[vertical list of authors]

[cover art/text goes here]

Contents

Copyright

Second Edition (July 2005)

About this guide

This section describes who this guide is for as well as how to use it.

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

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.
- *Derby Network Server advanced topics*

  Describes advanced topics for Derby Network Server users.

Part two: Derby Administration Guide

- *Checking database consistency*

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

  Describes how to back up a database when it is online.
- *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 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 utilizes 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 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 *Derby 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 *Derby Developer's Guide* for more information.

Administrative tools:

Derby provides some tools and features to assist database administrators, including:

- Consistency checker
- Online backup
- The ability to put a database's log on a separate device

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
- Through the servlet interface
- 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.

Managing the Derby Network Server remotely by using the servlet interface explains how to use the servlet interface to manage the Network Server.

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 Java, 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.

```
/* loading the client driver boots the client driver only*/
Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
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 (see Setting Network Server properties), or you can start it by using a program. 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.put("user","usr");
props.put("password","pwd");

Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
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 and tuning. Before reading this guide, you should first learn about basic Derby functionality by reading the *Derby Developer's Guide* . Also, because multi-user environments typically have performance and tuning issues, you should read *Tuning Derby* .

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 JVMs

The Derby Network Server is compatible with Java(TM) 2 Platform, Standard Edition, v 1.3.1 (J2SE) 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 files to your server classpath:

- derbynet.jar

  This jar file contains the Network Server code. It is only necessary for the process that starts the Network Server in addition to the standard Derby .jar files.

- derby.jar

  This file must be in your classpath to use any of the Derby Network Server functions.

- derbyclient.jar

  This jar file must be in your class path to use the Network Client driver. The jar file is necessary for client-side communication with the Network Server using the Derby Network Client driver. It needs to be in the client-side classpath to use the Network Client driver to access Derby.

Derby provides script files for setting the classpath to work with the Network Server. The scripts are located in the $DERBY_INSTALL\frameworks\NetworkServer\bin directory.

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

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

Starting the Network Server

Note that you should always properly shut down the Network Server after use, because failure to do so might result in unpredictable side-effects, such as blocked ports on the server.

Use the startNetworkServer.bat script to start the Network Server on Windows machines and the startNetworkServer.ksh script to start the Network Server on UNIX systems. These scripts are located in $DERBY_INSTALL/frameworks/NetworkServer/bin, where $DERBY_INSTALL is the directory where you installed Derby.

You can run NetworkServerControl commands only from the host that started the Network Server.

To start the Network Server, you run the appropriate script from the command line. For example, on a Windows system, if you have installed Derby in the default directory on the C drive and you have set up your classpath correctly, type the following command:

```
$DERBY_INSTALL\frameworks\NetworkServer\bin\startNetworkserver.bat
```

The default system directory is the directory in which Derby was started. (See the *Derby Developer's Guide* for more information about the default system directory.)
Tip: You can set $DERBY_INSTALL\frameworks\NetworkServer\bin in your path to shorten the command.

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.

Altering the startNetworkServer script

You can modify the startNetworkServer script in any of the following ways:

- 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
  ```

  where 1368 is the new port number.

- 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:

  Remember: Before using this option, you should run under the Java security manager and enable user authentication.

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

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

  On all interfaces, you can specify a host name, IP address or 0.0.0.0 to listen.

Starting the Network Server without using the script

If you don't want to use the StartNetworkServer script, you can start the Network Server by using the command line. The syntax for the command looks like this:

```
java org.apache.derby.drda.NetworkServerControl start
   [-h <hostname>] [-p <portNumber>]
```

Starting the Network Server from a Java application

Note that you should always properly shut down the Network Server 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);
```

Shutting down the Network Server

If user authentication is disabled, a Derby database will shut down normally when the Network Server is shut down. If user authentication is enabled, you must explicitly shut down the database *before* shutting down the Network Server by specifying a valid Derby user name and password.

The database can be shut down either directly, or by the Derby server.

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

```
stopNetworkServer.bat [-h <hostname>] [-p <portnumber>]
```

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

```
stopNetworkServer.ksh [-h <hostname>] [-p <portnumber>]
```

These scripts are located in the $DERBY_INSTALL/frameworks/NetworkServer/bin directory.

Shutting down by using the command line

From the command line, shut down the Network Server with the following command:

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

Shutting down by using the API

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

```
shutdown();
```

For example, the following command shuts down the Network Server running on the current machine using port 1527.

```
NetworkServerControl server = new NetworkServerControl();
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, the NetworkServerControl API, and through the servlet interface.

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

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

  ```
  sysinfo.bat [-h <hostname>][-p <portnumber>]
  ```

- Run the following **sysinfo** script to obtain information about the Network Server on a UNIX system:

  ```
  sysinfo.ksh [-h <hostname>] [<-p portnumber>]
  ```

Obtaining system information by using the command line

To run **sysinfo** from the command line, use the following command while the Network Server is running:

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

Administrative commands such as `org.apache.derby.drda.NetworkServerControl 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 **sysinfo** 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();
```

These methods return information about the Network Server running on the current machine on the default port number (1527).

Obtaining Network Server runtime information:

Use the **runtimeinfo** command or **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 getCurrent Properties 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 NetServer 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.

See Managing the Derby Network Server remotely by using the servlet interface for information about obtaining system information using the servlet interface.

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.

Table1. Standard JDBC DataSource properties

Property	Type	Description	URL attribute	Notes
databaseName	String	The name of the database. This property is required.	'	This property is also available using EmbeddedDataSource.
dataSourceName	String	The data source name.	'	This property is also available using EmbeddedDataSource.
description	String	A description of the data source.	'	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.	'	Default is "localhost".
portNumber	Integer	The port number where the server is listening for requests.	'	Default is "1527".

Table1. 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*	'
traceDirectory	String	The directory for the tracing output. Each connection will send output to a separate file. Setting this property turns on tracing. See Network client tracing .	*traceDirectory*	'
traceLevel	Integer	The level of client tracing if *traceFile* or *traceDirectory* are set.	*traceLevel*	The default is TRACE_ALL.
traceFileAppend	Boolean	Value is true if tracing output should append to the existing trace file.	*traceFileAppend*	The default is false.
securityMechanism	Integer	The security mechanism. See Network client security .	*securityMechanism*	The default is USER_ONLY _SECURITY.

Property	Type	Description	URL attribute	Notes
retrieveMessageText	Boolean	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.	*retrieveMessageText*	The default is true.

Table1. Server-Specific DataSource properties

Property	Type	Description	URL attributes	Notes
connectionAttributes	String	Set to the list of Derby embedded connection attributes separated by semicolons.	Various	This property is also available using EmbeddedDataSource. See the *Derby Reference Manual* for more information about the various connection attributes.

Note that *setAttributesAsPassword* , 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.

Security mechanisms supported by the Derby Network Client lists the security mechanisms that the Derby Network Client supports, and the corresponding property value to specify to obtain this securityMechanism. 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.

Table1. Security mechanisms supported by the Derby Network Client

Security and mechanism	`securityMechanism` property value	Comments
User id and password	ClientDataSource. CLEAR_TEXT_PASSWORD_SECURITY (0x03)	Default if password is set
User id only	ClientDataSource. USER_ONLY_SECURITY	Default if password is not set

Security and mechanism	`securityMechanism` property value	Comments
	(0x04)	
Encrypted user id and encrypted password	ClientDataSource. ENCRYPTED_USER_AND_ PASSWORD_SECURITY (0x09)	Encryption requires a JCE implementation that supports the Diffie-Helman algorithm with a prime of 32 bytes.

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` 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';
```

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.

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` or `traceDirectory` URL attributes to set these properties prior to creating the connection with the `DriverManager.getConnection()` method.

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` URL attribute:

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

Table1. 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	0x3

Trace level	Value
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
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
  ```

Network client driver examples

Example 1

The following example connects to the default server name localhost on the default port, 1527, and to the database sample. It specifies the user and password URL attributes. You must set these attributes before attempting to connect to the server.

```
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
```

Example 4

The following example shows how to use the network client driver to connect the network client to the Network Server:

```
String databaseURL = "jdbc:derby://localhost:1527/sample";
// Load Derby Network Client driver class
Class.forName("org.apache.derby.jdbc.ClientDriver");
// Set user and password properties
Properties properties = new Properties();
properties.put("user", "APP");
properties.put("password", "APP");
// Get a connection
Connection conn = DriverManager.getConnection(databaseURL, properties);
```

Accessing the Network Server by using the DB2 Universal Driver

You can use the DB2 Universal Driver instead of the Derby network client driver to connect to the Network Server. Your application needs to load the driver and connection URL that is specific to the Network Server. In addition, you specify a user name and password. If you have not set up authentication, you can use any value for the user name and password. The driver that you use to access the Network Server is:

```
com.ibm.db2.jcc.DB2Driver
```

You must have the following two jar files present in your classpath in order to use the DB2 Universal Driver:

- `db2jcc.jar`
- `db2jcc_license_c.jar`

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

```
jdbc:derby:net://<server>[:<port>]/
<databaseName>[;<Derby URL attribute>=<value> [;...]]
[:<Universal Driver attribute>=<value>; [...;]]
```

After you specify the database name and attributes, you can include attributes for the DB2 JDBC Driver. You must include a semicolon after the last Universal Driver attribute.

server
The name of the machine where the server is running. It can be the name of the machine (for example, *buffy*) or the IP address, for example, *158.58.62.225*.
Note: Unless the Network Server was started with the `-h` option or the *derby.drda.host* property set, this value must be `localhost`.

port
The port that the server is listening to. The default is 1527.

database name
The name of the database that you are connecting to. The database name can be a maximum of 18 characters. You must use quotation marks (") to include path information in the database name. Alternately, you can specify path information by setting the property *derby.system.home* in either the `derby.properties` file or in the Java environment when you start the Network Server. See the *Derby Developer's Guide* for more information about defining the system home.

derby URL attribute=value
Optional database connection URL attributes that are supported by Derby. See the *Derby Developer's Guide* for more information.

Universal Driver Attribute=value

Optional database connection URL attributes that are supported by the DB2 Universal JDBC Driver.

The DB2 JDBC Universal Driver requires that you set the Universal Driver user and password attributes to non-null values.

The following DB2 Universal JDBC Driver attributes are available to you when running the Network Server:

user

User name (required by the Universal JDBC Driver).

password

User password (required by the Universal JDBC Driver).

portNumber

The TCP/IP port number where the Network Server listens for connection requests to this data source. The default is 1527.

retrieveMessagesFromServerOnGetMessage

Displays error messages from the server.

readOnly

Creates a read-only connection. The default is false.

logWriter

A character output stream. All logging and tracing messages print to the `logWriter` property.

traceLevel

Specifies the granularity of logging messages to the `logWriter` property.

traceFile

Provides an explicit file location for the trace output.

securityMechanism

Indicates what type of security mechanism is used.

deferPrepares

Controls when prepared statements are physically prepared in the database server. The default value is true.

Universal Driver System information

The Derby Network Server is compatible with the DB2 JDBC Universal Driver release 2.4 and higher.

DB2 Universal Driver examples:

Example 1

The following example connects to the default server name *localhost* on the default port, *1527*, and to the database *sample*. It specifies the URL attributes *user, password,* and *retrieveMessagesFromServerOnGetMessage*. You must set the Universal Driver Attributes user name and password.

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

Example 2

The following example specifies both Derby and Universal Driver Attributes:

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

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. The database name must be delimited by double quotes and you cannot specify Derby attributes on the URL.

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

Example 4

The following is a sample program fragment that connects to the Network Server using the Universal Driver:

```
String databaseURL = "jdbc:derby:net://localhost:1527/sample";
```

```
// Load IBM JDBC Universal Driver class
Class.forName("com.ibm.db2.jcc.DB2Driver");
// Set user and password properties
Properties properties = new Properties();
properties.put("user", "APP");
properties.put("password", "APP");
properties.put("retreiveMessagesFromServerOnGetMessage", "true");
// Get a connection
Connection conn = DriverManager.getConnection(databaseURL, properties);
```

Accessing the Network Server by using a DataSource

The Derby Network Client driver DataSources `org.apache.derby.jdbc.ClientDataSource` and `org.apache.derby.jdbc.ClientConnectionPoolDataSource` are supported by the Network Server.

DataSource access example

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

```
public static javax.sql.DataSource getDS(String database, String user,
String
password) throws SQLException
{
org.apache.derby.jdbc.ClientDataSource ds =
   new org.apache.derby.jdbc.ClientDataSource();

// DatabaseName can include Derby URL Attributes
ds.setDatabaseName(database);

if (user != null)
   ds.setUser(user);
if (password != null)
   ds.setPassword(password);

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

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

return ds;
}
```

The program then can connect:

```
javax.sql.DataSource ds = getDS("mydb;create=true", null, null);
// Note:  user and password are required on connection
Connection conn = ds.getConnection("usr2", "pass2");
```

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 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 (`org.apache.derby.jdbc.ClientXADataSource`).

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;

String driver = "org.apache.derby.jdbc.ClientDataSource";
ClientXADataSource ds = new ClientXADataSource();

ds.setDatabaseName ("sample;create=true");

ds.setServerName("localhost");

ds.setPortNumber(1527);

Class.forName(driver);

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:

1. Specify the `org.apache.derby.jdbc.ClientDriver` driver in any of the following ways:
   a. Use a script.

      Run the ij.bat script on Windows systems and the ij.ksh script on UNIX systems. These scripts are located in the $DERBY_INSTALL directory.

   b. Run the ij tool from the command line.

      ```
      java -Dij.driver='org.apache.derby.jdbc.ClientDriver'
         org.apache.derby.tools.ij
      ```

   c. Use the DRIVER command.

      ```
      ij> DRIVER 'org.apache.derby.jdbc.ClientDriver';
      ```

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, make sure the Network Server is running (see Starting the Network Server), and then include the necessary Derby and Network Client driver connection attributes as part of the database URL.

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

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 that 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. Note that 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. Some SQLStates may be null when using the network client, particularly for data conversion errors.
- Multiple SQL exceptions and warnings will only return the SQLState of the first exception when using the network client. The text of the additional exceptions will be appended to the text of the first exception. See Error message differences .
- There are no localized error messages for the network client.
- The network client driver fully materializes LOBS when retrieving a row.
- Scrollable cursors (`ResultSet.TYPE_SCROLL_SENSITIVE` or `ResultSet.TYPE_SCROLL_INSENSITIVE`) are not supported using the network client if the result set contains LOB data. `TYPE_FORWARD_ONLY` must be specified for result sets containing LOB data.
- 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

The functionality of updatable resultsets in a server environment are similar to an embedded environment in Derby, with the exception of the following differences:

- The Network Client requires that there be at least one column in the select list from the target table. For example, the following statement will fail in a server environment:

  ```
  select 1, 2 from t1 for update of c11
  ```

  The Network Client driver looks at both of the columns in the select list and cannot determine the target table for update/delete by looking at the column metadata. This requirement is not necessary in an embedded environment.
- The embedded driver allows for statement name changes when there is an open resultset on the statement object. This is not supported in a server environment.

Other differences between updatable resultsets in a server or embedded environment can be found in the following table.

Table1. Comparison of updatable resultsets features in server and embedded environments

Embedded environment	Server environment
updateString on SMALLINT, INTEGER, BIGINT, DECIMAL datatypes supported.	Not supported
updateBytes on CHAR, VARCHAR, LONG VARCHAR datatypes supported.	Not supported
updateTime on TIMESTAMP datatypes supported.	Not supported
updateObject with null values supported.	Not supported
updateClob and updateBlob supported.	Not supported

Error message differences

The Network Server reports only the first error or warning message if multiple errors or warnings occur for a given statement. For example:

```
ij> create table ai (x int, y int generated always as identity
        (increment by 200000000));
ij> insert into ai (x) values (1),(2),(3),(4),(5),(6),(7),
(8),(9),(10),(11),(12),(13),(14),(15),(16),(17),(18),(19);
```

The Network Server generates the following error message and appends the exception message to the error:

```
ERROR 42Z24: Overflow occurred in identity for column 'Y' in table 'AI':
SQLSTATE: 22003: The resulting value is outside the range
for the data type INTEGER.
```

The Derby embedded driver, however, would generate two SQL exceptions:

```
ERROR 42Z24: Overflow occurred in identity for column 'Y' in table 'AI'.
```

```
ERROR 22003: The resulting value is outside the range for the data type
INTEGER.
```

This is because the network client driver reports only one SQLException or one SQLWarning per statement.

User authentication differences

When running Derby in embedded mode or when using the Derby Network Server, you can enable or disable server-side user authentication. However, when using the Network Server, the default security mechanism (`CLEAR_TEXT_PASSWORD`) 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 using this mechanism, you must specify only the user property.
- *Encrypted UserID and encrypted password (org.apache.derby.jdbc.ClientDataSource.ENCRYPTED_USER AND_PASSWORD_SECURITY)*

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

The user's name that is specified upon connection is the default schema for the connection, if a schema with that name exists. See the *Derby 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 attribute) when making the connection.

Network Server user authentication when user authentication is on in Derby:

When user authentication is enabled in Derby, you can either use the default security mechanism (user name and password) or you can specify that the security mechanism be encrypted user and password.

Network Server user authentication when user authentication is off in Derby:

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 does not have to be one 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 IBM JCE (Java Cryptography Extension) 1.2.1 or later. You can use it with any version of IBM or Sun's Java(TM) 2 Platform, Standard Edition, Version 1.2 (J2SE).

IBM Developer Kit for the Java Platform 1.4 or later comes with IBM JCE, so you do not need to install IBM JCE separately. If you have an earlier version of IBM Developer Kit for the Java Platform or other Software Development Kits, complete the following steps:

1. Copy the following IBM JCE jar files to the jre/lib/ext directory of the IBM SDK's installation home:
   - ibmjceprovider.jar
   - ibmjcefw.jar
   - ibmpkderby.jar
   - ibmpkcs11.jar
2. Modify the java.security file in the jre/lib/security directory. In the section that lists providers (and preference order), replace the text with:

   ```
   security.provider.1=sun.security.provider.Sun
   security.provider.2=com.ibm.crypto.provider.IBMJCE
   ```

   Note: If you are installing the IBM JCE on a Sun Java Development Kit, you must specify both of these lines in the order shown.

3. To use the encrypted user id and password security mechanism during JDBC connection using the network client, specify the `securityMechanism` in the connection property.

   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.

Setting port numbers

By default, Derby using the 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 scripts
   - 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 run as a stand-alone server, with Derby as an embedded part of the application.

It can also be managed remotely from a web server by using a servlet interface. You can manage the Network Server by using shell scripts, the command line, or the Network Server API. See Managing the Derby Network Server remotely by using the servlet interface for information about starting and shutting down the Network Server using the servlet interface.

Overview

You start the Derby Network Server using the command line or using the Derby 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 two constructor methods for this class:

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

- NetworkServerControl()

  This constructor method 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 method 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 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);
```

Setting Network Server properties

You can specify Network Server properties in three ways:

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

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

derby.drda.host

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, Network Server will listen on all interfaces. Ensure that you are running under the 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

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. derby. The 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

Indicates whether to log connections and disconnections.

Syntax

```
derby.drda.logConnections=[true|false]
```

Default

False.

Example

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

Static or dynamic

Dynamic. System values can be changed by using commands or the servlet interface after the Network Server has been started.

derby.drda.maxThreads

Use the derby.drda.maxThreads property to set a maximum number of connection threads that 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

1

Example

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

Static or dynamic

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

derby.drda.minThreads

Use the derby.drda.minThreads property to set the minimum number of connection threads that Network Server will allocate. By default, connection threads are allocated as needed.

Syntax

```
derby.drda.minThreads=numthreads
```

Default

1

Example

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

Static or dynamic

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

derby.drda.portNumber

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.startNetworkServer

Use the derby.drda.startNetworkServer property to simplify embedding the Network Server in your application. When you set derby.drda.startNetworkServer, the Network Server will automatically start when you start Derby. Only one Network Server can be started in a JVM.

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.timeslice

Use the derby.drda.timeslice property to set 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.

Syntax

```
derby.drda.timeslice=milliseconds
```

Default

1

Example

```
derby.drda.timeslice=2000
```

Static or dynamic

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

derby.drda.traceAll

Turns tracing on for all sessions.

Syntax

```
derby.drda.traceAll=[true|false]
```

Default

False.

Example

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

Static or dynamic

Dynamic. System values can be changed by using commands or the servlet interface after the Network Server has been started.

derby.drda.traceDirectory

Indicates the location of tracing files.

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. System values can be changed by using commands or the servlet interface 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 scripts 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 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;
        }
```

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 a Web server, and derby.system.home must be known to the Web server.

A Web application archive (WAR) file, *derby.war*, for the Derby Network Server is available in *$DERBY_INSTALL/frameworks/NetworkServer/war.* 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 `cns`, the URL of the server is:

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

Note: A servlet engine is not part of the Network Server.

The servlet takes the following optional configuration parameters:

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*.

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, boot or shut down databases. 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.)
- Test the connection. (If you choose this option, the Test connection page is displayed.)
- Specify threading parameters for Network Server. (If you choose this option, the Thread parameters page is displayed.)
- Stop the application 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 push 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 push the Set Directory button.

Set Network Server parameters

The first page is displayed if the thread parameter button is pressed. 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 parameters are 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 will only listen 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 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 host machine *sampleserver.sampledomain.com* and also listens for clients from other hosts. Clients must specify the server in the URL or DataSource as sampleserver.sampledomain.com:

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

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

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

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

In addition, 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.

Running the Network Server under the security manager

You should run the Network Server under the Java security manager. An sample security policy file is shown in the following examples. Fine tune this policy to suit your needs.

CAUTION: Opening up the server to all clients without limiting access by using a policy similar to the one in the following example is a severe security risk.

```
//Recommended set of permissions to start and use the Network Server,
//assuming the 'd:/derby/lib' directory has been secured.
//Fine tune based on your environment settings
grant codeBase "file:d:/derby/lib/-" {
permission java.io.FilePermission "${derby.system.home}${/}-",
        "read, write, delete";
permission java.io.FilePermission "${user.dir}${/}-", "read, write,
delete";
permission java.util.PropertyPermission "derby.*", "read";
permission java.util.PropertyPermission "user.dir", "read";
permission java.lang.RuntimePermission "createClassLoader";
permission java.net.SocketPermission "myclientmachine", "accept";
};

//Required set of permissions to stop the Network Server, assuming you
have
// secured the 'd:/derby/lib' directory
//Remember to fine tune this as per your environment.
grant codeBase "file:d:/derby/lib/-"  {
```

```
//Following is required when server is started with "-h localhost"
//or without the -h option
permission java.net.SocketPermission "localhost", "accept, connect,
resolve";
permission java.net.SocketPermission "127.0.0.1", "accept, connect,
resolve";
//The following is only required if the server is started with the -h
<host>
//option (else shutdown access will be denied).
permission java.net.SocketPermission "<host>:*", "accept, connect,
resolve";
};
```

- The following example shows how to start the Network Server in the default security manager (listening to clients from `localhost` only, which is the default behavior if the `-h` option is not used to start the server). This example assumes that the policy file exists in `d:/nwsvr.policy`.

  ```
  java -Djava.security.manager -Djava.security.policy=d:/nwsvr.policy
  org.apache.derby.drda.NetworkServerControl start
  ```

- You can also achieve the same behavior by using the `-h` option when starting the server as shown in the following example:

  ```
  java -Djava.security.manager -Djava.security.policy=d:/nwsvr.policy
  org.apache.derby.drda.NetworkServerControl start -h localhost
  ```

- The following example shows how to start the Network Server (assuming that you start the server on the host machine *myserver*) in the default security manager (listening to client requests originating from other machines only). This example assumes that the policy file exists in `d:/nwsvr.policy`.

  ```
  java -Djava.security.manager -Djava.security.policy=d:/nwsvr.policy
  org.apache.derby.drda.NetworkServerControl start -h myserver
  ```

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 or by using the servlet interface.

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:

  ```
  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:

```
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).

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 and disconnections when the property `derby.drda.logConnections` is set to `true`. 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`.

- To turn on connection and disconnection logging, you can use the servlet interface or you can issue the following command:

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

- To turn connection logging off you can use the servlet interface or you can issue the following command:

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

See the *Derby 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.

See Managing the Derby Network Server remotely by using the servlet interface for information about managing the trace facility using the servlet interface.

Turning on the trace facility

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

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

   Alternatively, while the Network Server is running, you can use the following command 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>
```

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.

3. While the Network Server is running, enter the following command to set the trace directory:

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

Turning off the trace facility

Enter the following command to turn off tracing:

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

The tracing files are named ServerX.trace, where X is a connection number.

Derby Network Server sample programs

This section describes several Derby Network Server 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.
- Creates the *NsSampledb* database if not already 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.

You must install the following files in the `%DERBY_INSTALL%\demo\nserverdemo\` directory before you can run the sample program:

- `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 shutdown 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:

1. Open a command prompt and change directories to the %DERBY_INSTALL%\demo\ directory, where %DERBY_INSTALL% is the directory where you installed Derby.
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 that shown in the following table:

```
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 directories and files:

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.

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

To run the sample program, install the following files in the `%DERBY_INSTALL%\demo\nserverdemo\` directory:

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

Running the SimpleNetworkServerSample program

To run the Derby Network Server sample program:

1. Open a command prompt and change directories to the %DERBY_INSTALL%\demo\nserverdemo directory, where %DERBY_INSTALL% is the directory where you installed Derby.
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 exampleS:

   ```
   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 directories and files:

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:

- Loads the network client driver
- Obtains a client connection by using the DriverManager
- Obtains a client connection by using a DataSource
- Tests the database connections by running a sample query
- Closes the connections and then exits the program

You must install the following files in the `%DERBY_INSTALL%\demo\nserverdemo\` directory before you can run the sample program:

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

Running the SimpleNetworkClientSample program

To connect to the Network Server that has been started with the SimpleNetworkServerSample program:

1. Open a command prompt and change directories to the%DERBY_INSTALL%\demo\nserverdemo directory, where %DERBY_INSTALL% is the directory where you installed Derby.
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 SimpleNetworkClientSample 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.

Checking database consistency

If you experience hardware or operating system failure, you can use the SYSCS_UTIL.SYSCS_CHECK_TABLE function to verify that the database is still consistent.

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.

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 online. You can also restore a full backup from a specified location.

While the backup is in progress, update operations are temporarily blocked, but read operations can still proceed.

Backing up a database

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. You must shut down the database prior to performing 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\2005-06-01`:

```
xcopy d:\mydatabases\sample c:\mybackups\2005-06-01\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.

Online backups

Use online backups to back up a database while it is running. During the interval that the backup is running, the database can be read, but writes to the database are blocked.

You can perform online backups by using the backup procedure or by using operating systems commands with the freeze and unfreeze system procedures.

Using the backup procedure to perform an online backup:

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE procedure locks the database so that any connection trying to write to the database will be frozen until the backup completes. Database reads can continue while the backup is running.

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE procedure takes 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/2005-06-01` 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/2005-06-01')
```

The *SYSCS_UTIL.SYSCS_BACKUP_DATABASE()* procedure puts the database into a state in which it can be safely copied, 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.

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:/mybacksup/" +
   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 2005-06-01, the previous commands copy the current database to a directory of the same name in *c:/mybackups/2005-06-01*.

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.

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 Derby uses the IBM Application Developer Kit file-copying routines, 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

Note: Read Logging on a separate device to find out about the default location of the database log.

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\2005-06-01\sample /s /i
xcopy h:\janet\tourslog\log c:\mybackups\2005-06-01\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 *service.properties* 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 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
```

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
```

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 the calls in the previous example 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 *deleteOnlineArchivedLogFiles* 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.

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:

By using the full backup copy, archived logs, and active logs, you can restore a database to its most recent state by performing roll-forward recovery. You perform a roll-forward recovery by specifying a connection URL attribute *rollForwardRecoveryFrom=<BackupPath>* at boot time. This brings the database to its most recent state by using full backup copy, archived logs, and active logs. All the log files should be in the database log path directory.

Backing up a database:

In the following example, a database named wombat is backed up to the d:/backup directory with log archive mode enabled:

```
connect 'jdbc:derby:wombat;create=true';

create table t1(a int not null primary key);
------------------DML/DDL Operations
CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE
('d:/backup', 0);
insert into t1 values(19);
create table t2(a int);
-----------------DML/DDL Operations
-----------------Database Crashed (Media Corruption on data disks)
```

Restoring a database using roll-forward recovery:

In the following example, the database is restored using roll-forward recovery after a media failure:

```
connect 'jdbc:derby:wombat;rollForwardRecoveryFrom=d:/backup/wombat';
select * from t1;
---------------DML/DDL Operations
```

The following attribute can be specified in the JDBC boot time connection URL:

rollForwardRecoveryFrom=<Path>

For more information, see the rollForwardRecoveryFrom=<Path> section in the *Derby Reference Manual* .

After a database is restored from full backup, transactions from the online archived logs and active logs are replayed.

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* 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 attribute

To specify a non-default location for the log directory, set the logDevice attribute on the database connection URL when you create the database.

This attribute is meaningful only when you are creating a database. You can specify `logDevice` as either an absolute path or as a path that is relative to the directory where the JVM is executed.

Setting logDevice on the database connection URL 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` attribute as a database property by using the following statement:

```
VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('logDevice')
```

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:

```
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 NT 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.

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 how to set properties, and information about the specific properties that are mentioned in this topic, see *Tuning Derby* .

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 the section that discusses controlling Derby application behavior in the *Derby 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_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 in committed. SYSCS_INPLACE_COMPRESS 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_COMPRESS_TABLE, the SYSCS_INPLACE_COMPRESS procedure uses no temporary files and moves rows around within the same conglomerate.

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_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 *Derby 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.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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.