the `tar` command might provide faster backups than the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE` procedure.

To use operating system commands for online database backups, call the `SYSCS_UTIL.SYSCS_FREEZE_DATABASE` system procedure. The `SYSCS_UTIL.SYSCS_FREEZE_DATABASE` system procedure puts the database into a state in which it can be safely copied. After the database has been copied, use the `SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE` system procedure to continue working with the database. Only after `SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE` has been specified can transactions once again write to the database. Read operations can proceed while the database is frozen.

Note: To ensure a consistent backup of the database, Derby might block applications that attempt to write to a frozen database until the backup is completed and the `SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE` system procedure is called.

The following example demonstrates how the freeze and unfreeze procedures are used to surround an operating system copy command:

```
public static void backUpDatabaseWithFreeze(Connection conn)
 throws SQLException {
    Statement s = conn.createStatement();
    s.executeUpdate(
        "CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
    //copy the database directory during this interval
    s.executeUpdate(
        "CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
    s.close();
}
```

When the log is in a non-default location

If you put the database log in a non-default location prior to backing up the database, be aware of the following requirements.

- If you are using an operating system command to back up the database, you must explicitly copy the log file as well, as shown in the following example:

  ```
  xcopy d:\mydatabases\sample c:\mybackups\2012-04-01\sample /s /i
  xcopy h:\janet\tourslog\log c:\mybackups\2012-04-01\sample\log /s /i
  ```

  If you are not using Windows, substitute the appropriate operating system command for copying a directory and all of its contents to a new location.
- Edit the `logDevice` entry in the `service.properties` file of the database backup so that it points to the correct location for the log. In the previous example, the log was moved to the default location for a log, so you can remove the `logDevice` entry entirely, or leave the `logDevice` entry as is and wait until the database is restored to edit the entry.

See Logging on a separate device for information about the default location of the database log and about putting the log in a non-default location.

Backing up encrypted databases

When you back up an encrypted database, both the backup and the log files remain encrypted.

To restore an encrypted database, you must know the boot password.

Restoring a database from a backup copy

To restore a database by using a full backup from a specified location, specify the `restoreFrom=`*`path`* attribute in the boot-time connection URL.

If a database with the same name exists in the `derby.system.home` location, the system will delete the database, copy it from the backup location, and then restart it.

The log files are copied to the same location they were in when the backup was taken. You can use the `logDevice` attribute in conjunction with the `restoreFrom=`*`path`* attribute to store logs in a different location.

For example, to restore the sample database by using a backup copy in `c:\mybackups\sample`, the connection URL should be:

```
jdbc:derby:sample;restoreFrom=c:\mybackups\sample
```

For more information, see "restoreFrom=path attribute" in the *Java DB Reference Manual*.

Creating a database from a backup copy

To create a database from a full backup copy at a specified location, specify the `createFrom=path` attribute in the boot-time connection URL.

If there is already a database with the same name in `derby.system.home`, an error will occur and the existing database will be left intact. If there is not an existing database with the same name in the current `derby.system.home` location, the system will copy the whole database from the backup location to `derby.system.home` and start it.

The log files are also copied to the default location. You can use the `logDevice` attribute in conjunction with the `createFrom=path` attribute to store logs in a different location. With the `createFrom=path` attribute, you do not need to copy the individual log files to the log directory.

For example, to create the sample database from a backup copy in `c:\mybackups\sample`, the connection URL should be:

```
jdbc:derby:sample;createFrom=c:\mybackups\sample
```

For more information, see "createFrom=path attribute" in the *Java DB Reference Manual*.

Roll-forward recovery

Derby supports roll-forward recovery to restore a damaged database to the most recent state before a failure occurred.

Derby restores a database from full backup and replays all the transactions after the backup. All the log files after a backup are required to replay the transactions after the backup. By default, the database keeps only logs that are required for crash recovery. For roll-forward recovery to be successful, all log files must be archived after a backup. Log files can be archived using the backup function calls that enable log archiving.

In roll-forward recovery, the log archival mode ensures that all old log files are available. The log files are available only from the time that the log archival mode is enabled.

Derby uses the following information to restore the database:

- The backup copy of the database
- The set of archived logs
- The current online active log

You cannot use roll-forward recovery to restore individual tables. Roll-forward recovery recovers the entire database.

To restore a database by using roll-forward recovery, you must already have a backup copy of the database, all the archived logs since the backup was created, and the active log files. All the log files should be in the database log directory.

There are two types of log files in Derby: active logs and online archived logs.

Active logs
: Active logs are used during crash recovery to prevent a failure that might leave a database in an inconsistent state. Roll-forward recovery can also use the active logs to recover to the end of the log files. Active logs are located in the database log path directory.

Online archived logs

Log files that are stored for roll-forward recovery use when they are no longer needed for crash recovery. Online archived logs are also kept in the database log path directory.

Enabling log archival mode

Online archive logs are available only if the database is enabled for log archival mode. You can use the following system procedure to enable the database for log archival mode:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE
(IN BACKUPDIR VARCHAR(32672), IN SMALLINT DELETE_ARCHIVED_LOG_FILES)
```

The input parameters for this procedure specify the location where the backup should be stored and specify whether or not the database should keep online archived logs for the backup. Existing online archived log files that were created before this backup will be deleted if the input parameter value for the `DELETE_ARCHIVED_LOG_FILES` parameter is non-zero. The log files are deleted only after a successful backup.
Note: Make sure to store the backup database in a safe place when you choose the log file removal option.

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE` procedure will issue an error if there are any unlogged operations in the same transaction as the backup procedure.

If any unlogged operations are in progress in other transactions in the system when the backup starts, this procedure will block until those transactions are complete before performing the backup. Derby automatically converts unlogged operations to logged mode if they are started while the backup is in progress (except operations that maintain application jar files in the database). Procedures to install, replace, and remove jar files in a database are blocked while the backup is in progress.

If you do not want backup to block until unlogged operations in other transactions are complete, use the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT` procedure. This procedure issues an error immediately at the start of the backup if there are any transactions in progress with unlogged operations, instead of waiting for those transactions to complete.

Disabling log archival mode

After you enable log archival mode, the database will always have the log archival mode enabled even if it is subsequently booted or backed up. The only way to disable the log archive mode is to run the following procedure:

```
SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE
(IN SMALLINT DELETE_ARCHIVED_LOG_FILES)
```

This system procedure disables the log archive mode and deletes any existing online archived log files if the input parameter `DELETE_ARCHIVED_LOG_FILES` is non-zero.

Performing roll-forward recovery

If you have a backup made by using `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE` or `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT`, you can restore it to its most recent state by using the full backup copy, archived logs, and active logs. You perform a roll-forward recovery by specifying the connection URL attribute `rollForwardRecoveryFrom=`*`path`* at boot time. All the log files should be in the database log path directory.

The steps involved are as follows. They do not show the commands to start `ij`.

1. **Back up the database with log archive mode enabled.**

   For example, you could back up a database named `wombat` to the `/backup` directory as follows. After many operations, the database crashes.

   ```
   ij> connect 'jdbc:derby:wombat;create=true';
   ij> create table t1(a int not null primary key);
   0 rows inserted/updated/deleted
   ------------------DML/DDL Operations
   ij> CALL
    SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE
   ('/backup', 0);
   0 rows inserted/updated/deleted
   ij> insert into t1 values(19);
   1 row inserted/updated/deleted
   ij> create table t2(a int);
   0 rows inserted/updated/deleted
   ------------------DML/DDL Operations
   ------------------Database Crashed (Media Corruption on data disks)
   ```

2. **Prepare the database for restoration.**

   Before you restore the database, shut down the original database and rename the original database directory. For example, after shutdown, you could issue the following commands in a Linux shell:

   ```
   mv /databases/wombat /databases/brokenwombat
   cd /databases
   ```

3. **Restore the database using roll-forward recovery.**

   Since you moved the database, you need to specify the `logDevice=`*`logDirectoryPath`* attribute in addition to the `rollForwardRecoveryFrom=`*`path`* attribute when you restore the database using roll-forward recovery. Use commands like the following (the connection URL must be all on one line):

   ```
   ij> connect
    'jdbc:derby:wombat;rollForwardRecoveryFrom=/backup/wombat;
   logDevice=/databases/brokenwombat';
   ij> select * from t1;
   A
   -----------
   19

   1 row selected
   ---------------DML/DDL Operations
   ```

   After a database is restored from full backup, transactions from the online archived logs and active logs are replayed. This brings the database to its most recent state. All the log files should be in the directory specified by the `logDevice=`*`logDirectoryPath`* attribute.

For more information, see "rollForwardRecoveryFrom=path attribute" and "logDevice=logDirectoryPath attribute" in the *Java DB Reference Manual*.

Importing and exporting data

You can import and export large amounts of data between files and the Derby database. Instead of having to use INSERT and SELECT statements, you can use Derby system procedures to import data directly from files into tables and to export data from tables into files.

The Derby system procedures import and export data in delimited data file format.

- Use the export system procedures to write data from a database to one or more files that are stored outside of the database. You can use a procedure to export data from a table into a file or export data from a SELECT statement result into a file.
- Use the import system procedures to import data from a file into a table. If the target table already contains data, you can replace or append to the existing data.

Methods for running the import and export procedures

You can run the import and export procedures from within an SQL statement using `ij` or any Java application.

The import and export procedures read and write text files, and if you use an external file when you import or export data, you can also import and export blob data. The import procedures do not support read-once streams (live data feeds), because the procedures read the first line of the file to determine the number of columns, then read the file again to import the data.

Note: The import and export procedures are server-side utilities that exhibit different behavior in client/server mode. Typically, you use these procedures to import data into and export data from a locally running Derby database. However, you can use the import and export procedures when Derby is running in a server framework if you specify import and export files that are accessible to the server.

Bulk import and export requirements and considerations

There are requirements and limitations that you must consider before you use the Derby import and export procedures.

Database transactions
Derby issues a COMMIT or a ROLLBACK statement after each import and export procedure is run (a COMMIT if the procedure completes successfully, a ROLLBACK if it fails). For this reason, you should issue either a COMMIT or ROLLBACK statement to complete all transactions and release all table-level locks before you invoke an import or export procedure. An error in an import or export procedure and the ensuing ROLLBACK would throw away any changes performed before the procedure was called, and vice versa: any unsound changes before the import or export procedure call that should not be committed could be committed automatically.

Database connections
To invoke a Derby import or export procedure, you must be connected to the database into which the data is imported or from which the data is exported. Other user applications that access the table with a separate connection do not need to disconnect.

Classpath
You must have the `derbytools.jar` file in your classpath before you can use the import or export procedures from `ij`.

The table must exist
To import data into a table, the table must already exist in Derby. The table does not have to be empty. If the table is not empty, bulk import performs single row inserts, which result in slower performance.

Create indexes, keys, and unique constraints before you import
To avoid a separate step, create the indexes, keys (primary and foreign), and unique constraints on tables before you import data. However, if your memory and disk space resources are limited, you can build the indexes and primary keys after importing data.

Data types

Derby implicitly converts the strings to the data type of the receiving column. If any of the implicit conversions fail, the whole import is aborted. For example, "3+7" cannot be converted into an integer. An export that encounters a runtime error stops.
Note: You cannot import or export the XML data type.

Locking during import
Import procedures use the same isolation level as the connection in which they are executed to insert data into tables. During import, the entire table is exclusively locked irrespective of the isolation level.

Locking during export
Export procedures use the same isolation level as the connection in which they are executed to fetch data from tables.

Import behavior on tables with triggers
The import procedures enable INSERT triggers when data is appended to the table. The REPLACE parameter is not allowed when triggers are enabled on the table.

Restrictions on the REPLACE parameter
If you import data into a table that already contains data, you can either replace or append to the existing data. You can use the REPLACE parameter on tables that have dependent tables. The replaced data must maintain referential integrity; otherwise, the import operation will be rolled back. You cannot use the REPLACE parameter if the table has triggers enabled.

Restrictions on tables
You cannot use import procedures to import data into a system table or a declared temporary table.

Bulk import and export of large objects

You can import and export large objects (LOBs) using the Derby system procedures.

Importing and exporting CLOB and BLOB data

CLOB and BLOB data can be exported to the same file as the rest of the column data, or the LOB column data can be exported to a separate external file. When the LOB column data is exported to a separate external file, reference to the location of the LOB data is placed in the LOB column in the main export file.

Importing and exporting LOB data using a separate external file might be faster than storing the LOB data in the same file as the rest of the column data:

- The CLOB data does not have to be scanned for the delimiters inside the data
- The BLOB data does not need to be converted to hexadecimal format

Importing and exporting other binary data

When you export columns that contain the data types CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, and LONG VARCHAR FOR BIT DATA, the column data is always exported to the main export file. The data is written in hexadecimal format. To import data into a table that has columns of these data types, the data in the import file for those columns must be in hexadecimal format.

Importing LOB data from a file that contains all of the data

You can use the `SYSCS_UTIL.SYSCS_IMPORT_TABLE` and `SYSCS_UTIL.SYSCS_IMPORT_DATA` procedures to import data into a table that contains a LOB column. The LOB data must be stored in the same file as the other column data that you are importing. If you are importing data from a file that was exported from a non-Derby source, the binary data must be in hexadecimal format.

Importing LOB data from a separate external file

You can use the `SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE` and `SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE` procedures to import LOB

data that is stored in a file that is separate from the main import file. These procedures read the LOB data using the reference that is stored in the main import file. If you are importing data from a non-Derby source, the references to the LOB data must be in the main import file in the format *lobsFileName.Offset.length/*. This is the same method that the Derby export procedures use to export the LOB data to a separate external file.

Exporting LOB data to the same file as the other column data

You can use the SYSCS_UTIL.SYSCS_EXPORT_TABLE and SYSCS_UTIL.SYSCS_EXPORT_QUERY procedures to write LOB data, along with the rest of the column data, to a single export file.

CLOB column data is treated same as other character data. Character delimiters are allowed inside the CLOB data. The export procedures write the delimiter inside the data as a double-delimiter.

BLOB column data is written to the export file in hexadecimal format. For each byte of BLOB data, two characters are generated. The first character represents the high nibble (4 bits) in hexadecimal and the second character represents the low nibble.

Exporting LOB data to a separate external file from the other column data

You can use the SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE and SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE procedures to write LOB data to a separate external file. These procedures include the LOBSFILENAME parameter, which specifies the name of the external file for the LOB data.

When you use these procedures, the location of the LOB data is written to the main export file. The format of the reference to the LOB stored in the main export file is *lobsFileName.Offset.length/*.

- *Offset* is the position in the external file in bytes
- *length* is the size of the LOB column data in bytes

If a LOB column value is NULL, *length* is written as -1. No data conversion is performed when you export LOB data to an external file. BLOB data is written in binary format, and CLOB data is written using the codeset that you specify.

See Examples of bulk import and export for examples using each of the import and export procedures.

File format for input and output

There are specific requirements for the format of the input and output files when you import and export data.

The default file format is a delimited text file with the following characteristics:

- Rows are separated by a newline
- Fields are separated by a comma (,)
- Character-based fields are delimited with double quotes (")

Restriction: Before you perform import or export operations, you must ensure that the chosen delimiter character is not contained in the data to be imported or exported. If you chose a delimiter character that is part of the data to be imported or exported, unexpected errors might occur. The following restrictions apply to column and character delimiters:

- Delimiters are mutually exclusive
- A delimiter cannot be a line-feed character, a carriage return, or a blank space
- The default decimal point (.) cannot be a character delimiter
- Delimiters cannot be hexadecimal characters (0-9, a-f, A-F).

The record delimiter is assumed to be a newline character. The record delimiter should not be used as any other delimiter.

Character delimiters are permitted with the character-based fields (CHAR, VARCHAR, and LONG VARCHAR) of a file during import. Any pair of character delimiters found between the enclosing character delimiters is imported into the database. For example, suppose that you have the following character string:

```
"What a ""great"" day!"
```

The preceding character string gets imported into the database as:

```
What a "great" day!
```

During export, the rule applies in reverse. For example, suppose you have the following character string:

```
"The boot has a 3" heel."
```

The preceding character string gets exported to a file as:

```
"The boot has a 3""heel."
```

The following example file shows four rows and four columns in the default file format:

```
1,abc,22,def
22,,,"a is a zero-length string, b is null"
13,"hello",454,"world"
4,b and c are both null,,
```

The export procedure outputs the following values:

```
1,"abc",22,"def"
22,,,"a is a zero-length string, b is null"
13,"hello",454,"world"
4,"b and c are both null",,
```

Importing data using the built-in procedures

You can use the Derby import procedures to import all of the data from a table or query, or to import LOB data separately from the other data.

1. Use the following table to choose the correct procedure for the type of import that you want to perform. For examples of these procedures, see Examples of bulk import and export.

   Table 11. Using the built-in import procedures

Type of Import	Procedure to Use
To import all the data to a table, where the import file contains the LOB data	`SYSCS_UTIL.SYSCS_IMPORT_TABLE (IN SCHEMANAME VARCHAR(128), IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128), IN REPLACE SMALLINT)`
To import the data to a table, where the LOB data is stored in a separate file and the main import file contains	`SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE (IN SCHEMANAME VARCHAR(128), IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),`

Type of Import	Procedure to Use

all of the other data with a reference to the LOB data	`  IN CODESET VARCHAR(128),`
	`  IN REPLACE SMALLINT)`
	
	The import utility looks in the main import file for a reference to the location of the LOB data. The format of the reference to the LOB stored in the main import file must be *`lobsFileName.Offset.length/`*.

To import data from a file to a subset of columns in a table, where the import file contains the LOB data	`SYSCS_UTIL.SYSCS_IMPORT_DATA`
	`(IN SCHEMANAME VARCHAR(128),`
	`IN TABLENAME VARCHAR(128),`
	`IN INSERTCOLUMNS VARCHAR(32672),`
	`IN COLUMNINDEXES VARCHAR(32672),`
	`IN FILENAME VARCHAR(32672),`
	`IN COLUMNDELIMITER CHAR(1),`
	`IN CHARACTERDELIMITER CHAR(1),`
	`IN CODESET VARCHAR(128),`
	`IN REPLACE SMALLINT)`
	
	You must specify the `INSERTCOLUMNS` parameter on the table into which data will be imported. You must specify the `COLUMNINDEXES` parameter to import data fields from a file to a column in a table.

To import data to a subset of columns in a table, where the LOB data is stored in a separate file and the main import file contains all of the other data with a reference to the LOB data	`SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE`
	`(IN SCHEMANAME VARCHAR(128),`
	`IN TABLENAME VARCHAR(128),`
	`IN INSERTCOLUMNS VARCHAR(32672),`
	`IN COLUMNINDEXES VARCHAR(32672),`
	`IN FILENAME VARCHAR(32672),`
	`IN COLUMNDELIMITER CHAR(1),`
	`IN CHARACTERDELIMITER CHAR(1),`
	`IN CODESET VARCHAR(128),`
	`IN REPLACE SMALLINT)`
	
	The import utility looks in the main import file for a reference to the location of the LOB data. The format of the reference to the LOB stored in the main import file must be *`lobsFileName.Offset.length/`*.

Parameters for the import procedures

The Derby import procedures use specific parameters.

`SCHEMANAME`
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The `SCHEMANAME` parameter takes an input argument that is a VARCHAR(128) data type.

`TABLENAME`
Specifies the name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. The string must exactly match the case of the table name. Specifying a NULL value results in an error. The `TABLENAME` parameter takes an input argument that is a VARCHAR(128) data type.

`INSERTCOLUMNS`
Specifies the comma-separated column names of the table into which the data will be imported. You can specify a NULL value to import into all columns of the table. The `INSERTCOLUMNS` parameter takes an input argument that is a VARCHAR(32672) data type.

`COLUMNINDEXES`
Specifies the comma-separated column indexes (numbered from one) of the input data fields that will be imported. You can specify a NULL value to use all input data fields in the file. The `COLUMNINDEXES` parameter takes an input argument that is a VARCHAR(32672) data type.

FILENAME
Specifies the name of the file that contains the data to be imported. If the path is omitted, the current working directory is used. The specified location of the file should refer to the server side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR(32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter takes an input argument that is a CHAR(1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR(1) data type.

CODESET
Specifies the code set of the data in the input file. The code set name should be one of the Java supported character encoding sets. Data is converted from the specified code set to the database code set (UTF-8). You can specify a NULL value to interpret the data file in the same code set as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR(128) data type.

REPLACE
A non-zero value for the REPLACE parameter will import in REPLACE mode, while a zero value will import in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the table and inserts the imported data. The table definition and the index definitions are not changed. You can import with REPLACE mode only if the table already exists. INSERT mode adds the imported data to the table without changing the existing table data. Specifying a NULL value results in an error. The REPLACE parameter takes an input argument that is a SMALLINT data type.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all uppercase characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Import into tables that contain identity columns

You can use the either the SYSCS_UTIL.SYSCS_IMPORT_DATA procedure or the SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE procedure to import data into a table that contains an identity column. The approach that you take depends on whether the identity column is GENERATED ALWAYS or GENERATED BY DEFAULT.

Identity columns and the REPLACE parameter

If the REPLACE parameter is used during import, Derby resets its internal counter of the last identity value for a column to the initial value defined for the identity column.

Identity column is GENERATED ALWAYS

If the identity column is defined as GENERATED ALWAYS, an identity value is always generated for a table row. When a corresponding row in the input file already contains a value for the identity column, the row cannot be inserted into the table and the import operation will fail.

To prevent such failure, the following examples show how to specify parameters in the SYSCS_UTIL.SYSCS_IMPORT_DATA and SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE procedures to ignore data for the identity column from the file, and omit the column name from the insert column list.

The following table definition contains an identity column, c2, and is used in the examples below:

```
CREATE TABLE tab1 (c1 CHAR(30), c2 INT GENERATED ALWAYS AS IDENTITY,
   c3 REAL, c4 CHAR(1))
```

- Suppose that you want to import data into tab1 from a file, myfile.del, that does not have identity column information. The myfile.del file contains three fields with the following data:

  ```
  Robert,45.2,J
  Mike,76.9,K
  Leo,23.4,I
  ```

  To import the data, you must explicitly list the column names in the tab1 table, except for the identity column c2, when you call the procedure. For example:

  ```
  CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', 'C1,C3,C4',
      null, 'myfile.del',null, null, null, 0)
  ```
- Suppose that you want to import data into tab1 from a file, empfile.del, that also has identity column information. The file contains three fields with the following data:

  ```
  Robert,1,45.2,J
  Mike,2,23.4,I
  Leo,3,23.4,I
  ```

  To import the data, you must explicitly specify an insert column list without the identity column c2 and specify the column indexes without identity column data when you call the procedure. For example:

  ```
  CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', 'C1,C3,C4',
      '1,3,4', 'empfile.del',null, null, null, 0)
  ```

Identity column is GENERATED BY DEFAULT

If the identity column is defined as GENERATED BY DEFAULT, an identity value is generated for a table row only if no explicit value is given. This means that you have several options, depending on the contents of your input file and the desired outcome of the import processing:

- You may omit the identity column from the insert column list, in which case Derby will generate a new value for the identity column for each input row. You may use this option whether or not the input file contains values for the identity column, but note that if the input file contains values for the identity column, you must also then omit the identity column from the column indexes when you call the procedure.
- You may include the identity column in the insert column list, in which case Derby will use the column values from the input file. Of course, this option is available only if the input file actually contains values for the identity column.

The following table definition contains an identity column, c2, and is used in the examples below:

```
CREATE TABLE tab1 (c1 CHAR(30),
   c2 INT GENERATED BY DEFAULT AS IDENTITY,
   c3 REAL, c4 CHAR(1))
```

- Suppose that you want to import data into tab1 from a file, myfile.del, that does not have identity column information. The myfile.del file contains three fields with the following data:

  ```
  Robert,45.2,J
  Mike,76.9,K
  ```

```
Leo,23.4,I
```

To import the data, you must explicitly list the column names in the `tab1` table, except for the identity column `c2`, when you call the procedure. For example:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', 'C1,C3,C4',
    null, 'myfile.del',null, null, null, 0)
```

- Suppose that you want to import data into `tab1` from a file, `empfile.del`, that also has identity column information. The file contains three fields with the following data:

```
Robert,1,45.2,J
Mike,2,23.4,I
Leo,3,23.4,I
```

In this case, suppose that you wish to use the existing identity column values from the input file. To import the data, you may simply pass `null` for the insert column list and column indexes parameters when you call the procedure. For example:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', NULL,
    NULL, 'empfile.del',null, null, null, 0)
```

- Suppose (again) that you want to import data into `tab1` from a file, `empfile.del`, that also has identity column information, but in this case, suppose that you do **not** wish to use the identity column values from the input file, but would prefer to allow Derby to generate new identity column values instead. In this case, to import the data, you must specify an insert column list without the identity column `c2` and specify the column indexes without identity column data when you call the procedure. For example:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', 'C1,C3,C4',
    '1,3,4', 'empfile.del',null, null, null, 0)
```

Exporting data using the built-in procedures

You can use the Derby export procedures to export all of the data from table or query, or to export LOB data separately from the other data.

1. Use the following table to choose the correct procedure for the type of export that you want to perform. For examples of these procedures, see Examples of bulk import and export.

Table 12. Using the built-in export procedures

Type of Export	Procedure to Use
To export all the data from a table to a single export file, including the LOB data	`SYSCS_UTIL.SYSCS_EXPORT_TABLE (IN SCHEMANAME VARCHAR(128), IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128))`
To export all the data from a table, and place the LOB data into a separate export file	`SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE (IN SCHEMANAME VARCHAR(128), IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128), IN LOBSFILENAME VARCHAR(32672))` A reference to the location of the LOB data is placed in the LOB column in the main export file.

Type of Export	Procedure to Use
To export the result of a SELECT statement to a single file, including the LOB data	`SYSCS_UTIL.SYSCS_EXPORT_QUERY (IN SELECTSTATEMENT VARCHAR(32672), IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128))`
To export the result of a SELECT statement to a main export file, and place the LOB data into a separate export file	`SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE (IN SELECTSTATEMENT VARCHAR(32672), IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128), IN LOBSFILENAME VARCHAR(32672))` A reference to the LOB data is written to the main export file.

Parameters for the export procedures

The Derby export procedures use specific parameters.

`SCHEMANAME`
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The `SCHEMANAME` parameter takes an input argument that is a VARCHAR(128) data type.

`SELECTSTATEMENT`
Specifies the SELECT statement query that returns the data to be exported. Specifying a NULL value will result in an error. The `SELECTSTATEMENT` parameter takes an input argument that is a VARCHAR(32672) data type.

`TABLENAME`
Specifies the table name of the table or view from which the data is to be exported. This table cannot be a system table or a declared temporary table. The string must exactly match the case of the table name. Specifying a NULL value results in an error. The `TABLENAME` parameter takes an input argument that is a VARCHAR(128) data type.

`FILENAME`
Specifies the file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export utility overwrites the contents of the file; it does not append the information. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The `FILENAME` parameter takes an input argument that is a VARCHAR(32672) data type.

`COLUMNDELIMITER`
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The `COLUMNDELIMITER` parameter must be a CHAR(1) data type.

`CHARACTERDELIMITER`
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The `CHARACTERDELIMITER` parameter takes an input argument that is a CHAR(1) data type.

`CODESET`
Specifies the code set of the data in the export file. The code set name should be one of the Java supported character encoding sets. Data is converted from the database code page to the specified code page before writing to the file. You can specify a NULL value to write the data in the same code page as the JVM in which it is being executed. The `CODESET` parameter takes an input argument that is a VARCHAR(128) data type.

LOBSFILENAME

Specifies the file that the large object data is exported to. If the path is omitted, the LOB file is created in the same directory as the main export file. If you specify the name of an existing file, the export utility overwrites the contents of the file. The data is not appended to the file. If you are using the Network Server, the file should be in a server-side location. Specifying a NULL value results in an error. The `LOBSFILENAME` parameter takes an input argument that is a VARCHAR(32672) data type.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the export procedure using all uppercase characters. If you created a schema or table name as a delimited identifier, you must pass the name to the export procedure using the same case that was used when it was created.

Examples of bulk import and export

All of the examples in this section are run using the `ij` utility.

Example: Importing all data from a file

The following example shows how to import data into the STAFF table in a sample database from the `myfile.del` file. The data will be appended to the existing data in the table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE(
    null,'STAFF','myfile.del',null,null,null,0);
```

Example: Importing all data from a delimited file

The following example shows how to import data into the STAFF table in a sample database from a delimited data file, `myfile.del`. This example defines the percentage character (%) as the string delimiter, and a semicolon as the column delimiter. The data will be appended to the existing data in the table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE(
    null,'STAFF','c:\output\myfile.del',';','%',null,0);
```

Example: Importing all data from a table, using a separate import file for the LOB data

The following example shows how to import data into the STAFF table in a sample database from a delimited data file, `staff.del`. The import file `staff.del` is the main import file and contains references that point to a separate file which contains the LOB data. This example specifies a comma as the column delimiter. The data will be appended to the existing data in the table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE(
    null,'STAFF','c:\data\staff.del',',','"','UTF-8',0);
```

Example: Importing data into specific columns, using a separate import file for the LOB data

The following example shows how to import data into several columns of the STAFF table. The STAFF table includes a LOB column in a sample database. The import file, `staff.del`, is a delimited data file. The `staff.del` file contains references that point to a separate file which contains the LOB data. The data in the import file is formatted using double quotation marks (") as the string delimiter and a comma (`,`) as the column delimiter. The data will be appended to the existing data in the STAFF table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE(
    null, 'STAFF', 'NAME,DEPT,SALARY,PICTURE', '2,3,4,6',
    'c:\data\staff.del', ',','"','UTF-8',  0);
```

Example: Exporting all data from a table to a single export file

The following example shows how to export data from the STAFF table in a sample database to the file `myfile.del`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE(
    null,'STAFF','myfile.del',null,null,null);
```

Example: Exporting data from a table to a single delimited export file

The following example shows how to export data from the STAFF table to a delimited data file, `myfile.del`, with the percentage character (%) as the character delimiter, and a semicolon as the column delimiter from the STAFF table.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE(
    null,'STAFF','c:\output\myfile.del',';','%',null);
```

Example: Exporting all data from a table, using a separate export file for the LOB data

The following example shows how to export data from the STAFF table in a sample database to the main file, `staff.del`, and the LOB export file, `pictures.dat`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE(null,'STAFF',
    'c:\data\staff.del',',','"','UTF-8', 'c:\data\pictures.dat');
```

Example: Exporting data from a query to a single export file

The following example shows how to export employee data in department 20 from the STAFF table in a sample database to the file `awards.del`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY(
    'SELECT * FROM STAFF WHERE dept=20',
    'c:\output\awards.del',null,null,null);
```

Example: Exporting data from a query, using a separate export file for the LOB data

The following example shows how to export employee data in department 20 from the STAFF table in a sample database to the main file, `staff.del`, and the LOB data to the file `pictures.dat`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE(
    'SELECT * FROM STAFF WHERE dept=20',
    'c:\data\staff.del', ',' ,'"',
    'UTF-8','c:\data\pictures.dat');
```

Running import and export procedures from JDBC

You can run import and export procedures from a JDBC program.

The following code fragment shows how you might call the `SYSCS_UTIL.SYSCS_EXPORT_TABLE` procedure from a Java program. In this example, the procedure exports the data in the `staff` table in the default schema to the `staff.dat` file. A percentage (%) character is used to specify the column delimiter.

```
PreparedStatement ps = conn.prepareStatement(
    "CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE (?,?,?,?,?,?)");
ps.setString(1,null);
ps.setString(2,"STAFF");
ps.setString(3,"staff.dat");
ps.setString(4,"%");
ps.setString(5,null);
ps.setString(6,null);
ps.execute();
```

How the import and export procedures process NULL values

In a delimited file, a NULL value is exported as an empty field.

The following example shows the export of a four-column row where the third column is empty:

```
7,95,,Happy Birthday
```

The import procedures work the same way; an empty field is imported as a NULL value.

CODESET values for import and export procedures

Import and export procedures accept arguments to specify codeset values. You can specify the codeset (character encoding) for import and export procedures to override the system default.

For a table that shows a sample of the character encodings supported by the Java Development Kit, see "derby.ui.codeset property" in the *Java DB Tools and Utilities Guide*. To review the complete list of character encodings, refer to your Java documentation.

Examples: Specifying the codeset in import and export procedures

The following example shows how to specify UTF-8 encoding to export to the `staff.dat` table:

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE(
    NULL,'STAFF','staff.dat',NULL,NULL,'UTF-8')
```

The following example shows how to specify UTF-8 encoding to import from the `staff.dat` table:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE(
    NULL,'STAFF','staff.dat',NULL,NULL,'UTF-8',0)
```

Replicating databases

Replication is an important feature of a robust database management system. In Derby, you start database replication by using connection URL attributes.

The replication capability of Derby has the following features:

- **One master, one slave**: A replicated database resides in two locations and is managed by two different Derby instances. One of these Derby instances has the *master* role for this database, and the other has the *slave* role. Typically, the master and slave run on different nodes, but this is not a requirement. Together, the master and its associated slave represent a *replication pair*.
- **Roll-forward shipped log**: Replication is based on shipping the Derby transaction log from the master to the slave, and then rolling forward the operations described in the log to the slave database.
- **Asymmetry**: Only the master processes transactions. The slave processes no transactions, not even read operations.
- **Asynchronicity**: Transactions are committed on the master without waiting for the slave. The shipping of the transaction log to the slave is performed regularly, and is completely decoupled from the transaction execution at the master. This may lead to a few lost transactions if the master crashes.
- **Shared nothing**: Apart from the network line, no hardware is assumed to be shared.

- **Replication granularity**: The granularity for replication is exactly one database. However, one Derby instance may have different roles for different databases. For example, one Derby instance may have the following roles, all at the same time:
  - The master role for one database D1 replicated to one node
  - The slave role for a database D2 replicated from another node
  - The normal, non-replicated, role for a database D3

Replication builds on Derby's ability to recover from a crash by starting with a backup and rolling forward Derby's transaction log files. The master sends log records to the slave using a network connection. The slave then writes these log records to its local log and redoes them.

If the master fails, the slave completes the recovery by redoing the log that has not already been processed. The state of the slave after this recovery is close to the state the master had when it crashed. However, some of the last transactions performed on the master may not have been sent to the slave and may therefore not be reflected. When the slave has completed the recovery work, it is transformed into a normal Derby instance that is ready to process transactions. For more details, see Forcing a failover and Replication and security.

Several Derby properties allow you to specify the size of the replication log buffers and the intervals between log shipments, as well as whether replication messages are logged. See the *Java DB Reference Manual* for details.

You can perform replication on a database that runs in either embedded mode or Network Server mode.

Starting and running replication

Each replicated database is replicated from a master to a slave version of that database.

Initially there is no replication; a master database must be created before it can be replicated. The database may, of course, be empty when replication starts. On the other hand, replication does not need to be specified immediately after the database is created; it can be initiated at any time after the database is created.

Before you start replication, you must shut down the master database and then copy the database to the slave location. Follow these steps to start replication:

1. Make sure that the database on the master system is shut down cleanly.
2. Copy the database to the slave location.
3. Start slave replication mode on the Derby instance that is acting as the slave for the database. To start slave replication, use the `startSlave=true` attribute and, optionally, the `slaveHost=`*`hostname`* and `slavePort=`*`portValue`* attributes. For example, for a database named `wombat`, you might use the following connection URL:

   ```
   jdbc:derby:wombat;startSlave=true
   ```
4. Start master replication mode on the Derby instance that is acting as the master for the database. To start replication, connect to the database on the master system using the `startMaster=true` attribute in conjunction with the `slaveHost=`*`hostname`* attribute (and, optionally, the `slavePort=`*`portValue`* attribute). For example, you might use the following connection URL:

   ```
   jdbc:derby:wombat;startMaster=true;slaveHost=myremotesystem
   ```

   A successful use of the `startMaster=true` attribute will also start the database.

See the *Java DB Reference Manual* for details about these attributes.

After replication has been started, the slave is ready to receive logged operations from the master. The master can now continue to process transactions. From this point on, the master forwards all logged operations to the slave in chunks. The slave repeats these operations by applying the contents of the Derby transaction log, but does not process any other operations. Attempts to connect to the slave database are refused. In case of failure, the slave can recover to the state the master was in at the time the last chunk of the transaction log was sent.

While replication is running, neither the slave or the master database is permitted to be shut down. Replication must be stopped before you can shut down either the slave or the master database. There is one exception to this rule: if the entire system is shut down, the peer that is shut down notifies the other replication peer that replication is stopped.

If you install jar files on the master system while replication is running, the same jars are not automatically installed on the slave. But because the transaction log information sent to the slave system includes the jar file installation, the slave database has a record of the jar files, even though they are not actually there. Therefore, you must install the jar files on the former slave after a failover by calling either `SQLJ.remove_jar` followed by `SQLJ.install_jar`, or `SQLJ.replace_jar`. (For information on installing jar files, see "Loading classes from a database" in the *Java DB Developer's Guide* and "System procedures for storing jar files in a database" in the *Java DB Reference Manual*.)

If the jar files must be available to clients immediately after a failover, you must stop replication and then start replication over again from the beginning, so that the slave database will have the same jar files as the master.

Stopping replication

To stop replication of a database, connect to the master database using the `stopMaster=true` connection URL attribute.

The master sends the remaining log records that await shipment, and then sends a stop replication command to the slave. The slave then writes all logs to disk and shuts down the database. For example, for a database named `wombat`, you might specify the following connection URL:

```
jdbc:derby:wombat;stopMaster=true
```

To stop replication on the slave system if the connection to the master is lost, use the `stopSlave=true` connection URL attribute.

See the *Java DB Reference Manual* for details about these attributes.

You cannot resume replication after it has been stopped. You need to start replication over again from the beginning using the `startMaster=true` attribute, as described in Starting and running replication.

Forcing a failover

At any time, you can transform the Derby database that has the slave role into a normal Derby database that can process transactions. This transformation from being a slave to becoming an active Derby database is called *failover*.

During failover, the slave applies the parts of the transaction log that have not yet been processed. It then undoes operations that belong to uncommitted transactions, resulting in a transaction-consistent state that includes all transactions whose commit log record has been sent to the slave.

You perform failover from the master system. To do so, you connect to the database on the master system using the `failover=true` connection URL attribute. For example, for a database named `wombat`, you might specify the following connection URL:

```
jdbc:derby:wombat;failover=true
```

If the network connection between the master system and the slave system is lost, you can perform failover from the slave system.

See the *Java DB Reference Manual* for details about the `failover=true` attribute.

There is no automatic failover or restart of replication after one of the instances has failed.

Replication and security

If you want to perform replication with the security manager enabled, you must modify the security policy file on both the master and slave systems to allow the master-slave network connection.

The section to be modified is the one following this line:

```
grant codeBase "${derby.install.url}derby.jar"
```

Add the following permission to the policy file on the master system:

```
permission java.net.SocketPermission "slaveHost:slavePort",
 "connect,resolve";
```

Add the following permissions to the policy file on the slave system:

```
permission java.net.SocketPermission "slaveHost", "accept,resolve";
permission java.net.SocketPermission "localhost:slavePort", "listen";
```

slaveHost and *slavePort* are the values you specify for the `slaveHost=`*`hostname`* and `slavePort=`*`portValue`* attributes, which are described in the *Java DB Reference Manual*.

See Basic Network Server security policy for details on the security policy file.

Depending on the security mode Derby is running under, the measures described in the following table are enforced when you specify the replication-related connection URL attributes.

Table 13. Replication behavior with Derby security

Security Mode	Replication Attribute Requirements
No security	Anyone may specify the replication attributes
Authentication is turned on	Normal Derby connection policy: specify valid `user=`*`userName`* and `password=`*`userPassword`* attributes
Authorization is turned on	The `user=`*`userName`* and `password=`*`userPassword`* attributes must be valid, and the user must be the owner of the replicated database

Replication failure handling

Replication can encounter several failure situations. The following table lists these situations and describes the actions that Derby takes as a result.

Table 14. Replication failure handling

Failure Situation	Action Taken
Master loses connection with slave.	Transactions are allowed to continue processing while the master tries to reconnect with the slave. Log records generated while the connection is down are buffered in main memory. If the log buffer reaches its size limit before the connection can be reestablished, the master replication functionality is stopped. You can use the property `derby.replication.logBufferSize` to configure the size limit of the buffer; see the *Java DB Reference Manual* for details.
Slave loses connection with master.	The slave tries to reestablish the connection with the master by listening on the specified host and port. It will not give up until it is explicitly requested to do so by either the `failover=true` or `stopSlave=true` connection URL attribute. If a failover is requested, the slave applies all received log records and boots the database as described in Forcing a failover. If the `stopSlave=true` attribute is specified, the slave database is shut down without further actions.
Two different masters of database D try to replicate to the same slave.	The slave will only accept the connection from the first master attempting to connect. Note that authentication is required to start both the slave and the master, as described in Replication and security.
The master and slave Derby instances are not at the same Derby version.	An exception is raised and replication does not start.
The master Derby instance crashes, then restarts.	Replication must be restarted, as described in Starting and running replication.
The master Derby instance is not able to send log data to the slave at the same pace as the log is generated. The main memory log buffer gradually fills up and eventually becomes full.	The master notices that the main memory log buffer is filling up. It first tries to increase the speed of the log shipment to keep the amount of log in the buffer below the maximum. If that is not enough to keep the buffer from getting full, the response time of transactions may increase for as long as log shipment has trouble keeping up with the amount of generated log records. You can use properties to tune both the log buffer size and the minimum and maximum interval between consecutive log shipments. See "derby.replication.logBufferSize", "derby.replication.maxLogShippingInterval", and "derby.replication.minLogShippingInterval" in the *Java DB Reference Manual* for details.
The slave Derby instance crashes.	The master sees this as a lost connection to the slave. The master tries to reestablish the connection until the replication log buffer is full. Replication is then stopped on the master. Replication must be restarted, as described in Starting and running replication.

Failure Situation	Action Taken
An unexpected failure is encountered.	Replication is stopped. The other Derby instance of the replication pair is notified of the decision if the network connection is still alive.

Logging on a separate device

You can improve the performance of update-intensive, large databases by putting a database's log on a separate device, which reduces I/O contention.

By default, the transaction log is in the `log` subdirectory of the database directory. Use either of the following methods to store this `log` subdirectory in another location:

- Specify the non-default location by using the `logDevice=`*`logDirectoryPath`* attribute on the database connection URL when you create the database.
- If the database is already created, move the log manually and update the `service.properties` file.

Using the logDevice=logDirectoryPath attribute

To specify a non-default location for the log directory, set the `logDevice=`*`logDirectoryPath`* attribute on the database connection URL.

This attribute is meaningful when you are creating a database or when you are restoring a database using roll-forward recovery. You can specify `logDevice=`*`logDirectoryPath`* as either an absolute path or as a path that is relative to the directory where the JVM is executed.

Setting `logDevice=`*`logDirectoryPath`* on the database connection URL when you create the database adds an entry to the `service.properties` file. If you ever move the log manually, you will need to alter the entry in `service.properties`. If you move the log back to the default location, remove the `logDevice` entry from the `service.properties` file.

To check the log location for an existing database, you can retrieve the `logDevice=`*`logDirectoryPath`* attribute as a database property by using the following statement:

```
VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('logDevice')
```

For more information, see Roll-forward recovery in this manual and "logDevice=logDirectoryPath attribute" in the *Java DB Reference Manual*.

Example of creating a log in a non-default location

The following database connection URL creates a database in the directory `d:/mydatabases`, but puts the database log directory in `h:/janets/tourslog`.

```
jdbc:derby:d:/mydatabases/toursDB;
    create=true;logDevice=h:/janets/tourslog
```

Example of moving a log manually

If you want to move the log to `g:/bigdisk/tourslog`, move the log with operating system commands.

For example, you could use the following command:

```
move h:\janets\tourslog\log\*.* g:\bigdisk\tourslog\log
```

Then, alter the `logDevice` entry in `service.properties` to read as follows:

```
logDevice=g:/bigdisk/toursLog
```

Note: You can use either a single forward slash or double back slashes for a path separator.

If you later want to move the log back to its default location (in this case, `d:\mydatabases\toursDB\log`), move the log manually as follows:

```
move g:\bigdisk\tourslog\log\*.* d:\mydatabases\toursDB\log
```

Then, delete the `logDevice` entry from `service.properties`.

Note: This example uses commands that are specific to the Windows operating system. Use commands appropriate to your operating system to copy a directory and all of its contents to a new location.

Issues for logging in a non-default location

When the log is not in the default location, backing up and restoring a database can require extra steps.

See Backing up and restoring databases for details.

Obtaining locking information

Derby provides a tool to monitor and display locking information.

This tool can help you create applications that minimize deadlock. It can also help you locate the cause of deadlock when it does occur.

To diagnose locking problems, constantly monitor locking traffic by logging all deadlocks by using the `derby.locks.monitor` property, which is described in the *Java DB Reference Manual*.

Monitoring deadlocks

The `derby.stream.error.logSeverityLevel` property determines the level of error that you are informed about.

By default, `derby.stream.error.logSeverityLevel` is set to 40000. If `derby.stream.error.logSeverityLevel` is set to display transaction-level errors (that is, if it is set to a value less than 40000), deadlock errors are logged to the `derby.log` file. If it is set to a value of 40000 or higher, deadlock errors are not logged to the `derby.log` file.

The `derby.locks.monitor` property ensures that deadlock errors are logged regardless of the value of `derby.stream.error.logSeverityLevel`. When `derby.locks.monitor` is set to true, all locks that are involved in deadlocks are written to `derby.log` along with a unique number that identifies the lock.

To see a thread's stack trace when a lock is requested, set `derby.locks.deadlockTrace` to `true`. This property is ignored if `derby.locks.monitor` is set to false.

Note: Use `derby.locks.deadlockTrace` with care. Setting this property can alter the timing of the application, severely affect performance, and produce a very large `derby.log` file.

For information about working with properties, see the *Java DB Developer's Guide*. For information about the specific properties that are mentioned in this topic, see the *Java DB Reference Manual*.

Here is an example of an error message when Derby aborts a transaction because of a deadlock:

```
--SQLException Caught--

SQLState: 40001 =
Error Code: 30000
Message: A lock could not be obtained due to a deadlock,
cycle of locks and waiters is: Lock : ROW, DEPARTMENT, (1,14)
Waiting XID : {752, X} , APP, update department set location='Boise'
 where deptno='E21'
Granted XID : {758, X} Lock : ROW, EMPLOYEE, (2,8)
Waiting XID : {758, U} , APP, update employee set bonus=150 where
 salary=23840
Granted XID : {752, X} The selected victim is XID : 752
```

Note: You can use the `derby.locks.waitTimeout` and `derby.locks.deadlockTimeout` properties to configure how long Derby waits for a lock to be released, or when to begin deadlock checking. For more information about these properties, see "Controlling Derby application behavior" in the *Java DB Developer's Guide*.

Reclaiming unused space

A Derby table or index (sometimes called a *conglomerate*) can contain unused space after large amounts of data have been deleted or updated.

This happens because, by default, Derby does not return unused space to the operating system. After a page has been allocated to a table or index, Derby does not automatically return the page to the operating system until the table or index is dropped, even if the space is no longer needed. However, Derby does provide a way to reclaim unused space in tables and associated indexes.

If you determine that a table and its indexes have a significant amount of unused space, use either the `SYSCS_UTIL.SYSCS_COMPRESS_TABLE` or `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` procedure to reclaim that space. `SYSCS_UTIL.SYSCS_COMPRESS_TABLE` is guaranteed to recover the maximum amount of free space, at the cost of temporarily creating new tables and indexes before the statement is committed. `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` attempts to reclaim space within the same table, but cannot guarantee it will recover all available space. The difference between the two procedures is that unlike `SYSCS_UTIL.SYSCS_COMPRESS_TABLE`, the `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` procedure uses no temporary files and moves rows around within the same conglomerate.

You can use the `SYSCS_DIAG.SPACE_TABLE` diagnostic table to estimate the amount of unused space in a table or index by examining, in particular, the values of the NUMFREEPAGES and ESTIMSPACESAVING columns. For example:

```
SELECT * FROM TABLE(SYSCS_DIAG.SPACE_TABLE('APP', 'FLIGHTAVAILABILITY'))
 AS T
```

For more information about `SYSCS_DIAG.SPACE_TABLE` see "SYSCS_DIAG diagnostic tables and functions" in the *Java DB Reference Manual*.

As an example, after you have determined that the `FlightAvailability` table and its related indexes have too much unused space, you could reclaim that space with the following command:

```
call SYSCS_UTIL.SYSCS_COMPRESS_TABLE('APP', 'FLIGHTAVAILABILITY', 0);
```

The third parameter in the `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` procedure determines whether the operation will run in sequential or non-sequential mode. If you specify `0` for the third argument in the procedure, the operation will run in non-sequential mode. In sequential mode, Derby compresses the table and indexes sequentially, one at a time. Sequential compression uses less memory and disk space but is slower. To force the operation to run in sequential mode, substitute a non-zero SMALLINT value for the third argument. The following example shows how to force the procedure to run in sequential mode:

```
call SYSCS_UTIL.SYSCS_COMPRESS_TABLE('APP', 'FLIGHTAVAILABILITY', 1);
```

For more information about this command, see the *Java DB Reference Manual*.

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.