Everybody's building Web applications these days. But what if you just want a little bit of data? Do you really need to go to the trouble of creating a whole application? Doesn't Derby provide a tool for that? Well, of course it does. Derby's main command line tool, ij, is designed for just that: manipulating and retrieving the data in the database from the command line. But ij is more than just a place to enter select statements. It provides a fairly comprehensive suite of tools and commands, such as the use and manipulation of cursors. This article explains the use of ij's advanced scripting capabilities. You may never go back to a Web application again.

What's the ij tool's purpose?

The Apache Derby database (and its commercial cousin, IBM® Cloudscape™) is perfectly positioned for embedding in an application. It's small, compliant with both Java™ Database Connectivity (JDBC) and Open Database Connectivity (ODBC), and built on the Java platform, so it's available on a multitude of operating systems. But that versatility masks one of its most powerful tools.

The ij tool provides an easy means for simple tasks, such as creating databases and executing SQL statements. For example, you can create two simple databases -- one to hold projects and one to hold developers (see Listing 1).

### Listing 1. Creating databases and tables

```
ij> connect 'jdbc:derby:projectDB;create=true;';
ij> create table projects (  
id integer,  
project_name varchar(75),  
status integer,  
comments varchar(255)
);
0 rows inserted/updated/deleted
ij> insert into projects values (1, 'Wheel O Fish', 0, 'Waiting for contract');
1 row inserted/updated/deleted
ij> insert into projects values (2, 'Bass O Matic', 1, 'In production');
1 row inserted/updated/deleted
ij> insert into projects values (3, 'Patty O Furniture', -1, 'Need clarification. Does anybody know what this is?');
1 row inserted/updated/deleted
ij> connect 'jdbc:derby:developerDB;create=true;';
ij(CONNECTION1)> create table developers (  
developer_id integer,  
developer_name varchar(75),  
development_status integer,  
development_date date)
0 rows inserted/updated/deleted
ij(CONNECTION1)> insert into developers values (1, 'Jason', 'Developer', '2006-07-18');
1 row inserted/updated/deleted
ij(CONNECTION1)> insert into developers values (2, 'Mia', 'Project Manager', '2006-07-19');
1 row inserted/updated/deleted
```
id integer,
    developer_name varchar(75),
    availability varchar(255)
);
0 rows inserted/updated/deleted

ij(CONNECTION1)> insert into developers values (1, 'Jim Bacon', 'Open');
1 row inserted/updated/deleted
ij(CONNECTION1)> insert into developers values (2, 'Brady James', 'Open');
1 row inserted/updated/deleted
ij(CONNECTION1)> insert into developers values (3, 'Michelle Rappaport', 'Busy until October');
1 row inserted/updated/deleted
ij(CONNECTION1)> insert into developers values (4, 'Aaron Templeton', 'A joy every time we use him. Hire him.');
1 row inserted/updated/deleted

ij(CONNECTION1)

(Yes, normally we would put these tables into the same database. Please indulge me for the sake of demonstrating concepts.)

If creating databases, tables, and data were all you could do with ij, it would still be useful. But that's not the case. ij enables you to do many of the same tasks you might think you can only accomplish using a full-fledged JDBC application, such as manipulating cursors and creating and executing prepared statements.

But first, notice how ij created a second connection when you created the second database. The first connection, to projectDB, is still active, but you need to specifically tell ij you want to use it.

Let's see how that works.

Connections and isolation levels

One of the things ij enables you to do is to control the overall database system and the connection to the database. For example, we have two database connections right now, and we can choose between them using the set connection command (see Listing 2).

Listing 2. Choosing between database connections

ij(CONNECTION1)> select * from projects;
ERROR 42X05: Table 'PROJECTS' does not exist.
ij(CONNECTION1)> show connections;
CONNECTION0 -   jdbc:derby:projectDB
CONNECTION1* -  jdbc:derby:developerDB
* = current connection

ij(CONNECTION1)> set connection connection0;

ij(CONNECTION0)> select * from projects;

+--------+-----------+-------+-------------------+
<table>
<thead>
<tr>
<th>ID</th>
<th>PROJECT_NAME</th>
<th>STATUS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wheel O Fish</td>
<td>0</td>
<td>Waiting for contract</td>
</tr>
<tr>
<td>2</td>
<td>Bass O Matic</td>
<td>1</td>
<td>In production</td>
</tr>
<tr>
<td>3</td>
<td>Patty O Furniture</td>
<td>1</td>
<td>Need clarification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Does anybody know what this is?</td>
</tr>
</tbody>
</table>
+--------+-----------+-------+-------------------+
3 rows selected
Notice that if you start by trying to select from the projects table, you get an error even though that connection is still open because you're currently using the connection to the developerDB database. You can see that by issuing the `show connections` command. To change connections, you use the `set connection` command followed by the name of the database as shown by `show connections`.

`ij` also enables you to set overall database properties. For example, you can set the `derby.database.propertiesOnly` property for the database, as shown below:

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

This command makes sure that any properties specifically set on this database, such as authentication requirements, aren't overruled by other values in the overall system configurations.

Derby recognizes several dozen system properties, such as `derby.database.defaultConnectionMode`, `derby.authentication.ldap.searchAuthDN`, `derby.authentication.ldap.searchAuthPW`, and `derby.user.UserName`.

Now let's look at some ways of dealing with data.

**Working with cursors**

**Cursors or RecordSets?**

Cursors and RecordSets provide many of the same functions when it comes to manipulating multiple rows of data. So which is which? In a JDBC application, you create a RecordSet, and in `ij` you create a cursor. But, in fact, `ij` uses a RecordSet to implement the cursor on the back end.

When you select a set of data from a table or tables in a database, the returned data is known as a cursor. (Popular legend holds that this is an abbreviation of "current set of rows".) The documentation on isolation levels mentions cursors all the time, but you don't often see talk about manipulating cursors directly through `ij`. The fact is that many of the operations that might seem to require manipulation of a Java RecordSet can be accomplished directly in `ij` using a cursor.

**Forward-only cursors**

The simplest type of cursor is the *firehose*, or forward-only cursor. This cursor includes all of the data and enables you to move through each of the records in the cursor, but doesn't enable you to move backwards. For example, see **Listing 3**.

**Listing 3. The forward-only cursor**

```sql
ij(CONNECTION0)> get cursor projectCursor as 'select * from projects';
ij(CONNECTION0)> next projectCursor;
ID |PROJECT_NAME       |STATUS |COMMENTS
----------------------------------------------
1  |Wheel O Fish       |0      |Waiting for contract
ij(CONNECTION0)> next projectCursor;
ID |PROJECT_NAME       |STATUS |COMMENTS
```
IJ ERROR: PREVIOUS is not allowed on a forward-only cursor. Here you use the get cursor command to create a cursor, feeding it an SQL statement that represents the data you want to include. Note that here I'm just pulling data from a single table, but you can get as complex as Derby will allow in these SQL statements.

After you have the cursor, you can tell it to give you the next record, in which case it moves down the list of records, in the order specified by the SQL statement. Once you get to the bottom, ij tells you there are no more records. Notice that while you can move down the list, you can't move back; a standard cursor is, as I said before, forward only. But that doesn't mean you're doomed to always move in a single direction.

**Scroll-insensitive cursors**

Sometimes a forward-only cursor isn't good enough because you want to move around among the records. To do that, you can create a scroll-insensitive cursor, as seen in **Listing 4**.

**Listing 4. The scroll-insensitive cursor**

```sql
ij(CONNECTION0)> get scroll insensitive cursor projectScroll as 'select * from projects';
ij(CONNECTION0)> absolute 2 projectScroll;
ID |PROJECT_NAME |STATUS |COMMENTS
---------------------------------|---------|-------|-------------
2  |Bass O Matic |1      |In production
ij(CONNECTION0)> relative -1 projectScroll;
ID |PROJECT_NAME |STATUS |COMMENTS
---------------------------------|---------|-------|-------------
1  |Wheel O Fish |0      |Waiting for contract
ij(CONNECTION0)> relative 2 projectScroll;
ID |PROJECT_NAME |STATUS |COMMENTS
---------------------------------|---------|-------|-------------
3  |Patty O Furniture |-1     |Need clarification. Does anybody know what this is?
ij(CONNECTION0)> previous projectScroll;
ID |PROJECT_NAME |STATUS |COMMENTS
---------------------------------|---------|-------|-------------
2  |Bass O Matic |1      |In production
ij(CONNECTION0)> before first projectScroll;
No current row
ij(CONNECTION0)> next projectScroll;
ID |PROJECT_NAME |STATUS |COMMENTS
---------------------------------|---------|-------|-------------
```
Note that autocommit must be off to use a scroll-insensitive cursor.

Here you use the `get scroll insensitive` cursor command, once again specifying an SQL statement for the data to include. Once you have this scroll-insensitive cursor, you have much more freedom of movement. You can jump to the second row using the `absolute` command, move a specified number of rows forward and backward using the `relative` command, move to the previous or next row, or use the `before first` and `after last` commands to position the cursor at the top or bottom of the data.

**Updateable cursors**

All of this is fine, but the real value of directly manipulating a cursor is the ability to update the database at a particular position. For example, you can loop through projects and update their statuses based on the current cursor position (see Listing 5).

**Listing 5. An updateable cursor**

```
ij(CONNECTION0)> autocommit off;
ij(CONNECTION0)> get cursor updateProjects as
   'select * from projects for update';
ij(CONNECTION0)> next updateProjects;
ID | PROJECT_NAME       | STATUS | COMMENTS
---------------------------------------------
 1  | Wheel O Fish       | 0      | Waiting for contract
ij(CONNECTION0)> update projects set status = 1, comments = 'Contract received, starting work.' where current of updateProjects;
1 row inserted/updated/deleted
ij(CONNECTION0)> next updateProjects;
ID | PROJECT_NAME       | STATUS | COMMENTS
---------------------------------------------
 2  | Bass O Matic       | 1      | In production
ij(CONNECTION0)> next updateProjects;
ID | PROJECT_NAME       | STATUS | COMMENTS
---------------------------------------------
 3  | Patty O Furniture  | -1     | Need clarification. Does anybody know what this is?
ij(CONNECTION0)> delete from projects where current of updateProjects;
1 row inserted/updated/deleted
ij(CONNECTION0)> select * from projects;
ID | PROJECT_NAME       | STATUS | COMMENTS
---------------------------------------------
 1  | Wheel O Fish       | 1      | Contract received, starting work.
 2  | Bass O Matic       | 1      | In production
2 rows selected
ij(CONNECTION0)> commit;
ij(CONNECTION0)> close updateProjects;
```

First of all, you must make absolutely certain that autocommit is off before creating an updateable cursor. Second, an updateable cursor must be forward only and must include the `for update` clause in the `select` statement.
From there, you manipulate the cursor as before, but you can also execute update and delete statements that have as their where clause the current position of the cursor rather than any particular data requirements.

Finally, when you're finished, close the cursor to free up resources.

**Using prepared statements**

Another task you don't normally associate with the command line is the use of prepared statements. Prepared statements are recompiled for better performance and are convenient when you need to do the same thing over and over again. For example, you can create a statement that sets all current developers to be available, as in **Listing 6**.

**Listing 6. Preparing a statement**

```
ij(CONNECTION0)> set connection connection1;
ij(CONNECTION1)> prepare allDevelopersOpen as
    'update developers set availability = ''open''';
ij(CONNECTION1)> execute allDevelopersOpen;
4 rows inserted/updated/deleted
ij(CONNECTION1)> select * from developers;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>DEVELOPER_NAME</th>
<th>AVAILABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jim Bacon</td>
<td>open</td>
</tr>
<tr>
<td>2</td>
<td>Brady James</td>
<td>open</td>
</tr>
<tr>
<td>3</td>
<td>Michelle Rappaport</td>
<td>open</td>
</tr>
<tr>
<td>4</td>
<td>Aaron Templeton</td>
<td>open</td>
</tr>
</tbody>
</table>

4 rows selected

Here you switch back to the other connection and create a prepared statement with a specific, hard-coded SQL statement, and then execute it.

What would be even more useful, however, is to create a statement that can be used for different rows and different values, such as that shown in **Listing 7**.
Listing 7. Prepared statements with values

ij(CONNECTION1)\> prepare updateOneDeveloper as
  'update developers set availability = ? where id = ?';
ij(CONNECTION1)\> execute updateOneDeveloper using
  'VALUES (''Im really serious here. Hire this guy!'', 4)';
1 row inserted/updated/deleted
ij(CONNECTION1)\> commit;
ij(CONNECTION1)\> select * from developers;

<table>
<thead>
<tr>
<th>ID</th>
<th>DEVELOPER_NAME</th>
<th>AVAILABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jim Bacon</td>
<td>open</td>
</tr>
<tr>
<td>2</td>
<td>Brady James</td>
<td>open</td>
</tr>
<tr>
<td>3</td>
<td>Michelle Rappaport</td>
<td>open</td>
</tr>
<tr>
<td>4</td>
<td>Aaron Templeton</td>
<td>Im really serious here. Hire this guy!</td>
</tr>
</tbody>
</table>

4 rows selected

As with prepared statements in a JDBC application, the values get substituted for each placeholder in order of appearance in the statement.

You can even create prepared statements that can be used with multiple sets of data (see Listing 8).

Listing 8. Prepared statements with multiple sets of values

ij(CONNECTION1)\> prepare addDeveloper as
  'insert into developers values (?, ?, ?)';
ij(CONNECTION1)\> execute addDeveloper using
  'VALUES (5, 'Corben Deeto', 'Available this summer'),
   (6, 'Seetha Pio', 'Open')';
1 row inserted/updated/deleted
1 row inserted/updated/deleted
ij(CONNECTION1)\> select * from developers;

<table>
<thead>
<tr>
<th>ID</th>
<th>DEVELOPER_NAME</th>
<th>AVAILABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jim Bacon</td>
<td>open</td>
</tr>
<tr>
<td>2</td>
<td>Brady James</td>
<td>open</td>
</tr>
<tr>
<td>3</td>
<td>Michelle Rappaport</td>
<td>open</td>
</tr>
<tr>
<td>4</td>
<td>Aaron Templeton</td>
<td>Im really serious here. HIRE this guy!</td>
</tr>
<tr>
<td>5</td>
<td>Corben Deeto</td>
<td>Available this summer</td>
</tr>
<tr>
<td>6</td>
<td>Seetha Pio</td>
<td>Open</td>
</tr>
</tbody>
</table>

6 rows selected

Using prepared statements with a cursor

Perhaps the most useful prepared statements are those that not only take input values, but also operate on the current row of a cursor. For example, you can create a statement that only affects the current developer (see Listing 9).
### Listing 9. A prepared statement with an updateable cursor

```sql
ij(CONNECTION1)> get cursor updateDevelopers as 'select * from developers for update';
ij(CONNECTION1)> prepare busyDeveloper as 'update developers set availability = ''Busy'' where current of updateDevelopers';
ij(CONNECTION1)> next updateDevelopers;
ID |DEVELOPER_NAME |AVAILABILITY
----------------------------------------
 1 |Jim Bacon       |open
ij(CONNECTION1)> next updateDevelopers;
ID |DEVELOPER_NAME |AVAILABILITY
----------------------------------------
 2 |Brady James     |open
ij(CONNECTION1)> execute busyDeveloper;
1 row inserted/updated/deleted
ij(CONNECTION1)> next updateDevelopers;
ID |DEVELOPER_NAME |AVAILABILITY
----------------------------------------
 3 |Michelle Rappaport |open
ij(CONNECTION1)> next updateDevelopers;
ID |DEVELOPER_NAME |AVAILABILITY
----------------------------------------
 4 |Aaron Templeton |Im really serious here. HIRE this guy!
ij(CONNECTION1)> execute busyDeveloper;
1 row inserted/updated/deleted
ij(CONNECTION1)> select * from developers;
ID |DEVELOPER_NAME |AVAILABILITY
----------------------------------------
 1 |Jim Bacon       |open
 2 |Brady James     |Busy
 3 |Michelle Rappaport |open
 4 |Aaron Templeton |Busy
 5 |Corben Deeto    |Available this summer
 6 |Seetha Pio      |Open
6 rows selected
ij(CONNECTION1)>
```

Technically, you're not doing anything different here. You create the updateable cursor and the prepared statement. The key is that the SQL statement in the prepared statement references the current position of the cursor, so you can execute it multiple times and update different rows. The task here may have been trivial, but this could just as easily have been an SQL statement that spanned multiple tables, executed triggers, and so on.

### Running ij scripts

An article on scripting ij wouldn’t be complete without actually discussing the running of ij scripts. An ij script is essentially a compilation of ij commands in a file, which you can then feed to the ij application. For example, you can create a script that adds a number of new developers and projects (see Listing 10):
Listing 10. A script to add new developers and projects

```sql
connect 'jdbc:derby:projectDB;';
insert into projects values (4, 'Movie Addict', 0, 'Waiting for contract');
insert into projects values (5, 'Tree Huggers Anonymous', 0, 'Waiting for contract');
select * from projects;
connect 'jdbc:derby:developerDB;';
insert into developers values (7, 'Frank Stein', 'Busy');
insert into developers values (8, 'Wolff Mann', 'Open');
select * from developers;
```

The script itself is pretty simple, just executing the same types of commands you might access from the command line.

To run the script, you have several alternatives. The first is to run it from within the ij application itself (see Listing 11).

Listing 11. Running an ij script from within the ij application

```sql
ij> run 'C:\myScripts\derbyscript.sql';
ij> connect 'jdbc:derby:projectDB;';
ij> insert into projects values (4, 'Movie Addict', 0, 'Waiting for contract');
  1 row inserted/updated/deleted
ij> insert into projects values (5, 'Tree Huggers Anonymous', 0, 'Waiting for contract');
  1 row inserted/updated/deleted
ij> select * from projects;
ID         |PROJECT_NAME          |STATUS     |COMMENTS
--------------------------------------------------------------------
1          |Wheel O Fish          |1          |Contract received, starting work.
2          |Bass O Matic          |1          |In production
4          |Movie Addict          |0          |Waiting for contract
5          |Tree Huggers Anonymous|0          |Waiting for contract
4 rows selected
ij> connect 'jdbc:derby:developerDB;';
ij(CONNECTION1)> insert into developers values (7, 'Frank Stein', 'Busy');
  1 row inserted/updated/deleted
```

Notice that ij echoes back each of the commands.

Another alternative is to reference the script when you first start up the ij application, essentially replacing stdin with the contents of the file (see Listing 12):
Listing 12. Feeding the ij script to the ij application

```
C:\SW\db-derby-10.1.2.1-bin\frameworks\embedded\bin>java
-Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
  org.apache.derby.tools.ij derbyscript.sql
ij version 10.1
ij> connect 'jdbc:derby:projectDB;'
ij> insert into projects values (4, 'Movie Addict', 0, 'Waiting for contract');
1 row inserted/updated/deleted
ij> insert into projects values (5, 'Tree Huggers Anonymous', 0, 'Waiting for contract');
1 row inserted/updated/deleted
ij> select * from projects;
|ID         |PROJECT_NAME       |STATUS     |COMMENTS                               
|-----------|-------------------|-----------|---------------------------------------
|1          |Wheel O Fish       |1          |Contract received, starting work.      
|2          |Bass O Matic       |1          |In production                          
```

When the script completes, the ij application exits. That's not the case when you run a script from within ij itself.

In that case, the program continues as normal until someone types `exit`.

In some cases, you'll want to save the output of a script running in ij. To do that, simply redirect the output to a file, or use the `ij.outputfile` property (see Listing 13):

### Listing 13. Saving the output of an ij script to a file

```
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
-Dij.outfile=results.txt org.apache.derby.tools.ij
derbyscript.sql
```

Note that if you redirect the output to a file, you won't see it on the command line.

**Summary**

In this article, you learned about some of the lesser-used features of the ij administration tool provided with the Apache Derby project. In addition to the normal functions of creating databases and executing screen captures, ij enables you to create and manipulate both forward-only and scroll-insensitive cursors as well as update them. You can create and execute prepared statements or create and run external scripts. All of these capabilities can be combined to create a fairly functional -- and useful -- application right in ij, without having to build any external infrastructure.
Related topics

- Learn more about Derby stored procedures by reading the "Create a wiki system using Derby, Part 2: Java functions, procedures, and triggers" tutorial (developerWorks, October 2005).
- Visit the developerWorks Apache Derby content area for Derby resources and articles.
- See Robert Brunner's series of articles, Developing with Apache Derby -- Hitting the Trifecta.
- Access the Apache Derby project tutorial for details on how to download and install Apache Derby.
- Browse all the Apache Derby articles and free tutorials available in the developerWorks Open source zone.
- Download the Apache Derby database.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)