[vertical list of authors]

[cover art/text goes here]

Contents

Copyright

First Edition (July 2005)

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* .

Purpose of this guide

This book 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 procedures and how you can deploy Derby applications.

Audience

This book 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 chapters:

- *JDBC applications and Derby basics*

  Basic details for using Derby, including loading the JDBC driver, specifying a database URL, and starting Derby.

- *After installing*

  Explains the installation layout.

- *Deploying Derby applications*

  An overview of different deployment scenarios, and tips for getting the details right when deploying applications.

- *Controlling Derby application behavior*

  JDBC, cursors, locking and isolation levels, and multiple connections.

- *Using Derby as a J2EE 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.

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 10.1).
- allows use of new features such as SYNONYMS with the upgraded database.

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

Preparing to upgrade

Upgrading occurs the first time the new Derby software connects to the old database. Before connecting with 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* .

   If you do not perform a soft upgrade, then once the database is upgraded, it cannot be reverted back to the previous version.

Upgrading a database

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

Do not attempt to upgrade a database without first backing it up.

To request an upgrade when connecting to the database:

1. Use the `upgrade=true` database connection URL attribute, as shown in the following example:

   ```
   jdbc:derby:sample;upgrade=true
   ```

   Once the upgrade is complete, you cannot connect to the database with an older version of Derby.

You can find out the version of Derby using sysinfo:

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

Note that this is the version of Derby, not the version of the database. Sysinfo uses information found in the Derby jar files, so verify that only one version of Derby's jar files are in your class path when you run this tool.

Soft upgrade

Soft upgrade allows you 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.

If you perform a soft upgrade, you will not be able to perform certain functions that are not available in older versions of Derby. For example, the following Derby Version 10.1 features cannot be used in a database that has been soft upgraded:

- Synonym functionality
- Creating tables using the GENERATED BY DEFAULT option for identity columns
- Reclaiming unused space using the SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE procedure

Other new features in Derby that do not affect database structure, such as using timestamp arithmetic, are allowed in a soft upgraded database.

To perform a soft upgrade on a database created using an earlier version of Derby:

1. Simply connect to the database, as shown in the following example:

   ```
   connect 'jdbc:derby:sample'
   ```

   In this example, the sample database is a Version 10.0 database.

JDBC applications and Derby basics

This chapter 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 JDBC, the application programming interface that makes it possible to access relational databases from Java programs. The JDBC API is part of the Java(TM) 2 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 JDBC. 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 chapter 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 JDBC 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-92 is the version of SQL standardized by ANSI and ISO in 1992; Derby supports entry-level SQL-92 as well as some higher-level features. Entry-level SQL-92 is a subset of full SQL-92 specified by ANSI and ISO that is supported by nearly all major DBMSs today. This chapter 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 JDBC that allows Derby to serve a resource manager in a J2EE compliant system.

Derby embedded basics

This section discusses the basics of the Derby database.

Derby JDBC driver

Derby consists of both the database engine and an embedded JDBC driver. Applications use JDBC to interact with a database. Applications must load the driver in order to work with the database.

In an embedded environment, loading the driver also starts Derby.

In a Java application, you typically load the driver with the static *Class.forName* method or with the `jdbc.drivers` system property. For example:

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

For detailed information about loading the Derby JDBC driver, see *"java.sql.Driver"* in the *Derby Reference Manual* . See also javadoc for org.apache.derby.util.DriverUtil.

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.

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 the *Derby Server and Administration Guide* for more information on the Network Server.

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 "Examples". For detailed reference about attributes and values, as well as syntax of the database connection URL, see the "Derby Database Connection URL Syntax" in the *Derby Reference Manual* .

An example use of the connection URL:

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

Derby system

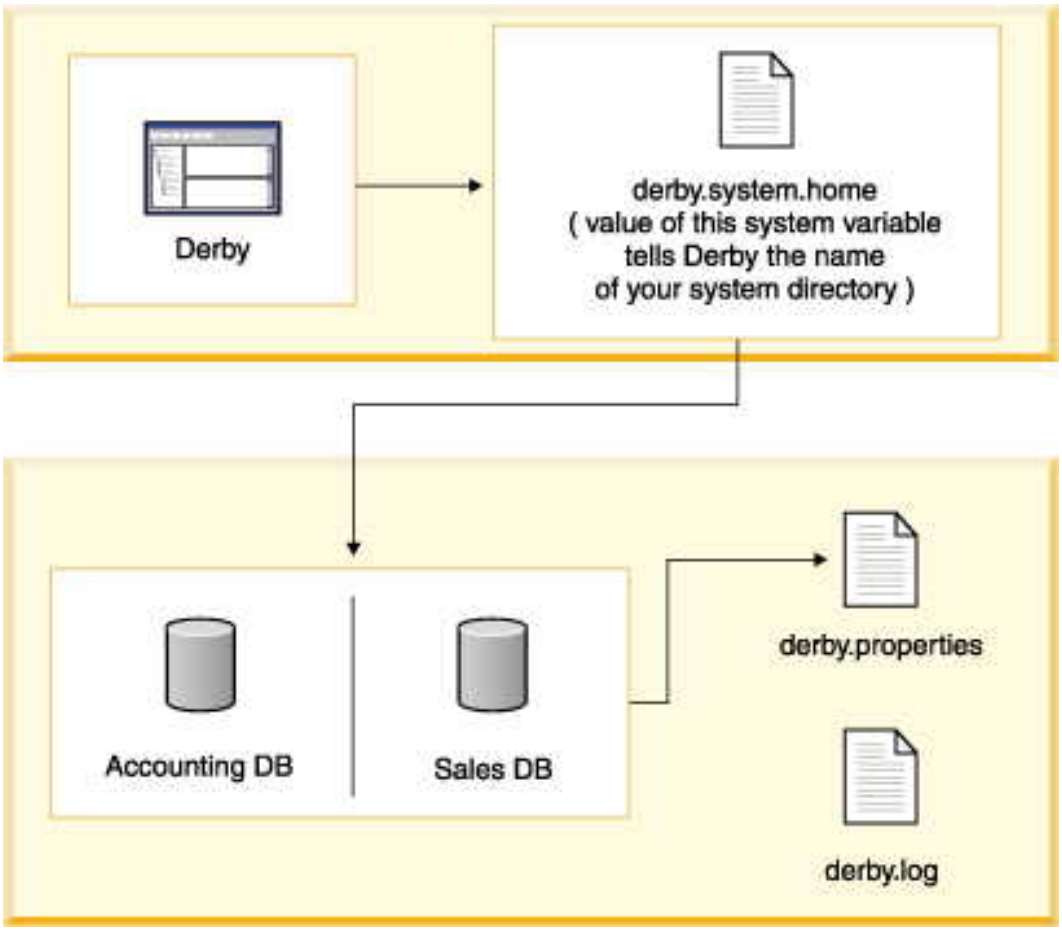
A Derby database exists within a *system*.

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 system - as well as its directory, which you name - is an essential part of a running database or databases. Understanding the Derby system is essential to successful development and deployment of Derby applications.

Figure1. Derby databases live in a system, which includes system-wide properties, an error log, and one or more databases.



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).

You can use a property (see *"derby.service"* in *Tuning Derby*) to include databases in other directories or in subdirectories of the system directory in the current system when you start it up.

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

One Derby instance for each Java Virtual Machine

You could potentially have two instances of a Derby system (JVM) running on the same machine at the same time. Each instance must run in a different JVM. Two separate instances of Derby must not access the same database. For example, in an embedded environment, an application that accesses Derby databases starts up the local JDBC driver, which starts up an instance of Derby. If you start another application, such as ij, and connect to the same database, severe database corruption can result. See Double-booting system behavior .

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 *Tuning Derby* 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, it prints a message in the error log:

```
2005-06-10 03:54:06.196 GMT: Booting Derby version Apache Derby
- 10.0.0.1 - (29612): instance c013800d-00fd-0cb0-e736-ffffd1025a25 on
database directory sample
```

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 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:cs;shutdown=true");
```

Shutdown commands always raise *SQLExceptions*.

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

```
Sat Jan 10 14:31:54 PDT 2005:
Shutting down instance 80000001-00d0-8bdf-d115-000a0a0b2d00
```

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 JVM needs to unload *org.apache.derby.jdbc.EmbeddedDriver*, so that it can reload it when it restarts Derby. (Loading the local driver starts Derby.)

You cannot explicitly request that the JVM unload a class, but you can ensure that the *EmbeddedDriver* class is unloaded by using a `System.gc()` to force it to garbage collect classes that are no longer needed. Running with `-nogc` or `-noclassgc` definitely *prevents* the class from being unloaded and makes you unable to restart Derby in the same JVM.

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 *Tuning Derby* for more information on setting 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, if any. Typically, Derby writes this information to a log called *derby.log* in the system directory, although you can also have Derby send messages to a stream, using a property. 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 this and other properties, see *Tuning Derby* .

derby.properties

The text file *derby.properties* contains the definition of properties, or configuration parameters valid for the entire system. This file is not automatically created; if you wish to set Derby properties with this file, you need to create it yourself. The 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 *Tuning Derby* .

Double-booting system behavior

Derby attempts to prevent two instances of Derby from booting the same database by using a file called *db.lck* inside the database directory (see "The Database Directory").

On all platforms running with a JDK of 1.4 or higher, Derby can successfully prevent a second instance of Derby from booting the database and thus prevents corruption.

On some platforms running with a JDK lower than 1.4, Derby may prevent a second instance of Derby from booting the database (previous to JDK 1.4 the ability to do this was OS dependent).

If this is the case, you will see an *SQLException* like the following:

```
ERROR XJ040: Failed to start database 'sample', see the next exception
for details.
ERROR XSDB6: Another instance of Derby might have already booted
the databaseC:\databases\sample.
```

The error is also written to the error log.

If you are running a JVM prior to 1.4, Derby issues a warning message on some platforms if an instance of Derby attempts to boot a database that already has a running instance of Derby attached to it. However, it does not prevent the second instance from booting, and thus potentially corrupting, the database. (You can change this behavior with the property `derby.database.forceDatabaseLock` .)

If a warning message has been issued, corruption might already have occurred. Corruption can occur even if one of the two booting systems has "readonly" access to the database.

The warning message looks like this:

```
WARNING: Derby
(instance 80000000-00d2-3265-de92-000a0a0a0200) is
attempting to boot the database /export/home/sky/wombat
even though Derby
(instance 80000000-00d2-3265-8abf-000a0a0a0200) might still be active.
```

```
Only one instance of Derby
should boot a database at a time. Severe and non-recoverable corruption
can
result and might have already occurred.
```

The warning is also written to the error log.

If you see this warning, you should close the connection and exit the JVM, minimizing the risk of a corruption. Close all instances of Derby, then restart one instance of Derby and shut down the database properly so that the *db.lck* file can be removed. The warning message continues to appear until a proper shutdown of the Derby system can delete the *db.lck* file.

When developing applications, you might want to configure Derby to append to the log. Doing so will help you detect when you have inadvertently started more than one instance of Derby in the same system. For example, when the *derby.infolog.append* property is set to true for a system, booting two instances of Derby in the same system produces the following in the log:

```
Sat Aug 14 09:42:51 PDT 2005:
Booting Derby version Apache Derby - 10.0.0.1 - (29612):

instance 80000000-00d2-1c87-7586-000a0a0b1300 on database at
directory C:\tutorial_system\sample
------------------------------------------------------------
Sat Aug 14 09:42:59 PDT 2005:
Booting Derby version Apache Derby - 10.0.0.1 - (29612):
instance 80000000-00d2-1c87-9143-000a0a0b1300 on database at
directory C:\tutorial_system\HelloWorldDB
```

Derby allows you to boot databases that are not in the system directory. While this might seem more convenient, check that you do not boot the same database with two JVMs. If you need to access a single database from more than one JVM, you will need to put a server solution in place. You can allow multiple JVMs that need to access that database to connect to the server. The Derby Network Server is provided as a server solution. 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, and only one instance of Derby should boot a database at one time. Keeping databases in the system directory makes it less likely that you would use more than one instance of Derby.
- 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.

A database directory contains the following, as shown in Derby database directories contain files and directories used by the software. :

- *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

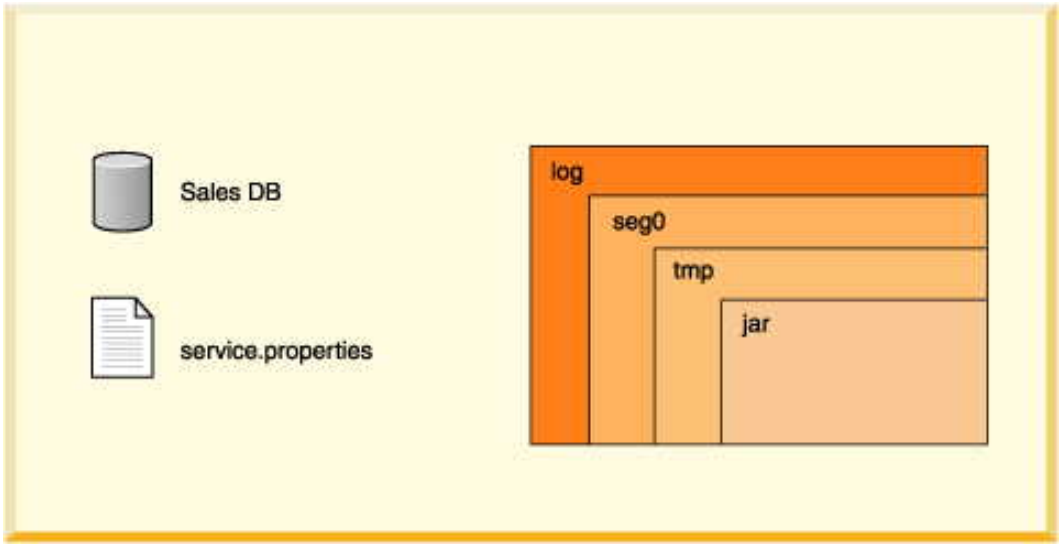
  (might not exist.) A temporary directory used by Derby for large sorts and deferred updates and deletes. Sorts are used by a variety of SQL statements. For databases on read-only media, you might need to set a property to change the location of this directory. See "Creating Derby Databases for Read-Only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

Figure1. Derby database directories contain files and directories used by the software.



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:

Table1. Size Limits to Derby Database Objects

Type of Object	Limit
tables per database	*java.lang.Long.MAX_VALUE*

Type of Object	Limit
	Some operating systems impose a limit to the number of files allowed in a single directory.
indexes per table	32,767 or storage
columns per table	1,012
number of columns on an index key	16
rows per 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, and therefore limit 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 (see Database connection examples).

There is no drop database command. To drop a database, 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* (see Offering Connection Choices to the User).

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. See Shutting down Derby or an individual database .

Storage and recovery

A Derby 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 *Tuning Derby* .) 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 "Database-Wide Properties" in *Tuning Derby* .

Note: You should work with database-level properties wherever possible.

Database limitations

Derby databases have these limitations:

- *Indexes are not supported for columns defined on CLOB, BLOB, and LONG VARCHAR 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.

  It is generally not recommended to create indexes on long columns. It is best to 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* .

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

  A "LogFull" error or some sort of `IOException` will occur in the *derby.log* if the system runs out of space. If the system has no more disk space to append to the *derby.log*, 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 (see Derby JDBC database connection URL). You specify a path to the database within this connection URL.

Standard connections

Connecting to databases within the system:

The standard way to access databases is in the file system by specifying the path to the database, either absolute or relative to the system directory. In a client/server environment, this path 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, just specify the database name 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 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 (see *java.io.File.isAbsolute*). You must specify a path for a database in a directory other than the system directory even if you have defined the `derby.service` property to have Derby boot the database automatically (see *"derby.service"* in *Tuning Derby*).

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 to, 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 that no other instance of Derby should access that database, among other things. It is recommended that you connect to databases only in the system directory. See Recommended practices for suggestions about working with a Derby system.

Conventions for specifying the database path

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

The path must do one of the following:

- refer to a previously created Derby database
- specify the *create=true* attribute

The path separator in the connection URL is / (forward slash), as in the standard *file://* URL protocol.

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 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 and its relative path 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.

If the path to the database is ambiguous, i.e., potentially the same as that to a database that is available on the classpath (see "Special Database Access"), use the *directory:* subsubprotocol to specify the one in the file system. For example:

```
jdbc:derby:directory:myDB
```

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.

Accessing databases from the classpath:

In most cases, you access databases from the file system as described above. 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, specify the name of the database relative to the directory in the classpath. You can use the classpath subprotocol if such a database is ambiguous within the directory system. See Embedded Derby JDBC database connection URL for more information.

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:/sample
```

The forward slash is required before *sample* to indicate that it is relative to *C:\derby\demo\databases* directory.

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

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

Accessing databases from a jar or zip file:

It is possible to access databases from a jar file. The jar file can be, but 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:/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. This information also applies to the client connection URL in a client/server environment. For reference information about client connection URLs, see *"java.sql.Connection"* in the *Derby Reference Manual* .

- *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.* If a jar file storing databases were in the user's classpath, this could also be a path within the jar file.

- *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.

- *jdbc:derby:/myDB*

  Access *myDB* (which is directly in a directory in the classpath) as a read-only database.

- *jdbc:derby:classpath:/myDB*

  Access *myDB* (which is directly in a directory in the classpath) as a read-only database. The reason for using the subsubprotocol is that it might have the same path as a database in the directory structure.

- *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. The reason for using the `directory:` subsubprotocol is that it might happen to have the same path as a database in the classpath.

Working with the database connection URL attributes

You specify attributes on the Derby connection URL (see Derby JDBC database 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".

All attributes are optional. For detailed information about the connection URL syntax and attributes, see "Derby Database Connection URL Syntax"in the *Derby Reference Manual* .

You can specify the following attributes:

- *bootPassword=key*
- *create=true*
- *databaseName=nameofDatabase*
- *dataEncryption=true*
- *encryptionProvider=providerName*
- *encryptionAlgorithm=algorithm*
- *territory=ll_CC*
- *logDevice=logDirectoryPath*
- *createFrom=BackupPath*
- *restoreFrom=BackupPath*
- *rollForwardrecoveryFrom=BackupPath*
- *password=userPassword*
- *shutdown=true*
- *user=userName*

Using the databaseName attribute

```
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:/databasePathWithinArchive
```

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 must not 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.

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");
```

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 connection URL attributes (see *user=userName* and *password=userPassword*).

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

For more information, see Working with user authentication .

Encrypting a database when you create it

If your environment is configured properly, you can create your database as an encrypted database (one in which the database is encrypted on disk). To do this, you use the *dataEncryption=true* attribute to turn on encryption and the `bootPassword=key` attribute to specify a key for the encryption. You can also specify an encryption provider and encryption algorithm other than the defaults with the `encryptionProvider=providerName` and `encryptionAlgorithm=algorithm` attributes For more information about data encryption, see Encrypting databases on disk .

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
bootPassword=DBpassword
```

Booting an encrypted database

You must specify the encryption key with the *bootPassword=key* attribute for an

encrypted database when you boot it (which is the first time you connect to it within a JVM session or after shutting it down within the same JVM session). For more information about data encryption, see Encrypting databases on disk .

```
jdbc:derby:encryptedDB;bootPassword=DBpassword
```

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:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Properties p = new Properties();
p.put("user", "sa");
p.put("password", "manager");

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

After installing

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

Review the `install.html` file that is installed with Derby for information on installing the Derby development environment. 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

The installation program installs the Derby software in a directory of your choice. See the `install.html` file for information on how to install Derby.

The installer automatically creates setup scripts that include an environment variable called DERBY_INSTALL. The installer's value is set to the Derby base directory.

```
C:>echo %DERBY_INSTALL%
C:\DERBY_INSTALL
```

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 installer for Derby installs 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*.
- */demo* contains some sample applications, useful scripts, and prebuilt databases.
  - */databases* includes prebuilt sample databases.
  - */programs* includes sample applications.
- */doc* contains the on-line documentation (including this document).
- */frameworks* contains utilities and scripts for running Derby.
- */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 */frameworks/embedded/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 can require modification in order to run properly on certain platforms.

Derby and JVMs

Derby is a database engine written completely in Java; it will run in any JVM, version 1.3 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

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.

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.

Embedded Derby JDBC driver

The Derby driver class name for the embedded environment is *org.apache.derby.jdbc.EmbeddedDriver*. In a Java application, you typically load the driver with the static `Class.forName` method or with the `jdbc.drivers` system property. For more information, see "Starting Derby as an Embedded Database".

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

Embedded Derby JDBC database connection URL

The standard Derby JDBC connection URL, which you can use for tasks besides connecting to a database, is

```
jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]*
```

Subsubprotocol, which is not typically specified, determines how Derby looks for a database: in a directory, in a class path, or in a jar file. Subsubprotocol is one of the following:

- *directory* The default. Specify this explicitly only to distinguish a database that might be ambiguous with one on the class path.
- *classpath* Databases are treated as read-only databases, and all *databaseNames* must begin with at least a slash, because you specify them "relative" to the classpath directory.
- *jar* Databases are treated as read-only databases. *DatabaseNames* might require a leading slash, because you specify them "relative" to the jar file.

jar requires an additional element immediately before the database name:

```
(pathToArchive)
```

pathToArchive is the path to the jar or zip file that holds the database.

For examples of using this syntax, see Accessing a read-only database in a zip/jar file .

You typically pass the connection URL as an argument to the JDBC *DriverManager.getConnection* method call. For example:

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

You can specify attributes and attribute values to a connection URL. For more information about what you can specify with the Derby connection URL, see Database

connection examples . For detailed reference about attributes and values, see the *Derby Reference Manual* .

Getting a nested connection

When you are executing a method within SQL, that method might need to reuse the current connection to the database in order to execute more SQL statements. Such a connection is called a *nested connection*. The way for a method to get a nested connection is to issue a connection request using the connection URL.

```
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.

Starting Derby as an embedded database

To start Derby, you start the Derby JDBC driver. Starting the Derby driver starts up the complete Derby system within the current JVM.

For example, when using the JBDC driver manager directly within Java code, you typically start a JDBC driver in one of two ways:

- Specify the *jdbc.drivers* system property, which allows users to customize the JDBC drivers used by their applications. For example:

  ```
  java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
  applicationClass
  ```

- Load the class directly from Java code using the static method *Class.forName*. For example:

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

For more details, see *"java.sql.Driver"* in the *Derby Reference Manual* .

Once the Derby JDBC driver class has been loaded, you can connect to any Derby database by passing the embedded connection URL with the appropriate attributes to the *DriverManager.getConnection* method.

For example:

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

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.

Figure1. Derby embedded in a single-user Java application

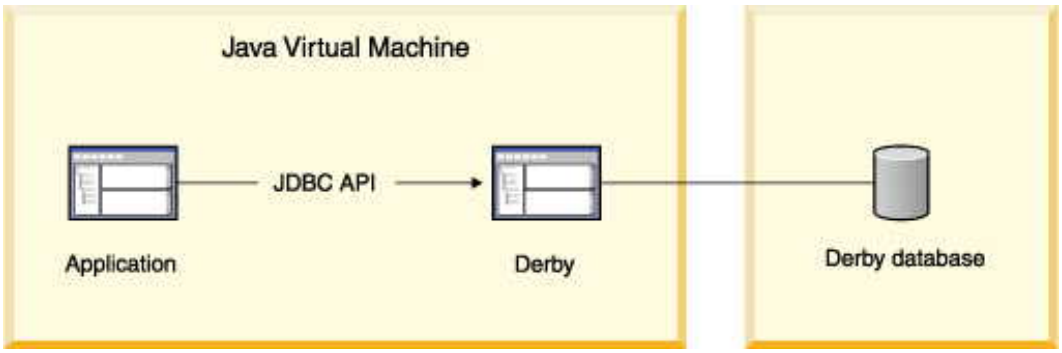
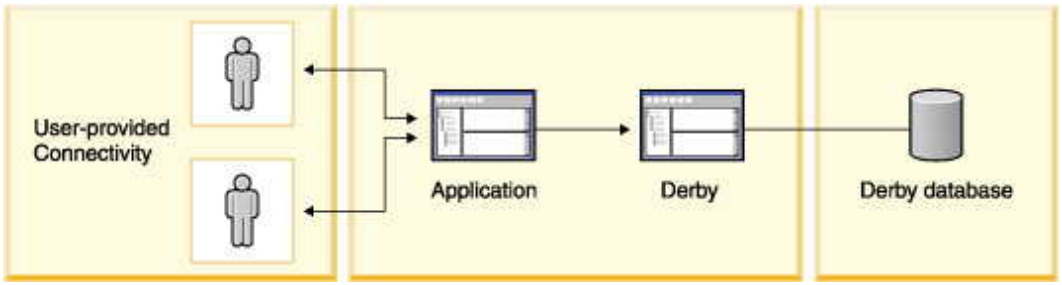


Figure1. Derby embedded in a multi-user Java application server



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

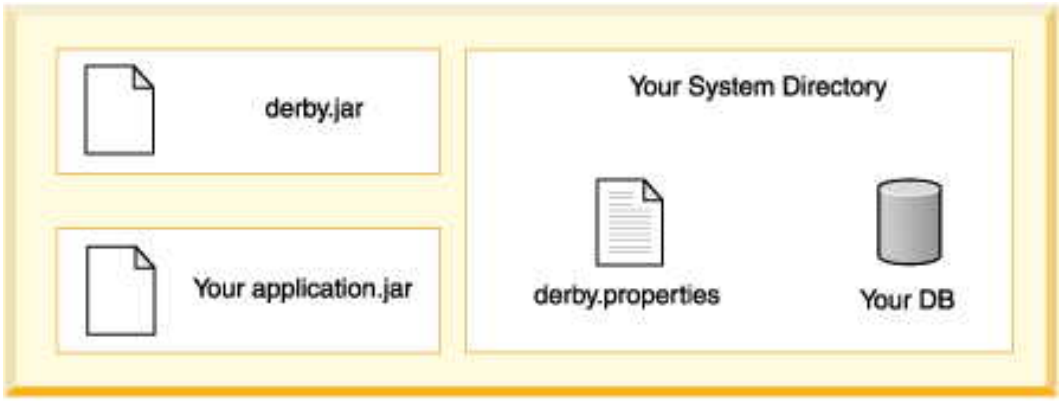
You can "embed" Derby in any Java application (single- or multi-user) by deploying the following package:

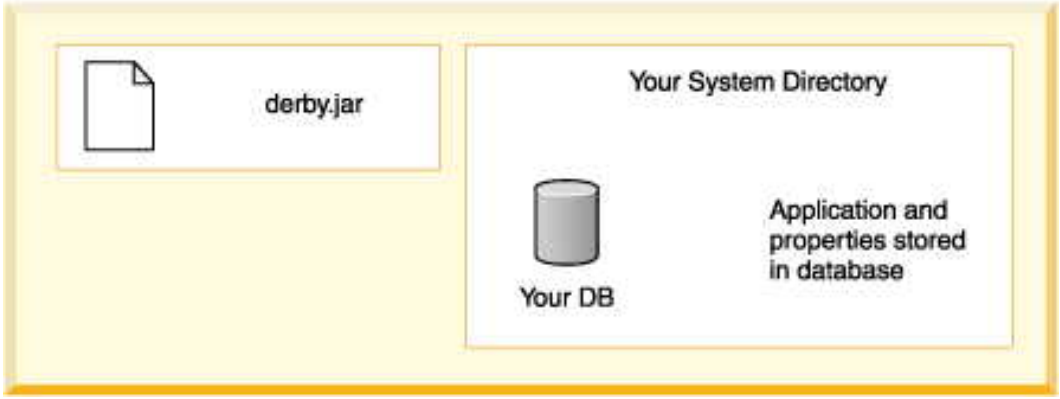
- the Derby library (*derby.jar*)
- the application's libraries

  You have the option of storing these libraries in the database. (See Loading classes from a database .)

- the database or databases used by the application, in the context of their system directory (see Embedded systems and properties)

Figure1. Deploying an application, embedded Derby software, and the database. Storing the application in the database and setting properties as database-wide properties simplify deployment.





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.

Before deploying your application, read Recommended practices .

Extra steps are required for deploying an application and an embedded database on read-only media. See Creating Derby databases for read-only use .

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:

1. Create and populate the database on read-write media.
2. Commit all transactions and shut down Derby in the prescribed manner (see Shutting down Derby or an individual database). 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 *Tuning Derby* .

   ```
   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, see *Tuning Derby* .

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

Be sure to set these properties so that they are deployed with the database. For more information, see Embedded systems and properties .

Deploying the database on the read-only media

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. Follow the instructions for creating a database for use on read-only media. See Creating and preparing the database for read-only use .
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.

Databases in a jar or zip file are not booted automatically when Derby is started, unless they are explicitly listed as *derby.service* properties.

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 do not have to specify 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 one of the following connection URLs:

   Standard syntax:
   ```
   jdbc:derby:/databasePathWithinArchive
   ```

   Syntax with subsubprotocol:
   ```
   jdbc:derby:classpath:/databasePathWithinArchive
   ```

For example:

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

Connecting to databases with ambiguous paths to databases in the file system

Use the basic connection URL syntax only if the database path specified does not also point to a Derby database in the file system. If this is the case, the connection attempt might fail or connect to the wrong database. Use the form of the syntax with the subsubprotocol to distinguish between the databases.

For example:

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

Connecting to databases when the path is ambiguous because of databases in the classpath

To connect to a database in the file system when the connection URL that you would use would be ambiguous with a database in the classpath, use the following form of the connection URL:

```
jdbc:derby:directory:databasePathInFileSystem
```

For example,

```
jdbc:derby:directory:/products/boiledfood
```

Apart from the connection URL, databases in archives in the classpath behave just like databases in archives accessed through the file system. However, databases in archives are read-only.

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:

1. Create jar files for your application
2. Add the jar file or files to the database
3. Enable database class loading with a property
4. Code your applications

Note: If you are interested in making changes to jar files stored in the database or changing the database jar "classpath" of your application without having to re-boot, read Dynamic changes to jar files or to the database jar classpath .

Signed jar files

For information about how Derby handles signed jar files, see Signed jar files

Create jar files for your application

Include any Java classes in a jar file intended for Derby class loading except:

- 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.

- those supplied with your Java environment (for example, *sun.**)

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 in your jar file.*

  Best if you need only a small subset of the classes in the third-party jar file.

- *Store the third-party jar file in the database.*

  Best if 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.*

  Best if the classes are often already installed on a user's machine (for example, Objectspace's JGL classes).

Include the class files and resources needed for your application.

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 *SQL92Identifier* .

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 the *Derby Tools and Utilities Guide* for reference information about the utility and complete syntax.

Installing jar files:

```
-- 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:

```
-- SQL  statement
CALL sqlj.remove_jar(
    'APP.Sample1', 0)
```

Replacing jar files:

```
-- 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 class path, 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 two-part names for the jar files (schema name and jar name). Set the property as a database-level property for the database. The *first time* you set the property, you must reboot to load the classes.

Example:

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

See *"derby.database.classpath"* in *Tuning Derby* 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 cause classes to be loaded the way you normally would:

- Indirectly referencing them in the code
- Directly using *java.lang.Class.forName*

In your applications, 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 you 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

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.

These features include such programming as database-side JDBC procedures and triggers.

Programming database-side JDBC procedures

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

An application-side method can be exactly the same as a database-side procedure. 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 procedures and nested connections

Most database-side JDBC Procedures need to share the same transaction space as the statements that called them for the following reasons:

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

In order to use the same transaction, the procedure must use the same connection as the parent SQL statement in which the method 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 Connection 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` :

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

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

Requirements for database-side JDBC procedures using nested connections
In order to preserve transactional atomicity, database-side JDBC procedures that use nested connections:

- cannot issue a commit or rollback, unless called within a CALL statement.
- cannot change connection attributes such as auto-commit.
- cannot modify the data in a table used by the parent statement that called the procedure, 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 procedure.
- 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 procedures using non-nested connections

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

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

Note: If database-side JDBC procedures 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 the CallableStatement method. You can invoke the procedure in an SQL statement such as the following:

```
CALL MYPROC()
```

Note: You cannot roll back this statement, because commits occur within the procedure itself. Procedures that use nested connections, on the other hand, are not permitted to commit or roll back and can therefore be rolled back after the calling statement.

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

Database-side JDBC procedures and SQLExceptions

It is possible to code database-side procedures, like application-side methods, to catch *SQLExceptions*. `SQLExceptions` that are caught within a procedure 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 procedure 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 in the *Derby Reference Manual* goes into detail of 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 `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);)
```

A trigger action does have some limitations, though; for example, it cannot contain dynamic parameters or alter the table on which the trigger is defined. See "TriggerAction" in the *Derby Reference Manual* for details.

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 to obtain this

functionality, rather than writing triggers. See the *Derby Reference Manual* for more information on referential integrity.

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" in the *Derby Reference Manual* .

Examples

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 MODE DB2SQL
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');
```

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 to 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.

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 procedures, 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 procedures 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 (see Derby 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 (see Derby JDBC database connection URL).

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*. For more information about how to use cursors and `ResultSets` , see SQL and JDBC ResultSet/Cursor mechanisms .

Nested connections

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

For more information, see Programming database-side JDBC procedures .

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 (see Using auto-commit), 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. (See Holdable cursors for more information.) It also releases any database locks currently held by the `Connection` , whether or not these objects were created in different threads.

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 `ResultSet` , the statement completes when the last row of the `ResultSet` has been retrieved or 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 *close cursors on commit* option set.

  Auto-commit automatically closes cursors, which are explicitly opened with the *close on commit* option, when you do any in-place updates or deletes. For more information about cursors, see SQL and JDBC ResultSet/Cursor mechanisms .

  A cursor declared to be held across commit can execute updates and issue multiple commits before closing the cursor, but the cursor must be repositioned before any statement following the commit. If this is attempted with auto-commit on, an error is generated.

- *Database-side JDBC Procedures (procedures using nested connections)*

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

  Procedures that use nested connections are not permitted to turn auto-commit on or off or to commit or roll back.

- *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.*

  Table1. Summary of 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() begins a transaction.
Database-side JDBC procedures (routines using nested connections)	Auto-commit is turned off.	Works (no explicit commits or rollbacks are allowed).
Updatable cursors	Does not work.	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. Not required by the JDBC program.

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

For example, close a *Statement* object using its `close` method; close a `Connection` object using its `close` method. If auto-commit is disabled, active transactions need to be explicitly committed or rolled back before closing the connection

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 the *Derby Reference Manual* for more information.

SQL and JDBC ResultSet/Cursor mechanisms

A cursor provides you with the ability to step through and process the rows in a *ResultSet* one by one. A `java.sql.ResultSet` object constitutes a cursor. You do not need to use a language construct, such as SQL-92's DECLARE CURSOR, to work with cursors in a Java application. In Derby, any SELECT statement generates a cursor.

Simple non-updatable ResultSets

Here is an excerpt from a sample JDBC application that generates a *ResultSet* 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 cursors

Cursors are read-only by default. For a cursor to be updatable, you must specify SELECT . . . FOR UPDATE. Use FOR UPDATE only when you will be modifying rows to avoid excessive locking of tables.

Requirements for updatable cursors

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* .

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 cursor example

```
String URL = "jdbc:derby:sample";
// autocommit must be turned off for updatable cursors
conn.setAutoCommit(false);
Statement s3 = conn.createStatement();
// name the statement so we can reference the result set
// it generates
s3.setCursorName("UPDATABLESTATEMENT");
// Updatable statements have some requirements
// for example, select must be on a single table
ResultSet Updatable = s3.executeQuery(
```

```
    "SELECT firstnme, lastname, workdept, bonus" +
    "FROM employee FOR UPDATE of bonus");
// we need a separate statement to do the
// update PreparedStatement
PreparedStatement ps2 = conn.prepareStatement("UPDATE employee " +
// we could use the cursor name known by the system,
// as the following line shows
//"SET bonus = ? WHERE CURRENT OF " + Updatable.getCursorName());
// but we already know the cursor name
"SET bonus = ? WHERE CURRENT OF UPDATABLESTATEMENT");
String theDept="E21";
while (Updatable.next()) {
    String firstnme = Updatable.getString("FIRSTNME");
    String lastName = Updatable.getString("LASTNAME");
    String workDept = Updatable.getString("WORKDEPT");
    BigDecimal bonus = Updatable.getBigDecimal("BONUS");
    if (workDept.equals(theDept)) {
    // if the current row meets our criteria,
    // update the updatable column in the row
        ps2.setBigDecimal(1, bonus.add(new BigDecimal(250)));
        ps2.executeUpdate();
        System.out.println("Updating bonus in employee" +
        " table for employee " + firstnme +
        ", department " + theDept );
    }
    }
Updatable.close();
s3.close();
ps2.close();
conn.commit();
```

ResultSets and auto-commit

Except for the result sets associated with holdable cursors (see Holdable cursors for more information), issuing a commit will cause all result sets on your connection to be closed.

The JDBC program is not required to have auto-commit off when using updatable ResultSets.

Scrolling insensitive ResultSets

JDBC 2.0 adds a new kind of *ResultSet* , one that allows you to scroll in either direction or to move the cursor to a particular row. Derby implements scrolling insensitive *ResultSets* . When you use a scroll insensitive *ResultSets* cursor to facilitate the insensitive scrolling action, Derby materializes in memory all rows from the first one in the result set up to the one with the biggest row number.

```
//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())
    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();
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 cursors

Note: Non-holdable cursors are only available in Java 2 Platform, Standard Edition, v 1.4 (J2SE) environments.

The holdable cursor feature permits an application to keep cursors open after implicit or explicit commits. By default, the cursors are held open after a commit. Starting with Java 2 Platform, Standard Edition, v 1.4 (J2SE), cursors can be created with close when a commit occurs option. Such cursors will be automatically closed when a commit happens. Cursors are automatically closed when a transaction aborts, whether or not they have been specified to be held open.

Note: Holdable cursors do not work with XA transactions, in Derby Version 10.1, therefore cursors should be opened with holdability false when working with XA transactions.

To specify whether a cursor 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*

  Cursors are closed when an implicit or explicit commit is performed.

- *HOLD_CURSORS_OVER_COMMIT*

  Cursors are held open when a commit is performed, implicitly or explicitly. This is the default behavior.

The method *Statement.getResultSetHoldability()* indicates whether a cursor 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, *ResultSets* that hold cursors open behave as follows:

- Open *ResultSets* remain open. The cursor is positioned before the next logical row of the result set.
- When the session is terminated, the *ResultSet* is closed and destroyed.
- All locks are released, except locks protecting the current cursor position of open cursors specified to stay open after commits.
- Immediately following a commit, the only valid operations that can be performed on the *ResultSet* are:
  - positioning the *ResultSet* to the next valid row in the result with `ResultSet.next()`.
  - closing the *ResultSet* with `ResultSet.close()`.

When a rollback occurs either explicitly or implicitly, the following behavior applies:

- All open *ResultSets* are closed.
- All locks acquired during the unit of work are released.

Holdable cursors and autocommit

When autocommit is on, positioned updates and deletes are not supported for *ResultSet* objects that hold cursors open. If you attempt an update or delete, an exception is thrown.

Non-holdable cursor 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.setIsolationLevel* method or the SQL SET ISOLATION statement. The names of the isolation levels are different, depending on whether you use a JDBC method or SQL statement. Mapping of JDBC transaction isolation levels to Derby isolation levels shows the equivalent names for isolation levels whether they are set through the JDBC method or an SQL statement.

Table1. 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 Transaction Anomalies .

Table1. 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.)

Non-Repeatable Reads
 	
Non-repeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Non-repeatable 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
 	
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.
	Transaction A begins.
	
	`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.

When Transaction Anomalies Are Possible shows which anomalies are possible under the various locking schemas and isolation levels.

Table1. When Transaction Anomalies Are Possible

Isolation Level	Table-Level Locking	Row-Level Locking
TRANSACTION_READ_UNCOMMITTED	Dirty reads, nonrepeatable reads, and phantom	Dirty reads, nonrepeatable reads, and phantom reads

Isolation Level	Table-Level Locking	Row-Level Locking
	reads possible	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*. For more information about range locks, see Scope of locks .

- `TRANSACTION_REPEATABLE_READ`

  `RS` from SQL.

  TRANSACTION_REPEATABLE_READ means that Derby issues locks to prevent only dirty reads and non-repeatable 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 the "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.

Note: For information about how to choose a particular isolation level, see *Tuning Derby* .

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 *Tuning Derby* . 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".

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) 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, and transactions using the TRANSACTION_SERIALIZABLE or TRANSACTION_REPEATABLE_READ isolation level downgrade it to a shared lock until the transaction is committed. (For update locks, the TRANSACTION_READ_UNCOMMITTED isolation level acts the same way as TRANSACTION_READ_COMMITTED.)

Update locks help minimize deadlocks.

Lock compatibility

Lock Compatibility Matrix lists compatibility between lock types. + means compatible, - means incompatible.

Table1. Lock Compatibility Matrix

'	Shared	Update	Exclusive
Shared	+	-	-
Update	+	-	-
Exclusive	-	-	-

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.

  This section applies only to row-level locking systems.

For TRANSACTION_READ_COMMITTED or TRANSACTION_REPEATABLE_READ isolation, Derby treats rows as cursors for SELECT statements. It 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).

  This section applies only to 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.

Table1. Possible Types and Scopes of Locking

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
Connection. TRANSACTION_ READ_UNCOMMITED (SQL: UR)	For SELECT statements, table-level locking is never requested using this isolation level. For other statements, same as for TRANSACTION_ READ_ COMMITTED.	SELECT statements get no locks. For other statements, same as for TRANSACTION_ READ_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_	Same as for TRANSACTION_	SELECT statements get shared locks on the rows that satisfy the WHERE clause (but do

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
REPEATABLE_READ (SQL: RS)	SERIALIZABLE	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).
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).

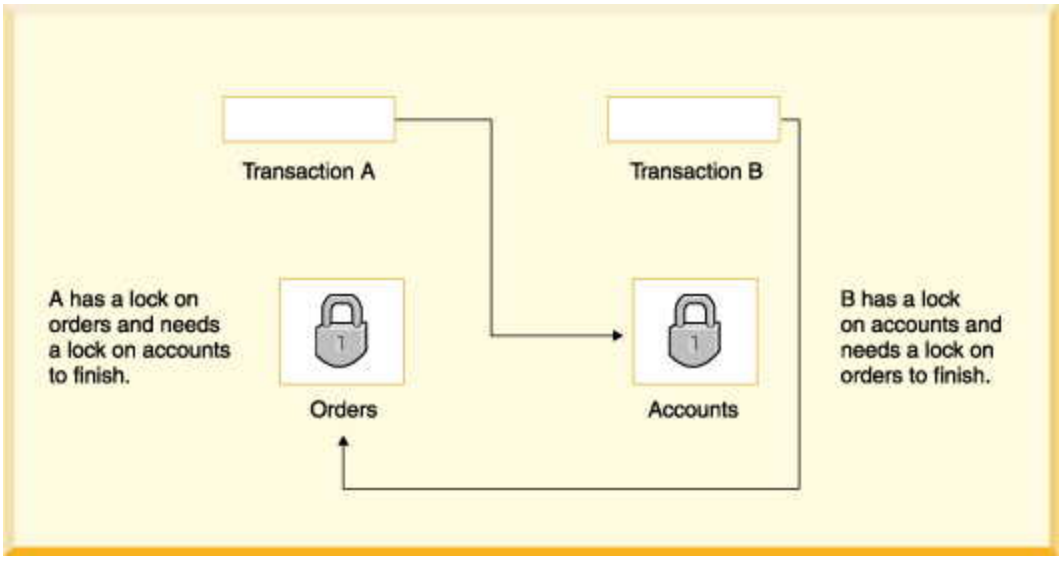
The table and examples in this section do not take performance-based lock escalation into account. Remember that the system can choose table-level locking for performance reasons.

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.

Figure1. A deadlock.



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.

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*. (For information about configuring the lock wait timeout, see Configuring deadlock detection and lock wait timeouts .)

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.

Figure1. One possible configuration: deadlock checking occurs when a transaction has waited 30 seconds; no lock wait timeouts occur.

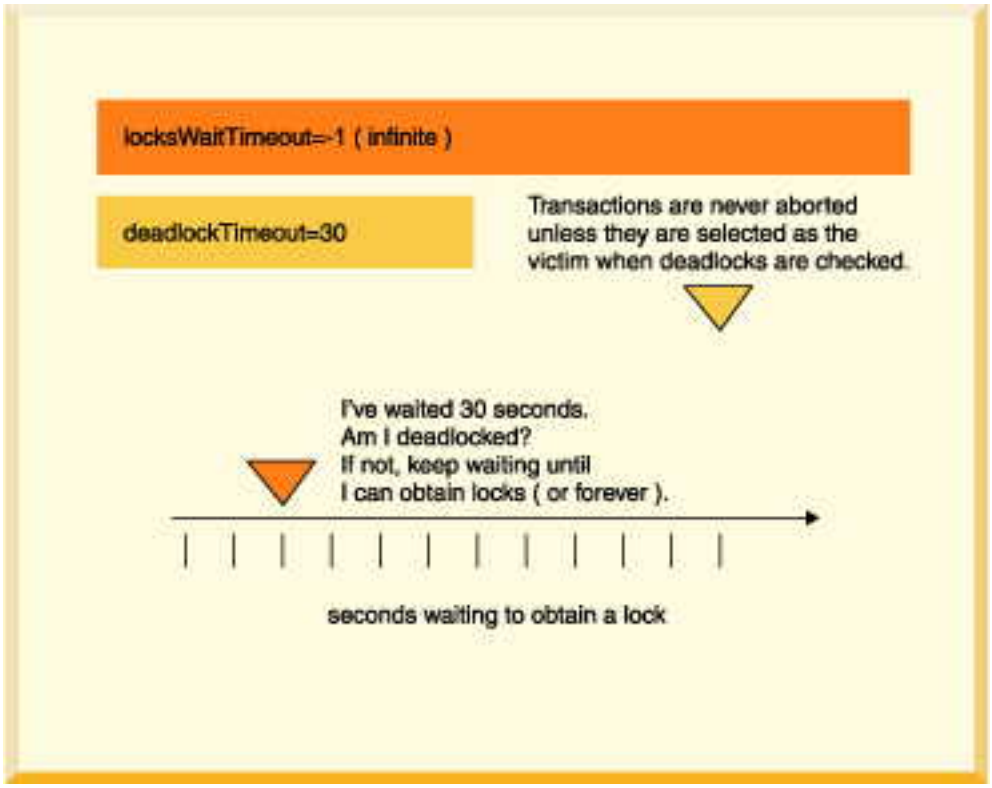


Figure1. Another typical configuration: deadlock checking occurs after a transaction has waited 60 seconds for a lock; after 90 seconds, the transaction times out and is rolled back.

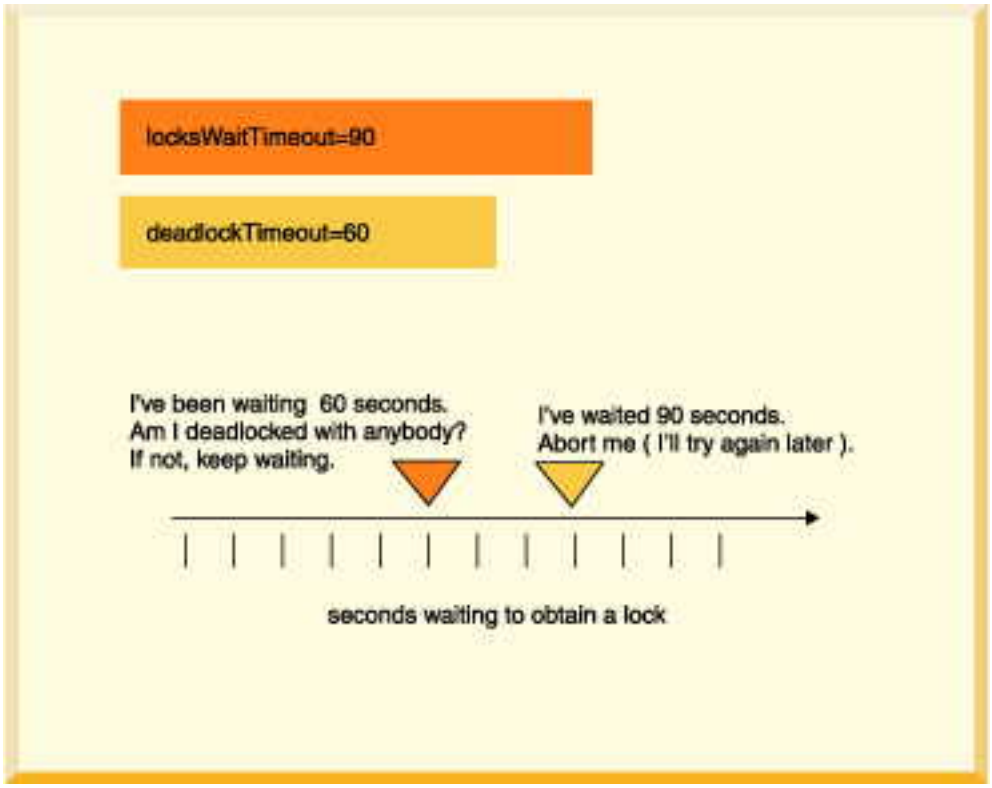
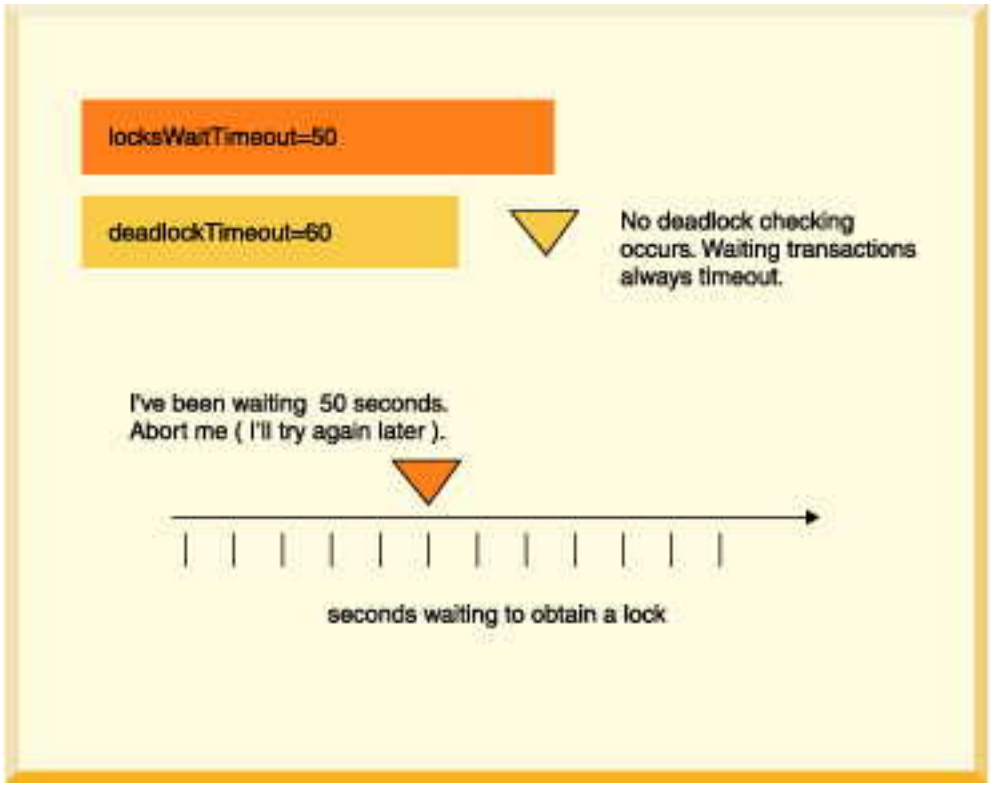


Figure1. A configuration in which no deadlock checking occurs: transactions time out after they have waited 50 seconds. No deadlock checking occurs.



Debugging Deadlocks

If deadlocks occur frequently in your multi-user system with a particular application, you might need to do some debugging. Derby provides a class to help you in this situation, *org.apache.derby.diag.LockTable*. 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 the Tuning Guide for more information on this property. For information, see the *Derby Server and Administration Guide* .

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 this. To do this, test for *SQLExceptions* with *SQLStates* of 40001 (deadlock timeout) or 40XL1 or 40XL2 (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 procedures 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 the following 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 Threading and Connection Modes .

Table1. Threading and Connection Modes

Connection mode	Embedded	Server
Multi-Threaded From an application, using a *single* `Connection` 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. For more information, see Transactions .	Server frameworks can automatically multi-thread operations. Remote client applications can multi-thread if desired.
Multi-Connection From an application, using multiple connections to a Derby database and issuing requests against those connections on multiple threads.	Create individual connections within a single application and use the 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.	Remote client applications can establish the multiple connections desired.
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. When using a pre-1.4 JVM, Derby might not prevent multiple applications from concurrently accessing the same Derby system, but do not allow this because such access can corrupt the databases involved.	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. For more information, see Locking, concurrency, and isolation .

Multiple connections from a single application

A single application can work with multiple *Connections* to the same database and assign them to different threads. The application programmer can avoid concurrency and deadlock problems 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

Some programmers might share a *Connection* among multiple threads because they 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 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

Do not use `interrupt` calls to notify threads that are accessing a database, because Derby will catch the `interrupt` call and close the connection to the database. Use `wait` and `notify` calls instead.

This will not happen in a client/server environment, but if you want your application to work in either environment it is good practice to follow this rule.

There are special considerations when working with more than one database thread in an application. See *Working with multiple connections to a single database* and *Working with multiple threads sharing a single connection* .

Working with Derby SQLExceptions in an Application

JDBC generates exceptions 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 SQLExceptions

Derby provides the message, *SQLState* values, and error codes. Use the `getSQLState` and `getMessage` methods to view the `SQLState` and error messages. Use `getErrorCode` to see the error code. The error code defines the severity of the error and is not unique to each exception. The severity levels are described in `org.apache.derby.types.ExceptionSeverity` .

Applications should also check for and process *java.sql.SQLWarnings*, 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

In addition, a single error can generate more than one *SQLException*. Use a loop and the `getNextException` method to process all `SQLExceptions` 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();
    }
}
```

See also "Derby Exception Messages and SQL States", in the *Derby Reference Manual* .

Using Derby as a J2EE resource manager

J2EE, or the Java 2 Platform, Enterprise Edition, 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 2 Platform, Standard Edition (J2SE), J2EE adds support for Enterprise Java Beans (EJBs), Java Server Pages (JSPs), Servlets, XML and many more. The J2EE architecture is used to bring together existing technologies and enterprise applications in a single, manageable environment.

Derby is a J2EE-conformant component in a distributed J2EE system. As such, it 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 J2EE and how to work in this environment, see the J2EE specification available at *http://java.sun.com/j2ee/docs.html* .

Note: This chapter 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 J2EE system, not to end-user application developers.

In order to qualify as a resource manager in a J2EE system, J2EE requires these basic areas of support. These three areas of support involve implementation of APIS and are described in "J2EE 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.

Note: All of the classes described in this chapter require a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Classes that pertain to resource managers

See the javadoc for each class for more information.

- *org.apache.derby.jdbc.EmbeddedDataSource*

  Implements *javax.sql.DataSource* interface, which a JNDI server can reference. Typically this is the object that you work with as a `DataSource` .

- *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource*

  Implements *javax.sql.ConnectionPoolDataSource*. A factory for `PooledConnection` objects.

- *org.apache.derby.jdbc.EmbeddedXADataSource*

  Derby's implementation of a *javax.sql.XADataSource*.

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. If a database named wombat located at /local1/db/wombat is to be accessed, then one should call *setDatabaseName*("/local1/db/wombat") on the data source object.

- *CreateDatabase*

  Optional. Sets a property to create a database the next time the *XADataSource.getXAConection()* method 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 `XADataSource.getXAConnection().getConnection()` method is called.

- *DataSourceName*

  Optional. Name for ConnectionPooledDataSource or XADataSource. Not used by the data source object. Used for informational purpose only.

- *Description*

  Optional. Description of the data source. Not used by the data source object. Used for informational purpose 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 the following 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.

Derby and Security

Derby can be deployed in a number of ways and in a number of different environments. The security needs of the Derby system are also diverse. Derby supplies or supports the following optional security mechanisms:

- *User authentication*

  Derby verifies user names and passwords before permitting them access to the Derby system. See Working with user authentication .

- *User authorization*

  A means of granting specific users permission to read a database or to write to a database. See User authorization .

- *Disk encryption*

  A means of encrypting Derby data stored on disk. See Encrypting databases on disk .

- *Validation of Certificate for Signed Jar Files*

  In a Java 2 environment, Derby validates certificates for classes loaded from signed jar files. See Signed jar files .

Figure1. Some of the Derby security mechanisms at work in a client/server environment

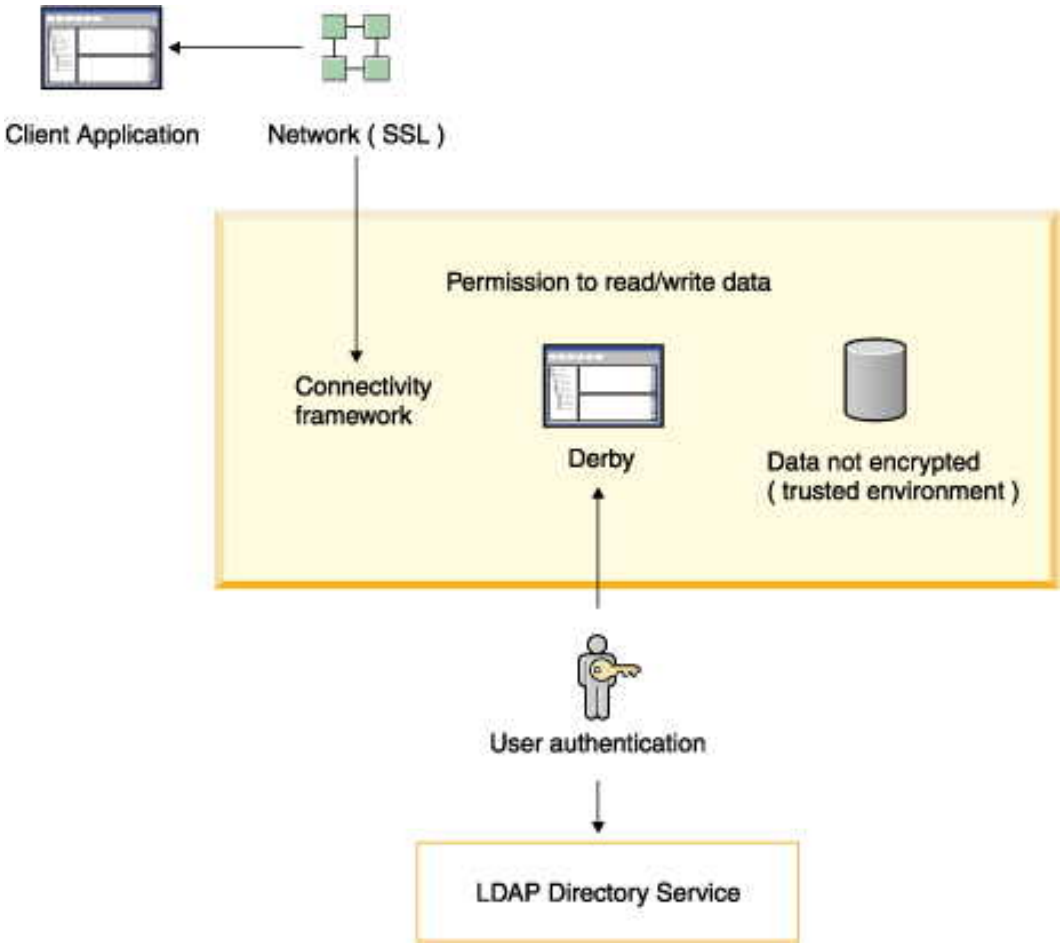
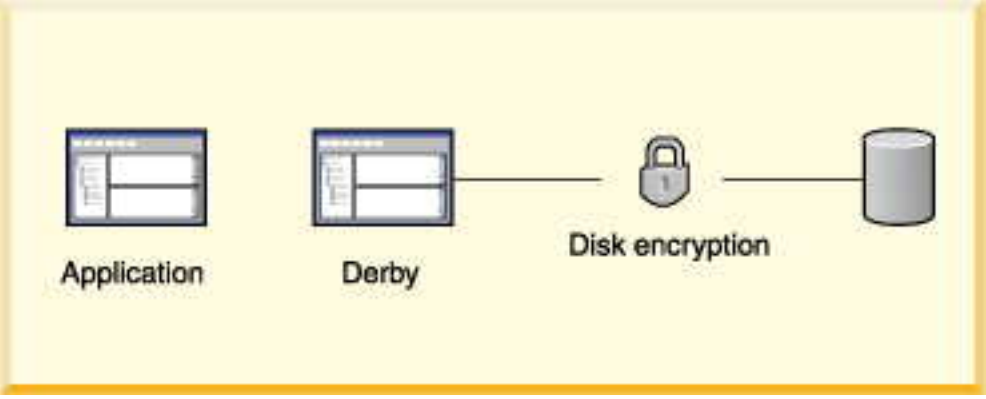


Figure1. Another Derby security mechanism, disk encryption, protects data when the recipient might not know how to protect data. It is useful for databases deployed in an embedded environment.



Configuring security for your environment

In most cases, you enable Derby's security features through the use of properties. It is important to understand the best way of setting properties for your environment.

Configuring security

Derby does not come with a built-in superuser. For that reason, be careful when configuring Derby for user authentication and user authorization.

1. When first working with security, work with system-level properties only so that you can easily override them if you make a mistake.
2. Be sure to create at least one valid user, and grant that user full (read-write) access. For example, you might always want to create a user called *sa* with the password *derby* while you are developing.
3. Test the authentication system while it is still configured at the system level. Be absolutely certain that you have configured the system correctly before setting the properties as database-level properties.
4. Before disabling system-level properties (by setting *derby.database.propertiesOnly* to true), test that at least one database-level read-write user (such as *sa*) is valid. If you do not have at least one valid user that the system can authenticate, you will not be able to access your database.

Configuring security in a client/server environment

This procedure requires a system with multiple databases and some administrative resources. For systems that have a single database and for which there are no administrative resources, follow the instructions in Configuring security in an embedded environment .

1. Configure security features as system properties. See *Tuning Derby* .
2. Provide administrative-level protection for the *derby.properties* file and Derby databases. For example, you can protect these files and directories with operating system permissions and firewalls.
3. Turn on user authentication for your system. All users must provide valid user IDs and passwords to access the Derby system. See Working with user authentication for information. If you are using Derby's built-in users, configure users for the system in the *derby.properties* file. Provide the protection for this file.
4. Configure user authorization for sensitive databases in your system. Only designated users will be able to access sensitive databases. You typically configure

user authorization with database-level properties. See User authorization for information. It is also possible to configure user authorization with system-level properties. This is useful when you are developing systems or when all databases have the same level of sensitivity.

Configuring security in an embedded environment

In an embedded environment, typically there is only one database per system and there are no administrative resources to protect databases.

1. Encrypt the database when you create it.
2. Configure all security features as database-level properties. These properties are stored in the database (which is encrypted). See *Tuning Derby* .
3. Turn on protection for database-level properties so that they cannot be overridden by system properties by setting the *derby.database.propertiesOnly* property to TRUE.
4. To prevent unauthorized users from accessing databases once they are booted, turn on user authentication for the database and configure user authorization for the database. See Working with user authentication and User authorization for more information.
5. If you are using Derby's built-in users, configure each user as a database-level property so that user names and passwords can be encrypted.

Working with user authentication

Derby provides support for user authentication. *User authentication* means that Derby authenticates a user's name and password before allowing that user access to the system.

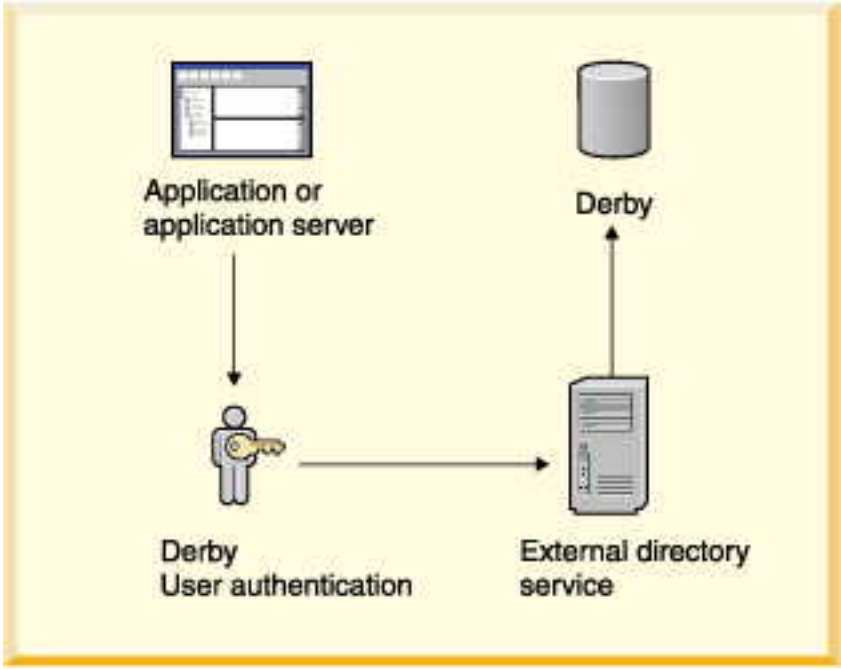
When user authentication is enabled (which it is not by default), the user requesting a connection must provide a valid name and password, which Derby verifies against the repository of users defined for the system. Once Derby authenticates the user, it grants the user access to the Derby system but not necessarily access to the database made in the connection request. In the Derby system, access to a database is determined by *user authorization*. For information, see User authorization .

Derby allows you to provide a repository of users in a number of different ways. For example, you can hook Derby up to an external directory service elsewhere in your enterprise, create your own, use Derby's simple mechanism for creating a built-in repository of users.

You can define a repository of users for a particular database or for an entire system, depending on whether you use system-wide or database-wide properties. See Configuring security for your environment for more information.

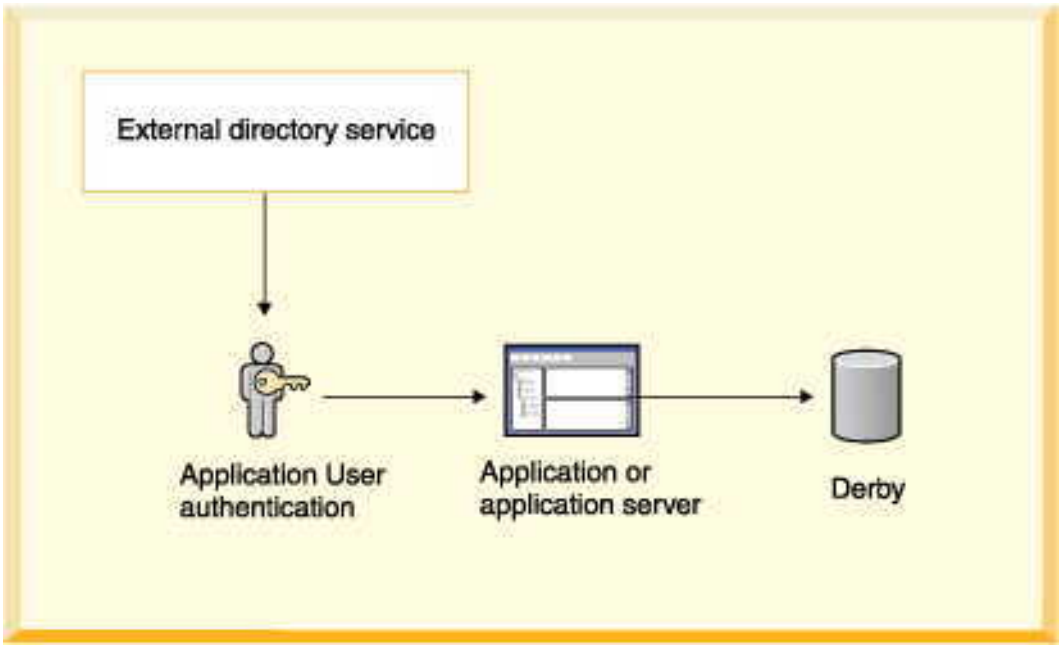
When Derby user authentication is enabled and Derby uses an external directory service, the architecture looks something like that shown in the Figure below:

Figure1. Derby user authentication using an external service. The application can be a single-user application with an embedded Derby engine or a multi-user application server.



Derby always runs embedded in another Java application, whether that application is a single-user application or a multiple-user application server or connectivity framework. A database can be accessed by only one JVM at a time, so it is possible to deploy a system in which the application in which Derby is embedded, not Derby, handles the user authentication by connecting to an external directory service.

Figure1. The application provides the user authentication using an external service. The application can be a single-user application with an embedded Derby engine or a multi-user application server.



Enabling user authentication

To enable user authentication, set the *derby.connection.requireAuthentication* property to true. Otherwise, Derby does not require a user name and password. You can set this property as a system-wide property or as a database-wide property.

For a multi-user product, you would typically set it for the system in the *derby.properties* file for your server, since it is in a trusted environment.

Note: If you start a Derby system with user authentication enabled but without defining at least one user, you will not be able to shut down the system gracefully. When Derby is running in a connectivity server and user authentication is turned on, stopping the server requires a user name and password. You will need to alter shutdown scripts accordingly.

Defining users

Derby provides several ways to define the repository of users and passwords. To specify which of these services to use with your Derby system, set the property *derby.authentication.provider* to the appropriate value as discussed in the appropriate section listed below.

Setting the property as a system-wide property creates system-wide users. Setting the property as a database-wide property creates users for a single database only.

- External directory service : LDAP directory service . This includes Windows NT domain user authentication through the Netscape NT Synchronization Service.
- User-defined class
- Built-in Derby users

Note: Shutting down the Derby system (for example, using the *shutdown=true* form of the connection URL without specifying a particular database) when user authentication is turned on requires that you define at least one user as a system-wide user.

External directory service

A directory service stores names and attributes of those names. A typical use for a directory service is to store user names and passwords for a computer system. Derby uses the Java naming and directory interface (JNDI) to interact with external directory services that can provide authentication of users' names and passwords.

LDAP directory service

You can allow Derby to authenticate users against an existing LDAP directory service within your enterprise. LDAP (lightweight directory access protocol) provides an open directory access protocol running over TCP/IP. An LDAP directory service can quickly authenticate a user's name and password.

To use an LDAP directory service, set *derby.authentication.provider* to *LDAP*.

Examples of LDAP service providers are:

- Netscape Directory Server

  Netscape Directory Server is an LDAP directory server. In addition, the Netscape Directory Synchronization Service synchronizes entries in a Windows NT directory with the entries in Netscape's Directory Server. It allows you to use the Windows NT directory as a repository for Derby users.

- UMich slapd (freeware for the UNIX platform from the University of Michigan)
- AE SLAPD for Windows NT, from AEInc

Libraries for LDAP user authentication:

To use an LDAP directory service with Derby, you need the following libraries in your classpath:

- *jndi.jar*

  JNDI classes

- *ldap.jar*

  LDAP provider from Sun

- *providerutil.jar*

  JNDI classes for a provider

Derby does not provide these libraries; they are available from Sun on the JNDI page. Use the 1.1.x versions of these libraries, not the 1.2.x versions. You might need to do two separate downloads to obtain all the required libraries.

Setting up Derby to use your LDAP directory service:

When specifying LDAP as your authentication service, you must specify the location of the server and its port number.

- *derby.authentication.server*

  Set the property *derby.authentication.server* to the location and port number of the LDAP server. For example:

  ```
  derby.authentication.server=godfrey:389
  ```

Guest access to search for DNs:

In an LDAP system, users are hierarchically organized in the directory as a set of entries. An *entry* is a set of name-attribute pairs identified by a unique name, called a DN (distinguished name). An entry is unambiguously identified by a DN, which is the concatenation of selected attributes from each entry in the tree along a path leading from the root down to the named entry, ordered from right to left. For example, a DN for a user might look like this:

```
cn=mary,ou=People,o=FlyTours.com
uid=mary,ou=People,o=FlyTours.com
```

The allowable entries for the name are defined by the entry's *objectClass*.

An LDAP client can bind to the directory (successfully log in) if it provides a user ID and password. The user ID must be a DN, the fully qualified list of names and attributes. This means that the user must provide a very long name.

Typically, the user knows only a simple user name (e.g., the first part of the DN above, *mary*). With Derby, you do not need the full DN, because an LDAP client (Derby) can go to the directory first as a guest or even an anonymous user, search for the full DN, then rebind to the directory using the full DN (and thus authenticate the user).

Derby typically initiates a search for a full DN before binding to the directory using the full DN for user authentication. Derby does not initiate a search in the following cases:

- You have set *derby.authentication.ldap.searchFilter* to *derby.user*.
- A user DN has been cached locally for the specific user with the *derby.user.UserName* property.

For more information, see *derby.authentication.ldap.searchFilter* in *Tuning Derby* .

Some systems permit anonymous searches; other require a user DN and password. You can specify a user's DN and password for the search with the properties listed below. In addition, you can limit the scope of the search by specifying a filter (definition of the

object class for the user) and a base (directory from which to begin the search) with the properties listed below.

- *derby.authentication.ldap.searchAuthDN (optional)*

  Specifies the DN with which to bind (authenticate) to the server when searching for user DNs. This parameter is optional if anonymous access is supported by your server. If specified, this value must be a DN recognized by the directory service, and it must also have the authority to search for the entries.

  If not set, it defaults to an anonymous search using the root DN specified by the *derby.authentication.ldap.searchBase* property. For example:

  ```
  uid=guest,o=FlyTours.com
  ```

- *derby.authentication.ldap.searchAuthPW (optional)*

  Specifies the password to use for the guest user configured above to bind to the directory service when looking up the DN. If not set, it defaults to an anonymous search using the root DN specified by the *derby.authentication.ldap.searchBase* property.

  ```
  myPassword
  ```

- *derby.authentication.ldap.searchBase (optional)*

  Specifies the root DN of the point in your hierarchy from which to begin a guest search for the user's DN. For example:

  ```
  ou=people,o=FlyTours.com
  ```

  When using Netscape Directory Server, set this property to the root DN, the special entry to which access control does not apply (optional).

To narrow the search, you can specify a user's *objectClass*.

- *derby.authentication.ldap.searchFilter (optional)*

  Set *derby.authentication.ldap.searchFilter* to a logical expression that specifies what constitutes a user for your LDAP directory service. The default value of this property is `objectClass=inetOrgPerson` . For example:

  ```
  objectClass=person
  ```

LDAP performance issues:

For performance reasons, the LDAP directory server should be in the same LAN as Derby. Derby does not cache the user's credential information locally and thus must connect to the directory server every time a user connects.

Connection requests that provide the full DN are faster than those that must search for the full DN.

Considerations when using Windows NT with LDAP:

Netscape provides LDAP functionality for Windows NT systems with its Netscape Directory Synchronization service, which synchronizes the Windows NT users with the Netscape Directory Server. SSL is recommended in this configuration.

LDAP restrictions:

Derby does not support LDAP groups.

JNDI-specific properties for external directory services

Derby allows you to set a few advanced JNDI properties, which you can set in any of the supported ways of setting Derby properties. Typically you would set these at the same level (database or system) for which you configured the external authentication service.

The list of supported properties can be found in Appendix A: JNDI Context Environment in the Java Naming and Direction API at *http://java.sun.com/products/jndi/reference/api/index.html* . The external directory service must support the property.

Each JNDI provider has its set of properties that you can set within the Derby system.

For example, you can set the property *java.naming.security.authentication* to allow user credentials to be encrypted on the network if the provider supports it. You can also specify that SSL be used with LDAP (LDAPS).

User-defined class

Set *derby.authentication.provider* to the full name of a class that implements the public interface *org.apache.derby.authentication.UserAuthenticator*.

By writing your own class that fulfills some minimal requirements, you can hook Derby up to an *external authentication service* other than LDAP. To do so, specify an external authentication service by setting the property `derby.authentication.provider` to a class name that you want Derby to load at startup.

The class that provides the external authentication service must implement the public interface *org.apache.derby.authentication*.*UserAuthenticator* and throw exceptions of the type *java.sql.SQLException* where appropriate.

Using a user-defined class makes Derby adaptable to various naming and directory services.

Example of setting a user-defined class:

A very simple example of a class that implements the *org.apache.derby.authentication* interface:

```
import org.apache.derby.authentication.UserAuthenticator;
import java.io.FileInputStream;
import java.util.Properties;
import java.sql.SQLException;
/**
  * A simple example of a specialized Authentication scheme.
  * The system property 'derby.connection.requireAuthentication'
  * must be set
  * to true and 'derby.connection.specificAuthentication' must
  * contain the full class name of the overriden authentication
  * scheme,  i.e., the name of this class.
  *
  * @see org.apache.derby.authentication.UserAuthenticator
  */

public class MyAuthenticationSchemeImpl implements
UserAuthenticator {
    private static final String USERS_CONFIG_FILE = "myUsers.cfg";
    private static Properties usersConfig;

    // Constructor
    // We get passed some Users properties if the
    //authentication service could not set them as
    //part of System properties.
    //
    public MyAuthenticationSchemeImpl() {
    }
    /* static block where we load the users definition from a
users configuration file.*/

    static {
       /* load users config file as Java properties
```

```
         File must be in the same directory where
         Derby  gets started.
        (otherwise full path must be specified) */
        FileInputStream in = null;
        usersConfig = new Properties();
        try {
            in = new FileInputStream(USERS_CONFIG_FILE);
            usersConfig.load(in);
            in.close();
        } catch (java.io.IOException ie) {
            // No Config file. Raise error message
            System.err.println(
              "WARNING: Error during Users Config file
retrieval");
            System.err.println("Exception: " + ie);
        }
    }
    /**
     * Authenticate the passed-in user's credentials.
     * A more complex class could make calls
     * to any external users directory.
     *
     * @param userName              The user's name
     * @param userPassword          The user's password
     * @param databaseName          The database
     * @param infoAdditional jdbc connection info.
     * @exception SQLException on failure
     */
    public boolean authenticateUser(String userName,
     String userPassword,
     String databaseName,
     Properties info)
        throws SQLException
    {
        /* Specific Authentication scheme logic.
         If user has been authenticated, then simply return.
         If user name and/or password are invalid,
         then raise the appropriate exception.

        This example allows only users defined in the
        users config properties object.

        Check if the passed-in user has been defined for the system.
        We expect to find and match the property corresponding to
        the credentials passed in. */
        if (userName == null)
            // We do not tolerate 'guest' user for now.
            return false;
        //
        // Check if user exists in our users config (file)
        // properties set.
        // If we did not find the user in the users config set, then
        // try to find if the user is defined as a System property.
        //
        String actualUserPassword;
        actualUserPassword = usersConfig.getProperty(userName);
        if (actualUserPassword == null)
            actualUserPassword = System.getProperty(userName);
        if (actualUserPassword == null)
             // no such passed-in user found
             return false;
             // check if the password matches
        if (!actualUserPassword.equals(userPassword))
        return false;
        // Now, check if the user is a valid user of the database
        if (databaseName != null)
        {
              /* if database users restriction lists present, then check
             if there is one for this database and if so,
             check if the user is a valid one of that database.
             For this example, the only user we authorize in database
             DarkSide is user 'DarthVader'. This is the only database
             users restriction list we have for this example.
             We authorize any valid (login) user to access the
             OTHER databases in the system.
             Note that database users ACLs could be set in the same
             properties file or a separate one and implemented as you
             wish. */
//
            if (databaseName.equals("DarkSide")) {
               // check if user is a valid one.
               if (!userName.equals("DarthVader"))
                   // This user is not a valid one of the passed-in
                   return false;
            }
        }
        // The user is a valid one in this database
        return true;
```

```
    }
}
```

Built-in Derby users

Derby provides a simple, built-in repository of user names and passwords.

To use the built-in repository, set *derby.authentication.provider* to *BUILTIN*. Using built-in users is an alternative to using an external directory service such as LDAP.

```
derby.authentication.provider=BUILTIN
```

You can create user names and passwords for Derby users by specifying them with the *derby.user.UserName* property.

Note: These user names are case-sensitive for user authorization. User names are *SQL92Identifiers*. Delimited identifiers are allowed:

```
derby.user."FRed"=java
```

For more information on user names and *SQL92Identifiers*, see Users and authorization identifiers .

Note: For passwords, it is a good idea not to use words that would be easily guessed, such as a login name or simple words or numbers. A password should be a mix of numbers and upper- and lowercase letters.

Database-level properties

When you create users with database-level properties, those users are available to the specified database only.

You set the property once for each user. To delete a user, set that user's password to null.

```
-- adding the user sa with password 'derbypass'
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.sa', 'derbypass')
-- adding the user mary with password 'little7xylamb'
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.mary', 'little7xylamb')
-- removing mary by setting password to null
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.mary', null)
```

System-level properties

When you create users with system-level properties, those users are available to all databases in the system.

You set the value of this system-wide property once for each user, so you can set it several times. To delete a user, remove that user from the file.

You can define this property in the usual ways- typically in the *derby.properties* file. For more information about setting properties, see *Tuning Derby* .

Here is a sample excerpt from the *derby.properties* file:

```
# Users definition
#
derby.user.sa=derbypass
derby.user.mary=little7xylamb
```

List of user-authentication properties

User Authentication Properties summarizes the various properties related to user authentication.

Table1. User Authentication Properties

Property Name	Use
derby.connection.requireAuthentication	Turns on user authentication.
derby.authentication.provider	Specifies the kind of user authentication to use.
derby.authentication.server	For LDAP user authentication, specifies the location of the server.
derby.authentication.ldap.searchAuthDN, *derby.authentication.ldap.searchAuthPW*, *Derby.authentication.ldap.searchFilter*, and `Derby.authentication. ldap.searchBase`	Configures the way DN searches are performed.
derby.user.UserName	Creates a user name and password for Derby's built-in user repository.
*javax.naming.**	JNDI properties

Programming applications for Derby user authentication

This section discusses programming user authentication into applications for use with Derby.

Programming the application to provide the user and password

In the *DriverManager.getConnection* call, an application can provide the user name and password:

- Separately as arguments to the following signature of the method: *getConnection(String url, String user, String password)*

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", "mary", "little7xylamb");
```

- As attributes to the database connection URL

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB;user=mary;password=little7xylamb");
```

- By setting the user and password properties in a Properties object as with other connection URL attributes

```
Properties p = new Properties();
p.put("user", "mary");
p.put("password", "little7xylamb");
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", p);
```

Note: The password is not encrypted. When you are using Derby in the context of a server framework, the framework should be responsible for encrypting the password across the network. If your framework does not encrypt the password, consider using SSL.

For information about the treatment of user names within the Derby system, see Users and authorization identifiers .

Login failure exceptions with user authentication

If user authentication is turned on and a valid user name and password are not provided, *SQLException* 08004 is raised.

```
ERROR 08004: Connection refused : Invalid authentication.
```

Users and authorization identifiers

User names within the Derby system are known as *authorization identifiers*. The authorization identifier is a string that represents the name of the user, if one was provided in the connection request. For example, the built-in function CURRENT_USER returns the authorization identifier for the current user.

Once the authorization identifier is passed to the Derby system, it becomes an *SQL92Identifier*. `SQL92Identifiers` -the kind of identifiers that represent database objects such as tables and columns-are case-insensitive (they are converted to all caps) unless delimited with double quotes, are limited to 128 characters, and have other limitations.

User names must be valid authorization identifiers even if user authentication is turned off, and even if all users are allowed access to all databases.

For more information about *SQL92Identifiers*, see the *Derby Reference Manual* .

Authorization identifiers, user authentication, and user authorization

When working with both user authentication and user authorization, you need to understand how user names are treated by each system. If an external authentication system is used, the conversion of the user's name to an authorization identifier does not happen until *after* authentication has occurred but *before* user authorization (see User authorization). Imagine, for example, a user named Fred.

- Within the user authentication system, Fred is known as *FRed*. Your external user authorization service is case-sensitive, so Fred must always type his name that way.

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

- Within the Derby user authorization system, Fred becomes a case-insensitive authorization identifier. Fred is known as *FRED*.
- When specifying which users are authorized to access the accounting database, you must list Fred's authorization identifier, *FRED* (which you can type as *FRED*, *FREd*, or *fred*, since the system automatically converts it to all-uppercase).

  ```
  derby.fullAccessUsers=sa,FRED,mary
  ```

Let's take a second example, where Fred has a slightly different name within the user authentication system.

- Within the user authentication system, Fred is known as *Fred!*. You must now put double quotes around the name, because it is not a valid `SQL92Identifier` . (Derby knows to remove the double quotes when passing the name to the external authentication system.)

  ```
  Connection conn = DriverManager.getConnection(
      "jdbc:derby:myDB", "\"Fred!\"", "flintstone");
  ```

- Within the Derby user authorization system, *Fred* becomes a case-sensitive authorization identifier. Fred is known as *Fred!*.

- When specifying which users are authorized to access the accounting database, you must list Fred's authorization identifier, *"Fred!"* (which you must always delimit with double quotation marks).

```
derby.fullAccessUsers=sa,"Fred!",manager
```

As shown in the first example, your external authentication system may be case-sensitive, whereas the authorization identifier within Derby may not be. If your authentication system allows two distinct users whose names differ by case, delimit all user names within the connection request to make all user names case-sensitive within the Derby system. In addition, you must also delimit user names that do not conform to *SQL92Identifier* rules with double quotes.

User names and schemas

User names can affect a user's default schema. For information about user names and schemas, see "SET SCHEMA statement" in the *Derby Reference Manual* .

Exceptions when using authorization identifiers

Specifying an invalid authorization identifier in a database user authorization property raises an exception. Specifying an invalid authorization identifier in a connection request raises an exception.

User authorization

Derby provides user authorization, which is a means of granting users permission to access a database (or system). Derby allows you to distinguish between full (read-write) access and read-only access. (Derby Version 10.1 does not support the SQL GRANT and REVOKE features, which allow you to set permissions to specific database objects or specific SQL actions.)

Setting user authorization

Typically, you configure user authorization for a particular database. However, Derby also allows you to configure user authorization for the system, which is useful during development and for specifying a secure default access for all databases in the system.

To control access to a particular database, set database-level properties that specify which users have full (read-write) access to the database and which users have read-only access to the database. Users not specified by either property inherit the default access for the database (none, read-only, or full read-write access). When not explicitly set, the default access for a database is full (read-write) access.

Setting the default access mode

To specify the default access mode for the database, use the *derby.database.defaultConnectionMode* property. You can set the property to the following values:

- *noAccess*
- *readOnlyAccess*
- *fullAccess* (the default)

Derby validates the authorization configuration properties when users set them. It raises an exception if a user attempts to set the properties to invalid values (see User authorization exceptions).

Setting the access mode for particular users

To specify which particular users have full (read-write) access to a database, use the *derby.database.fullAccessUsers* property. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'sa,mary')
```

To specify which particular users have read-only access to a database, use the *derby.database.readOnlyAccessUsers* property. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers', 'guest,"Fred!"')
```

For these properties, you specify users as a comma-separated list (no spaces between the comma and the next user).

For users not specified with either property the access is specified by the *derby.database.defaultConnectionMode* property.

Note: It is possible to configure a database so that it cannot be changed (or even accessed) using the *derby.database.defaultConnectionMode* property. If you set this property to `noAccess` or `readOnlyAccess` , be sure to allow at least one user full access.

See the Javadoc for the utility or Examples of user authorization for more details.

Notes on user authorization

All the authorization properties are set for a connection when it is created. Changing any of the authorization properties does not affect existing connections. However, all future connections are affected by the change.

For more information about authorization identifiers, see Users and authorization identifiers .

User authorization exceptions

If a user is not authorized to connect to the database specified in the connection request, *SQLException* 04501 is raised.

If a user with *readOnlyAccess* attempts to write to a database, *SQLException* 08004 - `connection refused` is raised.

Read-only and full access permissions

Permissions for Read-Only and Full-Access Users shows which actions read-only and full-access users are permitted to perform on regular or source databases and on target databases.

Table1. Permissions for Read-Only and Full-Access Users

Action	Read-Only Users	Full-Access Users
Executing SELECT statements	X	X
Reading database properties	X	X
Loading database classes from jar files	X	X
Executing INSERT, UPDATE, or DELETE statements	'	X
Executing DDL statements	'	X

Action	Read-Only Users	Full-Access Users
Adding or replacing jar files	'	X
Setting database properties	'	X

Examples of user authorization

This example shows the property settings to configure a database to support:

- Full access for a single user named "sa"
- Read-only access for anyone else who connects to the database

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    'readOnlyAccess')
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'sa')
```

The following example shows the settings to configure a database to support:

- Full access for a single user named "Fred!" (case-sensitive) with full (read-write) access
- Read-only access for mary and guest
- No access for other users

The example also demonstrates the use of delimited identifiers for user names.

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    'noAccess')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', '"Fred!"')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers', 'mary,guest')
```

Encrypting databases on disk

Derby provides a way for you to encrypt your data on disk.

Typically, database systems encrypt and decrypt data in transport over the network, using industry-standard systems. This system works well for client/server databases; the server is assumed to be in a trusted, safe environment, managed by a system administrator. In addition, the recipient of the data is trusted and should be capable of protecting the data. The only risk comes when transporting data over the wire, and data encryption happens during network transport only.

However, Derby databases are platform-independent files that are designed to be easily shared in a number of ways, including transport over the Internet. Recipients of the data might not know how, or might not have the means, to properly protect the data.

This data encryption feature provides the ability to store user data in an encrypted form. The user who boots the database must provide a boot password.

Note: Jar files stored in the database are not encrypted.

Requirements for Derby encryption

Derby supports disk encryption, but you must supply the following:

- An implementation of the Java Cryptographic Extension (JCE) package version 1.2.1 or higher.

  Derby does not support earlier, non-exportable, versions of JCE (such as JCE 1.2). More information on JCE 1.2.1, including a product download, can be found at: *http://java.sun.com/products/jce/index.html* .

  Any attempt to create or access an encrypted database without the libraries for an implementation of JCE of the proper version, or without Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher, raises an exception; you will not be able to create or boot the database.

  Note: The JCE installation documentation describes configuring (registering) the JCE software. You do not need to do this; Derby registers JCE dynamically.
- The encryption provider

  An encryption provider implements the Java cryptography concepts. The JRE for J2SE 1.4 or J2EE 1.4 includes JCE and one or more default encryption providers.

Working with encryption

This section describes using encryption in Derby.

Encrypting databases on creation

Derby allows you to configure a database for encryption when you create it. To do so, you specify *dataEncryption=true* on the connection URL.

The Java Runtime Environment (JRE) determines the default encryption provider, as follows:

- For J2SE/J2EE 1.4 or higher, the JRE's provider is the default.
- For an IBM Corp J2SE/J2EE 1.3 JRE, the default provider is *com.ibm.crypto.provider*.
- For a Sun Microsystem J2SE/J2EE 1.3 JRE, the default provider is *com.sun.crypto.provider.SunJCE*.
- For any other J2SE/J2EE 1.3 JRE, a provider must be specified.

You have the option of specifying an alternate encryption provider; see Specifying an alternate encryption provider . The default encryption algorithm is DES, but you have the option of specifying an alternate algorithm; see Specifying an alternate encryption algorithm .

Creating the boot password

When you encrypt a database you must also specify a boot password, which is an alpha-numeric string used to generate the encryption key. The length of the encryption key depends on the algorithm used:

- AES (128, 192, and 256 bits)
- DES (the default) (56 bits)
- DESede (168 bits)
- All other algorithms (128 bits)

Note: The boot password should have at least as many characters as number of bytes in the encryption key (56 bits=8 bytes, 168 bits=24 bytes, 128 bits=16 bytes). The minimum number of characters for the boot password allowed by Derby is eight.

It is a good idea not to use words that would be easily guessed, such as a login name or simple words or numbers. A *bootPassword*, like any password, should be a mix of numbers and upper- and lowercase letters.

You turn on and configure encryption and specify the corresponding boot password on

the connection URL for a database when you create it:

```
jdbc:derby:encryptionDB1;create=true;dataEncryption=true;
    bootPassword=clo760uds2caPe
```

Note: If you lose the *bootPassword* and the database is not currently booted, you will not be able to connect to the database anymore. (If you know the current *bootPassword*, you can change it. See Changing the boot password .)

Specifying an alternate encryption provider:

You can specify an alternate provider when you create the database with the *encryptionProvider=providerName* attribute.

You must specify the full package and class name of the provider, and you must also add the libraries to the application's classpath.

```
-- using the the provider library jce_jdk13-10b4.zip|
-- available from www.bouncycastle.org
jdbc:derby:encryptedDB3;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe;
encryptionProvider=org.bouncycastle.jce.provider.BouncyCastleProvider;
encryptionAlgorithm=DES/CBC/NoPadding

-- using a provider
-- available from
-- http://jcewww.iaik.tu-graz.ac.at/download.html
jdbc:derby:encryptedDB3;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe;
encryptionProvider=iaik.security.provider.IAIK;encryptionAlgorithm=
DES/CBC/NoPadding
```

Specifying an alternate encryption algorithm:

Derby supports the following encryption algorithms:

- DES (the default)
- DESede (also known as triple DES)
- Any encryption algorithm that fulfills the following requirements:
  - It is symmetric
  - It is a block cipher, with a block size of 8 bytes
  - It uses the *NoPadding* padding scheme
  - Its secret key can be represented as an arbitrary byte array
  - It requires exactly one initialization parameter, an initialization vector of type *javax.crypto.spec.IvParameterSpec*
  - It can use *javax.crypto.spec.SecretKeySpec* to represent its key

  For example, the algorithm *Blowfish* implemented in the Sun JCE package fulfills these requirements.

By Java convention, an encryption algorithm is specified like this:

```
algorithmName/feedbackMode/padding
```

The only feedback modes allowed are:

- CBC
- CFB
- ECB
- OFB

By default, Derby uses the DES algorithm of *DES/CBC/NoPadding*.

Specify an alternate encryption algorithm when you create a database with the

encryptionAlgorithm=algorithm attribute. If the algorithm you specify is not supported by the provider you have specified, Derby throws an exception.

Booting an encrypted database

Once you have created an encrypted database, you must supply the boot password to reboot it. Encrypted databases cannot be booted automatically along with all other system databases on system startup (see *"derby.system.bootAll"* in *Tuning Derby*). Instead, you boot encrypted databases when you first connect to them.

For example, to access an encrypted database called *wombat,* created with the boot password clo760uds2caPe*,* you would use the following connection URL:

```
jdbc:derby:wombat;bootPassword=clo760uds2caPe
```

Once the database is booted, all connections can access the database without the boot password. Only a connection that boots the database requires the key.

For example, the following connections would boot the database and thus require the boot password:

- The first connection to the database in the JVM session
- The first connection to the database after the database has been explicitly shut down
- The first connection to the database after the system has been shut down and then rebooted

Note: The boot password is not meant to prevent unauthorized connections to the database once it has been booted. To protect a database once it has been booted, turn on user authentication (see Working with user authentication).

Changing the boot password

You can change the boot password for the current database.

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'bootPassword', 'oldbpw , newbpw');
```

where *oldbpw* is the current boot password and *newbpw* is the new boot password. This call commits immediately; it is not transactional.

Note: `PropertyInfo.getDatabaseProperty("bootPassword")` , or VALUES `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('bootPassword')`, will not return the boot password.

Signed jar files

In a Java 2 environment, Derby can detect digital signatures on jar files. When attempting to load a class from a signed jar file stored in the database, Derby will verify the validity of the signature.

Note: The Derby class loader only validates the integrity of the signed jar file and that the certificate has not expired. Derby cannot ascertain whether the validity/identity of declared signer is correct. To validate identity, use a Security Manager (i.e., an implementation of *java.lang.SecurityManager*).

When loading classes from an application jar file in a Java 2 environment, Derby behaves as follows:

- *If the class is signed, Derby will:*
  - Verify that the jar was signed using a X.509 certificate (i.e., can be represented by the class *java.security.cert.X509Certificate*). If not, throw an exception.
  - Verify that the digital signature matches the contents of the file. If not, throw an

exception.
- Check that the set of signing certificates are all valid for the current date and time. If any certificate has expired or is not yet valid, throw an exception.
- Pass the array of certificates to the *setSigners()* method of `java.lang.ClassLoader` . This allows security managers to obtain the list of signers for a class (using `java.lang.Class.getSigners`) and then validate the identity of the signers using the services of a Public Key Infrastructure (PKI).

Note: Derby does not provide a security manager.

For more information about signed jar files, see the Java 2 specifications at *http://java.sun.com* .

For more information about Java 2 security, go to *http://java.sun.com/security/* .

Notes on the Derby security features

Because Derby does not support traditional grant and revoke features, the security model has some basic limitations. For both embedded and client/server systems, it assumes that users are trusted. You must trust your full-access users not to perform undesirable actions. You lock out non full-access users with database properties, which are stored in the database (and in an encrypted database these properties are also encrypted). Note, however, for a distributed/embedded system that a sophisticated user with the database encryption key might be able to physically change those properties in the database files.

In addition, in the Derby system, it is not necessary to have a specific connection (or permission to access a particular database) to shut down the system. Any authenticated user can shut down the system.

Other security holes to think about are:

- JVM subversion, running the application under a home-grown JVM.
- Trolling for objects
- Class substitution, locating a class that has access to sensitive data and replacing it with one that passes on information

User authentication and authorization examples

This section provides examples on using user authentication and authorization in Derby in either a client/server environment or in an embedded environment.

User authentication example in a client/server environment

In this example, Derby is running in a user-designed application server. Derby provides the user authentication, not the application server. The server is running in a secure environment, the application server encrypts the passwords, and a database administrator is available. The administrator configures security using system-level properties in the *derby.properties* file and has protected this file with operating system tools. Derby connects to an existing LDAP directory service within the enterprise to authenticate users.

The default access mode for all databases is set to *fullAccess* (the default).

The *derby.properties* file for the server includes the following entries:

```
# turn on user authentication
```

```
derby.connection.requireAuthentication=true
# set the authentication provider to an external LDAP server
derby.authentication.provider=LDAP
# the host name and port number of the LDAP server
derby.authentication.server=godfrey:389
# the search base for user names
derby.authentication.ldap.searchBase=o=oakland.mycompany.com
# explicitly show the access mode for databases (this is default)
derby.database.defaultAccessMode=fullAccess
```

With these settings, all users must be authenticated by the LDAP server in order to access any Derby databases.

The database administrator has determined that one database, *accountingDB*, has additional security needs. Within a connection to that database, the database administrator uses database-wide properties (which override properties set in the *derby.properties* file) to limit access to this database. Only the users *prez*, *cfo*, and *numberCruncher* have full (read-write) access to this database, and only *clerk1* and *clerk2* have read-only access to this database. No other users can access the database.

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultAccessMode', 'noAccess')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers',
    'prez,cfo,numberCruncher')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readAccessUsers', 'clerk1,clerk2')
```

The database administrator then requires all current users to disconnect and re-connect. These property changes do not go into effect for current connections. The database administrator can force current users to reconnect by shutting down the database

User authentication example in a single-user, embedded environment

In this example, Derby is embedded in a single-user application that is deployed in a number of different and potentially insecure ways. For that reason, the application developer has decided to encrypt the database and to turn on user authentication using Derby's built-in user authentication, which will not require connections to an LDAP server. The end-user must know the *bootPassword* to boot the database and the user name and password to connect to the database. Even if the database ended up in an e-mail, only the intended recipient would be able to access data in the database. The application developer has decided not to use any user authorization features, since each database will accept only a single user. In that situation, the default full-access connection mode is acceptable.

When creating the database, the application developer encrypts the database by using the following connection URL:

```
jdbc:derby:wombat;create=true;dataEncryption=true;
    bootPassword=sxy90W348HHn
```

Before deploying the database, the application developer turns on user authentication, sets the authentication provider to BUILTIN, creates a single user and password, and disallows system-wide properties to protect the database-wide security property settings:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.connection.requireAuthentication', 'true')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.provider', 'BUILTIN')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.enduser', 'red29PlaNe')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.propertiesOnly', true')
```

When the user connects (and boots) the database, the user has to provide the *bootPassword*, the user name, and the password. The following example shows how to provide those in a connection URL, although the application programmer would probably provide GUI windows to allow the end user to type those in:

```
jdbc:derby:wombat;bootPassword=sxy90W348HHn;
    user=enduser;password=red29PlaNe
```

User authentication and authorization extended examples

The following two examples from the *sample* database show how to turn on and turn off user authentication using Derby's built-in user authentication and user authorization.

```
/**
         * Turn on built-in user authentication and user authorization.
         *
         * @param conn a connection to the database.
         */

        public static void turnOnBuiltInUsers(Connection conn) throws
SQLException {
                System.out.println("Turning on authentication.");
                Statement s = conn.createStatement();


                // Setting and Confirming requireAuthentication
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.connection.requireAuthentication',
'true')");
                ResultSet rs = s.executeQuery(
                        "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY("
+
                        "'derby.connection.requireAuthentication')");
                rs.next();
                System.out.println(rs.getString(1));
                // Setting authentication scheme to Derby
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.authentication.provider', 'BUILTIN')");

                // Creating some sample users
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.user.sa', 'ajaxj3x9')");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.user.guest', 'java5w6x')");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.user.mary', 'little7xylamb')");

                // Setting default connection mode to no access
                // (user authorization)
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.defaultConnectionMode',
'noAccess')");
                // Confirming default connection mode
                rs = s.executeQuery (
                        "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY("
+
                        "'derby.database.defaultConnectionMode')");
```

```
                rs.next();
                System.out.println(rs.getString(1));

                // Defining read-write users
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.fullAccessUsers', 'sa,mary')");

                // Defining read-only users
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.readOnlyAccessUsers',
'guest')");

                // Confirming full-access users
                rs = s.executeQuery(
                        "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY("
+
                        "'derby.database.fullAccessUsers')");
                rs.next();
                System.out.println(rs.getString(1));

                // Confirming read-only users
                rs = s.executeQuery(
                        "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY("
+
                        "'derby.database.readOnlyAccessUsers')");
                rs.next();
                System.out.println(rs.getString(1));

                //we would set the following property to TRUE only
                //when we were ready to deploy.
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.propertiesOnly', 'false')");
                s.close();
        }


/**
         * Turn off built-in user authentication and user authorization.
         *
         * @param conn a connection to the database.
         */

        public static void turnOffBuiltInUsers(Connection conn) throws
SQLException {
                Statement s = conn.createStatement();
                System.out.println("Turning off authentication.");

                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.connection.requireAuthentication',
'false')");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.authentication.provider', null)");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.user.sa', null)");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.user.guest', null)");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.user.mary', null)");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.defaultConnectionMode',
'fullAccess')");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.fullAccessUsers', null)");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.readOnlyAccessUsers', null)");
                s.executeUpdate("CALL
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
                        "'derby.database.propertiesOnly', 'false')");

                // Confirming requireAuthentication
                ResultSet rs = s.executeQuery(
                        "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY("
+
                        "'derby.connection.requireAuthentication')");
                rs.next();
```

```
                System.out.println(rs.getString(1));

                // Confirming default connection mode
                rs = s.executeQuery(
                        "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY("
+
                        "'derby.database.defaultConnectionMode')");
                rs.next();
                System.out.println(rs.getString(1));
                System.out.println("Turned off all the user-related
properties.");
                s.close();
        }
}
```

Running Derby under a security manager

When running within an application or application server with a Java 2 Security Manager enabled, Derby must be granted certain permissions to execute and access database files.

For more information about permissions and examples of creating permission objects and granting permissions, see the Security Architecture specification at *http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html* .

Granting permissions to Derby

This section discusses which permissions should be granted to Derby (the code base derby.jar).

See Default Policy Implementation and Policy File Syntax at *http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html* for more information about creating policy files.

Mandatory permissions

permission java.lang.RuntimePermission createClassLoader
Mandatory. It allows Derby to execute SQL queries and supports loading class files from jar files stored in the database.

permission java.util.PropertyPermission "derby.*", read
Allows Derby to read individual Derby properties set in the JVM's system set. If the action is denied, properties in the JVM's system set are ignored.

Database access permissions

permission java.io.FilePermission "directory${/}/-", "read,write,delete"
Allows Derby to manage files within the database that maps to the directory specified. For read-only databases, only the "read" action needs to be granted.

Optional permissions

permission java.io.FilePermission "${derby.system.home}", "read,write"
Allows Derby to determine the system directory when set by d*b2j.system.home* and create it if needed. If the system directory already exists then only the "read" permission needs to be granted.

permission java.util.PropertyPermission "user.dir", "read"
Permits access to the system directory value if *derby.system.home* is not set or no permission has been granted to read the `derby.system.home` property.

permission java.io.FilePermission "${derby.system.home}${/}derby.properties", "read"
Allows Derby to read the system properties file from the system directory.

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

permission java.io.FilePermission "${user.dir}${/}derby.log", "read,write,delete"

Only one of these permissions is needed. Permits the application to read, write, and delete to the Derby log file, unless the log has been re-directed. (See the *derby.stream.error* properties in *Tuning Derby* for more information.) If one of the requested valid actions is denied, the Derby log will be `java.lang.System.err` .

Combining permissions

You might grant one *FilePermission* that encompasses several or all of the permissions instead of separately granting a number of the more specific permissions. For example:

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

This allows the Derby engine complete access to the system directory and any databases contained in the system directory.

Examples of Java 2 security policy files for embedded Derby

Java 2 security policy file example 1

```
/* Grants permission to run Derby and access all      */
/* databases under the Derby system home              */
/* when it is specified by the system property             */
/* Derby.system.home                                    */

/* Note Derby.system.home must be an absolute pathname */

grant codeBase "file://f:/derby/lib/derby.jar" {

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

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

Java 2 security policy file example 2

```
/* Grants permission to run Derby and access all      */
/* databases under the Derby system home              */
/* when it defaults to the current directory               */


grant codeBase "file://f:/derby/lib/derby.jar" {
  permission java.lang.RuntimePermission "createClassLoader";
  permission java.util.PropertyPermission "derby.*", "read";

  permission java.util.PropertyPermission "user.dir", "read";
  permission java.io.FilePermission "${user.dir}${/}-",
 "read,write,delete";

};
```

Java 2 security policy file example 3

```
/* Grants permission to run Derby and access a single */
/* database (salesdb) under the Derby system home     */

/* Note Derby.system.home must be an absolute pathname */

grant codeBase "file://f:/derby/lib/derby.jar" {

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

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


};
```

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 *Tuning Derby* .

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. For information about setting attributes in calls to `java.sql.DriverManager.getConnection` , see Database connection examples .

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

Here is some example code:

```
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, e.g.,
        // 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, as described in One Derby instance for each Java Virtual Machine (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*; see Embedded Derby JDBC driver) and connection URL (`jdbc:derby:databaseName` ; see Embedded Derby JDBC database connection URL). 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

Multiple rows

Derby supports the complete SQL-92 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:

```
-- send 5 rows at a time:
PREPARE p1 AS 'INSERT INTO ThreeColumnTable VALUES
(?,?,?), (?,?,?), (?,?,?), (?,?,?), (?,?,?)
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 Ohio
SELECT 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

Developers using Derby in existing applications might need to create "empty" queries with the right result shape for filling in bits of functionality 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 other components such as the tools and the installer.

It also provides a feature to support databases in many different languages, a feature which is independent of a particular territory.

When you create or upgrade a database, you can use the territory attribute to associate a non-default territory with the database. For information about how to use the territory 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.

Other components

Derby also provides locale support for the following:

- 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. See Messages libraries .

The locale of the error messages and of the tools is not determined by the database's localle set by the *locale=ll_CC* attribute when the database is created but instead 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 would not be returned in the language of the database's locale but 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

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_de_DE.jar German
- derbyLocale_es.jar - Spanish
- derbyLocale_fr.jar - French
- derbyLocale_it.jar - Italian
- derbyLocale_ja_JP.jar - Japanese
- derbyLocale_ko_KR.jar - Korean
- derbyLocale_pt_BR.jar - Brazilian Portuguese
- derbyLocale_zh_CN.jar - Simplified Chinese
- derbyLocale_zh_TW.jar - Traditional Chinese

Derby and standards

Derby adheres to SQL99 standards wherever possible. Below you will find a guide to 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:

Dynamic SQL

Derby uses JDBC's Prepared Statement, and does not provide SQL commands for dynamic SQL.

Cursors

Derby uses JDBC's Result Sets, and does not provide SQL for manipulating cursors except for positioned update and delete. Derby's scrolling insensitive cursors are provided through JDBC, not through SQL commands.

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.

Transactions

All operations in Derby are transactional. Derby supports transaction control using JDBC 3.0 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.

Stored routines and PSM

Derby supports external procedures using the Java programming language. Procedures are managed using the CREATE PROCEDURE and DROP PROCEDURE statements.

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.

Unique constraints and nulls

The SQL standard defines that unique constraints on nullable columns allow any number of nulls; Derby does not permit unique constraints on nullable columns.

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.

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.

CLOB, and BLOB

Derby supports the standard CLOB and BLOB data types. BLOB and CLOB values are limited to a maximum of 2,147,483,647 characters.

Expressions on LONGs

Derby permits expressions on LONG VARCHAR; however LONG VARCHAR data types are not allowed in:

- 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.

ALTER TABLE

Slightly different ALTER TABLE syntax for altering column defaults. SQL99 uses DROP and SET, we use DEFAULT.

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.