Derby Developer's Guide

Version 10.14

Derby Document build:
April 6, 2018, 6:13:39 PM (PDT)

Contents

Copyright

Related information

License

License

The Apache License, Version 2.0

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

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

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

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

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

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

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

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

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

   You may add Your own copyright statement to Your modifications

```
   and may provide additional or different license terms and
   conditions for use, reproduction, or distribution of Your
   modifications, or for any such Derivative Works as a whole,
   provided Your use, reproduction, and distribution of the Work
   otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state
   otherwise, any Contribution intentionally submitted for
   inclusion in the Work by You to the Licensor shall be under the
   terms and conditions of this License, without any additional
   terms or conditions.  Notwithstanding the above, nothing herein
   shall supersede or modify the terms of any separate license
   agreement you may have executed with Licensor regarding such
   Contributions.

6. Trademarks. This License does not grant permission to use the
   trade names, trademarks, service marks, or product names of the
   Licensor, except as required for reasonable and customary use
   in describing the origin of the Work and reproducing the content
   of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or
   conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or
   FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for
   determining the appropriateness of using or redistributing the
   Work and assume any risks associated with Your exercise of
   permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and
   grossly negligent acts) or agreed to in writing, shall any
   Contributor be liable to You for damages, including any direct,
   indirect, special, incidental, or consequential damages of any
   character arising as a result of this License or out of the use
   or inability to use the Work (including but not limited to
   damages for loss of goodwill, work stoppage, computer failure or
   malfunction, or any and all other commercial damages or losses),
   even if such Contributor has been advised of the possibility of
   such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by
   reason of your accepting any such warranty or additional
   liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.
```

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

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

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

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

About this guide

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

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

Purpose of this guide

This guide explains how to use the core Derby technology and is for developers building Derby applications.

It describes basic Derby concepts, such as how you create and access Derby databases through JDBC routines and how you can deploy Derby applications.

When an application embeds Derby, application users take on the role of database administrator and must maintain the integrity of the database. See "Part Two: Derby Administration Guide" in the *Derby Server and Administration Guide* for information on administrative tasks such as backing up and restoring databases. In particular, see the topic "Maintaining database integrity" for information on preventing database corruption. You will need to make this information available to users of your application.

Audience

This guide is intended for software developers who already know some SQL and Java.

Derby users who are not familiar with the SQL standard or the Java programming language will benefit from consulting books on those subjects.

How this guide is organized

This document includes the following sections.

- *After installing*

  Explains the installation layout.
- *Upgrades*

  Explains how to upgrade a database created with a previous version of Derby.
- *JDBC applications and Derby basics*

  Basic details for using Derby, including loading the JDBC driver, specifying a database URL, starting Derby, and working with Derby properties.
- *Deploying Derby applications*

  An overview of different deployment scenarios, and tips for getting the details right when deploying applications.
- *Derby server-side programming*

  Describes how to program database-side JDBC routines, triggers, and table functions.
- *Controlling Derby application behavior*

  JDBC, cursors, locking and isolation levels, and multiple connections.
- *Using Derby as a Java EE resource manager*

Information for programmers developing back-end components in a J2EE system.

- *Developing tools and using Derby with an IDE*

  Tips for tool designers.
- *SQL tips*

  Insiders' tricks of the trade for using SQL.
- *Localizing Derby*

  An overview of database localization.
- *Derby and standards*

  Describes those parts of Derby that are non-standard or not typical for a database system.

After installing

This section provides reference information about the installation directory, JVMs, classpath, upgrades, and platform-specific issues.

Review the `index.html` file at the top level of the Derby distribution for pointers to reference and tutorial information about Derby. See the Release Notes for information on platform support, changes that may affect your existing applications, defect information, and recent documentation updates. See *Getting Started with Derby* for basic product descriptions, information on getting started, and directions for setting the path and the classpath.

The installation directory

You may install the Derby software in a directory of your choice.

See the `index.html` file for pointers to information on Derby.

The distribution includes setup scripts that use an environment variable called DERBY_HOME. The variable's value is set to the Derby base directory.

```
C:>echo %DERBY_HOME%
C:\DERBY_HOME
```

If you want to set your own environment, *Getting Started with Derby* instructs you on setting its value to the directory in which you installed the Derby software.

The distribution for Derby contains all the files you need, including the documentation set, some example applications, and a sample database.

Details about the installation:

- *index.html* in the top-level directory is the top page for the on-line documentation.
- *RELEASE-NOTES.html*, in the top-level Derby base directory, contains important last-minute information. *Read it first*.
- */bin* contains utilities and scripts for running Derby.
- */demo* contains some sample applications, useful scripts, and prebuilt databases.
  - */databases* includes prebuilt sample databases.
  - */programs* includes sample applications.
- */docs* contains the on-line documentation (including this document).
- */javadoc* contains the documented APIs for the public classes and interfaces. Typically, you use the JDBC interface to interact with Derby; however, you can use some of these additional classes in certain situations.
- */lib* contains the Derby libraries.

Batch files and shell scripts

The */bin* directory contains scripts for running some of the Derby tools and utilities. To customize your environment, put the directory first in your path.

These scripts serve as examples to help you get started with these tools and utilities on any platform. However, they may require modification in order to run properly on certain platforms.

Derby and Java Virtual Machines (JVMs)

Derby is a database engine written completely in the Java programming language; it will run in any JVM that is version 6 or higher.

Derby libraries and classpath

Derby libraries are located in the */lib* subdirectory of the Derby base directory. You must set the classpath on your development machine to include the appropriate libraries.

Getting Started with Derby explains how to set the classpath in a development environment.

UNIX-specific issues

This section discusses Derby issues specifically related to UNIX platforms.

Configuring file descriptors

Derby databases create one file per table or index. Some operating systems limit the number of files an application can open at one time.

If the default is a low number, such as 64, you might run into unexpected *IOExceptions* (wrapped in *`SQLExceptions`*). If your operating system lets you configure the number of file descriptors, set this number to a higher value.

Scripts

Your installation contains executable script files that simplify invoking the Derby tools. On UNIX systems, these files might need to have their default protections set to include execute privilege.

A typical way to do this is with the command *chmod +x *.ksh*.

Consult the documentation for your operating system for system-specific details.

Upgrades

To connect to a database created with a previous version of Derby, you must first upgrade that database.

Upgrading involves writing changes to the system tables, so it is not possible for databases on read-only media. The upgrade process:

- marks the database as upgraded to the current release (Version 10.14).
- allows use of new features.

See the release notes for more information on upgrading your databases to this version of Derby.

Preparing to upgrade

Upgrading your database occurs the first time the new Derby software connects to the old database.

Before you connect to the database using the new software:

1. Back up your database to a safe location using Derby online/offline backup procedures.

   For more information on backup, see the *Derby Server and Administration Guide*.
2. Update your CLASSPATH with the latest jar files.
3. Make sure that there are no older versions of the Derby jar files in your CLASSPATH. You can determine if you have multiple versions of Derby in your CLASSPATH by using the sysinfo tool.

   To use the `sysinfo` tool, execute the following command:

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

   The `sysinfo` tool uses information found in the Derby jar files to determine the version of any Derby jar in your CLASSPATH. Be sure that you have only one version of the Derby jar files specified in your CLASSPATH.

Upgrading a database

To upgrade a database, you must explicitly request an upgrade the first time you connect to the database with the new version of Derby.

Ensure that you complete the prerequisite steps before you upgrade:

- Back up your database before you upgrade.
- Ensure that only the new Derby jar files are in your CLASSPATH.

When you upgrade the database, you can perform a full upgrade or soft upgrade:

- A full upgrade is a complete upgrade of the Derby database. When you perform a full upgrade, you cannot connect to the database with an older version of Derby and you cannot revert back to the previous version. Elsewhere in the documentation, when the term "upgrade" is used without any qualification, it means a full upgrade.
- A soft upgrade allows you to run a newer version of Derby against an existing database without having to fully upgrade the database. This means that you can continue to run an older version of Derby against the database. However, if you perform a soft upgrade, certain features will not be available to you until you perform a full upgrade. Specifically, new features that affect the structure of a

database are not available with a soft upgrade. For a list of the new features in a release, see the Release Notes for that release.

1. To upgrade the database, select the type of upgrade that you want to perform. The following table shows the upgrade types. In both examples, `sample` is a database from a previous version of Derby.

   Table 1. Upgrade types

Type of Upgrade	Action
Full upgrade	Connect to the database using the `upgrade=true` database connection URL attribute. For example: `jdbc:derby:sample;upgrade=true` See "upgrade=true attribute" in the *Derby Reference Manual* for more information about this attribute.
Soft upgrade	Connect to the database. For example: `connect 'jdbc:derby:sample'`

JDBC applications and Derby basics

This section describes the core Derby functionality. In addition, it details the most basic Derby deployment, Derby embedded in a Java application.

Application development overview

Derby application developers use the Java Database Connectivity (JDBC) API, the application programming interface that makes it possible to access relational databases from Java programs.

The JDBC API is part of the Java Platform, Standard Edition and is not specific to Derby. It consists of the *java.sql* and *javax.sql* packages, which is a set of classes and interfaces that make it possible to access databases (from a number of different vendors, not just Derby) from a Java application.

To develop Derby applications successfully, you will need to learn the JDBC API. This section does not teach you how to program with the JDBC API.

This section covers the details of application programming that are specific to Derby applications. For example, all JDBC applications typically start their DBMS's JDBC driver and use a connection URL to connect to a database. This section gives you the details of how to start Derby's JDBC driver and how to work with Derby's connection URL to accomplish various tasks. It also covers essential Derby concepts such as the Derby system.

You will find reference information about the particulars of Derby's implementation of the JDBC API in the *Derby Reference Manual*.

Derby application developers will need to learn SQL. *SQL* is the standard query language used with relational databases and is not tied to a particular programming language. No matter how a particular RDBMS has been implemented, the user can design databases and insert, modify, and retrieve data using the standard SQL statements and well-defined data types.

SQL is standardized by ANSI and ISO; Derby supports entry-level SQL as well as some higher-level features. Entry-level SQL is a subset of the full SQL specified by ANSI and ISO that is supported by nearly all major DBMSs today.

This section does not teach you SQL. You will find reference information about the particulars of Derby's implementation of SQL in the *Derby Reference Manual*.

Derby implements the JDBC API so as to allow Derby to serve as a resource manager in a Java EE compliant system.

When an application embeds Derby, application users take on the role of database administrator and must maintain the integrity of the database. See "Part Two: Derby Administration Guide" in the *Derby Server and Administration Guide* for information on administrative tasks such as backing up and restoring databases. In particular, see the topic "Maintaining database integrity" for information on preventing database corruption. You will need to make this information available to your users.

Derby embedded basics

This section explains how to use and configure Derby in an embedded environment.

Included in the installation is a sample application program, */demo/programs/simple*, which illustrates how to run Derby embedded in the calling program.

Derby JDBC driver

Derby consists of both the database engine and an embedded JDBC driver. Applications use JDBC to interact with a database.

In an embedded environment, the embedded driver is initially loaded and registered when the *java.sql.DriverManager* class is initialized. That typically happens on the first call to a *DriverManager* method such as *DriverManager.getConnection*, as described in Derby JDBC database connection URL. Loading the driver also starts Derby.

The Derby driver class name for the embedded environment is *org.apache.derby.jdbc.EmbeddedDriver*.

For detailed information about loading the Derby JDBC driver, see "java.sql.Driver interface" in the *Derby Reference Manual*.

If your application shuts down Derby or calls the *DriverManager.deregisterDriver* method, and you then want to reload the driver, call the *Class.forName().newInstance()* method. See Shutting down the system for more information.

Derby JDBC database connection URL

A Java application using the JDBC API establishes a connection to a database by obtaining a *Connection* object.

The standard way to obtain a *Connection* object is to call the method *DriverManager.getConnection*, which takes a *String* containing a connection URL (uniform resource locator). A JDBC connection URL provides a way of identifying a database. It also allows you to perform a number of high-level tasks, such as creating a database or shutting down the system.

The following example shows the use of the connection URL:

```
Connection conn = DriverManager.getConnection("jdbc:derby:sample");
```

An application in an embedded environment uses a different connection URL from that used by applications using the Derby Network Server in a client/server environment. See "Accessing the Network Server by using the network client driver" in the *Derby Server and Administration Guide* for more information.

However, all versions of the connection URL (which you can use for tasks besides connecting to a database) have common features:

- You can specify the name of the database you want to connect to.
- You can specify a number of attributes and values that allow you to accomplish tasks.

For more information about what you can specify with the Derby connection URL, see Connecting to databases. For details on the connection URL syntax, see "Syntax of database connection URLs for applications with embedded databases" in the *Derby Reference Manual*. For detailed reference information on connection URL attributes and values, see "Setting attributes for the database connection URL" in the *Derby Reference Manual*.

Derby system

A Derby database exists within a *system*.

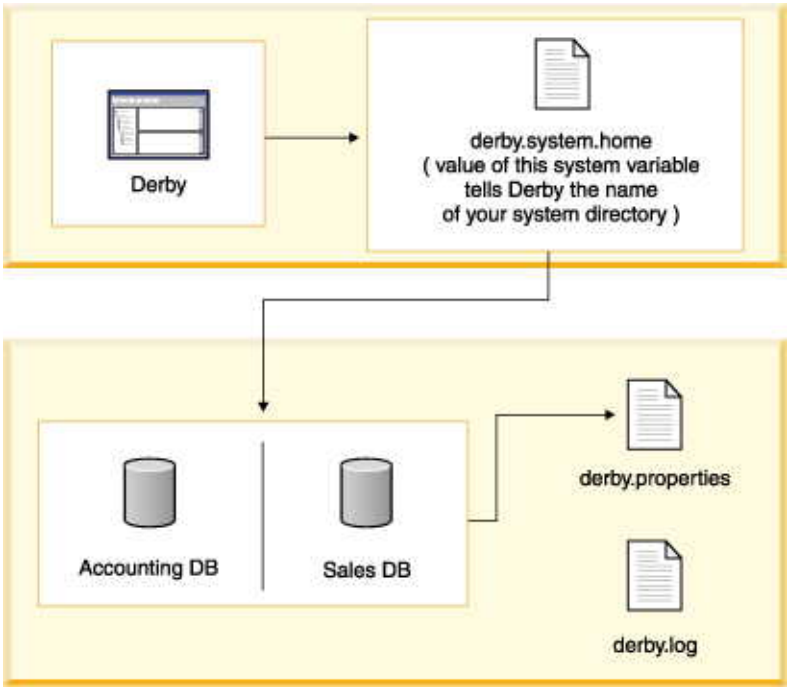
A Derby system is a single instance of the Derby database engine and the environment in which it runs. It consists of a system directory, zero or more databases, and a

system-wide configuration. The system directory contains any persistent system-wide configuration parameters, or properties, specific to that system in a properties file called derby.properties. This file is not automatically created; you must create it yourself.

The Derby system is not persistent; you must specify the location of the system directory at every startup.

However, the Derby system and the system directory is an essential part of a running database or databases. Understanding the Derby system is essential to successful development and deployment of Derby applications. As the following figure shows, Derby databases live in a system, which includes system-wide properties, an error log, and one or more databases.

Figure 1. Derby system



The system directory can also contain an error log file called *derby.log* (see The error log).

Each database within that system is contained in a subdirectory, which has the same name as the database (see A Derby database).

In addition, if you connect to a database outside the current system, it automatically becomes part of the current system.

When you use the embedded driver, Derby database files and log files normally have whatever default permissions you specify for your file system. If you are running with Java SE 7 or later, however, you can enhance security by restricting file access to the user who creates the database. To do this, set the system property *derby.storage.useDefaultFilePermissions* to false. See the *Derby Reference Manual* for details.

Note: In-memory databases do not appear in the system directory.

One Derby instance for each Java Virtual Machine (JVM)

You could potentially have two instances of a Derby system running on the same machine at the same time. Each instance must run in a different Java Virtual Machine (JVM).

If you use the embedded driver, two separate instances of Derby cannot access the same database. If a Derby instance attempts to access a running database, an error message appears, and a stack trace appears in the *derby.log* file. If you want more than one Derby instance to be able to access the same database, you can use the Network Server.

If a Derby instance uses the in-memory database capability for its database connection, the database exists only within the JVM of that Derby instance. Another Derby instance could refer to the same database name, but it would not be referring to the same actual database, and no error would result.

Booting databases

The default configuration for Derby is to *boot* (or start) a database when an application first makes a connection to it. When Derby boots a database, it checks to see if recovery needs to be run on the database, so in some unusual cases booting can take some time.

You can also configure your system to automatically boot all databases in the system when it starts up; see "*derby.system.bootAll*" in the *Derby Reference Manual*. Because of the time needed to boot a database, the number of databases in the system directory affects startup performance if you use that configuration.

Once a database has been booted within a Derby system, it remains active until the Derby system has been shut down or until you shut down the database individually.

When Derby boots a database, a message is added to the log file. The message includes the Derby version that the database was booted with, along with information about the Java version, the user's working directory, and the location of the Derby system directory, if the user specified it using the *derby.system.home* property. If *derby.system.home* was not specified, its value is reported as null, as in the following example:

```
Thu Sep 13 09:52:15 EDT 2012:
 Booting Derby version The Apache Software Foundation - Apache Derby
-  10.10.0.0 - (1384314): instance a816c00e-0139-bfe6-bff8-000000a155b8
on database directory C:\sampledb with class loader
sun.misc.Launcher$AppClassLoader@9931f5
Loaded from file:C:\db-derby-10.10.0.0-bin\lib\derby.jar
java.vendor=Oracle Corporation
java.runtime.version=1.7.0_07-b11
user.dir=C:\
os.name=Windows XP
os.arch=x86
os.version=5.1
derby.system.home=null
Database Class Loader started - derby.database.classpath=''
```

The number of databases running in a Derby system is limited only by the amount of memory available in the JVM.

Shutting down the system

In an embedded environment, when an application shuts down, it should first shut down Derby.

If the application that started the embedded Derby quits but leaves the Java Virtual Machine (JVM) running, Derby continues to run and is available for database connections.

In an embedded system, the application shuts down the Derby system by issuing the following JDBC call:

```
DriverManager.getConnection("jdbc:derby:;shutdown=true");
```

Shutdown commands always raise *SQLExceptions*.

When a Derby system shuts down, a message goes to the log file:

```
----------------------------------------------------------------
Thu Sep 13 09:53:21 EDT 2012: Shutting down Derby engine
----------------------------------------------------------------
Thu Sep 13 09:53:21 EDT 2012:
Shutting down instance a816c00e-0139-bfe6-bff8-000000a155b8 on
database directory C:\sampledb with class loader
sun.misc.Launcher$AppClassLoader@9931f5
----------------------------------------------------------------
```

If you are running with a security manager on JDK 8 or higher, you must grant Derby permission to deregister the embedded driver in order to fully shut down the system. See "Configuring Java security" in the *Derby Security Guide* for details.

Typically, an application using an embedded Derby engine shuts down Derby just before shutting itself down. However, an application can shut down Derby and later restart it in the same JVM session. To restart Derby successfully, the application needs to reload *org.apache.derby.jdbc.EmbeddedDriver* explicitly, as follows:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
```

This is an exception to the rule that you do not need to load the driver explicitly when starting Derby.

The JDBC specification does not recommend calling `newInstance()`, but adding a `newInstance()` call guarantees that Derby will be booted on any JVM.

Note: If your application will need to restart Derby, you can add the attribute *deregister=false* to the connection URL to avoid having to reload the embedded driver:

```
DriverManager.getConnection("jdbc:derby:;shutdown=true;deregister=false");
```

It is also possible to shut down a single database instead of the entire Derby system. See Shutting down Derby or an individual database. You can reboot a database in the same Derby session after shutting it down.

Defining the system directory

You define the system directory when Derby starts up by specifying a Java *system property* called *derby.system.home*.

If you do not specify the system directory when starting up Derby, the current directory becomes the system directory.

Derby uses the *derby.system.home* property to determine which directory is its system directory - and thus what databases are in its system, where to create new databases, and what configuration parameters to use. See the *Derby Reference Manual* for more information on this property.

If you specify a system directory at startup that does not exist, Derby creates this new directory - and thus a new system with no databases-automatically.

The error log

Once you create or connect to a database within a system, Derby begins outputting information and error messages to the error log. Typically, Derby writes this information to a file called *derby.log* in the system directory.

Alternatively, you can have Derby send messages to a stream, using the *derby.stream.error.method* or *derby.stream.error.field* property, or to a different file, using the *derby.stream.error.file* property. If you use any of these properties, the property setting will appear in the log.

By default, Derby overwrites *derby.log* when you start the system. You can configure Derby to append to the log with the *derby.infolog.append* property.

For information on setting all of these properties, see the *Derby Reference Manual*.

derby.properties

The text file *derby.properties* contains the definition of properties, or configuration parameters that are valid for the entire system.

The *derby.properties* file is not automatically created. If you want to set Derby properties with this file, you need to create the file yourself. The *derby.properties* file should be in the format created by the *java.util.Properties.save* method. For more information about properties and the *derby.properties* file, see Working with Derby properties and the *Derby Reference Manual*.

Double-booting system behavior

Derby prevents two instances of itself from booting the same database by using a file called *db.lck* inside the database directory.

If a second instance of Derby attempts to boot an already running database, the following error messages appear:

```
ERROR XJ040: Failed to start database 'firstdb', see the next exception
for details.
ERROR XSDB6: Another instance of Derby may have already booted the
database /home/myself/DERBYTUTOR/firstdb.
```

In addition, a stack trace appears in the *derby.log* file. For help diagnosing a double boot problem, use the *derby.stream.error.logBootTrace* property to obtain information about both successful and unsuccessful boot attempts. The property is described in the *Derby Reference Manual*.

If Derby is not able to create the *db.lck* file when booting a database, the database will be booted in read-only mode. This may happen due to lack of disk space or access rights for the database directory. The boot message in the *derby.log* will state that the database has been booted in READ ONLY mode. See also Creating Derby databases for read-only use.

If you need to access a single database from more than one Java Virtual Machine (JVM), you will need to put a server solution in place. You can allow applications from multiple JVMs that need to access that database to connect to the server. The Derby Network Server is provided as a server solution. For basic information on starting and using the Network Server, see *Getting Started with Derby*. See the *Derby Server and Administration Guide* for more information on the Network Server.

Recommended practices

When developing Derby applications, create a single directory to hold your database or databases.

Give this directory a unique name, to help you remember that:

- All databases exist within a system.
- System-wide properties affect the entire system, and persistent system-wide properties live in the system directory.
- You can boot all the databases in the system, and the boot-up times of all databases affect the performance of the system.
- You can preboot databases only if they are within the system. (Databases do not necessarily have to live inside the system *directory*, but keeping your databases there is the recommended practice.)

- Once you connect to a database, it is part of the current system and thus inherits all system-wide properties.
- Only one instance of Derby can run in a JVM at a single time.
- The error log is located inside the system directory.

A Derby database

A Derby database contains dictionary objects such as tables, columns, indexes, and jar files. A Derby database can also store its own configuration information.

The database directory

A Derby database is stored in files that live in a directory of the same name as the database. Database directories typically live in *system* directories.

Note: An in-memory database does not use the file system, but the size limits listed in the table later in this topic still apply. For some limits, the maximum value is determined by the available main memory instead of the available disk space and file system limitations.

A database directory contains the following, as shown in the following figure.

- *log* directory

  Contains files that make up the database transaction log, used internally for data recovery (not the same thing as the error log).
- *seg0* directory

  Contains one file for each user table, system table, and index (known as conglomerates).
- *service.properties* file

  A text file with internal configuration information.
- *tmp* directory

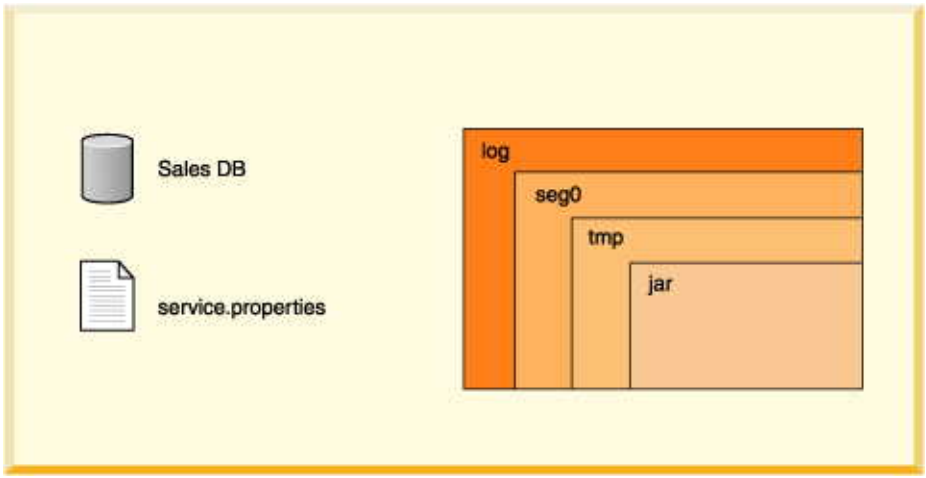
  (might not exist.) A temporary directory used by Derby for large sorts and deferred updates and deletes. Sorts are used by a variety of SQL statements. For databases on read-only media, you might need to set a property to change the location of this directory. See "Creating Derby Databases for Read-Only Use".
- *jar* directory

  (might not exist.) A directory in which jar files are stored when you use database class loading.

Read-only database directories can be archived (and compressed, if desired) into jar or zip files. For more information, see Accessing a read-only database in a zip/jar file.

The following figure shows the files and directories in the Derby database directories that are used by the Derby software.

Figure 2. An example of a Derby database directory and file structure



Derby imposes relatively few limitations on the number and size of databases and database objects. The following table shows some size limitations of Derby databases and database objects.

Table 2. Size limits for Derby database objects

Type of Object	Limit
Tables in each database	*java.lang.Long.MAX_VALUE* Some operating systems impose a limit to the number of files allowed in a single directory.
Indexes in each table	32,767 or storage
Columns in each table	1,012
Number of columns on an index key	16
Rows in each table	No limit.
Size of table	No limit. Some operating systems impose a limit on the size of a single file.
Size of row	No limit. Rows can span pages. Rows cannot span tables so some operating systems impose a limit on the size of a single file, which results in limiting the size of a table and size of a row in that table.

For a complete list of restrictions on Derby databases and database objects, see the *Derby Reference Manual*.

Creating, dropping, and backing up databases

You create new databases and access existing ones by specifying attributes to the Derby connection URL.

If you use an in-memory database, you can use a connection URL attribute to drop it. For a file system database, however, there is no drop attribute. To drop a database on the file system, delete the database directory with operating system commands. The database must not be booted when you remove a database. You can get a list of booted databases with *getPropertyInfo*.

To back up a database, you can use the online backup utility. For information on this utility, see the *Derby Server and Administration Guide*.

You can also use *roll-forward recovery* to recover a damaged database. Derby accomplishes roll-forward recovery by using a full backup copy of the database, archived logs, and active logs from the most recent time before a failure. For more information on roll-forward recovery see the *Derby Server and Administration Guide*.

Single database shutdown

An application can shut down a single database within a Derby system and leave the rest of the system running.

Storage and recovery

A Derby database (unless it is an in-memory database) provides persistent storage and recovery. Derby ensures that all committed transactions are durable, even if the system fails, through the use of a database transaction log.

Whereas inserts, updates, and deletes may be cached before being written to disk, log entries tracking all those changes are never cached but always forced to disk when a transaction commits. If the system or operating system fails unexpectedly, when Derby next starts up it can use the log to perform recovery, recovering the "lost" transactions from the log and rolling back uncommitted transactions. *Recovery* ensures that all committed transactions at the time the system failed are applied to the database, and all transactions that were active are rolled back. Thus the databases are left in a consistent, valid state.

In normal operation, Derby keeps the log small through periodic checkpoints. Checkpointing marks the portions of the log that are no longer useful, writes changed pages to disk, then truncates the log.

Derby checkpoints the log file as it fills. It also checkpoints the log when a shutdown command is issued. Shutting down the JVM in which Derby is running without issuing the proper shutdown command is equivalent to a system failure from Derby's point of view.

Booting a database means that Derby checks to see if recovery needs to be run on a database. Recovery can be costly, so using the proper shutdown command improves connection or startup performance.

Log on separate device

You can put a database's log on a separate device when you create it.

For more information, see the *Derby Server and Administration Guide*.

Database pages

Derby tables and indexes, known as conglomerates, consist of two or more pages.

A page is a unit of storage whose size is configurable on a system-wide, database-wide, or conglomerate-specific basis. By default, a conglomerate grows one page at a time until eight pages of user data (or nine pages of total disk use, which includes one page of internal information) have been allocated. (You can configure this behavior; see "*derby.storage.initialPages*" in the *Derby Reference Manual*.) After that, it grows eight pages at a time.

The size of a row or column is not limited by the page size. Rows or columns that are longer than the table's page size are automatically wrapped to overflow pages.

Database-wide properties

You can set many Derby properties as database-level properties. When set in this way, they are stored in the database and "travel" with the database unless overridden by a system property.

For more information, see Scope of properties and Setting database-wide properties.

Derby database limitations

Derby databases have a few limitations.

Indexes

Indexes are not supported for columns defined on CLOB, BLOB, LONG VARCHAR, and XML data types.

If the length of the key columns in an index is larger than half the page size of the index, creating an index on those key columns for the table fails. For existing indexes, an insert of new rows for which the key columns are larger than half of the index page size causes the insert to fail.

Avoid creating indexes on long columns. Create indexes on small columns that provide a quick look-up to larger, unwieldy data in the row. You might not see performance improvements if you index long columns. For information about indexes, see *Tuning Derby*.

System shutdowns

The system shuts down if the database log cannot allocate more disk space.

A "LogFull" error or some sort of `IOException` occurs in the `derby.log` file when the system runs out of space. If the system has no more disk space to append to the `derby.log` file, you might not see the error messages.

Connecting to databases

You connect to a database using a form of the Derby connection URL as an argument to the *DriverManager.getConnection* call.

For details on the syntax of the connection URL, see "Syntax of database connection URLs for applications with embedded databases" in the *Derby Reference Manual*.

You specify a path to the database within this connection URL.

Connecting to databases within the system

The standard way to access databases in the file system is by specifying the path name of the database. The path name can be either an absolute path name or a path name relative to the system directory. In a client/server environment, this path name is always on the *server* machine.

By default, you can connect to databases within the current system directory (see Defining the system directory). To connect to databases within the current system directory, just specify the base name of the database on the connection URL. For example, if your system directory contains a database called *myDB*, you can connect to that database with the following connection URL:

```
jdbc:derby:myDB
```

The full method call within a Java program would be:

```
Connection conn = DriverManager.getConnection("jdbc:derby:myDB");
```

Connecting to databases outside the system directory

You can also connect to databases in other directories (including subdirectories of the system directory) by specifying a relative or absolute path name to identify the database. The way you specify an absolute path is defined by the host operating system.

Using the connection URL as described here, you can connect to databases in more than one directory at a time.

Two examples:

```
jdbc:derby:../otherDirectory/myDB

jdbc:derby:c:/otherDirectory/myDB
```

Note: Once connected, such a database becomes a part of the Derby system, even though it is not in the system directory. This means that it takes on the system-wide properties of the system and no other instance of Derby should access that database. It is recommended that you connect to databases only in the system directory.

Conventions for specifying the database path name

When you access databases from the file system (instead of from the classpath or a jar file), any path name that is not absolute is interpreted as relative to the system directory.

The path name must do one of the following:

- Refer to a previously created Derby database
- Specify the *create=true* attribute

The path separator in the path name is a forward slash (`/`), even in Windows path names. The path name cannot contain a colon (`:`), except for the colon after the drive name in a Windows path name. See "Syntax of database connection URLs for applications with embedded databases" in the *Derby Reference Manual* for the full syntax.

You can specify only databases that are local to the machine on which the JVM is running. NFS file systems on UNIX and remote shared files on Windows (*//machine/directory*) are not guaranteed to work. Using *derby.system.home* and forward slashes is recommended practice for platform independent applications.

If two different database path name values, relative or absolute, refer to the same actual directory, they are considered equivalent. This means that connections to a database through its absolute path name and its relative path name are connections to the same database. Within Derby, the name of the database is defined by the canonical path of its directory from *java.io.File.getCanonicalPath*.

Derby automatically creates any intermediate directory that does not already exist when creating a new database. If it cannot create the intermediate directory, the database creation fails.

Special database access

You can also access databases from the classpath or from a jar file (in the classpath or not) as read-only databases.

You can create in-memory databases for use in testing and development and for processing temporary or reproducible data. See Using in-memory databases for details.

Accessing databases from the classpath:

In most cases, you access databases from the file system. However, it is also possible to access databases from the classpath. The databases can be archived into a jar or zip file or left as is.

All such databases are read-only.

To access an unarchived database from the classpath, use the *classpath* subsubprotocol.

For example, for a database called *sample* in *C:\derby\demo\databases*, you can put the *C:\derby\demo\databases* directory in the classpath and access *sample* like this:

```
jdbc:derby:classpath:sample
```

If only *C:\derby* were in the classpath, you could access *sample* (read-only) like this:

```
jdbc:derby:classpath:demo/databases/sample
```

Accessing databases from a jar or zip file:

It is possible to access databases from a jar file. The jar file does not have to be on the classpath.

Note: All such databases are read-only.

For example, suppose you have archived the database *jarDB1* into a file called *jar1.jar*. This archive is in the classpath before you start up Derby. You can access *jarDB1* with the following connection URL

```
jdbc:derby:classpath:jarDB1
```

To access a database in a jar file that is not on the classpath, use the *jar* subprotocol.

For example, suppose you have archived the database *jarDB2* into a file called *jar2.jar*. This archive is not in the classpath. You can access *jarDB2* by specifying the path to the jar file along with the *jar* subsubprotocol, like this:

```
jdbc:derby:jar:(c:/derby/lib/jar2.jar)jarDB2
```

For complete instructions and examples of accessing databases in jar files, see Accessing a read-only database in a zip/jar file.

Database connection examples

The examples in this section use the syntax of the connection URL for use in an embedded environment.

For reference information about connection URLs, see "Syntax of database connection URLs for applications with embedded databases" in the *Derby Reference Manual* or "Accessing the Network Server by using the network client driver" in the *Derby Server and Administration Guide*.

- *jdbc:derby:db1*

  Open a connection to the database *db1*. *db1* is a directory located in the system directory.
- *jdbc:derby:london/sales*

  Open a connection to the database *london/sales*. *london* is a subdirectory of the system directory, and *sales* is a subdirectory of the directory *london*.
- *jdbc:derby:/reference/phrases/french*

  Open a connection to the database */reference/phrases/french*.

  On a UNIX system, this would be the path of the directory. On a Windows system, the path would be *C:\reference\phrases\french* if the current drive were *C*.
- *jdbc:derby:a:/demo/sample*

  Open a connection to the database stored in the directory *\demo\sample* on drive *A* (usually the floppy drive) on a Windows system.
- *jdbc:derby:c:/databases/salesdb jdbc:derby:salesdb*

  These two connection URLs connect to the same database, *salesdb*, on a Windows platform if the system directory of the Derby system is *C:\databases*.
- *jdbc:derby:support/bugsdb;create=true*

  Create the database *support/bugsdb* in the system directory, automatically creating the intermediate directory *support* if it does not exist.

- *jdbc:derby:sample;shutdown=true*

  Shut down the *sample* database. (Authentication is not enabled, so no user credentials are required.)
- *jdbc:derby:memory:myDB*

  Access the in-memory database named *myDB*. The syntax for a client connection URL is different; see Using in-memory databases for details.
- *jdbc:derby:classpath:myDB*

  Access *myDB* (which is directly in a directory in the classpath) as a read-only database.
- *jdbc:derby:jar:(C:/dbs.jar)products/boiledfood*

  Access the read-only database *boiledfood* in the *products* directory from the jar file *C:/dbs.jar*.
- *jdbc:derby:directory:myDB*

  Access *myDB*, which is in the system directory.

Working with the database connection URL attributes

You specify attributes on the Derby connection URL.

The examples in this section use the syntax of the connection URL for use in an embedded environment. You can also specify these same attributes and values on the client connection URL if you are using Derby as a database server. For more information, see the *Derby Server and Administration Guide*.

You can also set these attributes by passing a *Properties* object along with a connection URL to `DriverManager.getConnection` when obtaining a connection; see Specifying attributes in a properties object. If you specify any attributes both on the connection URL and in a *Properties* object, the attributes on the connection URL override the attributes in the *Properties* object.

All attributes are optional.

For more information on working with connection URL attributes, see the following:

- "Configuring database encryption" in the *Derby Security Guide* for information on database encryption
- *Derby Server and Administration Guide* for information on tracing network clients, replicating databases, restoring databases from backup, and logging on separate devices

For complete information about the attributes, see "Setting attributes for the database connection URL" in the *Derby Reference Manual*.

For detailed information about the connection URL syntax, see "Syntax of database connection URLs for applications with embedded databases" in the *Derby Reference Manual*.

Using the databaseName attribute

You can use a `databaseName` attribute on a database connection URL to specify the path name of the database to which you want to connect.

```
jdbc:derby:;databaseName=databaseName
```

You can access read-only databases in jar or zip files by specifying `jar` as the subsubprotocol, like this:

```
jdbc:derby:jar:(pathToArchive)databasePathWithinArchive
```

Or, if the jar or zip file has been included in the classpath, like this:

```
jdbc:derby:classpath:databasePathWithinArchive
```

The path separator in the path name is a forward slash (`/`), even in Windows path names. The path name cannot contain a colon (`:`), except for the colon after the drive name in a Windows path name. See Conventions for specifying the database path name for more information.

Shutting down Derby or an individual database

Applications in an embedded environment shut down the Derby system by specifying the *shutdown=true* attribute in the connection URL. To shut down the system, you do not specify a database name, and you do not ordinarily specify any other attribute.

```
jdbc:derby:;shutdown=true
```

A successful shutdown always results in an *SQLException* to indicate that Derby has shut down and that there is no other exception.

If you are running with a security manager on JDK 8 or higher, you must grant Derby permission to deregister the embedded driver in order to fully shut down the system. See "Configuring Java security" in the *Derby Security Guide* for details.

If you have enabled user authentication at the system level, you will need to specify credentials (that is, username and password) in order to shut down a Derby system, and the supplied username and password must also be defined at the system level.

You can also shut down an individual database if you specify the *databaseName*.
You can shut down the database of the current connection if you specify the default connection instead of a database name(within an SQL statement).

```
// shutting down a database from your application
DriverManager.getConnection(
    "jdbc:derby:sample;shutdown=true");
```

If user authentication and SQL authorization are both enabled, only the database owner can shut down the database. (See the *Derby Security Guide* for details on authentication and authorization.)

```
// shutting down an authenticated database as database owner
DriverManager.getConnection(

 "jdbc:derby:securesample;user=joeowner;password=secret;shutdown=true");
```

If you previously called the *java.sql.DriverManager.setLoginTimeout* method to enable a login timeout, a shutdown of Derby or of an individual database can fail under circumstances like the following:

- Network problems which slow down LDAP authentication
- Heavily loaded databases which take a long time time to quiesce

Attention: It is good practice to close existing connections before shutting down the system or database. Connections created before the shutdown will not be usable after shutdown is performed. Attempting to access connections after shutdown may cause errors including instances of `NullPointerException` or protocol violations.

Creating and accessing a database

You create a database by supplying a new database name in the connection URL and specifying *create=true*.

Derby creates a new database inside a new subdirectory in the system directory. This system directory has the same name as the new database. If you specify a partial path, it is relative to the system directory. You can also specify an absolute path.

```
jdbc:derby:databaseName;create=true
```

For more details about *create=true*, see *"create=true"* in the *Derby Reference Manual*.

Providing a user name and password

When user authentication is enabled, an application must provide a user name and password. One way to do this is to use the *user=userName* and *password=userPassword* connection URL attributes.

```
jdbc:derby:sample;user=jill;password=toFetchAPail
```

Creating a database with locale-based collation

By default, Derby uses Unicode codepoint collation. However, you can specify locale-based collation when you create the database.

You can use the *collation=collation* and *territory=ll_CC* connection URL attributes to specify locale-based collation (see the *Derby Reference Manual* for details on these attributes). This type of collation applies only to user-defined tables. The system tables use the Unicode codepoint collation.

Restriction: The *collation=collation* and *territory=ll_CC* attributes can be specified only when you create a database. You cannot specify these attributes on an existing database or when you upgrade a database.

To create a database with locale-based collation, specify the language and country codes for the *territory=ll_CC* attribute, and the TERRITORY_BASED value for the *collation=collation* attribute, when you create the database.

For example, you could use the following connection URL:

```
jdbc:derby:MexicanDB;create=true;territory=es_MX;collation=TERRITORY_BASED
```

See the documentation of the *territory=ll_CC* and *collation=collation* attributes in the *Derby Reference Manual* for details on these attributes. See Creating a case-insensitive database for information on making the database use case-insensensitive searches.

Creating a case-insensitive database

When you create a database using locale-based collation, the *collation=collation* value TERRITORY_BASED uses the default collation strength for the locale, usually TERTIARY, which will consider character case significant in searches and comparisons.

To make the database use case-insensitive searches, specify an explicit strength lower than TERTIARY with the *collation=collation* attribute. The strength name is appended to TERRITORY_BASED with a colon to separate them.

For example, you could specify the following connection URL:

```
jdbc:derby:SwedishDB;create=true;territory=sv_SE;collation=TERRITORY_BASED:PRIMARY
```

With strength PRIMARY, the characters 'A' and 'a' will be considered equal, as well as 'à' ('a' with a grave accent). (This behavior is commonly the default with many other databases.) To make searches respect differences in accent, use strength SECONDARY.

The exact interpretation of the strength part of the attribute depends upon the locale.

For more information, see Creating a database with locale-based collation and the documentation of the *territory=ll_CC* and *collation=collation* attributes in the *Derby Reference Manual*.

Creating a customized collator

You may need to define a collation order different from that of the strengths provided by the *collation=collation* attribute.

To define a new collation order, follow these steps.

1. Create a class that extends the *java.text.spi.CollatorProvider* class and that returns a collator that orders strings the way you want it to.
2. Create a text file named *META-INF/services/java.text.spi.CollatorProvider* that contains one line with the name of your collator provider class.
3. Put the compiled class file and the text file in a jar file that you drop into your JRE's *lib/ext* directory or in one of the directories specified by the *java.ext.dirs* property.

For example, suppose you want to define a collation order to make Greek characters sort near their Latin equivalents ('#' near 'a', "#' near 'b', and so on). You could define another locale with a *CollatorProvider* that returns a *java.text.RuleBasedCollator* with ever rules you want. See the API documentation for the *RuleBasedCollator* class for details about how you specify rules. In its simplest form, a set of rules might look like `"a,A < b,B < c,C"`, which means more or less that 'a' and 'A' should be sorted before 'b' and 'B', which should be sorted before 'c' and 'C'. So to get the Greek characters sorted near similar Latin characters, define a *CollatorProvider* that looks like this one:

```
public class MyCollatorProvider extends CollatorProvider {
    public Locale[] getAvailableLocales() {
        return new Locale[] {
            new Locale("en", "US", "greek")
        };
    }

    public Collator getInstance(Locale locale) {
        StringBuilder rules = new StringBuilder();
        // alpha should go between a and b
        rules.append("< a,A < \u03b1,\u0391 < b,B");
        // beta should go between b and c
        rules.append("& b,B < \u03b2,\u0392 < c,C");
        // add more rules here ....

        try {
            return new RuleBasedCollator(rules.toString());
        } catch (ParseException pe) {
            throw new Error(pe);
        }
    }
}
```

Again, put the compiled class and the *META-INF/services/java.text.spi.CollatorProvider* file in a jar file, and start the `ij` tool with the `-Djava.ext.dirs=.` option in the directory where the jar file is located. Create a database that uses the new locale and insert some data with both Greek and Latin characters:

```
ij> connect
 'jdbc:derby:GreekDB;territory=en_US_greek;collation=TERRITORY_BASED;create=true';
ij> create table t (x varchar(12));
0 rows inserted/updated/deleted
ij> insert into t values 'a', 'b', 'c', '#', '#';
5 rows inserted/updated/deleted
ij> select * from t order by x;
X
------------
a
#
b
#
c

5 rows selected
```

The ordering is just as you wanted it, with the Greek characters between the Latin ones, and not at the end where they would normally be located.

One word of caution: If, after you have created a database, you update your custom *CollatorProvider* so that the ordering is changed, you will need to recreate the database. You must do this because the indexes in the database are ordered, and you may see strange results if the indexes are ordered with a different collator from the one your database is currently using.

Specifying attributes in a properties object

Instead of specifying attributes on the connection URL, you can specify attributes as properties in a `Properties` object that you pass as a second argument to the `DriverManager.getConnection` method.

For example, to set the user name and password:

```
Properties p = new Properties();

p.setProperty("user", "sa");
p.setProperty("password", "manager");
p.setProperty("create", "true");

Connection conn = DriverManager.getConnection(
    "jdbc:derby:mynewDB", p);
```

Note: If you specify any attributes both on the connection URL and in a *Properties* object, the attributes on the connection URL override the attributes in the *Properties* object.

Using in-memory databases

For testing and developing applications, or for processing transient or reproducible data, you can use Derby's in-memory database facility.

An in-memory database resides completely in main memory, not in the file system. It is useful for testing and developing applications, when you may want to create and discard databases that will never be used again. It is also useful when you need to process only temporary or reproducible data.

If you have the required memory available, you may also benefit from faster processing (no disk I/O) and from the simplicity of not having to explicitly delete databases you have finished with.

Creating an in-memory database

To create an in-memory database, specify `memory` as the JDBC subsubprotocol. For example, to create an in-memory database named `myDB` using the embedded driver, use the following connection URL:

```
jdbc:derby:memory:myDB;create=true
```

For the network client driver, use the following connection URL. Because the client driver does not understand the `memory` subsubprotocol, you must include it in the database name:

```
jdbc:derby://myhost:1527/memory:myDB;create=true
```

Be careful to specify a colon (`:`) after `memory`.

Referring to in-memory databases

When you create or refer to an in-memory database, any path that is not absolute is interpreted as relative to the system directory, just as with file system databases. For

example, if the system directory is `C:\myderby`, the following paths are regarded as equivalent:

```
jdbc:derby:memory:db
jdbc:derby:memory:C:\myderby\db
```

Similarly, Derby treats the following URLs as names for the same in-memory database:

```
jdbc:derby:memory:/home/myname/db
jdbc:derby:memory:/home/myname/../myname/db
```

Conventions for specifying the database path name has more information on database paths.

Using in-memory databases

When you use an in-memory database, you need to make sure to configure the heap and the Derby page cache size. See "Configure Derby to use an in-memory database" in *Tuning Derby* for details.

For examples of how to use an in-memory database, see some of the `ij` command examples in the *Derby Tools and Utilities Guide* (`execute` and `async`, for example).

Removing an in-memory database

To remove an in-memory database, use the connection URL attribute *drop* as follows:

```
jdbc:derby:memory:myDB;drop=true
```

```
jdbc:derby://myhost:1527/memory:myDB;drop=true
```

You can shut down an in-memory database using the *shutdown=true* attribute before you drop the database, but this is optional. Dropping the database also performs the shutdown.

When you drop the database, Derby issues what appears to be an error but is actually an indication of success. You need to catch error 08006, as described in "The WwdEmbedded program" in *Getting Started with Derby*.

If user authentication and SQL authorization are both enabled, only the database owner can drop the database. (See the *Derby Security Guide* for details on authentication and authorization.)

An in-memory database is automatically removed if any of the following happens:

- The Java Virtual Machine (JVM) is shut down normally (for example, if you exit the `ij` tool)
- The JVM crashes
- The machine you are running on crashes or shuts down

Persisting an in-memory database

If you create an in-memory database and then decided that you want to keep it after all, you can use one of the backup system procedures (SYSCS_UTIL.SYSCS_BACKUP_DATABASE, for example) to persist it. You can then boot it as an in-memory database at a later time, or use it as a normal file system database. See "Backing up and restoring databases" in *Derby Server and Administration Guide* for information on using the backup procedures.

Working with Derby properties

This section describes how to use Derby properties.

For details on specific properties, see the "Derby properties" section of the *Derby Reference Manual*.

Properties overview

Derby lets you configure behavior or attributes of a system, a specific database, or a specific *conglomerate* (a table or index) through the use of properties.

Examples of behavior or attributes that you can configure are:

- Whether to authorize users
- Page size of tables and indexes
- Where and whether to create an error log
- Which databases in the system to boot

Scope of properties

You use properties to configure a Derby system, database, or conglomerate.

- *system-wide*

  Most properties can be set on a *system-wide* basis; that is, you set a property for the entire system and all its databases and conglomerates, if this is applicable. Some properties, such as error handling and automatic booting, can be configured only in this way, since they apply to the entire system. (For information about the Derby system, see Derby system.)
- *database-wide*

  Some properties can also be set on a *database-wide* basis. That is, the property is true for the selected database only and not for the other databases in the system unless it is set individually within each of them.

For properties that affect conglomerates, changing the value of such properties affects only conglomerates that are created after the change. Conglomerates created earlier are unaffected.

Note: Database-wide properties are stored in the database and are simpler for deployment, in the sense that they follow the database. System-wide properties can be more practical during the development process.

Persistence of properties

A database-wide property always has persistence. That is, its value is stored in the database.

Typically, a database-wide property is in effect until you explicitly change the property or until you set a system-wide property with precedence over database-wide properties (see Precedence of properties).

To disable or turn off a database-wide property setting, set its value to null. This has the effect of removing the property from the list of database properties and restoring the system property setting, if there is one (and if *derby.database.propertiesOnly* has not been set; see Protection of database-wide properties).

A system-wide property might have persistence, depending on how you set it. If you set it programmatically, it persists only for the duration of the JVM of the application that set it. If you set it in the *derby.properties* file, a property persists until:

- That value is changed and the system is rebooted
- The file is removed from the system and the system is rebooted
- The database is booted outside of that system

Precedence of properties

The search order for properties is as follows.

1. System-wide properties set programmatically (as a command-line option to the JVM when starting the application or within application code)
2. Database-wide properties
3. System-wide properties set in the *derby.properties* file

This means, for example, that system-wide properties set programmatically override database-wide properties and system-wide properties set in the *derby.properties* file, and that database-wide properties override system-wide properties set in the *derby.properties* file.

Protection of database-wide properties:

There is one important exception to the search order for properties described above: When you set the *derby.database.propertiesOnly* property to *true*, database-wide properties cannot be overridden by system-wide properties.

This property ensures that a database's environment cannot be modified by the environment in which it is booted. Any application running in an embedded environment can set this property to `true` for security reasons.

See the "Derby properties" section of the *Derby Reference Manual* for details on the *derby.database.propertiesOnly* property.

Dynamic versus static properties

Most properties are dynamic; that means you can set them while Derby is running, and their values change without requiring a reboot of Derby. In some cases, this change takes place immediately; in some cases, it takes place at the next connection.

Some properties are static, which means changes to their values will not take effect while Derby is running. You must restart or set them before (or while) starting Derby.

For more information, see Making dynamic or static changes to properties.

Setting Derby properties

This section covers the different ways of setting properties.

Setting system-wide properties

You can set system-wide properties programmatically (as a command-line option to the JVM when starting the application or within application code) or in the text file *derby.properties*.

Changing the system-wide properties programmatically:

You can set properties programmatically -- either in application code before booting the Derby driver or as a command-line option to the Java Virtual Machine (JVM) when booting the application that starts up Derby.

When you set properties programmatically, these properties persist only for the duration of the application. Properties set programmatically are not written to the *derby.properties* file or made persistent in any other way by Derby.

Note: Setting properties programmatically works only for the application that starts up Derby; for example, for an application in an embedded environment or for the application server that starts up a server product. It does not work for client applications connecting to a server that is running.

You can set properties programmatically in the following ways:

- As a parameter to the JVM command line
- Using a Properties object within an application or statement

As a parameter to the JVM command line

You can set system-wide properties as parameters to the JVM command line when you start up the application or framework in which Derby is embedded. To do so, you typically use the *-D* option. For example:

```
java -Dderby.system.home=C:\home\Derby\
    -Dderby.storage.pageSize=8192 JDBCTest
```

Using a Properties object within an application or statement

In embedded mode, your application runs in the same JVM as Derby, so you can also set system properties within an application using a *Properties* object before loading the Derby JDBC driver. The following example sets *derby.system.home* on Windows.

```
Properties p = System.getProperties();
p.setProperty("derby.system.home", "C:\databases\sample");
```

Note: If you pass in a *Properties* object as an argument to the *DriverManager.getConnection* call when connecting to a database, those properties are used as database connection URL attributes, not as properties of the type discussed in this section. For more information, see Connecting to databases and Working with the database connection URL attributes as well as the *Derby Reference Manual*.

Changing the system-wide properties by using the derby.properties file:

You can set persistent system-wide properties in a text file called *derby.properties*, which must be placed in the directory specified by the *derby.system.home* property.

There is one *derby.properties* file per system, not one per database. The file must be created in the system directory. In a client/server environment, that directory is on the server. (For more information about a Derby system and the system directory, see Derby system.)

Derby does *not*:

- Provide this file
- Automatically create this file for you
- Automatically write any properties or values to this file

Instead, you must create, write, and edit this file yourself.

The file should be in the format created by the *java.util.Properties.save* method.

The following is the text of a sample properties file:

```
derby.infolog.append=true
derby.storage.pageSize=8192
derby.storage.pageReservedSpace=60
```

Properties set this way are persistent for the system until changed, until the file is removed from the system, or until the system is booted in some other directory (in which case Derby would be looking for *derby.properties* in that new directory). If a database is removed from a system, system-wide properties do not "travel" with the database unless explicitly set again.

Verifying system properties:

You can find out the value of a system property if you set it programmatically. You cannot find out the value of a system property if you set it in the *derby.properties* file.

For example, if you set the value of the `derby.storage.pageSize` system-wide property in your program or on the command line, the following code will retrieve its value from the System Properties object:

```
Properties sprops = System.getProperties();
```

```
System.out.println("derby.storage.pageSize value: "
+ sprops.getProperty("derby.storage.pageSize"));
```

You can also use Java Management Extensions (JMX) technology to obtain system information, including some settings that correspond to system properties. For details, see "Using Java Management Extensions (JMX) technology" in the *Derby Server and Administration Guide*.

Setting database-wide properties

Database-wide properties, which affect a single database, are stored within the database itself. This allows different databases within a single Derby system to have different properties and ensures that the properties are correctly retained when a database is moved away from its original system or copied.

You should use database-wide properties wherever possible for ease of deployment and for security.

You set and verify database-wide properties using system procedures within SQL statements.

To set a property, you connect to the database, create a statement, and then use the `SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` procedure, passing the name of the property and the value.

To check the current value of a property, you connect to the database, create a statement, and then use the `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function, passing in the name of the property.

If you specify an invalid value, Derby uses the default value for the property. (If you call the `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function, however, it displays the invalid value.)

See the *Derby Reference Manual* for more information on how to use these system functions and procedures.

Setting properties in a client/server environment

In a client/server environment, you must set the system properties for the *server*'s system. That means that when you are using the *derby.properties* file, the file exists in the *server*'s *derby.system.home* directory. Client applications can set database-wide properties because they are set via SQL statements.

The following table summarizes the ways to set properties.

Table 3. Ways to set properties

Type of Property	How You Set It
System-wide	• In *derby.properties* • As a command-line option when starting the JVM that holds the server or, if the server is started from within a program, programmatically by the program that hosts the server
Database-wide	Using system procedures and functions in an SQL statement

Making dynamic or static changes to properties

Properties set in the *derby.properties* file and on the command line of the application that boots Derby are *always* static, because Derby reads this file and those parameters only at startup.

Only properties set in the following ways have the potential to be dynamic:

- As database-wide properties
- As system-wide properties via a *Properties* object in the application in which the Derby engine is embedded

See the "Derby properties" section of the *Derby Reference Manual* for information about specific properties.

Properties case study

Derby allows you a lot of freedom in configuring your system. This freedom can be confusing if you do not understand how properties work. You also have the option of not setting any properties and instead using the Derby defaults, which are tuned for a single-user embedded system.

Imagine the following scenario of an embedded environment:

Your system has a *derby.properties* file, a text file in the system directory, which you have created and named *system_directory*. Your databases have also been created in this directory. The properties file sets the following property:

```
derby.storage.pageSize=8192
```

You start up your application, being sure to set the *derby.system.home* property appropriately:

```
java -Dderby.system.home=c:\system_directory MyApp
```

The command lines in this example assume that you are using a Windows system.

You then create a new table:

```
CREATE TABLE table1 (a INT, b VARCHAR(10))
```

Derby takes the page size of 8192 from the system-wide properties set in the *derby.properties* file, since the property has not been set any other way.

You shut down and then restart your application, setting the value of *derby.storage.pageSize* to 4096 programmatically, as a parameter to the JVM command line:

```
java -Dderby.system.home=c:\system_directory
    -Dderby.storage.pageSize=4096 MyApp

CREATE TABLE anothertable (a INT, b VARCHAR(10))
```

The page size for the *anothertable* table will be 4096 bytes.

You establish a connection to the database and set the value of the page size for all new tables to 32768 as a database-wide property:

```
CallableStatement cs =
  conn.prepareCall("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(?, ?)");
cs.setString(1, "derby.storage.pageSize");
cs.setString(2, "32768");
cs.execute();
cs.close();
```

You then create a new table that automatically inherits the page size set by the property:

```
CREATE TABLE table2 (a INT, b VARCHAR(10))
```

The page size for the *table2* table is 32768 bytes.

You shut down the application, then restart, this time forgetting to set the system-wide property programmatically (as a command-line option to the JVM):

```
java -Dderby.system.home=c:\system_directory MyApp
```

You then create another table:

```
CREATE TABLE table4 (a INT, b VARCHAR(10))
```

Derby uses the persistent database-wide property of 32768 for this table, since the database-wide property set in the previous session is persistent and overrides the system-wide property set in the *derby.properties* file.

What you have is a situation in which three different tables each get a different page size, even though the *derby.properties* file remained constant.

If you remove the *derby.properties* file from the system or remove the database from its current location (forgetting to move the file with it), you could get yet another value for a new table.

To avoid this situation, be consistent in the way you set properties.

Deploying Derby applications

Typically, once you have developed a Derby application and database, you package up the application, the Derby libraries, and the database in some means for distribution to your users. This process is called *deployment*.

This section discusses issues for deploying Derby applications and databases.

Deployment issues

This section discusses deployment options and details.

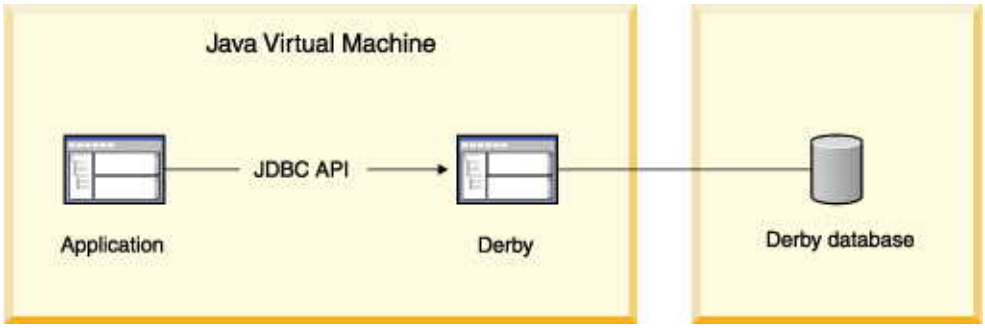
Embedded deployment application overview

In an embedded environment, Derby runs in the same JVM as the application.

The application can be a single-user application or a multi-user application server. In the latter case, Derby runs embedded in the user-provided server framework, and any client applications use user-provided connectivity or allow the application server to handle all database interaction.

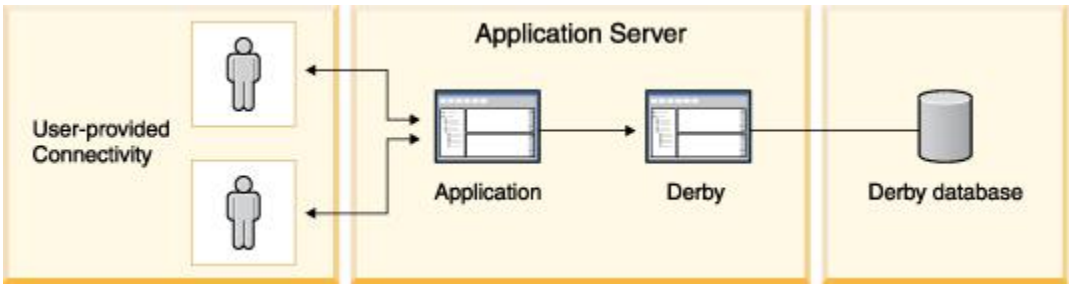
The following figure shows Derby embedded in a single-user Java application.

Figure 3. Derby embedded in a single-user Java application



The following figure shows Derby embedded in a multi-user Java application server.

Figure 4. Derby embedded in a multi-user Java application server



When a Derby database is embedded in a Java application, the database is dedicated to that single application. If you deploy more than one copy of the application, *each application has its own copy of the database and Derby software*. A Derby server framework can work in multi-threaded, multi-connection mode and can even connect to more than one database at a time. A server framework, such as the Derby Network

Server, can be used to manage multiple connections and handle network capabilities. Some server framework solutions, such as WebSphere Application Server, provide additional features such as web services and connection pooling. However, only one server framework at a time can operate against a Derby database.

The Derby application accesses an embedded Derby database through the JDBC API. To connect, an application makes a call to the local Derby JDBC driver. Accessing the JDBC driver automatically starts the embedded Derby software. The calling application is responsible for shutting down the embedded Derby database software.

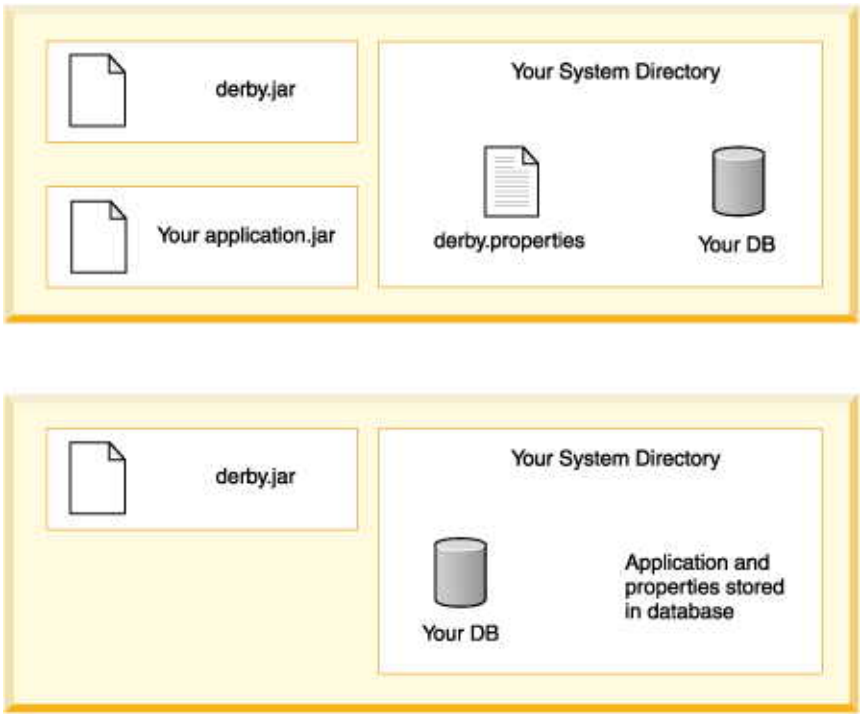
Deploying Derby in an embedded environment

You can embed Derby in any Java application (single- or multi-user) by deploying the following packages.

- The Derby library (*derby.jar*).
- The libraries for the application. You have the option of storing these libraries in the database.
- The database or databases used by the application, in the context of their system directory.

In the following figure, the top graphic shows the deployment of an application, where the application, the Derby software for embedded use, the derby.properties file and the database are four objects. The bottom graphic shows a simplified deployment by reducing the number of objects to two by storing the application and the properties file in the database.

Figure 5. Two approaches to deploying a Derby application in an embedded environment



Embedded systems and properties

Database-wide properties are stored in the database and are simpler for deployment, while system-wide parameters might be easier for development.

- If you are setting any system-wide properties, see if they can be set as database-wide properties instead.
- Are any properties being set in the *derby.properties* file? Some properties can only be set on a system-wide basis. If so, deploy the entire system directory along with the properties file. Deploy only those databases that you wish to include. Setting properties programmatically can simplify this step- you will not have to worry about deploying the system directory/properties file.

Extra steps are required for deploying an application and an embedded database on read-only media.

Creating Derby databases for read-only use

You can create Derby databases for use on read-only media such as CD-ROMs.

Derby databases in zip or jar files are also read-only databases. Typically, read-only databases are deployed with an application in an embedded environment.

Creating and preparing the database for read-only use

To create databases for use on read-only media, perform these steps.

1. Create and populate the database on read-write media.
2. Commit all transactions and shut down Derby in the prescribed manner. If you do not shut down Derby in the prescribed manner, Derby will need to perform recovery the next time the system boots. Derby cannot perform recovery on read-only media.
3. Delete the *tmp* directory if one was created within your database directory. If you include this directory, Derby will attempt to delete it and will return errors when attempting to boot a database on read-only media.
4. For the read-only database, set the property *derby.storage.tempDirectory* to a writable location.

   Derby needs to write to temporary files for large sorts required by such SQL statements as ORDER BY, UNION, DISTINCT, and GROUP BY. For more information about this property, see the *Derby Reference Manual*.

   ```
   derby.storage.tempDirectory=c:/temp/mytemp
   ```
5. Configure the database to send error messages to a writable file or to an output stream.

   For information on the *derby.stream.error.file* property, see the *Derby Reference Manual*.

   ```
   derby.stream.error.file=c:/temp/mylog.LOG
   ```

Be sure to set these properties so that they are deployed with the database.

Deploying the database on the read-only media

To deploy the database on read-only media, perform the following steps.

1. Move the database directory to the read-only media, including the necessary subdirectory directories (*log* and *seg0*) and the file `service.properties`.
2. Use the database as usual, except that you will not be able to insert or update any data in the database or create or drop dictionary objects.

Transferring read-only databases to archive (jar or zip) files

Once a database has been created in Derby, it can be stored in a jar or zip file and continue to be accessed by Derby in read-only mode.

This allows a read-only database to be distributed as a single file instead of as multiple files within a directory and to be compressed. In fact, a jar or zip file can contain any number of Derby databases and can also contain other information not related to Derby, such as application data or code.

You cannot store the *derby.properties* file in a jar or zip file.

To create a jar or zip file containing one or more Derby databases:

1. Create a database for use on read-only media.
2. From the directory that contains the database folder, archive the database directory and its contents. For example, for the database *sales* that lives in the system directory *C:\london*, issue the command from *london*. Do not issue the command from inside the database directory itself.

For example, archive the database folder and its contents using the JAR program from the JDK. You can use any zip or jar tool to generate the archive.

This command archives the database directory *sales* and its contents into a compressed jar file called *dbs.jar*.

```
cd C:\london
jar cMf C:\dbs.jar sales
```

You can add multiple databases with jar. For example, this command puts the *sales* databases and the *boiledfood* database (in the subdirectory products) into the archive.

```
cd C:\london
jar cMf C:\dbs.jar sales products\boiledfood
```

The relative paths of the database in the jar need not match their original relative paths. You can do this by allowing your archive tool to change the path, or by moving the original databases before archiving them.

The archive can be compressed or uncompressed, or individual databases can be uncompressed or compressed if your archive tool allows it. Compressed databases take up a smaller amount of space on disk, depending on the data loaded, but are slower to access.

Once the database is archived into the jar or zip file, it has no relationship to the original database. The original database can continue to be modified if desired.

Accessing a read-only database in a zip/jar file

To access a database in a zip/jar, you specify the jar in the subsubprotocol.

```
jdbc:derby:jar:(pathToArchive)databasePathWithinArchive
```

The *pathToArchive* is the absolute path to the archive file. The *databasePathWithinArchive* is the relative path to the database within the archive. For example:

```
jdbc:derby:jar:(C:/dbs.jar)products/boiledfood
jdbc:derby:jar:(C:/dbs.jar)sales
```

If you have trouble finding a database within an archive, check the contents of the archive using your archive tool. The *databasePathWithinArchive* must match the one in the archive. You might find that the path in the archive has a leading slash, and thus the URL would be:

```
jdbc:derby:jar:(C:/dbs.jar)/products/boiledfood
```

Databases in a jar or zip file are always opened read-only and there is currently no support to allow updates of any type.

Accessing databases within a jar file using the classpath

Once an archive containing one or more Derby databases has been created it can be placed in the classpath. This allows access to a database from within an application without the application's knowing the path of the archive.

When jar or zip files are part of the classpath, you specify the *classpath* subsubprotocol instead of the *jar* subsubprotocol to connect to them.

To access a database in a zip or jar file in the classpath:

1. Set the classpath to include the jar or zip file before starting up Derby:

   ```
   CLASSPATH="C:\dbs.jar;%CLASSPATH%"
   ```
2. Connect to a database within the jar or zip file with the following connection URL:

   ```
   jdbc:derby:classpath:databasePathWithinArchive
   ```

For example:

```
jdbc:derby:classpath:products/boiledfood
```

Databases on read-only media and DatabaseMetaData

Databases on read-only media return true for *DatabaseMetaData.isReadOnly*.

Loading classes from a database

You can store application logic in a database and then load classes from the database.

Application logic, which can be used by SQL functions and procedures, includes Java class files and other resources. Storing application code simplifies application deployment, since it reduces the potential for problems with a user's classpath.

In an embedded environment, when application logic is stored in the database, Derby can access classes loaded by the Derby class loader from stored jar files.

Class loading overview

You store application classes and resources by storing one or more jar files in the database. Then your application can access classes loaded by Derby from the jar file and does not need to be coded in a particular way. The only difference is the way in which you invoke the application.

Here are the basic steps.

Create jar files for your application

Include any Java classes in a jar file that are intended for Derby class loading, except the following classes.

- The standard Java packages (*java.**, *javax.**)

  Derby does not prevent you from storing such a jar file in the database, but these classes *are never loaded* from the jar file.
- The classes that are supplied with your Java environment

A running Derby system can load classes from any number of jar files from any number of schemas and databases.

Create jar files intended for Derby database class loading the same way you create a jar file for inclusion in a user's classpath. For example, consider an application targeted at travel agencies:

```
jar cf travelagent.jar travelagent/*.class.
```

Various IDEs have tools to generate a list of contents for a jar file based on your application. If your application requires classes from other jar files, you have a choice:

- *Extract the required third-party classes from their jar file and include only those classes in your jar file.*

  Use this option when you need only a small subset of the classes in the third-party jar file.
- *Store the third-party jar file in the database.*

  Use this option when you need most or all of the classes in the third-party jar file, since your application and third-party logic can be upgraded separately.
- *Deploy the third-party jar file in the user's class path.*

  Use this option when the classes are already installed on a user's machine (for example, Objectspace's JGL classes).

Add the jar file or files to the database

Use a set of procedures to install, replace, and remove jar files in a database. When you install a jar file in a database, you give it a Derby jar name, which is an *SQLIdentifier*.

Note: Once a jar file has been installed, you cannot modify any of the individual classes or resources within the jar file. Instead, you must replace the entire jar file.

Jar file examples:

See "System procedures for storing jar files in a database" in the *Derby Reference Manual* for reference information about the jar file system procedures and complete syntax.

Pay particular attention to the sections on execute privileges for the `sqlj.install_jar` and `sqlj.replace_jar` procedures. Since these procedures can be used to install arbitrary code (possibly from across the network) that runs in the same Java Virtual Machine as the Derby database engine, both authentication and SQL authorization should be enabled, and execute privileges should be granted only to trusted users.

Installing jar files:

The following examples show how to use the `sqlj.install_jar` procedure.

```
-- SQL  statement
CALL sqlj.install_jar(
    'tours.jar', 'APP.Sample1', 0)

-- SQL  statement
-- using a quoted identifier for the
-- Derby jar name
CALL sqlj.install_jar(
    'tours.jar', 'APP."Sample2"', 0)
```

Removing jar files:

The following example shows how to use the `sqlj.remove_jar` procedure.

```
-- SQL  statement
CALL sqlj.remove_jar(
```

```
    'APP.Sample1', 0)
```

Replacing jar files:

The following example shows how to use the `sqlj.replace_jar` procedure.

```
-- SQL  statement
CALL sqlj.replace_jar(
    'c:\myjarfiles\newtours.jar', 'APP.Sample1')
```

Enable database class loading with a property

Once you have added one or more jar files to a database, you must set the database jar classpath by including the jar file or files in the *derby.database.classpath* property to enable Derby to load classes from the jar files.

This property, which behaves like a classpath, specifies the jar files to be searched for classes and resources and the order in which they are searched. If Derby does not find a needed class stored in the database, it can retrieve the class from the user's classpath. (Derby first looks in the user's classpath before looking in the database.)

- Separate jar files with a colon (`:`).
- Use fully qualified identifiers for the jar files (schema name and jar name).
- Set the property as a database-level property for the database.

Example:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.classpath',
    'APP.ToursLogic:APP.ACCOUNTINGLOGIC')
```

See "*derby.database.classpath*" in the *Derby Reference Manual* for more information about the property.

Note: Derby's class loader looks first in the user's classpath for any needed classes, and then in the database. To ensure class loading with the database class loader, remove classes from the classpath.

Code your applications

In your applications, you load the classes either by indirectly referencing them in the code or by directly using *java.lang.Class.forName*.

You load resources the way you normally would, using the standard *java.lang.Class.getResourceAsStream*, a mechanism that allows an application to access resources defined in the classpath without knowing where or how they are stored.

You do not need to make any changes to the way code interacts with Derby and its JDBC driver. An application can safely attempt to boot Derby, even though it is already running, without any errors. Applications connect to Derby in the usual manner.

Note: The method *getResource* is not supported.

Dynamic changes to jar files or to the database jar classpath

When you store jar files in a single database and make those jar files available to that database, it is possible to make changes to jar files or to change the database jar "classpath" dynamically (without having to reboot).

That is, when you install or replace a jar file within an SQL statement or change the database jar "classpath" (the *derby.database.classpath* property), Derby is able to load the new classes right away without your having to reboot.

Requirements for dynamic changes

Certain conditions must be met for Derby to be able to load the new classes right away without your having to reboot.

- You *originally configured* database-level class loading for the database correctly. Turning on the database-level class loading property requires setting the *derby.database.classpath* property with valid two-part names, then rebooting.
- If changes to the *derby.database.classpath* property are needed to reflect new jar files, you change the property to a valid value.

If these requirements are not met, you will have to reboot to see the changes.

Notes on dynamic changes

Remember the following items when you make dynamic changes to jar files or to the classpath.

- When you are changing the *derby.database.classpath* property, all classes loaded from database jar files are reloaded, even for a jar file that has not changed.
- Remember that the user's classpath is searched first.
- Any existing prepared statements will use the previously loaded classes unless they require class loading, in which case they will fail with a *ClassNotFound* error.
- Cached objects do not match objects created with newly loaded classes. For example, an in-memory *Customer* object will not match a new *Customer* object if the *Customer* class has been reloaded, and it will raise a *ClassCastException*.

Derby server-side programming

This section discusses special programming for Derby.

In particular, this section discusses how to program database-side JDBC routines, triggers, and table functions.

Programming database-side JDBC routines

Methods invoked within an application are called application-side methods. Methods invoked within Derby are called database-side routines.

An application-side method can be exactly the same as a database-side routine. The only difference is where you invoke them. You write the method only once. Where you invoke the method--within the application or within an SQL statement--determines whether it is an "application-side" or a "database-side" method.

Database-side JDBC routines and nested connections

Most database-side JDBC routines need to share the same transaction space as the statements that called them.

The reasons for this are:

- to avoid blocking and deadlocks
- to ensure that any updates done from within the routine are atomic with the outer transaction

In order to use the same transaction, the routine must use the same connection as the parent SQL statement in which the routine was executed. Connections re-used in this way are called *nested connections*.

Use the connection URL *jdbc:default:connection* to re-use the current *Connection*.

The database donnection URL *jdbc:default:connection* allows a Java method to get the *Connection* of the SQL statement that called it. This is the standard (SQL standard, Part 13, SQL Routines and Java) mechanism to obtain the nested connection object. The method would get a *Connection* as follows:

```
Connection conn = DriverManager.getConnection(
    "jdbc:default:connection");
```

URL attributes are not supported as part of this connection URL. Any URL attributes specified in a Properties object, user name, or password that are passed to a *java.sql.DriverManager.getConnection()* call will be ignored.

Loading a JDBC driver in a database-side routine is not required.

Requirements for database-side JDBC routines using nested connections

In order to preserve transactional atomicity, database-side JDBC routines that use nested connections have the following limitations.

- Can issue a commit or rollback only within a procedure (not a function).
- Cannot change the auto-commit connection attribute.
- Cannot modify the data in a table used by the parent statement that called the routine, using INSERT, UPDATE, or DELETE. For example, if a SELECT statement using the *T* table calls the `changeTables` procedure, `changeTables` cannot modify data in the *T* table.

- Cannot drop a table used by the statement that called the routine.
- Cannot be in a class whose static initializer executes DDL statements.

In addition, the Connection object that represents the nested connection always has its auto-commit mode set to false.

Database-side JDBC routines using non-nested connections

A database-side JDBC routine can create a new connection instead of using a nested connection. Statements executed in the routine will be part of a different transaction, and so can issue commits and rollbacks.

Such a routine can connect to a database different from the one to which the parent SQL statement that called it is connected. The routine does not use the same transaction or *Connection*. It establishes a new *Connection* and transaction.

Note: If database-side JDBC routines do not use nested connections, this means that they are operating outside of the normal DBMS transaction control, so it is not good practice to use them indiscriminately.

Invoking a procedure using the CALL command

If a procedure uses only IN parameters, Derby can execute the procedure by using the SQL CALL command. A stored procedure with IN, OUT, or INOUT parameters can be invoked from a client application by using a *CallableStatement*.

You can invoke the procedure in an SQL statement such as the following:

```
CALL MYPROC()
```

Note: You can roll back a CALL statement only if no commits or rollbacks occur within the specified procedure.

You can also use the CALL command to execute a routine that returns a value, but you will not be able to access the value.

Database-side JDBC routines and SQLExceptions

It is possible to code database-side routines, like application-side methods, to catch *SQLExceptions*. SQLExceptions that are caught within a routine are hidden from the calling application code.

When such *SQLExceptions* are of transaction severity (such as deadlocks), this "hiding" of the exception causes unexpected problems.

This is because errors of transaction severity roll back work already done by a transaction (not just the piece executed by the called method) and silently begin a new transaction. When the method execution is complete, Derby detects that the outer statement was invalidated by a deadlock and rolls back any work done *in the new transaction* as well. This is the expected behavior, because all the statements in between explicit commits should be treated atomically; the new transaction implicitly begun by Derby's rollback was not intended by the application designer.

However, this is not the same behavior that would happen if the method were invoked in the application. In that situation, Derby would roll back the work done by the transaction and silently begin a new transaction. Work in the new transaction would not be rolled back when the method returned. However, coding the application in that way means that the transaction did not end where you expected it to and is probably a programming mistake. Coding in this manner is not recommended.

A method that catches a deadlock exception and then continues is probably making a mistake. Errors of transaction severity should be caught not by nested code, but only by

the outermost application code. That is the only way to ensure that transactions begin and end where you expect them to.

Not all database vendors handle nested deadlocks the same way. For this and other reasons, it is not possible to write portable SQL-invoking methods. However, it is possible to write SQL-invoking methods that behave identically *regardless of whether you invoke them in the application or as a routine in the database*.

In order to ensure identical application- and database-side handling of nested errors, code try-catch blocks to check for the severity of exceptions as follows:

```
try {
    preparedStatement.execute();
} catch (SQLException se ) {
    String SQLState = se.getSQLState();
    if ( SQLState.equals( "23505" ) ) {
        correctDuplicateKey();
    } else if ( SQLState.equals( "22003" ) ) {
        correctArithmeticOverflow();
    } else {
        throw se;
    }
}
```

Of course, users also have the choice of not wrapping SQL statements in try-catch blocks within methods. In that case, *SQLExceptions* are caught higher up in their applications, which is the desired behavior.

User-defined SQLExceptions

When the execution of a database-side method raises an error, Derby wraps that exception in an *SQLException* with an *SQLState* of 38000.

You can avoid having Derby wrap the exception if:

- The exception is an *SQLException*
- The range of the *SQLState* is 38001-38999

(This conforms to the SQL99 standard.)

Programming trigger actions

Derby allows you to create triggers. When you create a trigger, you define an action or set of actions that are executed when a database event occurs on a specified table. A *database event* is a delete, insert, or update operation.

For example, if you define a trigger for a delete on a particular table, the trigger action is executed whenever someone deletes a row or rows from the table.

The "CREATE TRIGGER statement" topic in the *Derby Reference Manual* shows the complete `CREATE TRIGGER` syntax. This section provides information on defining the trigger action itself, which is only one aspect of creating triggers.

This section refers to the actions defined by the `CREATE TRIGGER` statement as the *trigger actions*.

Trigger action overview

A trigger action is a simple SQL statement.

For example:

```
CREATE TRIGGER . . .
DELETE FROM FlightAvailability
  WHERE flight_id IN (SELECT flight_id FROM FlightAvailability
    WHERE YEAR(flight_date) < 2005);
```

The trigger action includes not only the *triggeredSQLStatement* described in the "triggeredSQLStatement" section of the "CREATE TRIGGER statement" topic in the *Derby Reference Manual*, but also the `[ FOR EACH { ROW | STATEMENT } ]` clause and the `WHEN` clause.

A trigger action has limitations; for example, it cannot contain dynamic parameters or alter the table on which the trigger is defined. These limitations are detailed in the "triggeredSQLStatement" section of the "CREATE TRIGGER statement" topic in the *Derby Reference Manual*.

Performing referential actions

Derby provides referential actions. Examples in this section are included to illustrate how to write triggers.

You can choose to use standard SQL referential integrity (using foreign keys) to obtain this functionality, rather than writing triggers. See the "CONSTRAINT clause" topic in the *Derby Reference Manual* for details.

Accessing before and after rows

Many trigger actions need to access the values of the rows being changed.

Such trigger actions need to know one or both of the following:

- The "before" values of the rows being changed (their values before the database event that caused the trigger to fire)
- The "after" values of the rows being changed (the values to which the database event is setting them)

Derby provides transition variables and transition tables for a trigger action to access these values. See "Referencing old and new values: the REFERENCING clause" under the "CREATE TRIGGER statement" topic in the *Derby Reference Manual*.

Examples of trigger actions

This section provides some examples of triggers.

The following trigger action copies a row from the `flights` table into the `flight_history` table whenever any row gets inserted into `flights` and adds the comment "inserted from trig1" in the `status` column of the `flight_history` table.

```
CREATE TRIGGER trig1
  AFTER UPDATE ON flights
  REFERENCING OLD AS UPDATEDROW
  FOR EACH ROW
  INSERT INTO flights_history
    VALUES (UPDATEDROW.FLIGHT_ID, UPDATEDROW.SEGMENT_NUMBER,
            UPDATEDROW.ORIG_AIRPORT, UPDATEDROW.DEPART_TIME,
            UPDATED ROW.DEST_AIRPORT, UPDATEDROW.ARRIVE_TIME,
            UPDATEDROW.MEAL, UPDATEDROW.FLYING_TIME, UPDATEDROW.MILES,
            UPDATEDROW.AIRCRAFT,'inserted from trig1');
```

The following trigger action updates the `FlightAvailability` table after an update of the `flights` table by setting the `flight_id` column to the value of the `flight_id` column in the modified row. The update of the `FlightAvailability` table happens only if the triggering update actually changed the value of FLIGHTS.FLIGHT_ID.

```
CREATE TRIGGER FLIGHTSUPDATE
  AFTER UPDATE ON flights
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  WHEN (OLD.FLIGHT_ID <> NEW.FLIGHT_ID)
  UPDATE FlightAvailability
    SET FLIGHT_ID = NEW.FLIGHT_ID
    WHERE FLIGHT_ID = OLD.FLIGHT_ID;
```

Triggers and exceptions

Exceptions raised by triggers have a statement severity; they roll back the statement that caused the trigger to fire.

This rule applies to nested triggers (triggers that are fired by other triggers). If a trigger action raises an exception (and it is not caught), the transaction on the current connection is rolled back to the point before the triggering event. For example, suppose Trigger A causes Trigger B to fire. If Trigger B throws an exception, the current connection is rolled back to the point before the statement in Trigger A that caused Trigger B to fire. Trigger A is then free to catch the exception thrown by Trigger B and continue with its work. If Trigger A does not throw an exception, the statement that caused Trigger A, as well as any work done in Trigger A, continues until the transaction in the current connection is either committed or rolled back. However, if Trigger A does not catch the exception from Trigger B, it is as if Trigger A had thrown the exception. In that case, the statement that caused Trigger A to fire is rolled back, along with any work done by both of the triggers.

Aborting statements and transactions

You might want a trigger action to be able to abort the triggering statement or even the entire transaction.

Triggers that use the current connection are not permitted to commit or roll back the connection, so how do you do that? The answer is: have the trigger throw an exception, which is by default a statement-level exception (which rolls back the statement). The application-side code that contains the statement that caused the trigger to fire can then roll back the entire connection if desired. Programming triggers in this respect is no different from programming any database-side JDBC method.

Programming Derby-style table functions

Derby lets you create table functions. Table functions are functions which package up external data to look like Derby tables. The external data can be an XML file, a table in a foreign database, a live data feed--in short, any information source that can be presented as a JDBC *ResultSet*.

Derby-style table functions let you efficiently import foreign data into Derby tables. Table functions let you join Derby tables with any of the following data sources:

- XML-formatted reports and logs
- Queries that run in foreign databases
- Streaming data from sensors
- RSS feeds

See "CREATE FUNCTION statement" in the *Derby Reference Manual* for the complete syntax needed to declare Derby-style table functions. The following topics provide information on how to write Java methods which wrap foreign data sources inside *ResultSet*s.

Overview of Derby-style table functions

A Derby-style table function is a method which returns a JDBC *ResultSet*.

Most of the *ResultSet* methods can be written as stubs which simply raise exceptions. However, the Derby-style table function must implement the following *ResultSet* methods:

- *next()*
- *close()*
- *wasNull()*
- *getXXX()* - When invoking a Derby-style table function at runtime, Derby calls a *getXXX()* method on each referenced column. The particular *getXXX()* method is based on the column's data type as declared in the `CREATE FUNCTION` statement. Preferred getXXX() methods for Derby-style table functions explains how Derby selects an appropriate *getXXX()* method. However, nothing prevents application code from calling other *getXXX()* methods on the *ResultSet*. The returned *ResultSet* needs to implement the *getXXX()* methods which Derby will call as well as all *getXXX()* methods which the application will call.

A Derby-style table function is materialized by a public static method which returns a *ResultSet*:

```
public static ResultSet read() {...}
```

The public static method is then bound to a Derby function name:

```
CREATE FUNCTION externalEmployees
()
RETURNS TABLE
(
  employeeId    INT,
  lastName      VARCHAR( 50 ),
  firstName     VARCHAR( 50 ),
  birthday      DATE
)
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
READS SQL DATA
EXTERNAL NAME 'com.example.hrSchema.EmployeeTable.read'
```

To invoke a table function, wrap it in a TABLE constructor in the FROM list of a query. Note that the table alias (in this example "s") is a required part of the syntax:

```
INSERT INTO employees
  SELECT s.*
    FROM TABLE (externalEmployees() ) s;
```

With a normal table function, you must select its entire contents. You can, however, write a restricted table function that lets you limit the rows and columns you select. A restricted table function can improve performance greatly. See Writing restricted table functions for details.

Preferred *getXXX()* methods for Derby-style table functions

While scanning a Derby-style table function, Derby calls a preferred *getXXX()* method for each column, based on the column's data type.

The following table lists the preferred *getXXX()* method for each Derby data type.

Table 4. *getXXX()* methods called for declared SQL types

Column Type Declared by CREATE FUNCTION	*getXXX()* Method Called by Derby
BIGINT	*getLong()*
BLOB	*getBlob()*
CHAR	*getString()*
CHAR FOR BIT DATA	*getBytes()*
CLOB	*getClob()*
DATE	*getDate()*
DECIMAL	*getBigDecimal()*
DOUBLE	*getDouble()*
DOUBLE PRECISION	*getDouble()*
FLOAT	*getDouble()*
INTEGER	*getInt()*
LONG VARCHAR	*getString()*
LONG VARCHAR FOR BIT DATA	*getBytes()*
NUMERIC	*getBigDecimal()*
REAL	*getFloat()*
SMALLINT	*getShort()*
TIME	*getTime()*
TIMESTAMP	*getTimestamp()*
VARCHAR	*getString()*
VARCHAR FOR BIT DATA	*getBytes()*
XML	Not supported

Example Derby-style table function

The following simple table function selects rows from a foreign database.

```
package com.example.hrSchema;

import java.sql.*;

/**
 * Sample Table Function for reading the employee table in an
 * external database.
 */
public class EmployeeTable
{
    public static ResultSet read()
        throws SQLException
    {
        Connection conn = getConnection();
        PreparedStatement ps = conn.prepareStatement(
            "select * from hrSchema.EmployeeTable" );

        return ps.executeQuery();
    }
```

```
    protected static Connection getConnection()
        throws SQLException
    {
        String EXTERNAL_DRIVER = "com.mysql.jdbc.Driver";

        try {
            Class.forName( EXTERNAL_DRIVER );
        }
        catch (ClassNotFoundException e) {
            throw new SQLException( "Could not find class "
                + EXTERNAL_DRIVER );
        }

        Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost/hr?user=root&password=mysql-passwd"
);

        return conn;
    }
}
```

Writing restricted table functions

Restricted table functions are Derby-style table functions which perform more efficiently because they can be told in advance which columns they will be asked to fetch along with simple limits on those columns. This feature exploits the expressiveness of the Java programming language and does not require any extensions to SQL.

A table function returns a rectangular chunk of data. If you use a restricted table function, Derby can tell the table function to return a shorter and narrower rectangle.

Consider the following scan of a table in a foreign database:

```
    select id, firstName, lastName
    from table( foreignDatabaseEmployeeTable() ) s
    where lastName = 'Stone'
```

If *foreignDatabaseEmployeeTable* is a restricted table function, Derby can tell the table function to fetch only the *id*, *firstName*, and *lastName* columns. In addition, Derby can tell the table function that it does not need to scan the entire foreign table; instead, the table function only needs to retrieve information for employees whose last name is "Stone".

Depending on the table function and query, this feature can support 1000X, 1000000X, or even greater performance improvements.

How to use restricted table functions

Creating and using a restricted table function involves the following steps:

1. **Implement** - You must write a class which implements both *java.sql.ResultSet* and the Derby-specific interface *org.apache.derby.vti.RestrictedVTI*. This interface defines an *initScan()* method. When executing a query, Derby uses that method to tell the table function what columns it will have to fetch and what bounds should be applied to those columns in order to reduce the number of rows returned. For the rest of this discussion, this user-written class will be referred to as *MyVTIClass*.
2. **Publish** - You must publish the table function by creating a public static method which returns a *MyVTIClass*. This is important. The Derby compiler must be able to see that the table function returns an object which implements both *java.sql.ResultSet* and *org.apache.derby.vti.RestrictedVTI*.
3. **Declare** - You declare the table function to Derby using the same CREATE FUNCTION syntax you are already familiar with. This syntax does not change.
4. **Invoke** - You then use the table function in a query. When Derby compiles the query, it sees that the return type of the table function implements

org.apache.derby.vti.RestrictedVTI. Armed with this information, at runtime Derby calls the *initScan()* method once before calling any of the *ResultSet* methods.

For example, you would declare the function as follows:

```
public class MyVTIClass implements ResultSet, RestrictedVTI
{
    ...

    public void initScan(java.lang.String[] columnNames,
        org.apache.derby.vti.Restriction restriction )
        throws SQLException {
         ...
    }
}
```

Then you publish the table function method:

```
public static MyVTIClass foreignDatabaseEmployeeTable()
    throws SQLException {
    ...
}
```

Then you declare the table function to Derby:

```
create function foreignDatabaseEmployeeTable()
returns table
(
    id int,
    birthday date,
    taxPayerID varchar( 50 ),
    firstName varchar( 50 ),
    lastName varchar( 50 )
)
language java
parameter style DERBY_JDBC_RESULT_SET
no sql
external name
 'com.example.portal.ForeignQueries.foreignDatabaseEmployeeTable'
```

Finally, you invoke the table function in a query:

```
select id, firstName, lastName
from table( foreignDatabaseEmployeeTable() ) s
where lastName = 'Stone'
```

When you invoke this query, Derby does the following:

- **Prepare** - When Derby prepares the query, Derby sees that the *foreignDatabaseEmployeeTable()* method returns an object which implements *org.apache.derby.vti.RestrictedVTI*. This is all that Derby needs to know in order to compile a plan which takes advantage of this feature.
- **Execute** - When Derby executes the query, Derby calls *initScan()*. In this example, Derby calls *initScan()* with the following arguments:

  ```
  initScan( new String[] { "ID", null, null, "FIRSTNAME", "LASTNAME"
   },
      new Restriction.ColumnQualifier(
          "LASTNAME", ORDER_OP_EQUALS, "Stone" ) )
  ```

  This, in turn, causes the following to happen:

- *Width* - The call to *initScan()* told the table function what columns should be fetched.
- *Length* - The call to *initScan()* told the table function how to filter the rows it returns.
- *Loop* - Derby then calls *MyVTIClass.next()* and retrieves rows until *MyVTIClass.next()* returns false. For each row, Derby calls:
  - *MyVTIClass.getInt(1)* to get the *id* column.
  - *MyVTIClass.getString(4)* to get the *firstName* column.
  - *MyVTIClass.getString(5)* to get the *lastName* column.

Contract

Derby calls *initScan()* before calling any other method on the *ResultSet*. The call to *initScan()* merely passes hints, which the restricted table function can exploit in order to perform better. Derby enforces the restriction outside the table function. Therefore, a restricted table function can still fetch extra columns and can ignore part or all of the restriction set by the call to *initScan()*.

Affected Operations

Compared to ordinary table functions, a restricted table function can perform better in queries involving the following comparisons of its columns to constants:

```
<
<=
=
!=
<>
>
>=
IS NULL
IS NOT NULL
```

In addition, performance gains can be realized for queries involving the following operators on the columns of the restricted table function:

```
LIKE
BETWEEN
```

However, this feature does not boost performance either for the IN operator, or in situations where Derby transforms OR lists into IN lists. See "Or transformations" in *Tuning Derby* for more information.

Writing context-aware table functions

A context-aware table function is able to access context information that is passed in to it from Derby.

Context-aware table functions are useful when both of the following are the case:

- You want to bind a single Java method to many table functions, each of which has a different row shape.
- You are able to determine the row shape, at runtime, from the schema-qualified name of the table function which is being invoked.

A context-aware table function makes use of the *org.apache.derby.vti.AwareVTI* interface and the *org.apache.derby.vti.VTIContext* class. The *VTIContext* class, which can be accessed through the *AwareVTI* interface, provides methods that return the unqualified table function name, the name of the schema which holds the table function, and

the text of the statement which invoked the table function. See the Derby public API documentation for more information about *AwareVTI* and *VTIContext*.

For example, the *ArchiveVTI* table function performs a task which many users have found useful: it provides a union of a main table with a set of archive tables. The archive tables are created at regular intervals. When a new archive table is created, the oldest rows from the main table are moved to the archive table.

To use the *ArchiveVTI* table function, you need to include *derbyTesting.jar* in your classpath along with other Derby jar files.

The following series of commands shows how to use the *archiveVTI* method, which is included in the Derby test code. The source code for the *ArchiveVTI* class is provided in the next topic.

In this example, the method is bound to two table functions; one function returns a three-column table, the other a two-column table.

```
java org.apache.derby.tools.ij
ij version 10.11
ij> connect 'jdbc:derby:memory:db;create=true';
ij> create table t1
(
    keyCol int,
    aCol int,
    bCol int
);
0 rows inserted/updated/deleted
ij> create table t1_archive_001 as select * from t1 with no data;
0 rows inserted/updated/deleted
ij> create table t1_archive_002 as select * from t1 with no data;
0 rows inserted/updated/deleted
ij> insert into t1_archive_002 values ( 1, 100, 1000 ), ( 2, 200, 2000 ),
    ( 3, 300, 3000 );
3 rows inserted/updated/deleted
ij> insert into t1_archive_001 values ( 4, 400, 4000 ), ( 5, 500, 5000 ),
    ( 6, 600, 6000 );
3 rows inserted/updated/deleted
ij> insert into t1 values ( 7, 700, 7000 ), ( 8, 800, 8000 ),
    ( 9, 900, 9000 );
3 rows inserted/updated/deleted
ij> create table t2
(
    keyCol int,
    aCol int
);
0 rows inserted/updated/deleted
ij> create table t2_arc_001 as select * from t2 with no data;
0 rows inserted/updated/deleted
ij> create table t2_arc_002 as select * from t2 with no data;
0 rows inserted/updated/deleted
ij> insert into t2_arc_002 values ( 1, 100 ), ( 2, 200 ),  ( 3, 300 );
3 rows inserted/updated/deleted
ij> insert into t2_arc_001 values ( 4, 400 ), ( 5, 500 ),  ( 6, 600 );
3 rows inserted/updated/deleted
ij> insert into t2 values ( 7, 700 ), ( 8, 800 ), ( 9, 900 );
3 rows inserted/updated/deleted
ij> create function t1( archiveSuffix varchar( 32672 ) ) returns table
(
    keyCol int,
    aCol int,
    bCol int
)
language java parameter style derby_jdbc_result_set reads sql data
external name
'org.apache.derbyTesting.functionTests.tests.lang.ArchiveVTI.archiveVTI';
0 rows inserted/updated/deleted
ij> create function t2( archiveSuffix varchar( 32672 ) ) returns table
```

```
(
    keyCol int,
    aCol int
)
language java parameter style derby_jdbc_result_set reads sql data
external name
'org.apache.derbyTesting.functionTests.tests.lang.ArchiveVTI.archiveVTI';
0 rows inserted/updated/deleted
ij> select * from table( t1( '_ARCHIVE_' ) ) s
where keyCol between 3 and 7
order by keyCol;
KEYCOL     |ACOL       |BCOL
-----------------------------------
3          |300        |3000
4          |400        |4000
5          |500        |5000
6          |600        |6000
7          |700        |7000

5 rows selected
ij> select * from table( t2( '_ARC_' ) ) s
where keyCol between 3 and 7
order by keyCol;
KEYCOL     |ACOL
-----------------------
3          |300
4          |400
5          |500
6          |600
7          |700

5 rows selected
```

ArchiveVTI source code

The code that defines the *archiveVTI* table function is as follows.

```
package org.apache.derbyTesting.functionTests.tests.lang;

import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

import org.apache.derby.vti.AwareVTI;
import org.apache.derby.vti.ForeignTableVTI;
import org.apache.derby.vti.ForwardingVTI;
import org.apache.derby.vti.RestrictedVTI;
import org.apache.derby.vti.Restriction;
import org.apache.derby.vti.VTIContext;

/**
 * <p>
 * This table function acts like a union view on a set of archive tables.
 * The idea is that the old contents of a main table are periodically
 * moved to archive tables whose names start with $tableName$suffix.
 * Each bulk move of rows results in the creation of a new archive table.
 * The archive tables live in the same schema as the main table and have
 * its shape. This table function unions the main table together with all
 * of its archived snapshots. So, for instance, you might have the
 * following set of tables, which this table function unions together:
 * </p>
 *
 * <pre>
 *  T1
 *  T1_ARCHIVE_1
 *  T1_ARCHIVE_2
 *   ...
```

```
 *  T1_ARCHIVE_N
 * </pre>
 *
 * <p>
 * This table function may appear in user documentation. If you change
 * the behavior of this table function, make sure that you adjust the
 * user documentation linked from DERBY-6117.
 * </p>
 */
public class ArchiveVTI extends ForwardingVTI implements AwareVTI,
        RestrictedVTI
{
    ////////////////////////////////////////////////////////////////////
    //
    // CONSTANTS
    //
    ////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    //
    // STATE
    //
    ////////////////////////////////////////////////////////////////////

    private Connection  _connection;
    private String      _archiveSuffix;
    private VTIContext  _vtiContext;
    private ArrayList<String>   _tableNames;
    private int         _tableIdx;

    private String[]    _columnNames;
    private Restriction _restriction;

    ////////////////////////////////////////////////////////////////////
    //
    //  TABLE FUNCTION
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * <p>
     * Entry point for creating an ArchiveVTI which is bound to a Derby
     * table function by a CREATE FUNCTION statement which looks like
     * this:
     * </p>
     *
     * <pre>
     * create function t1( archiveSuffix varchar( 32672 ) ) returns table
     * (
     *     keyCol int,
     *     aCol int,
     *     bCol int
     * )
     * language java parameter style derby_jdbc_result_set reads sql data
     * external name
     *
 'org.apache.derbyTesting.functionTests.tests.lang.ArchiveVTI.archiveVTI'
     * </pre>
     *
     * @param archiveSuffix All of the archive tables have names of the
     *                      form $tablename$archiveSuffix.
     */
    public  static  ArchiveVTI  archiveVTI( String archiveSuffix )
            throws SQLException
    { return new ArchiveVTI( archiveSuffix ); }

    ////////////////////////////////////////////////////////////////////
    //
    // CONSTRUCTOR
    //
```

```
    ////////////////////////////////////////////////////////////////////

    /** Construct from the suffix which flags all of the relevant
     *  tables.
     */
    public  ArchiveVTI( String archiveSuffix )    throws SQLException
    {
        _connection = DriverManager.getConnection(
                "jdbc:default:connection" );
        _archiveSuffix = archiveSuffix;
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  AwareVTI BEHAVIOR
    //
    ////////////////////////////////////////////////////////////////////

    public  VTIContext  getContext() { return _vtiContext; }
    public  void    setContext( VTIContext context )
    { _vtiContext = context; }

    ////////////////////////////////////////////////////////////////////
    //
    // RestrictedVTI BEHAVIOR
    //
    ////////////////////////////////////////////////////////////////////

    public  void    initScan
        ( String[] columnNames, Restriction restriction )
        throws SQLException
    {
        _columnNames = new String[ columnNames.length ];
        System.arraycopy( columnNames, 0, _columnNames, 0,
                columnNames.length );
        _restriction = restriction;
    }

    ////////////////////////////////////////////////////////////////////
    //
    // ResultSet BEHAVIOR
    //
    ////////////////////////////////////////////////////////////////////

    public boolean next()   throws SQLException
    {
        if ( _tableNames == null )
        {
            getTableNames();
            _tableIdx = 0;
            loadResultSet();
        }

        while ( !super.next() )
        {
            _tableIdx++;
            if ( _tableIdx >= _tableNames.size() ) { return false; }
            loadResultSet();
        }

        return true;
    }

    public  void    close() throws SQLException
    {
        if ( getWrappedResultSet() != null )
        {
            getWrappedResultSet().close();
        }
        wrapResultSet( null );
```

```
        _connection = null;
    }

    ////////////////////////////////////////////////////////////////////
    //
    // UTILITY METHODS
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * <p>
     * Get cursors on all the tables which we are going to union
     * together.
     * </p>
     */
    private void    getTableNames() throws SQLException
    {
        _tableNames = new ArrayList<String>();
        _tableNames.add( getContext().vtiTable() );

        DatabaseMetaData    dbmd = getConnection().getMetaData();
        ResultSet   candidates = dbmd.getTables
            ( null, getContext().vtiSchema(), getContext().vtiTable()
              + _archiveSuffix + "%", null );

        while ( candidates.next() )
        {
            _tableNames.add( candidates.getString( "TABLE_NAME" ) );
        }
        candidates.close();
    }

    /**
     * <p>
     * Compile the query against the next table and use its ResultSet
     * until it's drained.
     * </p>
     */
    private void    loadResultSet() throws SQLException
    {
        if ( getWrappedResultSet() != null )
        {
            getWrappedResultSet().close();
        }

        ForeignTableVTI     nextRS = new ForeignTableVTI
            ( getContext().vtiSchema(), _tableNames.get( _tableIdx ),
              getConnection() );
        nextRS.initScan( _columnNames, _restriction );

        wrapResultSet( nextRS );
    }

    /**
     * <p>
     * Get this database session's connection to the database.
     * </p>
     */
    private Connection  getConnection() throws SQLException
    {
        return _connection;
    }

}
```

Optimizer support for Derby-style table functions

This topic explains how to fine-tune the Derby optimizer's decision about where to place a table function in the join order.

By default, the Derby optimizer makes the following assumptions about a table function:

- **Expensive** - It is expensive to create and loop through the rows of the table function. This makes it likely that the optimizer will place the table function in an outer slot of the join order so that it will not be looped through often.
- **Repeatable** - The table function can be instantiated multiple times with the same results. This is probably true for most table functions. However, some table functions may open read-once streams. If the optimizer knows that a table function is repeatable, then the optimizer can place the table function in an inner slot where the function can be invoked multiple times. If a table function is not repeatable, then the optimizer must either place it in the outermost slot or invoke the function once and store its contents in a temporary table.

The user can override this optimizer behavior by giving the optimizer more information. Here's how to do this:

- **No-arg constructor** - The table function's class must have a public constructor whose signature has no arguments.
- **VTICosting** - The class must also implement *org.apache.derby.vti.VTICosting*. This involves implementing the following methods as described in Measuring the cost of Derby-style table functions and Example VTICosting implementation:
  - *getEstimatedCostPerInstantiation()* - This method estimates the cost of invoking the table function and looping through its rows. The returned value adds together two estimates:
    - Empty table - This is the cost of invoking the table function, even if it contains 0 rows. See the description of variable **E** in Measuring the cost of Derby-style table functions.
    - Scanning - This is the cost of looping through all of the rows returned by the table function. See the calculation of **P*N** in Measuring the cost of Derby-style table functions.
  - *getEstimatedRowCount()* - This guesses the number of rows returned by invoking the table function.
  - *supportsMultipleInstantiations()* - This returns false if the table function returns different results when invoked more than once.

Measuring the cost of Derby-style table functions

This topic shows how to measure the cost of a Derby-style table function.

The following formula describes how to estimate the value returned by *VTICosting.getEstimatedCostPerInstantiation()*:

C = I * A

where

- **C** = The estimated **C**ost for creating and running the table function. That is, the value returned by *VTICosting.getEstimatedCostPerInstantiation()*. In general, **C**ost is a measure of time in milliseconds.
- **I** = The optimizer's **I**mprecision. A measure of how skewed the optimizer's estimates tend to be in your particular environment. See below for instructions on how to estimate this **I**mprecision.
- **A** = The **A**ctual time in milliseconds which it takes to create and run this table function.

Calculating the optimizer's imprecision

We treat optimizer **I**mprecision as a constant across the runtime environment. The following formula describes it:

I = O / T

where

- **O** = The **O**ptimizer's estimated cost for a plan.
- **T** = The **T**otal runtime in milliseconds for the plan.

To estimate these values, turn on Derby statistics collection and run the following experiment several times, averaging the results:

- **Select** = Select all of the rows from a big table.
- **Record** = In the statistics output, look for the ResultSet which represents the table scan. That scan has a field labelled "optimizer estimated cost". That's **O**. Now look for the fields in that ResultSet's statistics labelled "constructor time", "open time", "next time", and "close time". Add up all of those fields. That total is **T**.

For example:

```
MAXIMUMDISPLAYWIDTH 7000;

CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);

select * from T;

values SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS();
```

Calculating the actual runtime cost of a table function

The following formula explains how to compute the **A**ctual runtime cost for the table function:

A = (P * N) + E

where

- **P** = The runtime spent **P**er row (in milliseconds).
- **N** = The **N**umber of rows in the table function.
- **E** = The time spent creating an **E**mpty instance of the table function which has no rows in it. Usually, **P * N** dwarfs **E**. That is, the table function instantiation cost is very small compared to the actual cost of looping through the rows. However, for some table functions, **E** may be significant and may dominate the table function's cost when **N** is small.

You may know that **E** is basically 0. If so, you can skip this step. Otherwise, to estimate **E**, turn on Derby statistics collection and run the following experiment several times, averaging the results:

- **Short-circuit** = Short-circuit the next() method of the *ResultSet* returned by your Derby-style table function so that it returns *false* the first time it is called. This makes it appear that the *ResultSet* has no rows.
- **Select** = Select all of the rows from the table function.
- **Record** = In the statistics output, look for the VTIResultSet which represents the table function scan. Add up the values of the fields in that VTIResultSet's statistics labelled "constructor time", "open time", "next time", and "close time". That total is **E**.

To estimate **P**, turn on Derby statistics collection and run the following experiment several times, averaging the results:

- **Select** = Select all of the rows from the table function.
- **Record** = In the statistics output, look for the VTIResultSet which represents the table function scan. Add up the values of the fields in that VTIResultSet's statistics

labelled "constructor time", "open time", "next time", and "close time". Subtract **E** from the result. Now divide by the value of the field "Rows seen". The result is **P**.

Computing the value returned by *getEstimatedCostPerInstantiation()*

Putting all of this together, the following formula describes the value returned by your table function's *VTICosting.getEstimatedCostPerInstantiation()* method.

C = O/T * [(P * N) + E]

Example VTICosting implementation

Once you have measured your table function's cost, you can write the *VTICosting* methods.

Optimizer fine-tuning can be added to the *EmployeeTable* table function shown before in Example Derby-style table function:

```
package com.example.hrSchema;

import java.io.Serializable;
import java.sql.*;

import org.apache.derby.vti.VTICosting;
import org.apache.derby.vti.VTIEnvironment;

/**
 * Tuned table function.
 */
public class TunedEmployeeTable extends EmployeeTable
    implements VTICosting
{
    public TunedEmployeeTable() {}

    public double getEstimatedRowCount( VTIEnvironment optimizerState )
        throws SQLException
    {
        return getRowCount( optimizerState );
    }

    public double getEstimatedCostPerInstantiation(
        VTIEnvironment optimizerState ) throws SQLException
    {
        double I = 100.0;  // optimizer imprecision
        double P = 10.0;   // cost per row in milliseconds
        double E = 0.0;    // cost of instantiating the external
                           //   ResultSet
        double N = getRowCount( optimizerState );

        return I * ( ( P * N ) + E );
    }

    public boolean supportsMultipleInstantiations(
        VTIEnvironment optimizerState ) throws SQLException
    {
        return true;
    }

    ////////////////////////////////////////////////////////////////////

    private double  getRowCount( VTIEnvironment optimizerState )
        throws SQLException
    {
        String ROW_COUNT_KEY = "rowCountKey";
        Double estimatedRowCount = (Double) getSharedState(
            optimizerState, ROW_COUNT_KEY );

        if ( estimatedRowCount == null )
```

```
        {
            Connection        conn = getConnection();
            PreparedStatement ps = conn.prepareStatement(
                "select count(*) from hrSchema.EmployeeTable" );
            ResultSet         rs = ps.executeQuery();

            rs.next();
            estimatedRowCount = new Double( rs.getDouble( 1 ) );

            setSharedState( optimizerState, ROW_COUNT_KEY,
                estimatedRowCount );

            rs.close();
            ps.close();
            conn.close();
        }

        return estimatedRowCount.doubleValue();
    }

    private Serializable getSharedState(
        VTIEnvironment optimizerState, String key )
    {
        return (Serializable) optimizerState.getSharedState( key );
    }

    private void setSharedState( VTIEnvironment optimizerState,
        String key, Serializable value )
    {
        optimizerState.setSharedState( key, value );
    }
}
```

Programming user-defined types

Derby allows you to create user-defined types. A user-defined type is a serializable Java class whose instances are stored in columns. The class must implement the *java.io.Serializable* interface, and it must be declared to Derby by means of a CREATE TYPE statement.

The key to designing a good user-defined type is to remember that data evolves over time, just like code. A good user-defined type has version information built into it. This allows the user-defined data to upgrade itself as the application changes. For this reason, it is a good idea for a user-defined type to implement *java.io.Externalizable* and not just *java.io.Serializable*. Although the SQL standard allows a Java class to implement only *java.io.Serializable*, this is bad practice for the following reasons:

- **Recompilation** - If the second version of your application is compiled on a different platform from the first version, then your serialized objects may fail to deserialize. This problem and a possible workaround are discussed in the "Version Control" section near the end of this Serialization Primer and in the last paragraph of the header comment for *java.io.Serializable*.
- **Evolution** - Your tools for evolving a class which simply implements *java.io.Serializable* are very limited.

Fortunately, it is easy to write a version-aware UDT which implements *java.io.Serializable* and can evolve itself over time. For example, here is the first version of such a class:

```
package com.example.types;

import java.io.*;
import java.math.*;

public class Price implements Externalizable
```

```
{
    // initial version id
    private static final int FIRST_VERSION = 0;

    public String currencyCode;
    public BigDecimal amount;

    // zero-arg constructor needed by Externalizable machinery
    public Price() {}

    public Price( String currencyCode, BigDecimal amount )
    {
        this.currencyCode = currencyCode;
        this.amount = amount;
    }

    // Externalizable implementation
    public void writeExternal(ObjectOutput out) throws IOException
    {
        // first write the version id
        out.writeInt( FIRST_VERSION );

        // now write the state
        out.writeObject( currencyCode );
        out.writeObject( amount );
    }

    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException
    {
        // read the version id
        int oldVersion = in.readInt();
        if ( oldVersion < FIRST_VERSION ) {
            throw new IOException( "Corrupt data stream." );
        }
        if ( oldVersion > FIRST_VERSION ) {
            throw new IOException( "Can't deserialize from the future."
 );
        }

        currencyCode = (String) in.readObject();
        amount = (BigDecimal) in.readObject();
    }
}
```

After this, it is easy to write a second version of the user-defined type which adds a new field. When old versions of `Price` values are read from the database, they upgrade themselves on the fly. Changes are shown in **bold**:

```
package com.example.types;

import java.io.*;
import java.math.*;
import java.sql.*;

public class Price implements Externalizable
{
    // initial version id
    private static final int FIRST_VERSION = 0;
    private static final int TIMESTAMPED_VERSION = FIRST_VERSION + 1;

    private static final Timestamp DEFAULT_TIMESTAMP = new Timestamp( 0L
 );

    public String currencyCode;
    public BigDecimal amount;
    public Timestamp timeInstant;
```

```
    // 0-arg constructor needed by Externalizable machinery
    public Price() {}

    public Price( String currencyCode, BigDecimal amount,
                  Timestamp timeInstant )
    {
        this.currencyCode = currencyCode;
        this.amount = amount;
        this.timeInstant = timeInstant;
    }

    // Externalizable implementation
    public void writeExternal(ObjectOutput out) throws IOException
    {
        // first write the version id
        out.writeInt( TIMESTAMPED_VERSION );

        // now write the state
        out.writeObject( currencyCode );
        out.writeObject( amount );
        out.writeObject( timeInstant );
    }

    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException
    {
        // read the version id
        int oldVersion = in.readInt();
        if ( oldVersion < FIRST_VERSION ) {
            throw new IOException( "Corrupt data stream." );
        }
        if ( oldVersion > TIMESTAMPED_VERSION ) {
            throw new IOException( "Can't deserialize from the future."
);
        }

        currencyCode = (String) in.readObject();
        amount = (BigDecimal) in.readObject();

        if ( oldVersion >= TIMESTAMPED_VERSION ) {
            timeInstant = (Timestamp) in.readObject();
        }
        else {
            timeInstant = DEFAULT_TIMESTAMP;
        }
    }
}
```

An application needs to keep its code in sync across all tiers. This is true for all Java code which runs both in the client and in the server. This is true for functions and procedures which run in multiple tiers. It is also true for user-defined types which run in multiple tiers. The programmer should code defensively for the case when the client and server are running different versions of the application code. In particular, the programmer should write defensive serialization logic for user-defined types so that the application gracefully handles client/server version mismatches.

Programming user-defined aggregates

Derby allows you to create custom aggregate operators, called user-defined aggregates (UDAs).

A UDA is a Java class that implements the *org.apache.derby.agg.Aggregator* interface.

The *org.apache.derby.agg.Aggregator* interface extends *java.io.Serializable*, so you must make sure that all of the state of your UDA is serializable. A UDA may be serialized

to disk when it performs grouped aggregation over a large number of groups. That is, intermediate results may be serialized to disk for a query like the following:

```
SELECT a, myAggregate( b ) FROM myTable GROUP BY a
```

The serialization will fail if the UDA contains non-serializable fields.

The following class provides an aggregate that computes the median value from a list of objects. This is a generic class. Its parameter must be a linear (*Comparable*) type.

```
import java.util.ArrayList;
import java.util.Collections;
import org.apache.derby.agg.Aggregator;

public class Median<V extends Comparable<V>>
        implements Aggregator<V,V,Median<V>>
{
    private ArrayList<V> _values;

    public Median() {}

    public void init() { _values = new ArrayList<V>(); }

    public void accumulate( V value ) { _values.add( value ); }

    public void merge( Median<V> other )
    {
        _values.addAll( other._values );
    }

    public V terminate()
    {
        Collections.sort( _values );

        int count = _values.size();

        if ( count == 0 ) { return null; }
        else { return _values.get( count/2 ); }
    }
}
```

Using this generic class, we can declare UDAs for all of the sortable Derby data types. For example:

```
create derby aggregate intMedian for int external name 'Median';
create derby aggregate varcharMedian for varchar( 32672 ) external name
  'Median';
```

We can then use these UDAs just like built-in Derby aggregates:

```
create table intValues( a int, b int );
create table varcharValues( a int, b varchar( 32672 ) );
insert into intValues values ( 1, 1 ), ( 1, 10 ), ( 1, 100 ),
  ( 1, 1000 ), ( 2, 5 ), ( 2, 50 ), ( 2, 500 ), ( 2, 5000 );
insert into varcharValues values ( 1, 'a' ), ( 1, 'ab' ), ( 1, 'abc' ),
  ( 2, 'a' ), ( 2, 'aa' ), ( 2, 'aaa' );

select a, intMedian( b ) from intValues group by a;
A          |2
-----------------------
1          |100
2          |500

select varcharMedian( b ) from varcharValues;
1
---
aaa
```

See "CREATE DERBY AGGREGATE statement" in the *Derby Reference Manual* for more information.

Controlling Derby application behavior

This section looks at some advanced Derby application concepts.

The JDBC connection and transaction model

Session and transaction capabilities for SQL are handled through JDBC routines, not by SQL commands.

JDBC defines a system session and transaction model for database access. A *session* is the duration of one connection to the database and is handled by a JDBC *Connection* object.

Connections

A *Connection* object represents a connection with a database.

Within the scope of one *Connection*, you access only a single Derby database. (Database-side JDBC routines can allow you to access more than one database in some circumstances.) A single application might allow one or more *Connections* to Derby, either to a single database or to many different databases, provided that all the databases are within the same system.

With DriverManager, you use the connection URL as an argument to get the *getConnection* method to specify which database to connect to and other details.

The following example shows an application establishing three separate connections to two different databases in the current system.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sample");
System.out.println("Connected to database sample");
conn.setAutoCommit(false);
Connection conn2 = DriverManager.getConnection(
    "jdbc:derby:newDB;create=true");
System.out.println("Created AND connected to newDB");
conn2.setAutoCommit(false);
Connection conn3 = DriverManager.getConnection(
    "jdbc:derby:newDB");
System.out.println("Got second connection to newDB");
conn3.setAutoCommit(false);
```

A *Connection* object has no association with any specific thread; during its lifetime, any number of threads might have access to it, as controlled by the application.

Statements

To execute SQL statements against a database, an application uses *Statements* (`java.sql.Statement`) and *PreparedStatements* (`java.sql.PreparedStatement`), or *CallableStatements* (`java.sql.CallableStatement`) for stored procedures.

Because *PreparedStatement* extends *Statement* and *CallableStatement* extends *PreparedStatement*, this section refers to both as *Statements*. *Statements* are obtained from and are associated with a particular *Connection*.

ResultSets and Cursors

Executing a *Statement* that returns values gives a `ResultSet` (`java.sql.ResultSet`), allowing the application to obtain the results of the statement.

Only one *ResultSet* can be open for a particular *Statement* at any time, as per the JDBC specification.

Thus, executing a *Statement* automatically closes any open *ResultSet* generated by an earlier execution of that *Statement*.

For this reason, you must use a different *Statement* to update a cursor (a named *ResultSet*) from the one used to generate the cursor.

The names of open cursors must be unique within a *Connection*.

Nested connections

SQL statements can include routine invocations. If these routines interact with the database, they must use a *Connection*.

Transactions

A *transaction* is a set of one or more SQL statements that make up a logical unit of work that you can either commit or roll back and that will be recovered in the event of a system failure.

All the statements in the transaction are *atomic*. A transaction is associated with a single *Connection* object (and database). A transaction cannot span *Connections* (or databases).

Derby permits schema and data manipulation statements (DML) to be intermixed within a single transaction. If you create a table in one transaction, you can also insert into it in that same transaction. A schema manipulation statement (DDL) is not automatically committed when it is performed, but participates in the transaction within which it is issued. Because DDL requires exclusive locks on system tables, keep transactions that involve DDL short.

Transactions when auto-commit is disabled

When auto-commit is disabled, you use a *Connection* object's *commit* and *rollback* methods to commit or roll back a transaction.

The *commit* method makes permanent the changes resulting from the transaction and releases locks. The *rollback* method undoes all the changes resulting from the transaction and releases locks. A transaction encompasses all the SQL statements executed against a single *Connection* object since the last *commit* or *rollback*.

You do not need to explicitly begin a transaction. You implicitly end one transaction and begin a new one after disabling auto-commit, changing the isolation level, or after calling *commit* or *rollback*.

Committing a transaction also closes all *ResultSet* objects excluding the *ResultSet* objects associated with cursors with holdability `true`. The default holdability of the cursors is `true` and *ResultSet* objects associated with them need to be closed explicitly. A commit will not close such *ResultSet* objects. It also releases any database locks currently held by the *Connection*, whether or not these objects were created in different threads.

Any outstanding violations of deferred constraints will be checked at commit time, so the call to *Connection.commit* may throw an exception. See "CONSTRAINT clause" in the *Derby Reference Manual* for information about deferrable constraints.

Using auto-commit

A new connection to a Derby database is in auto-commit mode by default, as specified by the JDBC standard.

Auto-commit mode means that when a statement is completed, the method *commit* is called on that statement automatically. Auto-commit in effect makes every SQL statement a transaction. The commit occurs when the statement completes or the next statement is executed, whichever comes first. In the case of a statement returning a forward only `ResultSet`, the statement completes when the last row of the `ResultSet` has been retrieved or the `ResultSet` has been closed explicitly. In the case of a statement returning a scrollable `ResultSet`, the statement completes only when the `ResultSet` has been closed explicitly.

Some applications might prefer to work with Derby in auto-commit mode; some might prefer to work with auto-commit turned off. You should be aware of the implications of using either model.

You should be aware of the following when you use auto-commit:

- *Cursors*

  You cannot use auto-commit if you do any positioned updates or deletes (that is, an update or delete statement with a WHERE CURRENT OF clause) on cursors which have the `ResultSet.CLOSE_CURSORS_AT_COMMIT` holdability value set.

  Auto-commit automatically closes cursors that are explicitly opened with the `ResultSet.CLOSE_CURSORS_AT_COMMIT` value, when you do any in-place updates or deletes.

  An updatable cursor declared to be held across commit (this is the default value) can execute updates and issue multiple commits before closing the cursor. After an explicit or implicit commit, a holdable forward-only cursor must be repositioned with a call to the `next` method before it can accessed again. In this state, the only other valid operation besides calling `next` is calling `close`.
- *Database-side JDBC routines (routines using nested connections)*

  You cannot execute functions within SQL statements if those functions perform a commit or rollback on the current connection. Since in auto-commit mode all SQL statements are implicitly committed, Derby turns off auto-commit during execution of database-side routines and turns it back on when the statement completes.

  Routines that use nested connections are not permitted to turn auto-commit on or off.
- *Table-level locking and the SERIALIZABLE isolation level*

  When an application uses table-level locking and the SERIALIZABLE isolation level, all statements that access tables hold at least shared table locks. Shared locks prevent other transactions that update data from accessing the table. A transaction holds a lock on a table until the transaction commits. *So even a SELECT statement holds a shared lock on a table until its connection commits and a new transaction begins.*

The following table summarizes how applications behave with auto-commit on or off.

Table 5. Application behavior with auto-commit on or off

Topic	Auto-Commit On	Auto-Commit Off
Transactions	Each statement is a separate transaction.	Commit() or rollback() completes a transaction.
Database-side JDBC routines (routines that use nested connections)	Auto-commit is turned off.	Works (no explicit commits or rollbacks are allowed).

Topic	Auto-Commit On	Auto-Commit Off
Updatable cursors	Works for holdable cursors; does not work for non-holdable cursors.	Works.
Multiple connections accessing the same data	Works.	Works. Lower concurrency when applications use SERIALIZABLE isolation mode and table-level locking.
Updatable ResultSets	Works.	Works.
Savepoints	Does not work.	Works.

Turning off auto-commit

You can disable auto-commit with the *Connection* class's *setAutoCommit* method.

```
conn.setAutoCommit(false);
```

Explicitly closing Statements, ResultSets, and Connections

You should explicitly close `Statements`, `ResultSets`, and `Connections` when you no longer need them, unless you declare them in a `try`-with-resources statement (available in JDK 7 and after).

Connections to Derby are resources external to an application, and the garbage collector will not close them automatically.

To close a `Statement`, `ResultSet`, or `Connection` object that is not declared in a `try`-with-resources statement, use its `close` method. If auto-commit is disabled, you must explicitly commit or roll back active transactions before you close the connection.

Statements, result sets, and connections extend `AutoCloseable` in JDK 7 and after. If you declare a connection in a `try`-with-resources statement and there is an error that the code does not catch, the JRE will attempt to close the connection automatically.

Note that a transaction-severity or higher exception causes Derby to abort an in-flight transaction. But a statement-severity exception does NOT roll back the transaction. Also note that Derby throws an exception if an attempt is made to close a connection with an in-flight transaction. Suppose now that a `Connection` is declared in a `try`-with-resources statement, a transaction is in-flight, and an unhandled statement-severity error occurs inside the `try`-with-resources block. In this situation, Derby will raise a follow-on exception as the JRE exits the *try*-with-resources block. (For details on error severity levels, see the documentation of the `derby.stream.error.logSeverityLevel` property in the *Derby Reference Manual*.)

It is therefore always best to catch errors inside the `try`-with-resources block and to either roll back or commit, as appropriate, to ensure that there is no pending transaction when leaving the `try`-with-resources block. This action also improves application portability, since DBMSs differ in their semantics when trying to close a connection with a pending transaction.

Statement versus transaction runtime rollback

When an SQL statement generates an exception, this exception results in a *runtime rollback*. A runtime rollback is a system-generated rollback of a statement or transaction by Derby, as opposed to an explicit `rollback` call from your application.

Extremely severe exceptions, such as disk-full errors, shut down the system, and the transaction is rolled back when the database is next booted. Severe exceptions, such as deadlock, cause transaction rollback; Derby rolls back all changes since the beginning of the transaction and implicitly begins a new transaction. Less severe exceptions, such as syntax errors, result in statement rollback; Derby rolls back only changes made by the statement that caused the error. The application developer can insert code to explicitly roll back the entire transaction if desired.

Derby supports partial rollback through the use of savepoints. See Using savepoints for more information.

Using savepoints

The *Savepoint* interface contains methods to set, release, or roll back a transaction to designated savepoints. Once a savepoint has been set, the transaction can be rolled back to that savepoint without affecting preceding work. Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction.

Setting and rolling back to a savepoint

The *Connection.setSavepoint* method sets a savepoint within the current transaction. The *Connection.rollback* method is overloaded to take a savepoint argument.

The code example below inserts a row into a table, sets the savepoint `svpt1`, and then inserts a second row. When the transaction is later rolled back to `svpt1`, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing '1' will be added to TABLE1.

```
conn.setAutoCommit(false); // Autocommit must be off to use savepoints.
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES(1)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("S1");
rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES (2)");
...
conn.rollback(svpt1);
...
conn.commit();
```

Releasing a savepoint

The method *Connection.releaseSavepoint* takes a *Savepoint* object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation will cause an *SQLException* to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints created after the savepoint in question.

Rules for savepoints

The savepoint cannot be set within a batch of statements to enable partial recovery. If a savepoint is set any time before the method *executeBatch* is called, it is set before any of the statements that have been added to the batch are executed.

A savepoint can be reused after it has been released explicitly (by issuing a release of the savepoint) or implicitly (by issuing a connection commit/rollback to that savepoint or to a savepoint declared earlier than that savepoint).

It is possible to nest savepoints, but only in an embedded environment.

Result set and cursor mechanisms

A result set maintains a cursor, which points to its current row of data. It can be used to step through and process the rows one by one.

In Derby, any SELECT statement generates a cursor which can be controlled by a *java.sql.ResultSet* object. Derby does not support SQL's DECLARE CURSOR language construct to create cursors. However, Derby supports positioned deletes and positioned updates of updatable cursors.

Simple non-updatable result sets

This example is an excerpt from a sample JDBC application that generates a result set with a simple SELECT statement and then processes the rows.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sample");
Statement s = conn.createStatement();
s.execute("set schema 'SAMP'");
//note that autocommit is on--it is on by default in JDBC
ResultSet rs = s.executeQuery(
    "SELECT empno, firstnme, lastname, salary, bonus, comm "
     + "FROM samp.employee");
/** a standard JDBC ResultSet. It maintains a
  *  cursor that points to the current row of data. The cursor
  *  moves down one row each time the method next() is called.
  *  You can scroll one way only--forward--with the next()
  *  method. When auto-commit is on, after you reach the
  *  last row the statement is considered completed
  *  and the transaction is committed.
  */
System.out.println( "last name" + "," + "first name" + ": earnings");
/* here we are scrolling through the result set
with the next() method.*/
while (rs.next()) {
    // processing the rows
    String firstnme = rs.getString("FIRSTNME");
    String lastName = rs.getString("LASTNAME");
    BigDecimal salary = rs.getBigDecimal("SALARY");
    BigDecimal bonus = rs.getBigDecimal("BONUS");
    BigDecimal comm = rs.getBigDecimal("COMM");
    System.out.println( lastName + ", " + firstnme + ": "
                        + (salary.add(bonus.add(comm))));
}
rs.close();
// once we've iterated through the last row,
// the transaction commits automatically and releases
//shared locks
s.close();
```

Updatable result sets

Updatable result sets in Derby can be updated by using result set update methods (`updateRow()`, `deleteRow()`, and `insertRow()`), or by using positioned update or delete queries.

Both scrollable and non-scrollable result sets can be updatable in Derby.

If the query which was executed to create the result set is not updatable, Derby will downgrade the concurrency mode to `ResultSet.CONCUR_READ_ONLY`, and add a warning about this on the `ResultSet`. The compilation of the query fails if the result set cannot be updatable, and contains a `FOR UPDATE` clause.

Positioned updates and deletes can be performed if the query contains `FOR UPDATE` or if the concurrency mode for the result set is `ResultSet.CONCUR_UPDATABLE`.

To use the result set update methods, the concurrency mode for the result set must be `ResultSet.CONCUR_UPDATABLE`. The query does not need to contain `FOR UPDATE` to use these methods.

Updatable cursors lock the current row with an update lock when positioned on the row, regardless of isolation level. Therefore, to avoid excessive locking of rows, only use concurrency mode `ResultSet.CONCUR_UPDATABLE` or the `FOR UPDATE` clause when you actually need to update the rows. For more information about locking, see Types and scope of locks in Derby systems.

Requirements for updatable result sets

Only specific SELECT statements- simple accesses of a single table-allow you to update or delete rows as you step through them.

For more information, see "SELECT statement" and "FOR UPDATE clause" in the *Derby Reference Manual*.

Forward only updatable result sets

A forward only updatable result set maintains a cursor which can only move in one direction (forward), and also update rows.

To create a forward only updatable result set, the statement has to be created with concurrency mode `ResultSet.CONCUR_UPDATABLE` and type `ResultSet.TYPE_FORWARD_ONLY`.
Note: The default type is `ResultSet.TYPE_FORWARD_ONLY`.

Example of using `ResultSet.updateXXX()` + `ResultSet.updateRow()` to update a row:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
  "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
  "FROM EMPLOYEE");

while (uprs.next()) {
    int newBonus = uprs.getInt("BONUS") + 100;
    uprs.updateInt("BONUS", newBonus);
    uprs.updateRow();
}
```

Example of using `ResultSet.deleteRow()` to delete a row:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
  "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
  "FROM EMPLOYEE");

while (uprs.next()) {
    if (uprs.getInt("WORKDEPT")==300) {
       uprs.deleteRow();
    }
}
```

Visibility of changes

- After an update or delete is made on a forward only result set, the result set's cursor is no longer on the row just updated or deleted, but immediately before the next row in the result set (it is necessary to move to the next row before

any further row operations are allowed). This means that changes made by `ResultSet.updateRow()` and `ResultSet.deleteRow()` are never visible.
- If a row has been inserted, i.e using `ResultSet.insertRow()` it may be visible in a forward only result set.

Conflicting operations

The current row of the result set cannot be changed by other transactions, since it will be locked with an update lock. Result sets held open after a commit have to move to the next row before allowing any operations on it.

Some conflicts may prevent the result set from doing updates/deletes:
- If the current row is deleted by a statement in the same transaction, calls to `ResultSet.updateRow()` will cause an exception, since the cursor is no longer positioned on a valid row.

Scrollable updatable result sets

A scrollable updatable result set maintains a cursor which can both scroll and update rows.

Derby only supports scrollable insensitive result sets. To create a scrollable insensitive result set which is updatable, the statement has to be created with concurrency mode `ResultSet.CONCUR_UPDATABLE` and type `ResultSet.TYPE_SCROLL_INSENSITIVE`.

Example of using result set update methods to update a row:

```
 Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
 ResultSet uprs = stmt.executeQuery(
   "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
   "FROM EMPLOYEE");

 uprs.absolute(5); // update the fifth row
 int newBonus = uprs.getInt("BONUS") + 100;
 uprs.updateInt("BONUS", newBonus);
 uprs.updateRow();
```

Example of using `ResultSet.deleteRow()` to delete a row:

```
 Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
 ResultSet uprs = stmt.executeQuery(
   "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
   "FROM EMPLOYEE");

 uprs.last();
 uprs.relative(-5); // moves to the 5th from the last row
 uprs.deleteRow();
```

Visibility of changes
- Changes caused by other statements, triggers and other transactions (others) are considered as other changes, and are not visible in scrollable insensitive result sets.
- Own updates and deletes are visible in Derby's scrollable insensitive result sets. **Note:** Derby handles changes made using positioned updates and deletes as own changes, so when made via a result set's cursor such changes are also visible in that result set.
- Rows inserted to the table may become visible in the result set.
- `ResultSet.rowDeleted()` returns true if the row has been deleted using the cursor or result set. It does not detect deletes made by other statements or

transactions. Note that the method will also work for result sets with concurrency CONCUR_READ_ONLY if the underlying result set is FOR UPDATE and a cursor was used to delete the row.
- `ResultSet.rowUpdated()` returns true if the row has been updated using the cursor or result set. It does not detect updates made by other statements or transactions. Note that the method will also work for result sets with concurrency CONCUR_READ_ONLY if the underlying result set is FOR UPDATE and a cursor was used to update the row.
- **Note:** Both `ResultSet.rowUpdated()` and `ResultSet.rowDeleted()` return true if the row first is updated and later deleted.

Please be aware that even if changes caused by others are not visible in the result set, SQL operations, including positioned updates, which access the current row will read and use the row data as it is in the database, not as it is reflected in the result set.

Conflicting operations

A conflict may occur in scrollable insensitive result sets if a row is updated/deleted by another committed transaction, or if a row is updated by another statement in the same transaction. The row which the cursor is positioned on is locked, however once it moves to another row, the lock may be released depending on transaction isolation level. This means that rows in the scrollable insensitive result set may have been updated/deleted by other transactions after they were fetched.

Since the result set is **insensitive**, it will not detect the changes made by others. When doing updates using the result set, conflicting changes on the columns being changed will be overwritten.

Some conflicts may prevent the result set from doing updates/deletes:
- The row has been deleted after it was read into the result set: Scrollable insensitive result sets will give a warning with *`SQLState`* `01001` .
- The table has been compressed: Scrollable insensitive result sets will give a warning with *`SQLState`* `01001`. A compress conflict may happen if the cursor is held over a commit. This is because the table intent lock is released on commit, and not reclaimed until the cursor moves to another row.

To avoid conflicts with other transactions, you may increase the transaction isolation level to repeatable read or serializable. This will make the transaction hold locks on the rows which have been read until it commits.
Note: When you use holdable result sets, be aware that the locks will be released on commit, and conflicts may occur regardless of isolation level. You should probably avoid using holdable result sets if your application relies on transactional behavior for the result set.

Inserting rows with updatable result sets

Updatable result set can be used to insert rows to the table, by using `ResultSet.insertRow()`.

When inserting a row, each column in the insert row that does not allow null as a value and does not have a default value must be given a value using the appropriate update method. If the inserted row satisfies the query predicate, it may become visible in the result set.

Example of using `ResultSet.insertRow()` to insert a row:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
  "SELECT firstname, lastname, workdept, bonus " +
  "FROM employee");
uprs.moveToInsertRow();
uprs.updateString("FIRSTNAME", "Andreas");
```

```
  uprs.updateString("LASTNAME", "Korneliussen");
  uprs.updateInt("WORKDEPT", 123);
  uprs.insertRow();
  uprs.moveToCurrentRow();
```

Naming or accessing the name of a cursor

There is no SQL language command to *assign* a name to a cursor. You can use the JDBC *setCursorName* method to assign a name to a *ResultSet* that allows positioned updates and deletes.

You assign a name to a *ResultSet* with the *setCursorName* method of the *Statement* interface. You assign the name to a cursor before executing the *Statement* that will generate it.

```
Statement s3 = conn.createStatement();
// name the statement so we can reference the result set
// it generates
s3.setCursorName("UPDATABLESTATEMENT");
// we will be able to use the following statement later
// to access the current row of the cursor
// a result set needs to be obtained prior to using the
// WHERE CURRENT syntax
ResultSet rs = s3.executeQuery("select * from
    FlightBookings FOR UPDATE of number_seats");
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE FlightBookings SET number_seats = ? " +
       "WHERE CURRENT OF UPDATABLESTATEMENT");
```

Typically, you do not assign a name to the cursor, but let the system generate one for you automatically. You can determine the system-generated cursor name of a *ResultSet* generated by a SELECT statement using the *ResultSet* class's *getCursorName* method.

```
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE employee SET bonus = ? WHERE CURRENT OF "+
        Updatable.getCursorName());
```

Extended updatable result set example

The following code example shows how to program updatable result sets.

```

Connection conn = DriverManager.getConnection("jdbc:derby:sample");
conn.setAutoCommit(false);

// Create the statement with concurrency mode CONCUR_UPDATABLE
// to allow result sets to be updatable
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE,
                                      ResultSet.CLOSE_CURSORS_AT_COMMIT);
// Updatable statements have some requirements
// for example, select must be on a single table
ResultSet uprs = stmt.executeQuery(
    "SELECT FIRSTNME, LASTNAME, WORKDEPT, BONUS " +
    "FROM EMPLOYEE FOR UPDATE of BONUS"); // Only bonus can be updated

String theDept="E21";

while (uprs.next()) {
    String firstnme = uprs.getString("FIRSTNME");
    String lastName = uprs.getString("LASTNAME");
    String workDept = uprs.getString("WORKDEPT");
    BigDecimal bonus = uprs.getBigDecimal("BONUS");
    if (workDept.equals(theDept)) {
        // if the current row meets our criteria,
        // update the updatable column in the row
```

```
        uprs.updateBigDecimal("BONUS",
 bonus.add(BigDecimal.valueOf(250L)));
        uprs.updateRow();
        System.out.println("Updating bonus for employee:" +
                           firstnme + lastName);
    }
}
conn.commit(); // commit the transaction
// close object
uprs.close();
stmt.close();
// Close connection if the application does not need it any more
conn.close();
```

Result sets and auto-commit

Except for the result sets associated with holdable cursors, issuing a commit will cause all result sets on your connection to be closed.

The JDBC application is not required to have auto-commit off when using update methods on updatable result set, even if the result set is not holdable. Positioned updates and deletes cannot be used in combination with autocommit and non-holdable result sets.

Scrollable result sets

JDBC provides two types of result sets that allow you to scroll in either direction or to move the cursor to a particular row. Derby supports one of these types: scrollable insensitive result sets (`ResultSet.TYPE_SCROLL_INSENSITIVE`).

When you use a result set of type of type `ResultSet.TYPE_SCROLL_INSENSITIVE`, Derby materializes rows from the first one in the result set up to the one with the biggest row number as the rows are requested. The materialized rows will be backed to disk if necessary, to avoid excessive memory usage.

Insensitive result sets, in contrast to sensitive result sets, cannot see changes made by others on the rows which have been materialized. Derby allows updates of scrollable insensitive result sets; see Visibility of changes, which also explains visibility of own changes.

Note: Derby does not support result sets of type `ResultSet.TYPE_SCROLL_SENSITIVE`.

```
//autocommit does not have to be off because even if
//we accidentally scroll past the last row, the implicit commit
//on the the statement will not close the result set because result sets
//are held over commit by default
conn.setAutoCommit(false);
Statement s4 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_READ_ONLY);
s4.execute("set schema 'SAMP'");
ResultSet scroller=s4.executeQuery(
    "SELECT sales_person, region, sales FROM sales " +
    "WHERE sales > 8 ORDER BY sales DESC");
if (scroller.first()) { // One row is now materialized
    System.out.println("The sales rep who sold the highest number" +
                       " of sales is " +
                       scroller.getString("SALES_PERSON"));
} else {
    System.out.println("There are no rows.");
}
scroller.beforeFirst();
scroller.afterLast();   // By calling afterlast(), all rows will be
 materialized
```

```
scroller.absolute(3);
if (!scroller.isAfterLast()) {
    System.out.println("The employee with the third highest number " +
                       "of sales is " +
                       scroller.getString("SALES_PERSON") + ", with " +
                       scroller.getInt("SALES") + " sales");
}
if (scroller.isLast()) {
    System.out.println("There are only three rows.");
}
if (scroller.last()) {
    System.out.println("The least highest number " +
                       "of sales of the top three sales is: " +
                       scroller.getInt("SALES"));
}
scroller.close();
s4.close();
conn.commit()
conn.close();
System.out.println("Closed connection");
```

Holdable result sets

The holdable result set feature permits an application to keep result sets open after implicit or explicit commits. By default, the cursor controlled by the result set is held open after a commit.

Note: Derby also supports non-holdable result sets.

When you create a statement, you can specify that the result set will be automatically closed when a commit occurs. Result sets are automatically closed when a transaction aborts, whether or not they have been specified to be held open.

To specify whether a result set should be held open after a commit takes place, supply one of the following *ResultSet* parameters to the `Connection` method `createStatement`, `prepareStatement`, or `prepareCall`:

- *CLOSE_CURSORS_AT_COMMIT*

  Result sets are closed when an implicit or explicit commit is performed.
- *HOLD_CURSORS_OVER_COMMIT*

  Result sets are held open when a commit is performed, implicitly or explicitly. This is the default behavior.

The method *Statement.getResultSetHoldability()* indicates whether a result set generated by the Statement object stays open or closes, upon commit. See the *Derby Reference Manual* for more information.

When an implicit or explicit commit occurs, result sets that hold cursors open behave as follows:

- Open result sets remain open. Non-scrollable result sets becomes positioned before the next logical row of the result set. Scrollable insensitive result sets keep their current position.
- When the session is terminated, the result set is closed and destroyed.
- All locks are released, including locks protecting the current cursor position.
- For non-scrollable result sets, immediately following a commit, the only valid operations that can be performed on the *ResultSet* object are:
  - positioning the result set to the next row with `ResultSet.next()`.
  - closing the result set with `ResultSet.close()`.

When a rollback or rollback to savepoint occurs, either explicitly or implicitly, the following behavior applies:

- All open result sets are closed.
- All locks acquired during the unit of work are released.

Note: Holdable result sets do not work with XA transactions in Derby. When working with XA transactions, the result set should be opened with holdability `ResultSet.CLOSE_CURSORS_AT_COMMIT`.

Holdable result sets and autocommit

When autocommit is on, a positioned update or delete statement will automatically cause the transaction to commit.

If the result set has holdability `ResultSet.CLOSE_CURSORS_AT_COMMIT`, combined with autocommit on, Derby gives an exception on positioned updates and deletes because the cursor is closed immediately before the positioned statement is commenced, as mandated by JDBC. In contrast, no such implicit commit is done when using result set updates methods.

Non-holdable result set example

The following example uses *Connection.createStatement* to return a *`ResultSet`* that will close after a commit is performed.

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_READ_ONLY,
                     ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

Locking, concurrency, and isolation

This section discusses topics pertinent to multi-user systems, in which concurrency is important.

Derby is configured by default to work well for multi-user systems. For single-user systems, you might want to tune your system so that it uses fewer resources; see Lock granularity.

Isolation levels and concurrency

Derby provides four transaction isolation levels. Setting the transaction isolation level for a connection allows a user to specify how severely the user's transaction should be isolated from other transactions.

For example, it allows you to specify whether transaction A is allowed to make changes to data that have been viewed by transaction B before transaction B has committed.

A connection determines its own isolation level, so JDBC provides an application with a way to specify a level of transaction isolation. It specifies four levels of transaction isolation. The higher the transaction isolation, the more care is taken to avoid conflicts; avoiding conflicts sometimes means locking out transactions. Lower isolation levels thus allow greater concurrency.

Inserts, updates, and deletes always behave the same no matter what the isolation level is. Only the behavior of select statements varies.

To set isolation levels you can use the JDBC *Connection.setTransactionIsolation* method or the SQL SET ISOLATION statement.

If there is an active transaction, the network client driver always commits the active transaction, whether you use the JDBC *Connection.setTransactionIsolation* method or the SQL SET ISOLATION statement. It does this even if the method call or statement does not actually change the isolation level (that is, if it sets the isolation

level to its current value). The embedded driver also always commits the active transaction if you use the SET ISOLATION statement. However, if you use the *Connection.setTransactionIsolation* method, the embedded driver commits the active transaction only if the call to *Connection.setTransactionIsolation* actually changes the isolation level.

The names of the isolation levels are different, depending on whether you use a JDBC method or SQL statement. The following table shows the equivalent names for isolation levels whether they are set through the JDBC method or an SQL statement.

Table 6. Mapping of JDBC transaction isolation levels to Derby isolation levels

Isolation Levels for JDBC	Isolation Levels for SQL
Connection.TRANSACTION_READ_UNCOMMITTED (ANSI level 0)	UR, DIRTY READ, READ UNCOMMITTED
Connection.TRANSACTION_READ_COMMITTED (ANSI level 1)	CS, CURSOR STABILITY, READ COMMITTED
Connection.TRANSACTION_REPEATABLE_READ (ANSI level 2)	RS
Connection.TRANSACTION_SERIALIZABLE (ANSI level 3)	RR, REPEATABLE READ, SERIALIZABLE

These levels allow you to avoid particular kinds of transaction anomalies, which are described in the following table.

Table 7. Transaction anomalies

Anomaly	Example

Dirty Reads
 	
A dirty read happens when a transaction reads data that is being modified by another transaction that has not yet committed.
	Transaction A begins.
	
	`UPDATE employee SET salary = 31650 WHERE empno = '000090'`
	
	Transaction B begins.
	
	`SELECT * FROM employee`
	
	(Transaction B sees data updated by transaction A. Those updates have not yet been committed.)

Nonrepeatable Reads
 	
Nonrepeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Nonrepeatable reads can occur when other transactions are modifying data that a transaction is reading.
	Transaction A begins.
	
	`SELECT * FROM employee WHERE empno = '000090'`
	
	Transaction B begins.
	
	`UPDATE employee SET salary = 30100 WHERE empno = '000090'`
	
	(Transaction B updates rows viewed by transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.

Phantom Reads	Transaction A begins.

Anomaly	Example

Records that appear in a set being read by another transaction. Phantom reads can occur when other transactions insert rows that would satisfy the WHERE clause of another transaction's statement.	`SELECT * FROM employee WHERE salary > 30000`
	Transaction B begins.
	`INSERT INTO employee (empno, firstnme, midinit, lastname, job, salary) VALUES ('000350', 'NICK', 'A','GREEN','LEGAL COUNSEL',35000)`
	Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

The transaction isolation level is a way of specifying whether these transaction anomalies are allowed. The transaction isolation level thus affects the quantity of data locked by a particular transaction. In addition, a DBMS's locking schema might also affect whether these anomalies are allowed. A DBMS can lock either the entire table or only specific rows in order to prevent transaction anomalies.

The following table shows which anomalies are possible under the various locking schemas and isolation levels.

Table 8. When transaction anomalies are possible

Isolation Level	Table-Level Locking	Row-Level Locking
TRANSACTION_READ_UNCOMMITTED	Dirty reads, nonrepeatable reads, and phantom reads possible	Dirty reads, nonrepeatable reads, and phantom reads possible
TRANSACTION_READ_COMMITTED	Nonrepeatable reads and phantom reads possible	Nonrepeatable reads and phantom reads possible
TRANSACTION_REPEATABLE_READ	Phantom reads not possible because entire table is locked	Phantom reads possible
TRANSACTION_SERIALIZABLE	None	None

The following *java.sql.Connection* isolation levels are supported:

- `TRANSACTION_SERIALIZABLE`

  RR, `SERIALIZABLE`, or `REPEATABLE READ` from SQL.

  TRANSACTION_SERIALIZABLE means that Derby treats the transactions as if they occurred serially (one after the other) instead of concurrently. Derby issues locks to prevent all the transaction anomalies listed in Transaction anomalies from occurring. The type of lock it issues is sometimes called a *range lock*.
- `TRANSACTION_REPEATABLE_READ`

  RS from SQL.

TRANSACTION_REPEATABLE_READ means that Derby issues locks to prevent only dirty reads and nonrepeatable reads, but not phantoms. It does not issue range locks for selects.

- `TRANSACTION_READ_COMMITTED`

  `CS` or `CURSOR STABILITY` from SQL.

  TRANSACTION_READ_COMMITTED means that Derby issues locks to prevent only dirty reads, not all the transaction anomalies listed in Transaction anomalies.

  TRANSACTION_READ_COMMITTED is the default isolation level for transactions.
- `TRANSACTION_READ_UNCOMMITTED`

  `UR`, `DIRTY READ`, or `READ UNCOMMITTED` from SQL.

  For a SELECT INTO, FETCH with a read-only cursor, full select used in an INSERT, full select/subquery in an UPDATE/DELETE, or scalar full select (wherever used), READ UNCOMMITTED allows:

  - Any row that is read during the unit of work to be changed by other application processes.
  - Any row that was changed by another application process to be read even if the change has not been committed by the application process.

  For other operations, the rules that apply to READ COMMITTED also apply to READ UNCOMMITTED.

Configuring isolation levels

If a connection does not specify its isolation level, it inherits the default isolation level for the Derby system. The default value is `CS`.

When set to `CS`, the connection inherits the TRANSACTION_READ_COMMITTED isolation level. When set to `RR`, the connection inherits the TRANSACTION_SERIALIZABLE isolation level, when set to `RS`, the connection inherits the TRANSACTION_REPEATABLE_READ isolation level, and when set to `UR`, the connection inherits the TRANSACTION_READ_UNCOMMITTED isolation level.

To override the inherited default, use the methods of *java.sql.Connection*.

In addition, a connection can change the isolation level of the transaction within an SQL statement. For more information, see "SET ISOLATION statement" in the *Derby Reference Manual*. You can use the WITH clause to change the isolation level for the current statement only, not the transaction. For information about the WITH clause, see "SELECT statement" in the *Derby Reference Manual*.

In all cases except when you change the isolation level using the WITH clause, changing the isolation level commits the current transaction. In most cases, the current transaction is committed even if you set the isolation level in a way that does not change it (that is, if you set it to its current value). See Isolation levels and concurrency for details.

Note: For information about how to choose a particular isolation level, see "Shielding users from Derby class-loading events" in *Tuning Derby* and Multi-thread programming tips.

Lock granularity

Derby can be configured for *table-level* locking. With table-level locking, when a transaction locks data in order to prevent any transaction anomalies, it always locks the entire table, not just those rows being accessed.

By default, Derby is configured for row-level locking. Row-level locking uses more memory but allows greater concurrency, which works better in multi-user systems. Table-level locking works best with single-user applications or read-only applications.

You typically set lock granularity for the entire Derby system, not for a particular application. However, at runtime, Derby may escalate the lock granularity for a particular transaction from row-level locking to table-level locking for performance reasons. You have some control over the threshold at which this occurs. For information on turning off row-level locking, see "*derby.storage.rowLocking*" in the *Derby Reference Manual*. For more information about automatic lock escalation, see "About the system's selection of lock granularity" and "Transaction-based lock escalation" in *Tuning Derby*. For more information on tuning your Derby system, see "Tuning databases and applications," also in *Tuning Derby*.

Types and scope of locks in Derby systems

There are several types of locks available in Derby systems, including exclusive, shared, and update locks.

Exclusive locks

When a statement modifies data, its transaction holds an *exclusive* lock on data that prevents other transactions from accessing the data.

This lock remains in place until the transaction holding the lock issues a commit or rollback. Table-level locking lowers concurrency in a multi-user system.

Shared locks

When a statement reads data without making any modifications, its transaction obtains a *shared lock* on the data.

Another transaction that tries to read the same data is permitted to read, but a transaction that tries to update the data will be prevented from doing so until the shared lock is released. How long this shared lock is held depends on the isolation level of the transaction holding the lock. Transactions using the TRANSACTION_READ_COMMITTED isolation level release the lock when the transaction steps through to the next row. Transactions using the TRANSACTION_SERIALIZABLE or TRANSACTION_REPEATABLE_READ isolation level hold the lock until the transaction is committed, so even a SELECT can prevent updates if a commit is never issued. Transactions using the TRANSACTION_READ_UNCOMMITTED isolation level do not request any locks.

Update locks

When a user-defined update cursor (created with the FOR UPDATE clause or by using concurrency mode `ResultSet.CONCUR_UPDATABLE`) reads data, its transaction obtains an *update* lock on the data.

If the user-defined update cursor updates the data, the update lock is converted to an exclusive lock. If the cursor does not update the row, when the transaction steps through to the next row, transactions using the TRANSACTION_READ_COMMITTED isolation level release the lock. (For update locks, the TRANSACTION_READ_UNCOMMITTED isolation level acts the same way as TRANSACTION_READ_COMMITTED.)

Update locks help minimize deadlocks.

Lock compatibility

The following table shows the compatibility between lock types. "Yes" means that the lock types are compatible, while "No" means that they are incompatible.

Table 9. Lock Compatibility Matrix

Lock Type	Shared	Update	Exclusive
Shared	Yes	Yes	No
Update	Yes	No	No
Exclusive	No	No	No

Scope of locks

The amount of data locked by a statement can vary.

Table locks

A statement can lock the *entire table*.

Table-level locking systems always lock entire tables.

Row-level locking systems can lock entire tables if the WHERE clause of a statement cannot use an index. For example, UPDATES that cannot use an index lock the entire table.

Row-level locking systems can lock entire tables if a high number of single-row locks would be less efficient than a single table-level lock. Choosing table-level locking instead of row-level locking for performance reasons is called *lock escalation*. For more information about this topic, see "About the system's selection of lock granularity" and "Transaction-based lock escalation" in *Tuning Derby*.

Single-row locks

A statement can lock only *a single row* at a time.

For row-level locking systems:

- For TRANSACTION_REPEATABLE_READ isolation, the locks are released at the end of the transaction.
- For TRANSACTION_READ_COMMITTED isolation, Derby locks rows only as the application steps through the rows in the result. The current row is locked. The row lock is released when the application goes to the next row.
- For TRANSACTION_SERIALIZABLE isolation, however, Derby locks the whole set before the application begins stepping through.
- For TRANSACTION_READ_UNCOMMITTED, no row locks are requested.

Derby locks single rows for INSERT statements, holding each row until the transaction is committed. If there is an index associated with the table, the previous key is also locked.

Range locks

A statement can lock *a range of rows* (range lock).

For row-level locking systems:

- For *any* isolation level, Derby locks *all the rows in the result* plus an entire range of rows for updates or deletes.
- For the TRANSACTION_SERIALIZABLE isolation level, Derby locks all the rows in the result plus an entire range of rows in the table for SELECTs to prevent nonrepeatable reads and phantoms.

For example, if a SELECT statement specifies rows in the *Employee* table where the *salary* is BETWEEN two values, the system can lock more than just the actual rows it returns in the result. It also must lock the entire *range* of rows between those two values to prevent another transaction from inserting, deleting, or updating a row within that range.

An index must be available for a range lock. If one is not available, Derby locks the entire table.

The following table summarizes the types and scopes of locking.

Table 10. Types and scopes of locking

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
Connection.TRANSACTION_READ_UNCOMMITTED (SQL: UR)	For SELECT statements, table-level locking is never requested using this isolation level. For other statements, same as for TRA	SELECT statements get no locks. For other statements, same as for TRANSACTION_RE D_COMMITTED
Connection.TRANSACTION_READ_COMMITTED (SQL: CS)	SELECT statements get a shared lock on the entire table. The locks are released when the user closes the *ResultSet*. Other statements get exclusive locks on the entire table, which are released when the transaction commits.	SELECTs lock and release single rows as the user steps through the *ResultSet*. UPDATEs and DELETEs get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).
Connection.TRANSACTION_REPEATABLE_READ (SQL: RS)	Same as for TRA	SELECT statements get shared locks on the rows that satisfy the WHERE clause (but do not prevent inserts into this range). UPDATEs and DELETEs get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows). IALIZABLE

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
Connection.TRANSACTION_SERIALIZABLE (SQL: RR)	SELECT statements get a shared lock on the entire table. Other statements get exclusive locks on the entire table, which are released when the transaction commits.	SELECT statements get shared locks on a range of rows. UPDATE and DELETE statements get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).

Notes on locking

In addition to the locks already described, foreign key lookups require briefly held shared locks on the referenced table (row or table, depending on the configuration).

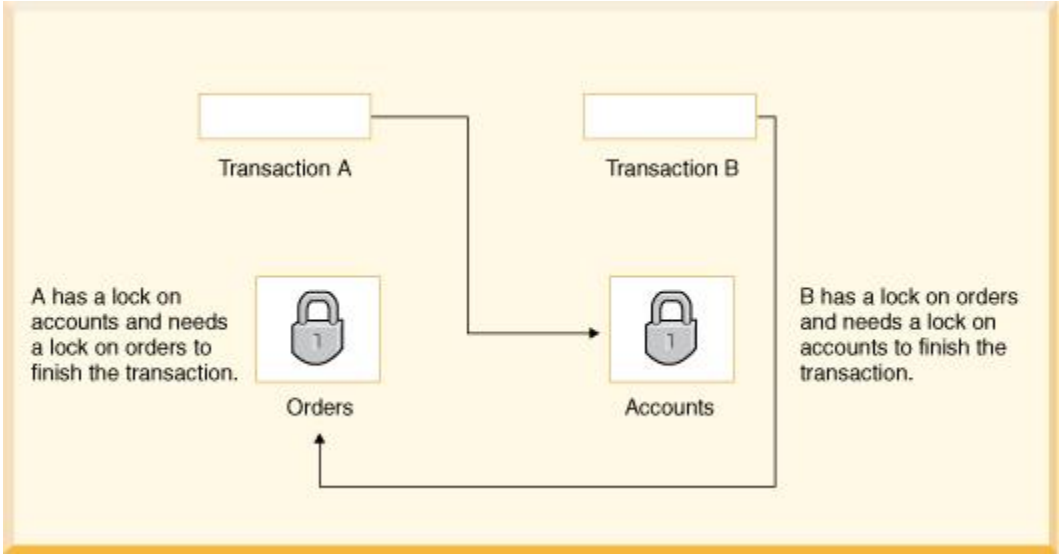
The table and examples in this section do not take performance-based lock escalation into account. Remember that the system can choose table-level locking for performance reasons.

Deadlocks

In a database, a deadlock is a situation in which two or more transactions are waiting for one another to give up locks.

For example, Transaction A might hold a lock on some rows in the *Accounts* table and needs to update some rows in the *Orders* table to finish. Transaction B holds locks on those very rows in the *Orders* table but needs to update the rows in the *Accounts* table held by Transaction A. Transaction A cannot complete its transaction because of the lock on *Orders*. Transaction B cannot complete its transaction because of the lock on *Accounts*. All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions. The following figure shows this situation.

Figure 6. A deadlock where two transactions are waiting for one another to give up locks



Avoiding deadlocks

Using both row-level locking and the TRANSACTION_READ_COMMITTED isolation level makes it likely that you will avoid deadlocks (both settings are Derby defaults). However, deadlocks are still possible.

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

The appropriate use of indexes can also help you to avoid deadlocks, since indexes make table scans less likely and reduce the number of locks obtained. For more information, see "CREATE INDEX statement" in the *Derby Reference Manual* and the topics under "Avoiding table scans of large tables" in *Tuning Derby*.

Another tool available to you is the LOCK TABLE statement. A transaction can attempt to lock a table in exclusive mode when it starts to prevent other transactions from getting shared locks on a table. For more information, see "LOCK TABLE statement" in the *Derby Reference Manual*.

Deadlock detection

When a transaction waits more than a specific amount of time to obtain a lock (called the deadlock timeout), Derby can detect whether the transaction is involved in a deadlock.

When Derby analyzes such a situation for deadlocks it tries to determine how many transactions are involved in the deadlock (two or more). Usually aborting one transaction breaks the deadlock. Derby must pick one transaction as the victim and abort that transaction; it picks the transaction that holds the fewest number of locks as the victim, on the assumption that transaction has performed the least amount of work. (This may not be the case, however; the transaction might have recently been escalated from row-level locking to table locking and thus hold a small number of locks even though it has done the most work.)

When Derby aborts the victim transaction, it receives a deadlock error (an *SQLException* with an *SQLState* of 40001). The error message gives you the transaction IDs, the statements, and the status of locks involved in a deadlock situation.

```
ERROR 40001: A lock could not be obtained due to a deadlock,
cycle of locks & 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
```

For information on configuring when deadlock checking occurs, see Configuring deadlock detection and lock wait timeouts.

Note: Deadlocks are detected only within a single database. Deadlocks across multiple databases are not detected. Non-database deadlocks caused by Java synchronization primitives are not detected by Derby.

Lock wait timeouts

Even if a transaction is not involved in a deadlock, it might have to wait a considerable amount of time to obtain a lock because of a long-running transaction or transactions holding locks on the tables it needs.

In such a situation, you might not want a transaction to wait indefinitely. Instead, you might want the waiting transaction to abort, or *time out*, after a reasonable amount of time, called a *lock wait timeout*.

Configuring deadlock detection and lock wait timeouts

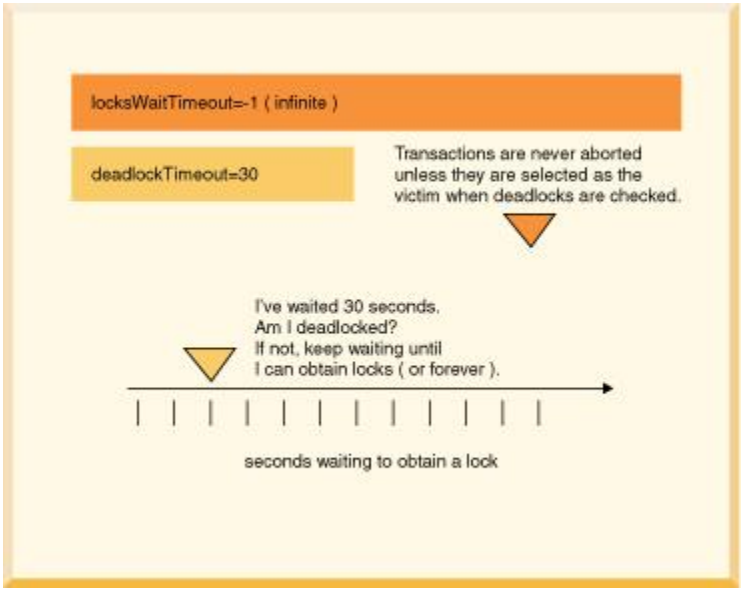
You configure the amount of time a transaction waits before Derby does any deadlock checking with the *derby.locks.deadlockTimeout* property.

You configure the amount of time a transaction waits before timing out with the *derby.locks.waitTimeout* property. When configuring your database or system, you should consider these properties together. For example, in order for any deadlock checking to occur, the *derby.locks.deadlockTimeout* property must be set to a value lower than the *derby.locks.waitTimeout* property. If it is set to a value equal to or higher than the *derby.locks.waitTimeout*, the transaction times out before Derby does any deadlock checking.

By default, *derby.locks.waitTimeout* is set to 60 seconds. -1 is the equivalent of no wait timeout. This means that transactions never time out, although Derby can choose a transaction as a deadlock victim.

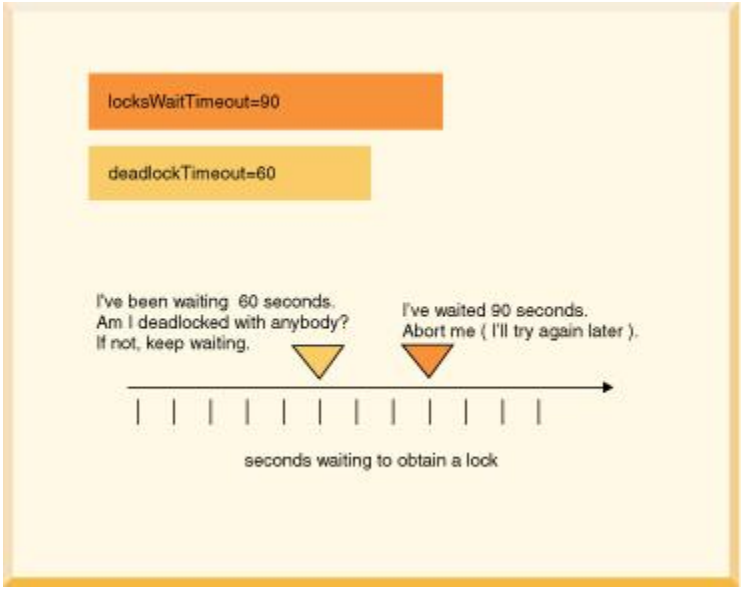
In the following figure, *derby.locks.deadlockTimeout* is set to 30 seconds, while *derby.locks.waitTimeout* has no limit.

Figure 7. Configuration with deadlock checking after 30 seconds and no lock wait timeouts



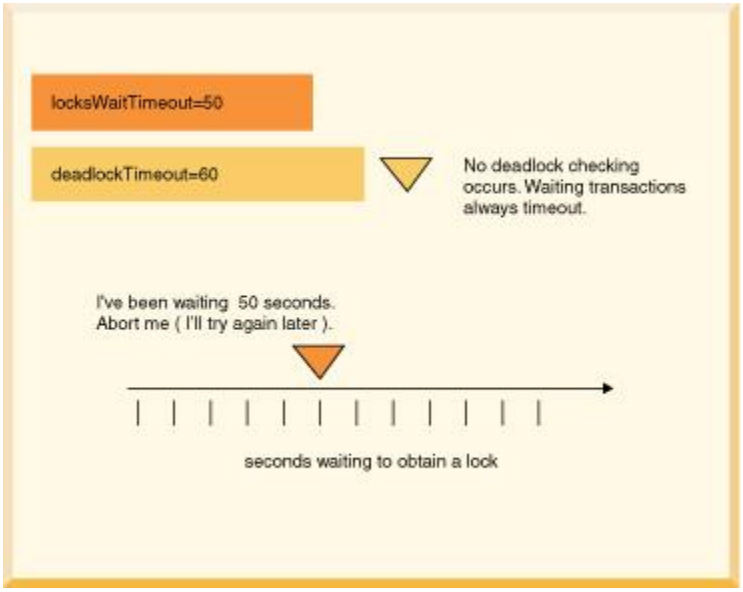
In the following figure, *derby.locks.deadlockTimeout* is set to 60 seconds, while *derby.locks.waitTimeout* is set to 90 seconds.

Figure 8. Configuration with deadlock checking after 60 seconds and lock wait timeout at 90 seconds



In the following figure, *derby.locks.deadlockTimeout* is set to 60 seconds, while *derby.locks.waitTimeout* is set to 50 seconds, lower than the deadlock timeout limit.

Figure 9. Configuration with no deadlock checking and a 50-second lock wait timeout



Debugging deadlocks

If deadlocks occur frequently in your multi-user system with a particular application, you might need to do some debugging.

Derby provides the SYSCS_DIAG.LOCK_TABLE diagnostic table to help you debug deadlocks. This diagnostic table shows all of the locks that are currently held in the Derby database. You can reference the SYSCS_DIAG.LOCK_TABLE diagnostic table directly in a statement.

For example:

```
SELECT * FROM SYSCS_DIAG.LOCK_TABLE
```

When the SYSCS_DIAG.LOCK_TABLE diagnostic table is referenced in a statement, a snapshot of the lock table is taken.

For more information about how to use this table, see "SYSCS_DIAG.LOCK_TABLE diagnostic table" in the *Derby Reference Manual*.

You can also set the property *derby.locks.deadlockTrace* to dump additional information to the *derby.log* file about any deadlocks that occur on your system. See *Derby Reference Manual* for more information on this property. Also see "Monitoring deadlocks" in the *Derby Server and Administration Guide*.

Additional general information about diagnosing locking problems can be found in the Derby Wiki at http://wiki.apache.org/db-derby/LockDebugging.

Programming applications to handle deadlocks

When you configure your system for deadlock and lockwait timeouts and an application could be chosen as a victim when the transaction times out, you should program your application to handle them.

To do this, test for *SQLExceptions* with *SQLStates* of 40001 (deadlock timeout) or 40XL1 (lockwait timeout).

In the case of a deadlock you might want to re-try the transaction that was chosen as a victim. In the case of a lock wait timeout, you probably do not want to do this right away.

The following code is one example of how to handle a deadlock timeout.

```
/// if this code might encounter a deadlock,
```

```
// put the whole thing in a try/catch block
// then try again if the deadlock victim exception
// was thrown
try {
    s6.executeUpdate(
         "UPDATE employee " +
         "SET bonus = 625 "
         "WHERE empno='000150'");
    s6.executeUpdate("UPDATE project " +
         "SET respemp = '000150' " +
         "WHERE projno='IF1000'");
}
// note: do not catch such exceptions in database-side methods;
// catch such exceptions only at the outermost level of
// application code.
// See Database-side JDBC routines and SQLExceptions.
catch (SQLException se) {
    if (se.getSQLState().equals("40001")) {
        // it was chosen as a victim of a deadlock.
        // try again at least once at this point.
        System.out.println( "Will try the transaction again.");
        s6.executeUpdate("UPDATE employee " +
        "SET bonus = 625 " +
        "WHERE empno='000150'");
        s6.executeUpdate("UPDATE project " +
        "SET respemp = 000150 " +
        "WHERE projno='IF1000'");
    }
    else throw se;
}
```

Working with multiple connections to a single database

This section discusses deploying Derby so that many connections can exist to a single database.

Deployment options and threading and connection modes

A database can be available to multiple connections in several situations.

- Multiple applications access a single database (possible only when Derby is running inside a server framework).
- A single application has more than one *Connection* to the same database.

The way you deploy Derby affects the ways applications can use multi-threading and connections, as shown in the following table.

Table 11. Threading and connection modes

Connection Mode	Embedded	Server
Multi-Threaded From an application, using a *singleConnection* to a Derby database and issuing requests against that connection in multiple threads.	Supply a single *Connection* object to separate threads. Derby ensures that only one operation is applied at a time for consistency. Server frameworks automatically manage multi-threaded operations.	Server frameworks can automatically multi-thread operations. Remote client applications can multi-thread if desired.
Multi-Connection	Create individual connections within a single application and use the	Remote client applications can establish the multiple connections desired.

Connection Mode	Embedded	Server
From an application, using multiple connections to a Derby database and issuing requests against those connections on multiple threads.	appropriate connection for each JDBC request. The connections can all be to the same database, or can be to different databases in the same Derby system.	
Multi-User Multiple applications (or JVMs) accessing the same Derby database. Each user application has its own connection or connections to the database.	Not possible. Only one application can access a database at a time, and only one application can access a specific system at a time.	Only one server should access a database at a time. Multiple remote client applications can access the same server, and thus can access the same database at the same time through that server.

Multi-user database access

Multi-user database access is possible if Derby is running inside a server framework.

If more than one client application tries to modify the same data, the connection that gets the table first gets the lock on the data (either specific rows or the entire table). The second connection has to wait until the first connection commits or rolls back the transaction in order to access the data. If two connections are only querying and not modifying data, they can both access the same data at the same time because they can each get a shared lock.

Multiple connections from a single application

A single application can work with multiple *Connections* to the same database and assign them to different threads.

You can avoid concurrency and deadlock problems in your application in several ways:

- Use the *TRANSACTION_READ_COMMITTED* isolation level and turn on row-level locking (the defaults).
- Beware of deadlocks caused by using more than one *Connection* in a single thread (the most obvious case). For example, if the thread tries to update the same table from two different `Connections`, a deadlock can occur.
- Assign *Connections* to threads that handle discrete tasks. For example, do not have two threads update the *Hotels* table. Have one thread update the *Hotels* table and a different one update the *Groups* table.
- If threads access the same tables, commit transactions often.
- Multi-threaded Java applications have the ability to self-deadlock without even accessing a database, so beware of that too.
- Use nested connections to share the same lock space.

Working with multiple threads sharing a single connection

JDBC allows you to share a single *Connection* among multiple threads.

Pitfalls of sharing a connection among threads

Here is a review of the potential pitfalls of sharing a single *Connection* among multiple threads.

- Committing or rolling back a transaction closes all open *ResultSet* objects and currently executing *Statements*, unless you are using held cursors.

  If one thread commits, it closes the *Statements* and *ResultSets* of all other threads using the same connection.
- Executing a *Statement* automatically closes any existing open *ResultSet* generated by an earlier execution of that *Statement*.

  If threads share *Statements*, one thread could close another's *ResultSet.*

In many cases, it is easier to assign each thread to a distinct *Connection*. If thread *A* does database work that is not transactionally related to thread *B*, assign them to different *Connections*. For example, if thread *A* is associated with a user input window that allows users to delete hotels and thread *B* is associated with a user window that allows users to view city information, assign those threads to different *Connections*. That way, when thread *A* commits, it does not affect any *ResultSets* or *Statements* of thread B.

Another strategy is to have one thread do queries and another thread do updates. Queries hold shared locks until the transaction commits in SERIALIZABLE isolation mode; use READ_COMMITTED instead.

Yet another strategy is to have only one thread do database access. Have other threads get information from the database access thread.

Multiple threads are permitted to share a *Connection*, *Statement*, or *ResultSet*. However, the application programmer must ensure that one thread does not affect the behavior of the others.

Recommended Practices

Here are some tips for avoiding unexpected behavior:

- Avoid sharing *Statements* (and their `ResultSets`) among threads.
- Each time a thread executes a *Statement*, it should process the results before relinquishing the `Connection`.
- Each time a thread accesses the *Connection*, it should consistently commit or not, depending on application protocol.
- Have one thread be the "managing" database *Connection* thread that should handle the higher-level tasks, such as establishing the `Connection`, committing, rolling back, changing `Connection` properties such as auto-commit, closing the `Connection`, shutting down the database (in an embedded environment), and so on.
- Close *ResultSets* and `Statements` that are no longer needed in order to release resources.

Multi-thread programming tips

You may be sharing a *Connection* among multiple threads because you have experienced poor concurrency using separate transactions.

Here are some tips for increasing concurrency:

- Use row-level locking.
- Use the *TRANSACTION_READ_COMMITTED* isolation level.
- Avoid queries that cannot use indexes; they require locking of all the rows in the table (if only very briefly) and might block an update.

In addition, some programmers might share a statement among multiple threads to avoid the overhead of each thread's having its own. Using the single statement cache, threads can share the same statement from *different connections*. For more information, see "Using the statement cache" in *Tuning Derby*.

Example of threads sharing a statement

This example shows what can happen if two threads try to share a single *Statement*.

```
PreparedStatement ps = conn.prepareStatement(
    "UPDATE account SET balance =  balance + ? WHERE id = ?");
/* now assume two threads T1,T2 are given this
java.sql.PreparedStatement object and that the following events
happen in the order shown (pseudojava code)*/
T1 - ps.setBigDecimal(1, 100.00);
T1 - ps.setLong(2, 1234);
T2 -  ps.setBigDecimal(1, -500.00);
// *** At this point the prepared statement has the parameters
// -500.00 and 1234
// T1 thinks it is adding 100.00 to account 1234 but actually
// it is subtracting 500.00
T1 - ps.executeUpdate();
T2 - ps.setLong(2, 5678);
// T2 executes the correct update
 T2 - ps.executeUpdate();
/* Also, the auto-commit mode of the connection can lead
to some strange behavior.*/
```

If it is absolutely necessary, the application can get around this problem with Java synchronization.

If the threads each obtain their own *PreparedStatement* (with identical text), their `setXXX` calls do not interfere with each other. Moreover, Derby is able to share the same compiled query plan between the two statements; it needs to maintain only separate state information. However, there is the potential for confusion in regard to the timing of the *commit*, since a single *commit* commits all the statements in a transaction.

Working with database threads in an embedded environment

As a rule, do not use *Thread.interrupt()* calls to signal possibly waiting threads that are also accessing a database, because Derby may catch the interrupt and close the connection to the database. Use *wait* and *notify* calls instead.

There are also special considerations when working with more than one database thread in an application, as described in Working with multiple threads sharing a single connection.

When queries, batches, and statements that wait for database locks run longer than expected, you can use interrupts to stop them. If you do, the connection will be closed and an exception will be thrown.

If you design an application whose database threads may see interrupts, you should plan for the following behavior:

- If a thread is interrupted and the interrupt status flag is not cleared before entering a Derby JDBC call, or if the thread is interrupted while inside a Derby JDBC call, the connection that is experiencing the interrupt will be terminated in the following situations:
  - If a query fetches rows from a database table after the interrupt has occurred
  - If the execution of a new element in a batched statement is attempted after the interrupt has occurred
  - If an interrupt is received while a transaction is waiting for a lock

If the connection is terminated, the application thread will experience the following consequences:

- The JDBC call will raise an *SQLException* with state "08000" ("Connection closed by unknown interrupt").
- Outstanding transactional work on that connection will be rolled back, and all of its locks will be released.
- The connection cannot be used to execute any further JDBC calls.

On return from the JDBC call, the *Thread.isInterrupted()* method of the thread will return *true*, whether or not an exception terminating the connection was thrown. That is, even if Derby does not heed an interrupt, the flag will remain set on exit from the JDBC call.

- All other connections will remain open. This includes other connections which the interrupted thread may be using. These connections will remain open at least until the thread tries to use one of its other connections. If the thead has neglected to clear its interrupted status flag, this connection is also subject to termination as described above.
- The application should normally be prepared to catch the 08000 exceptions, discard the dead connection, clear the interrupted status of the thread, and then restart the transaction in a new connection.

Working with Derby SQLExceptions in an application

JDBC generates exceptions that are refined subtypes of the type *java.sql.SQLException*.

To see the exceptions generated by Derby, retrieve and process the *SQLExceptions* in a catch block.

Information provided in SQL Exceptions

Derby provides the message, *SQLState* values, and error codes in SQL exceptions.

You can use methods of *java.lang.Throwable* to view the message issued by a SQL exception, including the *SQLState* and error messages.

Alternatively, you can use the *getSQLState* and *getMessage* methods to view the *SQLState* and error messages, and you can use *getErrorCode* to see the error code. The error code defines the severity of the error and is not unique to each exception.
Note: Severity is not standardized in Derby. Applications should not depend on the severity returned from SQL exceptions.

Applications should also check for and process *java.sql.SQLWarning*s, which are processed in a similar way. Derby issues an *SQLWarning* if the *create=true* attribute is specified and the database already exists.

Example of processing SQLExceptions

A single error can generate more than one *SQLException*. Usually, but not always, a call to the *printStackTrace* method displays all these exceptions.

To ensure that all exceptions are displayed, use a loop and the *getNextException* method to process all *SQLException*s in the chain. In many cases, the second exception in the chain is the pertinent one.

The following is an example:

```
        ...
    } catch (Throwable e) {
        System.out.println("exception thrown:");
        errorPrint(e);
```

```
    }
    ...
}

static void errorPrint(Throwable e) {
    if (e instanceof SQLException)
        SQLExceptionPrint((SQLException)e);
    else
        System.out.println("A non-SQL error: " + e.toString());
}

static void SQLExceptionPrint(SQLException sqle) {
    while (sqle != null) {
        System.out.println("\n---SQLException Caught---\n");
        System.out.println("SQLState:   " + (sqle).getSQLState());
        System.out.println("Severity: " + (sqle).getErrorCode());
        System.out.println("Message:  " + (sqle).getMessage());
        sqle.printStackTrace();
        sqle = sqle.getNextException();
    }
}
```

The *SQLException* may wrap another, triggering exception, such as an *IOException*. To inspect this additional, wrapped error, call the *SQLException*'s *getCause* method.

See also "Derby exception messages and SQL states" in the *Derby Reference Manual*.

Using Derby as a Java EE resource manager

The Java Platform, Enterprise Edition (the Java EE platform) is a standard for development of enterprise applications based on reusable components in a multi-tier environment. In addition to the features of the Java Platform, Standard Edition (the Java SE platform), the Java EE platform adds support for Enterprise JavaBeans (EJB) technology, the Java Persistence API, JavaServer Faces technology, Java Servlet technology, JavaServer Pages (JSP) technology, and many more. The Java EE platform architecture is used to bring together existing technologies and enterprise applications in a single, manageable environment.

Derby is a Java EE platform conformant component in a distributed Java EE system. As such, Derby is one part of a larger system that includes, among other things, a JNDI server, a connection pool module, a transaction manager, a resource manager, and user applications. Within this system, Derby can serve as the resource manager.

For more information on the Java EE platform, see http://www.oracle.com/technetwork/java/javaee/documentation/index.html.

Note: This section does not show you how to use Derby as a Resource Manager. Instead, it provides details specific to Derby that are not covered in the specification. This information is useful to programmers developing other modules in a distributed Java EE system, not to end-user application developers.

In order to qualify as a resource manager in a Java EE system, the Java EE platform requires three basic areas of support. These three areas of support involve implementation of APIs and are described in "Java EE Compliance: Java Transaction API and javax.sql Extensions" in the *Derby Reference Manual*.

This chapter describes the Derby classes that implement the APIs and provides some implementation-specific details.

Classes that pertain to resource managers

Derby provides two embedded variants of each *DataSource* interface defined by the JDBC API.

Most applications will use the first variant. However, applications that run on Java SE 8 Compact Profile 2 must use the second variant (the one whose class name begins with "Basic"). For more information, see "JDBC support for Java SE 8 Compact Profiles" in the *Derby Reference Manual*.

The Derby implementation classes for the *DataSource* interfaces are as follows:

- *org.apache.derby.jdbc.EmbeddedDataSource* and *org.apache.derby.jdbc.BasicEmbeddedDataSource40*

  These classes implement the *javax.sql.DataSource* interface, which a JNDI server can reference (except in the case of the second variant). Typically, this is the object that you work with as a *DataSource*.
- *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource* and *org.apache.derby.jdbc.BasicEmbeddedConnectionPoolDataSource40*

  These classes implement the *javax.sql.ConnectionPoolDataSource* interface and provide a factory for *PooledConnection* objects.
- *org.apache.derby.jdbc.EmbeddedXADataSource* and *org.apache.derby.jdbc.BasicEmbeddedXADataSource40*

These classes implement the *javax.sql.XADataSource* interface.

For more information, see the API documentation for each class.

Getting a DataSource

Normally, you can simply work with the interfaces for *javax.sql.DataSource*, *javax.sql.ConnectionPoolDataSource*, and *javax.sql.XADataSource*, as shown in the following examples.

```
import org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource;
import org.apache.derby.jdbc.EmbeddedDataSource;
import org.apache.derby.jdbc.EmbeddedXADataSource;

javax.sql.ConnectionPoolDataSource cpds = new
 EmbeddedConnectionPoolDataSource();
javax.sql.DataSource ds = new EmbeddedDataSource();
javax.sql.XADataSource xads = new EmbeddedXADataSource();
```

Derby provides six properties for a *DataSource*. These properties are in *org.apache.derby.jdbc.EmbeddedDataSource*. They are:

- *DatabaseName*

  This mandatory property must be set. It identifies which database to access. To access a database named *wombat* located at */local1/db/wombat*, call *setDatabaseName("/local1/db/wombat")* on the data source object.
- *CreateDatabase*

  Optional. Sets a property to create a database the next time the *getConnection* method of a data source object is called. The string *createString* is always "create" (or possibly null). (Use the method *setDatabaseName()* to define the name of the database.)
- *ShutdownDatabase*

  Optional. Sets a property to shut down a database. The string *shutDownString* is always "shutdown" (or possibly null). Shuts down the database the next time the *getConnection* method of a data source object is called.
- *DataSourceName*

  Optional. Name for *ConnectionPoolDataSource* or *XADataSource*. Not used by the data source object. Used for informational purposes only.
- *Description*

  Optional. Description of the data source. Not used by the data source object. Used for informational purposes only.
- *connectionAttributes*

  Optional. Connection attributes specific to Derby. See the *Derby Reference Manual* for a more information about the attributes.

Shutting down or creating a database

If you need to shut down or create a database, it is easiest just to work with the Derby-specific implementations of interfaces, as shown in these examples.

```
javax.sql.XADataSource xads = makeXADataSource(mydb, true);

// example of setting property directory using
// Derby's XADataSource object
import org.apache.derby.jdbc.EmbeddedXADataSource;
```

```
import javax.sql.XADataSource;
// dbname is the database name
// if create is true, create the database if not already created
XADataSource makeXADataSource (String dbname, boolean create)
{
    EmbeddedXADataSource xads = new EmbeddedXADataSource();
    // use Derby's setDatabaseName call
    xads.setDatabaseName(dbname);
    if (create)
        xads.setCreateDatabase("create");
    return xads;
}
```

Setting the property does not create or shut down the database. The database is not actually created or shut down until the next connection request.

Developing tools and using Derby with an IDE

Applications such as database tools are designed to work with databases whose schemas and contents are unknown in advance. This section discusses a few topics useful for such applications.

Offering connection choices to the user

JDBC's *java.sql.Driver.getPropertyInfo* method allows a generic GUI tool to determine the properties for which it should prompt a user in order to get enough information to connect to a database. Depending on the values the user has supplied so far, additional values might become necessary. It might be necessary to iterate though several calls to *getPropertyInfo*.

If no more properties are necessary, the call returns an array of zero length.

In a Derby system, do not use the method against an instance of *org.apache.derby.jdbc.EmbeddedDriver*. Instead, request the JDBC driver from the driver manager:

```
java.sql.DriverManager.getDriver(
    "jdbc:derby:").getPropertyInfo(URL, Prop)
```

In a Derby system, the properties returned in the *DriverPropertyInfo* object are connection URL attributes, including a list of booted databases in a system (the *databaseName* attribute).

Databases in a system are not automatically booted until you connect with them. You can configure your system to retain the former behavior, in which case the steps described in this section will continue to work. See "*derby.system.bootAll*" in the *Derby Reference Manual*.

getPropertyInfo requires a connection URL and a *Properties* object as parameters. Typically, what you pass are values that you will use in a future call to *java.sql.DriverManager.getConnection* when you actually connect to the database.

A call to *getPropertyInfo* with parameters that contain sufficient information to connect successfully returns an array of zero length. (Receiving this zero-length array does not *guarantee* that the *getConnection* call will succeed, because something else could go wrong.)

Repeat calls to *getPropertyInfo* until it returns a zero-length array or none of the properties remaining are desired.

The DriverPropertyInfo Array

When a non-zero-length array is returned by *getPropertyInfo*, each element is a `DriverPropertyInfo` object representing a connection URL attribute that has not already been specified. Only those that make sense in the current context are returned.

This *DriverPropertyInfo* object contains:

- *name of the attribute*
- *description*
- *current value*

  If an attribute has a default value, this is set in the value field of *DriverPropertyInfo*, even if the attribute has not been set in the connection URL or the `Properties`

object. If the attribute does not have a default value and it is not set in the URL or the `Properties` object, its value will be null.
- *list of choices*
- *whether required for a connection request*

Several fields in a *DriverPropertyInfo* object are allowed to be null.

DriverPropertyInfo array example

The following code shows how to use a *DriverPropertyInfo* array.

```
import java.sql.*;
import java.util.Properties;
// Start with the least amount of information
// to see the full list of choices.
// We could also enter with a URL and Properties
// provided by a user.
String url = "jdbc:derby:";
Properties info = new Properties();
Driver cDriver = DriverManager.getDriver(url);
for (;;)
        {
        DriverPropertyInfo[] attributes = cDriver.getPropertyInfo(
            url, info);
         // Zero length means a connection attempt can be made
        if (attributes.length == 0)
        break;
        // Insert code here to process the array; for example,
        // display all options in a GUI and allow the user to
        // pick and then set the attributes in info or URL.
        }
// Try the connection
Connection conn = DriverManager.getConnection(url, info);
```

Using Derby with IDEs

When you use an integrated development environment (IDE) to develop an embedded Derby application, you might need to run Derby within a server framework.

This is because an IDE might try connecting to the database from two different JVMs, whereas only a single JVM instance should connect to a Derby database at one time (multiple connections from the same JVM are allowed).

An "embedded Derby application" is one which runs in the same JVM as the application. Such an application uses the embedded Derby driver (*org.apache.derby.jdbc.EmbeddedDriver*) and connection URL (`jdbc:derby:databaseName`). If you use this driver name or connection URL from the IDE, when the IDE tries to open a second connection to the same database with the embedded Derby, the attempt fails. Two JVMs cannot connect to the same database in embedded mode.

IDEs and multiple JVMs

When you use an integrated development environment (IDE) to build a Java application, you can launch the application from within the IDE at any point in the development process. Typically, the IDE launches a JVM dedicated to the application. When the application completes, the JVM exits. Any database connections established by the application are closed.

During the development of a database application, most IDEs allow you to test individual database connections and queries without running the entire application. When you test an individual database connection or query (which requires a database connection), the IDE might launch a JVM that runs in a specialized testing environment. In this case,

when a test completes, the JVM remains active and available for further testing, and the database connection established during the test remains open.

Because of the behaviors of the IDE described above, if you use the embedded Derby JDBC driver, you may encounter errors connecting in the following situations:

- You test an individual query or database connection and then try to run an application that uses the same database as the tested feature.

  The database connection established by testing the connection or query stays open, and prevents the application from establishing a connection to the same database.
- You run an application, and before it completes (for example, while it waits for user input), you attempt to run a second application or to test a connection or query that uses the same database as the first application.

SQL tips

This section provides some examples of interesting SQL features. It also includes a few non-SQL tips.

Retrieving the database connection URL

Derby does not have a built-in function that returns the name of the database. However, you can use `DatabaseMetaData` to return the connection URL of any local `Connection`.

```
/* in java */
String myURL = conn.getMetaData().getURL();
```

Supplying a parameter only once

If you want to supply a parameter value once and use it multiple times within a query, put it in the FROM clause with an appropriate CAST.

```
SELECT  phonebook.*
 FROM phonebook, (VALUES (CAST(? AS INT), CAST(? AS VARCHAR(255))))
      AS Choice(choice, search_string)
 WHERE search_string = (case when choice = 1 then firstnme
                  when choice=2 then lastname
      when choice=3 then phonenumber end);
```

This query selects what the second parameter will be compared to based on the value in the first parameter. Putting the parameters in the FROM clause means that they need to be applied only once to the query, and you can give them names so that you can refer to them elsewhere in the query. In the example above, the first parameter is given the name *choice*, and the second parameter is given the name *search_string*.

Defining an identity column

An identity column is a column that stores numbers that increment by one with each insertion. Identity columns are sometimes called autoincrement columns.

Derby provides autoincrement as a built-in feature; see CREATE TABLE statement in the *Derby Reference Manual*.

Below is an example that shows how to use an identity column to create the MAP_ID column of the MAPS table in the *toursDB database*.

```
CREATE TABLE MAPS
(
MAP_ID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1,
 INCREMENT BY 1),
MAP_NAME VARCHAR(24) NOT NULL,
REGION VARCHAR(26),
AREA DECIMAL(8,4) NOT NULL,
PHOTO_FORMAT VARCHAR(26) NOT NULL,
PICTURE BLOB(102400),
UNIQUE (MAP_ID, MAP_NAME)
)
```

Using third-party tools

You can hook into any JDBC tool with just our JDBC Driver class name (*org.apache.derby.jdbc.EmbeddedDriver*) and Derby's JDBC connection URL.

Tricks of the VALUES clause

This section contains some tips to use with the VALUES clause.

Multiple rows

Derby supports the complete SQL VALUES clause; this is very handy in several cases.

The first useful case is that it can be used to insert multiple rows:

```
INSERT INTO OneColumnTable VALUES 1,2,3,4,5,6,7,8

INSERT INTO TwoColumnTable VALUES
    (1, 'first row'),
    (2, 'second row'),
    (3, 'third row')
```

Dynamic parameters reduce the number of times execute requests are sent across:

```
ij> -- send 5 rows at a time:
ij> PREPARE p1 AS 'INSERT INTO ThreeColumnTable VALUES
(?,?,?), (?,?,?), (?,?,?), (?,?,?), (?,?,?)';
ij> EXECUTE p1 USING 'VALUES (''1st'',1,1,''2nd'',2,2,''3rd'',
3,3,''4th'',4,4,''5th'',5,5)';
```

Mapping column values to return values

Multiple-row VALUES tables are useful in mapping column values to desired return values in queries.

```
-- get the names of all departments in OhioSELECT DeptName
FROM Depts,
(VALUES (1, 'Shoe'),
    (2, 'Laces'),
    (4, 'Polish'))
AS DeptMap(DeptCode,DeptDesc)
WHERE Depts.DeptCode = DeptMap.DeptCode
AND Depts.DeptLocn LIKE '%Ohio%'
```

You might also find it useful to store values used often for mapping in a persistent table and then using that table in the query.

Creating empty queries

You may need Derby to create "empty" queries in existing applications for filling in bits of functionality that Derby does not supply.

Empty queries of the right size and shape can be formed off a single values table and a "WHERE FALSE" condition:

```
SELECT *
FROM (VALUES ('',1,"TRUE")) AS ProcedureInfo(ProcedureName,NumParameters,

ProcedureValid)
WHERE 1=0
```

Localizing Derby

Derby offers support for locales.

The word *locale* in the Java platform refers to an instance of a class that identifies a particular combination of language and region. If a Java class varies its behavior according to *locale*, it is said to be locale-sensitive. Derby provides some support for locales for databases and for other components, such as the tools and the installer.

Derby also provides a feature to support databases in many different languages, a feature which is independent of a particular locale. When you create or upgrade a database, you can use the *territory=ll_CC* attribute to associate a non-default locale with the database. For information on how to use the *territory=ll_CC* attribute, see the *Derby Reference Manual*.

SQL parser support for Unicode

To support users in many different languages, Derby's SQL parser understands all Unicode characters and allows any Unicode character or number to be used in an identifier.

Derby does not attempt to ensure that the characters in identifiers are valid in the database's locale.

Character-based collation in Derby

A character set is a set of symbols and encodings. Character data types are represented as Unicode 2.0 sequences in Derby. Collation defines how character data is sorted.

How collation works in Derby

Derby supports a wide range of character sets and encodes all of the character sets by using the Unicode support provided by the *java.lang.Character* class in the Java Virtual Machine (JVM) in which the Derby database runs.

See the Java API documentation for the *java.lang.Character* class for the exact level of Unicode Standard that is supported.

A collation is a set of rules for comparing characters in a character set. In Derby, the collation rules affect comparisons of the CHAR and VARCHAR data types. Collation rules also affect how the LIKE Boolean operator processes the CHAR, VARCHAR, CLOB, and LONG VARCHAR data types.

The default Derby collation rule is based on the binary Unicode values of the characters. So a character is greater than (>), equal to (=), or less than (<) another character based on the numeric comparison of the Unicode values. This rule allows for very efficient comparisons of character strings.

Note: When LIKE comparisons are used, Derby compares one character at a time for non-metacharacters. This is different from the way Derby processes = comparisons. The comparisons with the = operator compare the entire character string on the left side of the = operator with the entire character string on the right side of the = operator. For details, see Differences between LIKE and equal (=) comparisons.

Locale-based collation

Derby supports the ability to define collation rules that are appropriate to a locale. This is referred to as *locale-based collation*.

Derby supports the locales that the Java programming language supports. In addition, it is possible to create customized locales. For details, see Creating a customized collator.

You can set the locale of a database when you create the database by specifying the *territory=ll_CC* attribute. (See Creating a database with locale-based collation for details.) If you do not specify a locale, Derby uses the default locale of the JVM in which the database is created.

At the same time, you can also specify locale-based collation by using the *collation=collation* attribute, and you can specify a particular collation strength to make the database either case-sensitive or case-insensitive, or to respect both case and accent in collation. See Creating a case-insensitive database for information on making the database use case-insensensitive searches.

Each JVM can support many locales that are independent of the default locale for the JVM. Collation support for these additional locales is provided through the *java.text.RuleBasedCollator* class and the set of rules for these locales. Refer to the JVM specification for details of how these rules are used to provide locale-specific collation.

The locale-based collation in Derby affects how the CHAR and VARCHAR data types are compared. Specifying locale-based collation also impacts how the LIKE Boolean operator processes CHAR, VARCHAR, CLOB, and LONG VARCHAR data.

Locale-based collation adds some extra processing overhead to all character-based comparison operations.

Database connection URL attributes that control collation

When you create a Derby database, the attributes that you set determine the collation that is used with all character data in the database.

The following table shows some examples.

Table 12. Database creation URL attributes that control collation

Example Create URLs	Collation Is Driven By
jdbc:derby:abcDB;create=true	Unicode codepoint collation (UCS_BASIC), which is the default collation for Derby databases.
jdbc:derby:abcDB;create=true;territory=es_MX	Unicode codepoint collation (UCS_BASIC). The *collation=collation* attribute is not set.
jdbc:derby:abcDB;create=true;collation=TERRITORY_BASED	The locale of the JVM, since the *territory=ll_CC* attribute is not set. **Tip:** To determine the locale of the JVM, call the *Locale.getDefault* method.
jdbc:derby:abcDB;create=true;territory=es_MX;collation=TER	The *territory=ll_CC* attribute.

Example Create URLs	Collation Is Driven By
jdbc:derby:abcDB;create=true;territory=es_MX;collation=TERRITORY_BASED	The *territory=ll_CC* attribute with collation strength PRIMARY, which makes the database case-insensitive (it typically means that only differences in base letters are considered significant).

Examples of case-sensitive and case-insensitive string sorting

These examples show the results of sorts on databases created with various collation and locale attributes.

With Unicode codepoint collation (UCS_BASIC), the default if you do not specify either *collation=collation* or *territory=ll_CC*, the numeric values of the Unicode encoding of the characters are used directly for ordering. For example, the FRUIT table contains the NAME column that uses the VARCHAR(20) data type. The contents of the NAME column are:

orange
apple
Banana
Pineapple
Grape

UCS_BASIC collation sorts all uppercase letters before lowercase letters. The statement `SELECT * FROM FRUIT ORDER BY NAME` returns the following:

Banana
Grape
Pineapple
apple
orange

The above result also appears if you specify *territory=ll_CC* but do not specify *collation=collation*.

If the database is created with the *territory=ll_CC* attribute set to *en_US* (English language, United States country code) and the *collation=collation* attribute set to TERRITORY_BASED, the statement `SELECT * FROM FRUIT ORDER BY NAME` returns:

apple
Banana
Grape
orange
Pineapple

The collation set for the database also impacts comparison operators on character data types. For example, the statement `SELECT * FROM FRUIT WHERE NAME > 'Banana' ORDER BY NAME` returns:

UCS_BASIC collation	Locale-based collation
Grape	Grape

Pineapple	orange
apple	Pineapple
orange	

For information on creating case-insensitive databases, see Creating a case-insensitive database.

Differences between LIKE and equal (=) comparisons

When you use locale-based collation, the comparisons can return different results when you use the LIKE and equal (=) operators.

For example, suppose that the Derby database is set to use a locale where the character 'z' has the same collation elements as 'xy'. Consider the following two WHERE clauses:

1. WHERE 'zcb' = 'xycb'
2. WHERE 'zcb' LIKE 'xy_b'

For WHERE clause 1, Derby returns TRUE, because the collation elements for the entire string 'zcb' will match the collation elements of the entire string 'xycb'.

For WHERE clause 2, Derby returns FALSE, because the collation element for the character 'z' does not match the collation element for the character 'x'. In addition, when a metacharacter such as an underscore is used with the LIKE operator, the metacharacter counts for one character in the string value. A clause such as `WHERE 'xycb' LIKE '_cb'` returns FALSE, because 'x' is compared to the metacharacter '_' and 'y' does not match 'c'.

Other components with locale support

Derby also provides locale support for the following components:

- Database error messages are in the language of the locale, if support is explicitly provided for that locale with a special library.

  For example, Derby explicitly supports Spanish-language error messages. If a database's locale is set to one of the Spanish-language locales, Derby returns error messages in the Spanish language.
- The Derby tools. In the case of the tools, locale support includes locale-specific interface and error messages and localized data display.

  For more information about localization of the Derby tools, see the *Derby Tools and Utilities Guide*.

Localized messages require special libraries.

The locale of the database is set by the *territory=ll_CC* attribute when the database is created. However, the locale of the error messages and tools is not determined by the locale of the database. The locale of the error messages and tools is determined by the default system locale. This means that it is possible to create a database with a non-default locale. In such a case, error messages are not returned in the language of the locale of the database but are returned in the language of the default locale instead.

Note: You can override the default locale for ij with a property on the JVM. For more information, see the *Derby Tools and Utilities Guide*.

Messages libraries

The following list describes the items required in order for Derby to provide localized messages.

- You must have the locale-specific Derby jar file. Derby provides such jars for only some locales. You will find the locale jar files in the */lib* directory in your Derby installation.
- The locale-specific Derby jar file must be in the classpath.

The locale-specific Derby jar file is named *derbyLocale_ll_CC.jar*, where *ll* is the two-letter code for language, and *CC* is the two-letter code for country. For example, the name of the jar file for error messages for the German locale is `derbyLocale_de_DE.jar`.

Derby supports the following locales:

- derbyLocale_cs.jar - Czech
- derbyLocale_de_DE.jar - German
- derbyLocale_es.jar - Spanish
- derbyLocale_fr.jar - French
- derbyLocale_hu.jar - Hungarian
- derbyLocale_it.jar - Italian
- derbyLocale_ja_JP.jar - Japanese
- derbyLocale_ko_KR.jar - Korean
- derbyLocale_pl.jar - Polish
- derbyLocale_pt_BR.jar - Brazilian Portuguese
- derbyLocale_ru.jar - Russian
- derbyLocale_zh_CN.jar - Simplified Chinese
- derbyLocale_zh_TW.jar - Traditional Chinese

Derby and standards

Derby adheres to SQL99 or newer standards wherever possible. This section describes those features currently in Derby that are not standard; these features are currently being evaluated and might be removed in future releases.

This section describes those parts of Derby that are non-standard or not typical for a database system.

ALTER TABLE syntax
Derby uses a slightly different ALTER TABLE syntax for altering column defaults. While SQL99 uses DROP and SET, Derby uses DEFAULT.

Calling functions and procedures
Derby supports the CALL (procedure) statement for calling external procedures declared by the CREATE PROCEDURE statement. Built-in functions and user-defined functions declared with the CREATE FUNCTION command can be called as part of an SQL select statement or by using either a VALUES clause or VALUES expression.

CLOB and BLOB data types
Derby supports the standard CLOB and BLOB data types. BLOB and CLOB values are limited to a maximum of 2,147,483,647 characters.

Cursors
Derby uses JDBC's result sets, and does not provide SQL for manipulating cursors except for positioned update and delete. Derby's scrollable insensitive cursors are provided through JDBC, not through SQL commands.

DECIMAL max precision
For Derby, the maximum precision for DECIMAL columns is 31 digits. SQL99 does not require a specific maximum precision for decimals, but most products have a maximum precision of 15-32 digits.

Dynamic SQL
Derby uses JDBC's Prepared Statement, and does not provide SQL commands for dynamic SQL.

Expressions on LONGs
Derby permits expressions on LONG VARCHAR; however, LONG VARCHAR data types are not allowed in the following clauses, operations, constraints, functions, and predicates:

- GROUP BY clauses
- ORDER BY clauses
- JOIN operations
- PRIMARY KEY constraints
- Foreign KEY constraints
- UNIQUE key constraints
- MIN aggregate function
- MAX aggregate function
- [NOT] IN predicate
- UNION, INTERSECT, and EXCEPT operators

SQL99 also places some restrictions on expressions on LONG types.

Information schema
Derby uses its own system catalog that can be accessed using standard JDBC DatabaseMetadata calls. Derby does not provide the standard Information Schema views.

NOT NULL characteristic
The SQL standard says NOT NULL is a constraint, and can be named and viewed in the information schema as such. Derby does not provide naming for NOT NULL, nor

does it present it as a constraint in the information schema, only as a characteristic of the column.

Stored routines and PSM

Derby supports external procedures using the Java programming language. Procedures are managed using the CREATE PROCEDURE and DROP PROCEDURE statements.

Transactions

All operations in Derby are transactional. Derby supports transaction control using JDBC Connection methods. This includes support for savepoints and for the four JDBC transaction isolation levels. The only SQL command provided for transaction control is SET TRANSACTION ISOLATION.

XML data types and operators

Derby supports the XML data type and a set of operators that work with the XML data type, but does not provide JDBC support for the XML data type. The XML data type and operators are based on a small subset of the SQL/XML specification.

The XML data type and operators are defined only in the SQL layer.

There is no JDBC-side support for XML data types. It is not possible to bind directly into an XML value or to retrieve an XML value directly from a result set. Instead, you must bind and retrieve the XML data as Java strings or character streams by explicitly specifying the appropriate XML operator as part of the SQL statements:

- Create a table with a XML data typed column. For example:

```
CREATE TABLE xml_data(xml_col XML);
```

- Use the XMLPARSE operator for binding data into XML values. For example:

```
INSERT INTO xml_data(xml_col)
  VALUES(XMLPARSE(DOCUMENT ' <name> Derby </name>' PRESERVE
 WHITESPACE));
```

  Note: You must insert the XML keywords DOCUMENT and PRESERVE WHITESPACE. Actual XML data should be inside single quotation marks, and values should be within the starting XML tag and the ending XML tag.

- Use the XMLSERIALIZE operator to retrieve XML values from a result set. For example:

```
SELECT XMLSERIALIZE(xml_col AS CLOB) FROM xml_data;
```

  Note: You can also specify `xml_col AS VARCHAR(25)`.

- Use non-XML data retrieved from a non-XML column to create an XML fragment. For example:

```
    SELECT '<my_self>' ||
    '<name>'|| my_name ||'</name>' ||
    '<age>'|| TRIM(CHAR(my_age)) ||'</age>'||
    '</my_self>'
    FROM my_non_xml_table;
```

  Note: This will result in XML fragments, which you must plug into an XML document.

Additionally, there is no JDBC metadata support for the XML data type.

The XML data type is not allowed in any of the clauses or operations that are described in the section on expressions on LONG data types in Derby and standards.

For the XML operators to work properly, Derby requires that the Java Virtual Machine (JVM) have working implementations of the *javax.xml.parsers.DocumentBuilderFactory* and *javax.xml.xpath.XPathFactory* classes. All supported JVMs implement these classes.

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.