Apache Derby: Introduction

Deep Dive with Apache Derby: Perl, PHP, and Python

OSCON

August 2, 2005

Dan Scott

dan.scott@acm.org / dan.scott@ca.ibm.com

Objectives

In this section of the tutorial, you will learn how to:

1. Set up and administer Apache Derby as a database server.
2. Connect to the Apache Derby Network Server.
3. Create a database that introduces a few of Apache Derby's features.
4. Enable some security features for our network environment.

Quick facts

A relational database implemented in Java with a 6 megabyte footprint, supporting many advanced features:

- stored procedures, triggers, user-defined functions
- transactions, views
- relational constraints, row-level locking
- database encryption
- database, LDAP, or user-defined authentication

IBM contributed the IBM Cloudscape source code to Apache under the Apache Software License 2.0 in September 2004, resulting in the first release 10.0.0.1. The Apache Derby project is currently preparing for the 10.1 release. The project is hosted at http://incubator.apache.org/derby/.

Setting up and administering Apache Derby as a database server

1. Starting Apache Derby as a network server

Apache Derby depends upon a Java Runtime Environment (JRE) and its behavior depends on having your environment variables set up correctly.

1. Open a new terminal window and extract the Apache Derby tarball:

```
bash$ tar xzf incubating-derby-snapshot-10.1.tar.gz
```

2. Extract the IBM JRE tarball:

```
bash$ tar xzf IBM-JRE-142.tar.gz
```

3. Change directories to derby/frameworks/NetworkServer/bin/:

```
bash$ cd derby/frameworks/NetworkServer/bin
```

4. Set up the network server environment variables:

```
bash$ . setNetworkServerCP.ksh
```

5. Start the network server:

```
bash$ ./startNetworkServerCP.ksh
```

Leave this terminal window open so that we can watch the connection requests as we develop our applications.

2. Connecting to and creating an Apache Derby database

This task currently requires you to connect to the database server using a JDBC connection, as the command to create a new database is issued by a JDBC connection property (interestingly enough). We will use the **ij** "interactive JDBC" tool to make the connection and issue the create database request. **ij** is a useful but rather rough command line environment for administering Apache Derby. After creating the database, the rest of our interactions with Apache Derby will be through Perl, PHP, and Python applications.

1. Open a new terminal window and change directories to derby/frameworks/NetworkServer/bin/:

   ```
   bash$ cd derby/frameworks/NetworkServer/bin
   ```

2. Set up the network client environment variables:

   ```
   bash$ . setNetworkClientCP.ksh
   ```

3. Start the **ij** interactive JDBC tool:

   ```
   bash$ ./ij.ksh
   ```

4. Issue your connection request using the **create=true;** connection attribute to force Apache Derby to create the database **MYDB** if it does not already exist:

   ```
   ij> connect 'jdbc:derby://localhost:1527/MYDB;create=true';
   ij> disconnect;
   ij> exit;
   ```

5. List the contents of the directory to ensure that a new subdirectory named **MYDB** has been created.

If you want to create a copy of the empty database, you can simply shut down the Apache Derby Network Server and copy the entire **MYDB** directory.

3. Creating the database tables

Our restaurant menu system will initially rely on a database containing three tables:

- FOOD -- stores the names and descriptions of individual food items
- PRICES -- stores the prices of food items
- PICTURES -- stores pictures of food items

Here are the SQL statements for creating the required tables:

```
CREATE TABLE menu.food (id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
name VARCHAR(128), description VARCHAR(1024));

CREATE TABLE menu.prices (id INTEGER, price DECIMAL(5,2), CONSTRAINT
prices_fk  FOREIGN KEY (id) REFERENCES menu.food(id));

CREATE TABLE menu.pictures (id INTEGER, picture BLOB(256K), CONSTRAINT
pictures_fk FOREIGN KEY (id) REFERENCES menu.food(id));
```

Create the tables using the **ij** utility. You can enter the statements individually, or you can create a text file that contains the SQL statements and use the **RUN** command to automatically execute all of the statements in the indicated text file.

Notice the features demonstrated by these three simple tables:

- schemas: the 'menu' part of **menu.**food is a schema qualifier. By default, new SQL objects are created in the schema of the user ID you used to connect to Apache Derby. For applications that support connections from multiple users, using an explicit schema for your SQL objects is strongly recommended.
- generated columns: the GENERATED ALWAYS AS IDENTITY clause automatically inserts the next integer value for the indicated column when you insert a new row into the table.
- relational constraints: the primary and foreign keys enforce referential integrity between different tables. If the row containing a primary key value is deleted, the default action is to delete the corresponding rows in any tables that defined a foreign key referencing the primary table.

4. Securing the Apache Derby Network Server

The Apache Derby Network Server is, in its default configuration, arguably rather secure: it only accepts connections from **localhost**. While that is useful for development purposes, most production database servers need to accept connections from other hosts on the network. Once your Apache Derby Network Server is ready to accept connections from other hosts, you need to secure your database by adding authenticated users and turning on authentication.

4a. Enabling connections from other hosts

By default, Apache Derby only allows connections from **localhost**. Connections from other hosts are rejected. The list of hosts is set when you start the Apache Derby Network Server, so to change the setting you must restart the Network Server.

The startNetworkServerCP.ksh script offers a number of ways to specify the list of hosts from which connections will be accepted; the easiest way is to set the DERBY_SERVER_HOST environment variable to a comma-delimited set of hostnames or IP addresses from which you want to accept connections. To accept connections from any host on the network, pass 0.0.0.0.

To set this property permanently, create an INI-style file called service.properties in the NetworkServer directory and set the **derby.drda.host** property.

Example:

```
--  accept connections from every host on the network
bash$ export DERBY_SERVER_HOST=localhost,incubator.apache.org,9.26.169.128
bash$ ./stopNetworkServer.ksh
bash$ ./startNetworkServer.ksh
```

4b. Adding built-in database users

By default, user authentication is not enabled, so Apache Derby accepts network connections for any combination of username and password you provide. This is not secure for a production environment, of course, so we will use Apache Derby's support for built-in database users to add an authentication requirement to our Apache Derby Network Server. Network connections through the DB2 client are automatically encrypted using the IBM Java Cryptographic Extension (JCE) on the network server side.

Built-in users are stored as database properties, so to add one or more built-in users for a given database we can call the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY stored

procedure. Note that these users have full INSERT/UPDATE/DELETE privileges on the entire database; Apache Derby does not provide a lower level of granularity for granting and revoking privileges on tables or views.

Example:

```
--  add a built-in user named 'neil' with the password 'diamond'
ij> CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.user.neil', 'diamond');
```

4c. Enabling built-in authentication

Apache Derby supports three different kinds of authentication: built-in users, LDAP, and custom authentication. We are using built-in authentication as it is the easiest to demonstrate. Authentication is a database property, so to turn it on you set the **derby.authentication.provider** to 'BUILTIN' and the **derby.connection.requireAuthentication** property to 'true'. Any subsequent connections to the database will require a correct username and password to connect successfully.

Note: if you turn authentication on and shut down the Apache Derby Network Server before defining any users, you will not be able to connect to the database so that you can define users. It's best not to get yourself into this situation, but if you do, you can define a system-wide user in the derby.properties file and use that user to correct the problem.

Example:

```
-- set authentication type to built-in
ij> CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
('derby.authentication.provider', 'BUILTIN');
-- turn on authentication
ij> CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
('derby.connection.requireAuthentication', 'true');
```

4d. Debugging information

The Apache Derby network server logs interesting debug information in a file called `derby.log`. Display the contents of that file in a text editor, and you should see the startup information, connection attempts, and any Java exceptions that were thrown by Apache Derby.

Administration: SQL interface

Apache Derby is very programmer-friendly, in that the majority of administrative tasks are performed by calling stored procedures or invoking SQL functions. This design approach makes it easy to write scripts in your favourite language to administer Apache Derby. Following is a complete list of the administrative stored procedures and functions.

Backing up a database

Backup support is provided by three stored procedures.

SYSCS_UTIL.SYSCS_FREEZE_DATABASE():

> Freezes all database operations so a backup at the level of the operating system can occur.

SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE():

> Brings database back to full operation after a backup at the level of the operating system.

SYSCS_UTIL.SYSCS_BACKUP_DATABASE(IN backupDir VARCHAR(32762)):

> Backs up the database to the directory specified by **backupDir**. Reads against Apache Derby can continue, but writes are blocked until the backup finishes.

Restoring a database is a matter of either connecting to the backed-up database, or stopping the current network server and renaming the original and backup directories.

Tuning performance

Performance diagnostics are provided by one function, and two stored procedures.

SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS():

> Returns a VARCHAR(32768) representing the execution plan for a prepared statement.

SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(IN enable SMALLINT):

> A positive integer value for **enable** turns the capturing of runtime statistics on so that you can return the execution plan by invoking the SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS() function.

SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(IN enable SMALLINT):

> A positive integer value for **enable** adds precise timings for each step of the execution plan. Has no effect if runtime statistics are not being collected.

Example:

```
-- turn on statistics collection
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
-- turn on statistics timing
CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);
-- retrieve the
VALUES SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS();
```

Moving data

Importing and exporting data is handled by four stored procedures:

SYSCS_UTIL.SYSCS_EXPORT_TABLE(IN schemaName VARCHAR(128), IN tableName VARCHAR(128), IN filename VARCHAR(32672), IN columnDelimiter CHAR(1), IN characterDelimiter CHAR(1), IN codeset VARCHAR(128)):

Exports the specified table to the specified filename. If **columnDelimiter** is NULL, the default value is a comma (,). If **characterDelimiter** is NULL, the default value is a double quotation mark ("). If codeset is NULL, the **codeset** of the running JVM is used.

SYSCS_UTIL.SYSCS_EXPORT_QUERY(IN selectStatement VARCHAR(128), IN filename VARCHAR(32672), IN columnDelimiter CHAR(1), IN characterDelimiter CHAR(1), IN codeset VARCHAR(128)):

Exports the results of the specified query to the specified filename. If **columnDelimiter** is NULL, the default value is a comma (,). If **characterDelimiter** is NULL, the default value is a double quotation mark ("). If **codeset** is NULL, the codeset of the running JVM is used.

SYSCS_UTIL.SYSCS_IMPORT_TABLE(IN schemaName VARCHAR(128), IN tableName VARCHAR(128), IN filename VARCHAR(32672), IN columnDelimiter CHAR(1), IN characterDelimiter CHAR(1), IN codeset VARCHAR(128), IN replace SMALLINT):

Imports the data from the specified file into the specified table.If **columnDelimiter** is NULL, the default value is a comma (,). If **characterDelimiter** is NULL, the default value is a double quotation mark ("). If **codeset** is NULL, the codeset of the running JVM is used. If the value of **replace** is a non-zero integer, all of the data in the table will be deleted and replaced by the contents of the file.

SYSCS_UTIL.SYSCS_IMPORT_DATA(IN schemaName VARCHAR(128), IN tableName VARCHAR(128), IN insertColumns VARCHAR(32672), IN columnIndexes VARCHAR (32672), IN filename VARCHAR(32672), IN columnDelimiter CHAR(1), IN characterDelimiter CHAR(1), IN codeset VARCHAR(128), IN replace SMALLINT):

Imports the data from the specified file into a subset of columns in the specified table.You must specify either a comma-delimited set of column names with **insertColumns** to specify the columns by name, or a comma-delimited set of column numbers with **columnIndexes** to specify the columns by number. If **columnDelimiter** is NULL, the default value is a comma (,). If **characterDelimiter** is NULL, the default value is a double quotation mark ("). If **codeset** is NULL, the codeset of the running JVM is used. If the value of **replace** is a non-zero integer, all of the data in the table will be deleted and replaced by the contents of the file.

Setting and getting database properties

SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(IN propertyName VARCHAR(128)):

Returns the current value of the specified database property.

SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(IN key VARCHAR(128), IN value VARCHAR(32672)):

Sets the database property **key** to the specified **value**.

Example:

```
--  log all connections to the database server
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.drda.logConnections',
'true');
-- check the value of the property to ensure the change was made
VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('derby.drda.logConnections');
```

Database health

SYSCS_UTIL.SYSCS_CHECK_TABLE((IN schemaName VARCHAR(128), IN tableName VARCHAR(128))):

> Returns an integer value of 1 if the specified table is consistent.

SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE():

> Flushes all cached data to disk -- the equivalent of calling **sync** on Linux / UNIX.

SYSCS_UTIL.SYSCS_COMPRESS_TABLE(IN schemaName VARCHAR(128), IN tableName VARCHAR(128), IN sequential SMALLINT):

> Returns unused space allocated for the specified table to the operating systems -- usually only a problem if many rows have been deleted from that table. The **sequential** argument, if set to a positive integer value, causes Apache Derby to use a slower but less memory-intensive method of reclaiming the unused space.

SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE(IN schemaName VARCHAR(128), IN tableName VARCHAR(128), IN purgeRows SMALLINT, IN defragmentRows SMALLINT, IN truncateEnd SMALLINT):

> Compresses the data within the existing table and index files, rather than creating new files. Passing a positive integer value for **purgeRows** and **defragmentRows** forces a scan of every page in the table; **truncateEnd** is used in conjunction with **defragmentRows** to reclaim the defragmented space at the end of the table.

Reference information

- Online documentation: http://incubator.apache.org/derby/manuals/index.html
- Mailing lists: http://incubator.apache.org/derby/derby_mail.html
- JIRA issue tracker: http://incubator.apache.org/derby/DerbyBugGuidelines.html