The in-memory back end is not considered experiment anymore. Consult the Derby documentation for more information.

This page may be deleted at some point.

This page describes the experimental in-memory storage factory for Derby. When used, Derby will store the database only in main memory. All the data will be lost on any of the following events:

- the Java VM crashes / exists
- the Java VM is shut down normally
- the machine crashes or shuts down (i.e. power failure, OS hang, reboot)

The in-memory storage back end must only be used for temporary or reproducible data.

How to use

To instruct Derby to use the in-memory storage factory, specify `memory` as the JDBC sub-sub-protocol:

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

Note that the client driver doesn't understand the memory sub-sub-protocol, so you have to include it in the database name:

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

The same applies when using the data sources instead of `DriverManager`.

How to delete an in-memory database

The mechanisms to delete an in-memory database are:

- restart the Java VM,
- drop database using the "drop=true" URL connection attribute.

Configuration and tuning

The most important aspect to configure, is the size of the Java heap. When using the in-memory storage back end, the memory requirement should be similar to the memory requirements of using the disk-based storage back end plus the size of the user data. The general overhead will be increased somewhat.

Sizing the Derby page cache still impacts the performance when using the in-memory storage back end. The default value of 1000 pages should give acceptable performance for most basic applications, but it is not recommended that a smaller number of pages is specified. The data has to pass through the page cache even though the user data is already stored in main memory. A larger page cache may increase the performance at the cost of increased memory usage.

To stop Derby from writing anything to `derby.log`, see one of the `derby.stream.error` properties (for instance `derby.stream.error.field`).

Known limitations and bugs

Known problems and issues will be listed in the table below.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Jira</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doesn't work on Windows</td>
<td>DERBY-4125</td>
<td>Due to broken path handling, the in-memory storage back end doesn't work on Windows.</td>
<td>Fixed in 10.5.0 RC2 (fix revision 760387).</td>
</tr>
</tbody>
</table>

User feedback

Users testing out the feature are welcome to provide feedback to the Derby developer list, on the DERBY-646 Jira issue or on this page.

Feedback, positive or negative, is appreciated!

Future features

A basic list of possible future features. Feel free to comment on the existing ones, or add a new feature you would like to see.

Proper delete mechanism
A documented and stable way of deleting an in-memory database is to be added. The following are possible mechanisms:

1. A JDBC connection URL attribute.
2. A JMX method / action.
3. A documented public static method.

The JDBC connection URL attribute (1) fits nicely into the current usage pattern. It can use the monitor to resolve database names, and it can also reuse the authentication and (parts of) the authorization code. It will also be possible to implement the functionality for other storage back ends. One current drawback is that Derby (as of 10.5) doesn’t have system privileges. This implies that only the database owner can delete a database for now, and it will be impossible to delegate this privilege to a different user. This restriction seems better than allowing all authenticated users to delete the database though.

The JMX solution (2) should be easy to implement, but has several drawbacks. It poses some challenges regarding authentication and authorization, and it also requires users of Derby to either write their own little piece of JMX code or use an external tool (for instance VisualVM) to delete an in-memory database.

The rather crude solution of documenting a public static method (3) is also very easy to implement, but has many of the same challenges as (2) when it comes to authentication and authorization.

For the reasons mentioned above, solution (1) seems like the best approach.

**Automatic database persistence on JVM shutdown**

When the JVM goes down gracefully (i.e. all non-daemon threads finish or `System.exit` is invoked), an in-memory database could be persisted to disk. This allows the data to survive the JVM shutdown, but note that no guarantee can be made that the data will be persisted under all circumstances. The data should still be considered volatile / transient. If the persisted data is to be used again, it can either be booted with the normal file/disk storage back end, or loaded into an in-memory database.

The most obvious solution would be to add a JVM shutdown hook. Care must be taken to ensure that the database is in a consistent state so that it can be safely persisted. Since many components of the JVM may have disappeared at the time of the hook execution, the hook code should ideally be dependent on as little other code as possible. For instance, is the Derby system still in a state so that the database can be frozen? Is freezing the database indeed required, or will the Derby system be down already due to the JVM shutdown?

Things to consider:

- Force persistence to a non-existing directory?
- Allow updates, that is to overwrite an existing version of the database on disk? If yes, a kind of “safe copy” should be used to avoid destroying the previous data.

**Automatic database persistence on database shutdown**

Could be a separate feature, or maybe merged together with persistence on JVM shutdown.

One difference from the JVM shutdown issue, is that on a normal database shutdown the state of the whole system is a lot more stable.

**JMX monitoring and/or management**

Suggested functionality:

- List databases stored in-memory
- Obtain size of databases stored in main memory
- Obtain size of a single database stored in main memory

Discarded functionality:

- Delete database from main memory: will most likely be provided by a different mechanism.

**Add log writer for discarding the error log**

Derby already provides several mechanisms for controlling what happens to the log messages that are by default written to derby.log. It has been discussed to add a log writer that discards all messages to the Derby code base, to make it even easier for users to get rid of derby.log. Note that setting the error stream properties will affect all databases handled by a given Derby system, not only a specific in-memory database.

It seems more feedback / requests are required before the fate of this feature is settled.

**Bounding the growth of an in-memory database**

Requested by a developer at a San Francisco JUG meeting.

Keeping track of the space occupied by the data pages should be rather straight forward. The question is what to do when the limit has been reached. Some possibilities (some mentioned on the Derby user list):

1. Abort transaction, throw exception. Would require manual intervention to add more data.
2. Some kind of automatic deletion of data.
3. Automatic compression of all user tables.

Additional features mentioned:
• Make it possible to change the bound.

Risky features

The text below describes some risky actions you can do with the in-memory back end. You, and your data, are on your own on this one.

It is possible to create an in-memory database, work with it, and then take a backup before the Java VM is shut down. This will result in the database being persisted to disk, and it can be restored into main memory using the in-memory back end and one of the `createFrom` and `restoreFrom` JDBC connection URL attributes.

If you do this and it fails, we'd like to hear about it, but we won't feel sorry for you 😏

Always consider your data as transient / volatile when using the in-memory storage back end.

Jira references

DERBY-646