Documentation

Table of contents

Copyright © 2004 Paul Ferraro All rights reserved.

1. Introduction

HA-JDBC is a JDBC proxy that enables a Java application to transparently access a cluster of identical databases through the JDBC API.

Normal database access via JDBC	Database cluster access via HA-JDBC

HA-JDBC has the following advantages over normal JDBC:

- High-Availability: The database cluster is available to service requests so long as at least one database node is active.
- Fault Tolerance: Because HA-JDBC operates via the JDBC API, it is transaction-aware and can survive a database node failure without failing or corrupting current transactions.
- Scalability: By balancing read requests across databases, HA-JDBC can meet increasing load by scaling horizontally (i.e. adding database nodes).

2. Requirements

- Java Runtime Environment 1.4 or greater
- Type IV JDBC driver for underlying databases
- An XML configuration file for each database cluster.
- ha-jdbc-*.jar and its dependencies must exist in the classpath.

2.1. Dependencies

Library	Description	Notes
jgroups-*.jar	JGroups	
jibx-run-*.jar	JiBX	
quartz-*.jar	Quartz	
slf4j-api-*.jar	SLF4J API	
Additional dependencies for Java 1.4 and 1.5 only.		
stax-api-*.jar	Streaming API for XML.	Required dependency of JiBX.
Additional dependencies for Java 1.4 only.		
backport-util-concurrent-*.jar	JDK 1.4 backport of java.util.concurrent package.	Required dependency of Retrotranslator.
jdbc-rowset-*.jar	JDBC RowSet API and implementations	

Copyright © 2004 Paul Ferraro All rights reserved.

retrotranslator-runtime-*.jar	Retrotranslator runtime.	

Table 1: /lib : Required dependencies

Library	Description	Notes
jcl104-over-slf4j-*.jar	An implementation of the commons-logging api that delegates to SLF4J. Provides gradual migration from commons-logging to SLF4J.	Commons-logging is a required dependency of JGroups and Quartz.
slf4j-jcl-*.jar	Commons-logging provider for SLF4J.	Used only in conjunction with commons-logging.jar.
slf4j-jdk14-*.jar	JDK logging provider for SLF4J.	
slf4j-log4j12-*.jar	Log4J provider for SLF4J.	
slf4j-simple-*.jar	System.err logging provider for SLF4J.	
Additional runtime dependencies for Java 1.4 and 1.5 only.:		
stax-*.jar	StAX reference implementation	Any StAX implementation will suffice.
Additional runtime dependencies for Java 1.4 only:		
jmx-*.jar	JMX API and reference implementation	Needed if your runtime environment does not provide a JMX implementation. Any JMX implementation will suffice.

Table 2: /lib/runtime : Optional runtime dependencies

Note:

You must include **exactly one** SLF4J implementation library (i.e. slf4j-*-*.jar) in your classpath.

Warning:

Never use jcl104-over-slf4j-*.jar in conjunction with slf4j-jcl-*.jar; this will cause a StackOverflowError.

3. <ha-jdbc> Configuration

Configuration for an HA-JDBC managed database cluster is contained in an XML file. The algorithm used to locate the configuration file resource at runtime is as follows:

Copyright © 2004 Paul Ferraro All rights reserved.

1. Determine the resource name from one of the following sources:
    1. A *config* property passed to DriverManager.getConnection(String, Properties), or the getConfig() property of the DataSource
    2. The *ha-jdbc.configuration* system property
    3. Use default value of *ha-jdbc-{0}.xml*
2. Format the parameterized resource name using the identifier of the cluster.
3. Attempt to interpret the resource name as a URL.
4. If the resource name cannot be converted to a URL, then search for the resource in the classpath using the following class loaders:
    1. Search for resource using the thread context class loader.
    2. Search for resource using the class loader of the current class.
    3. Search for resource using the system class loader.
5. If still not found, throw an exception back to the caller.

The root element of the configuration file has the following definition:

```
<!ELEMENT ha-jdbc (distributable?,sync+,cluster)>
```

3.1. <distributable>

Indicates that database clusters defined in this file will be accessed by multiple JVMs. HA-JDBC leverages JGroups to handle communication between database clusters across servers.

```
<!ELEMENT distributable EMPTY>
<!ATTLIST distributable
  config  CDATA "stacks.xml"
  stack   CDATA "udp-sync"
  timeout CDATA "1000"
>
```

config

Defines one of the following:

- Name of a system resource containing the JGroups XML configuration.
- URL of the JGroups XML configuration file.
- Path of the JGroups XML configuration on the local file system.

The jgroups config file is expected to use the new (as of 2.3) muliplexing format. See the JGroup's Wiki for assistance with customizing the protocol stack.

stack

The stack name from the jgroups configuration file.

timeout

Indicates the number of milliseconds allowed for JGroups operations.

Copyright © 2004 Paul Ferraro All rights reserved.

3.2. <sync>

Defines a strategy for synchronizing a database before activation. If the strategy contains JavaBean properties, you can override their default values.

```
<!ELEMENT sync (property*)>

<!ATTLIST sync
  id    ID    #REQUIRED
  class CDATA #REQUIRED
>
```

id
Uniquely identifies this synchronization strategy. Used when invoking activation methods.

class
Class name of an implementation of the net.sf.hajdbc.SynchronizationStrategy interface. Details here.

3.2.1. <property>

```
<!ELEMENT property (#PCDATA)>
<!ATTLIST property
  name CDATA #REQUIRED
>
```

name
The name of the property. The value of the property is defined inside this element's contents.

3.3. <cluster>

Defines the nodes and behavior of a database cluster.

```
<!ELEMENT cluster (database+|datasource+)>

<!ATTLIST cluster
  balancer                (simple|random|round-robin|load) #REQUIRED
  default-sync            IDREF                            #REQUIRED
  dialect                 CDATA                            "standard"
  meta-data-cache         (none|lazy|eager)                #REQUIRED
  transaction-mode        (parallel|serial)                #REQUIRED
  auto-activate-schedule  CDATA                            #IMPLIED
  failure-detect-schedule CDATA                            #IMPLIED
  min-threads             CDATA                            "0"
  max-threads             CDATA                            "100"
  max-idle                CDATA                            "60"
  detect-identity-columns (true|false)                     "false"
  detect-sequences        (true|false)                     "false"
```


Copyright © 2004 Paul Ferraro All rights reserved.

```
  eval-current-date       (true|false)                    "false"
  eval-current-time       (true|false)                    "false"
  eval-current-timestamp  (true|false)                    "false"
  eval-rand               (true|false)                    "false"
>
```

balancer

Defines the balancer implementation used to distribute read operations among the active nodes of this cluster.

default-sync

Defines the unique identifier of the synchronization strategy to use by default when activating nodes of this cluster.

dialect

The value of this attribute defines either:

- the class name of an implementation of the net.sf.hajdbc.Dialect interface.
- A pre-defined alias as enumerated here.

HA-JDBC references the configured dialect for any vendor specific SQL.

meta-data-cache

Defines the strategy to use for caching database meta data.

none

Meta data is loaded when requested and not cached.

lazy

Meta data is loaded and cached as it is requested.

eager

All necessary meta data is loaded and cached during HA-JDBC initialization.

transaction-mode

Indicates whether transactional writes should execute in serial or parallel. If your application uses distributed transactions coordinated via a transaction manager (typically provided by an application server) using the X/Open XA protocol then you should use serial mode. In serial mode, database writes execute in consistent order across databases in the cluster thereby avoiding deadlocks. If your application uses normal database transactions then you may use parallel mode. Parallel mode is obviously more efficient than serial mode. If you find that your application suffers unacceptably from SQLExceptions due to concurrent updates of the same data then you might want to use serial mode to improve fault tolerance.

parallel

Transactional writes are executed in parallel, for efficiency.

serial

Transactional writes are executed in serial, for improved fault tolerance.

auto-activate-schedule

Copyright © 2004 Paul Ferraro All rights reserved.

Defines a cron schedule for an asynchronous task that will automatically activate any database nodes that are alive, but inactive. Schedule should be defined in accordance with the documentation for Quartz CronTrigger.

failure-detect-schedule

Defines a cron schedule for an asynchronous task that will proactively detect failed database nodes and deactivate them. Schedule should be defined in accordance with the documentation for Quartz CronTrigger.

min-threads

Defines the minimum size of the thread pool used for executing write operations.

max-threads

Defines the maximum size of the thread pool used for executing write operations.

max-idle

Defines the amount of time for which a non-core idle thread will remain in the thread pool before it is discarded.

detect-identity-columns

Indicates whether or not identity columns should be detected (if the configured dialect supports them) and measures taken to ensure that they are replicated correctly. If enabled, you should also use an eager meta-data-cache since identity column detection requires several database meta-data queries.

detect-sequences

Indicates whether or not sequence operations should be detected (if the configured dialect supports them) and measures taken to ensure that they are replicated correctly.

eval-current-date

Indicates whether or not SQL statements containing non-deterministic CURRENT_DATE functions should be replaced with deterministic client-generated values.

eval-current-time

Indicates whether or not SQL statements containing non-deterministic CURRENT_TIME functions should be replaced with deterministic client-generated values.

eval-current-timestamp

Indicates whether or not SQL statements containing non-deterministic CURRENT_TIMESTAMP functions should be replaced with deterministic client-generated values.

eval-rand

Indicates whether or not SQL statements containing non-deterministic RAND() functions should be replaced with deterministic client-generated values.

3.3.1. <database>

Copyright © 2004 Paul Ferraro All rights reserved.

Defines the databases in this cluster that will be referenced via the java.sql.DriverManager facility.

```
<!ELEMENT database (driver?, url, property*, (user, password)?)>
```

driver
Defines the class name of the JDBC driver used to access this database.
url
Defines the JDBC url used to access this database.
property
Defines a property to be passed to the java.sql.Driver.connect() method.
user
Defines the user, if any, that HA-JDBC should use to connect to the database during synchronization and database failure detection.
password
Defines the password, if any, that HA-JDBC should use to connect to the database during synchronization and database failure detection.

```
<!ATTLIST database
        id     CDATA        #REQUIRED
        weight CDATA        "1"
        local  (true|false) "false"
>
```

id
Unique identifier for this database node.
weight
Defines the relative weight of this database node. The weight is used by the balancer implementation to determine which node will service a read request.
local
Indicates that this database resides on the local machine.

3.3.2. <datasource>, <pool-datasource>, <xa-datasource>

Defines the databases in this cluster that will be referenced via a javax.sql.DataSource.

```
<!ELEMENT datasource (name, property*, (user, password)?>
```

name
Defines one of the following:

- The JNDI name of this DataSource.
- The class name of the DataSource.

property
Depending on the value of <name>, properties are interpreted as:

- A JNDI environment property used when creating an InitialContext from which to lookup this data source.

Copyright © 2004 Paul Ferraro All rights reserved.

- A JavaBean property with which to initialize the data source.

user
Defines the user, if any, that HA-JDBC should use to connect to the database during synchronization and database failure detection.
password
Defines the password, if any, that HA-JDBC should use to connect to the database during synchronization and database failure detection.

```
<!ATTLIST datasource
        id     CDATA        #REQUIRED
        weight CDATA        "1"
        local  (true|false) "false"
>
<!ATTLIST pool-datasource
        id     CDATA        #REQUIRED
        weight CDATA        "1"
        local  (true|false) "false"
>
<!ATTLIST xa-datasource
        id     CDATA        #REQUIRED
        weight CDATA        "1"
        local  (true|false) "false"
>
```

id
Unique identifier for this database node.
weight
Defines the relative weight of this database node. The weight is used by the balancer implementation to determine which node will service a read request.
local
Indicates that this database resides on the local machine.

3.4. Dialect

The `dialect` attribute of a cluster determines the SQL syntax used for a given task. The value specified for this attribute is either the name of a class that implements net.sf.hajdbc.Dialect (custom implementations are allowed), or, more conveniently, a pre-defined, case-insensitive alias.

Vendor(s)	Dialect	Alias
Apache Derby	net.sf.hajdbc.dialect.DerbyDialec	derby
Firebird, InterBase	net.sf.hajdbc.dialect.FirebirdDiale	firebird
H2	net.sf.hajdbc.dialect.H2Dialect	h2
HSQLDB	net.sf.hajdbc.dialect.HSQLDBDia	hsqldb

Copyright © 2004 Paul Ferraro All rights reserved.

IBM DB2	net.sf.hajdbc.dialect.DB2Dialect	db2
Ingres	net.sf.hajdbc.dialect.IngresDialec	ingres
Mckoi	net.sf.hajdbc.dialect.MckoiDialec	mckoi
MySQL	net.sf.hajdbc.dialect.MySQLDiale	mysql
MySQL MaxDB	net.sf.hajdbc.dialect.MaxDBDiale	maxdb
Oracle	net.sf.hajdbc.dialect.OracleDialec	oracle
PostgreSQL	net.sf.hajdbc.dialect.PostgreSQL	postgresql
Sybase	net.sf.hajdbc.dialect.SybaseDiale	sybase
Standard (SQL-92 compliant)	net.sf.hajdbc.dialect.StandardDia	standard

Table 1: The HA-JDBC distribution contains the following dialect implementations:

Note:
Dialect contributions are more than welcome. Please submit any additions/updates here.

3.5. Balancer

When executing a read request from the cluster, HA-JDBC uses the configured balancer strategy to determine which database should service the request. Each database can define a weight to affect how it is prioritized by the balancer. If no weight is specified for a given database, it is assumed to be 1.

HA-JDBC supports four types of balancers:

simple
Requests are always sent to the node with the highest weight.
random
Requests are sent to a random node. Node weights affect the probability that a given node will be chosen. The probability that a node will be chosen = *weight* / *total-weight*.
round-robin
Requests are sent to each node in succession. A node of weight *n* will receive *n* requests before the balancer moves on to the next node.
load
Requests are sent to the node with the smallest load. Node weights affect the calculated load of a given node. The load of a node = *concurrent-requests* /

Copyright © 2004 Paul Ferraro All rights reserved.

weight.

Note:
In general, a node with a weight of 0 will never service a request unless it is the last node in the cluster.

3.6. Synchronization Strategy

Synchronization is performed before a database node is activated.

HA-JDBC provides several out-of-the-box database independent strategies for synchronizing a failed database:

net.sf.hajdbc.sync.FullSynchronizationStrategy
Each table in the inactive database is truncated and data is reinserted from an active database. This strategy is fastest if the database is way out of sync.
net.sf.hajdbc.sync.DifferentialSynchronizationStrategy
For each table in the inactive database is compared, row by row, with an active database and only changes are updated. This strategy is fastest if the database is more in sync than not.
net.sf.hajdbc.sync.PassiveSynchronizationStrategy
Does nothing. Should only be used if databases are known to be in sync.

Each synchronization strategy must be defined in the HA-JDBC configuration file. A strategy may contain any number of JavaBean properties that can be set in the config file.

e.g.

```
<ha-jdbc>
  <!-- ... -->
  <sync id="diff"
class="net.sf.hajdbc.sync.DifferentialSynchronizationStrategy">
    <property name="fetchSize">1000</property>
    <property name="maxBatchSize">100</property>
  </sync>
  <sync id="full" class="net.sf.hajdbc.sync.FullSynchronizationStrategy">
    <property name="fetchSize">1000</property>
    <property name="maxBatchSize">100</property>
  </sync>
  <sync id="passive"
class="net.sf.hajdbc.sync.PassiveSynchronizationStrategy"></sync>
  <!-- ... -->
</ha-jdbc>
```

Although the build-in strategies should be sufficient for most small databases, they are probably not feasible for large databases. Synchronizing a large database will typically require vendor specific functionality. Custom synchronization strategies may be written by implementing the net.sf.hajdbc.SynchronizationStrategy interface or by

Copyright © 2004 Paul Ferraro All rights reserved.

extending the functionality of one of the existing strategies. For example, I may want to improve the efficiency of the `FullSynchronizationStrategy` by dropping and re-creating indexes on my database tables.

e.g.

```
public class FasterFullSynchronizationStrategy extends
net.sf.hajdbc.sync.FullSynchronizationStrategy
{
    public void synchronize(SynchronizationContext context)
    {
        // For each table, drop all indexes

        super.synchronize(context);

        // For each table, recreate all indexes
    }
}
```

Any custom strategies that you plan to use should also be defined in the configuration file.

4. Using HA-JDBC

4.1. DriverManager-based Access

Just like your database's JDBC driver, the HA-JDBC driver must first be loaded. As of Java 1.6, the DriverManager uses the service provider mechanism to auto-load JDBC drivers on startup. Java versions prior to 1.6 must load the HA-JDBC driver manually. This can be accomplished in one of two ways:

- Add `net.sf.hajdbc.sql.Driver` to your `jdbc.drivers` system property.

  The JVM will automatically load and register each driver upon startup.
- Explicitly load the `net.sf.hajdbc.sql.Driver` class using `Class.forName(...)`.

  Per the JDBC specification, loading the HA-JDBC driver class automatically registers the driver with the `DriverManager`. The HA-JDBC driver will automatically load all underlying JDBC drivers defined within a given cluster. This means that you do not need to perform an additional `Class.forName(...)` to load your database's driver.

e.g.

The following is a sample HA-JDBC configuration that uses the DriverManager facility to obtain connections.

Copyright © 2004 Paul Ferraro All rights reserved.

```
<ha-jdbc>
  <!-- ... -->
  <cluster balancer="..." dialect="PostgreSQL" default-sync="..."
transaction-mode="...">
    <database id="database1">
      <driver>org.postgresql.Driver</driver>
      <url>jdbc:postgresql://server1/database</url>
      <user>postgres</user>
      <password>password</password>
    </datasource>
    <database id="database2">
      <driver>org.postgresql.Driver</driver>
      <url>jdbc:postgresql://server2/database</url>
      <user>postgres</user>
      <password>password</password>
    </datasource>
  </cluster>
</ha-jdbc>
```

The URL specified in subsequent calls to `DriverManager.getConnection(...)` has the following format:

`jdbc:ha-jdbc:`*cluster-id*

e.g.

```
Connection connection =
DriverManager.getConnection("jdbc:ha-jdbc:cluster1", "postgres",
"password");
```

The following is a sample Tomcat configuration that sets up a connection pool for the above HA-JDBC database cluster.

server.xml

```
<Context>
  <!-- ... -->
  <Resource name="jdbc/cluster" type="javax.sql.DataSource"
            username="postgres" password="password"
driverClassName="net.sf.hajdbc.sql.Driver"
            url="jdbc:ha-jdbc:cluster1"/>
  <!-- ... -->
</Context>
```

web.xml

```
<web-app>
  <!-- ... -->
  <resource-env-ref>
    <resource-env-ref-name>jdbc/cluster</resource-env-ref-name>
    <resource-env-ref-type>javax.sql.DataSource</resource-env-ref-type>
```


Copyright © 2004 Paul Ferraro All rights reserved.

```
  </resource-env-ref>
  <!-- ... -->
</web-app>
```

Pooled connections to the HA-JDBC cluster are now available via a DataSource at `java:comp/env/jdbc/cluster`.

4.2. DataSource-based Access

An HA-JDBC cluster can also wrap one or more DataSources.

Connections are made to the HA-JDBC cluster via the following code:

```
Context context = new InitialContext();
DataSource dataSource = (DataSource)
context.lookup("java:comp/env/jdbc/cluster");
Connection connection = dataSource.getConnection();
```

You can define underlying data sources either by JNDI names to which they are already bound, or explicitly in your configuration file.

4.2.1. JNDI-based Configuration

e.g.

The following is a sample HA-JDBC configuration that uses JNDI-bound DataSource facilities to obtain connections.

```
<ha-jdbc>
  <!-- ... -->
  <cluster balancer="..." dialect="PostgreSQL" default-sync="..."
transaction-mode="...">
    <datasource id="database1">
      <name>java:comp/env/jdbc/database1</name>
    </datasource>
    <datasource id="database2">
      <name>java:comp/env/jdbc/database2</name>
    </datasource>
  </cluster>
</ha-jdbc>
```

The corresponding Tomcat configuration might look like the following:

server.xml:

```
<Context>
  <!-- ... -->
  <Resource name="jdbc/database1" type="javax.sql.DataSource"
            username="postgres" password="password"
driverClassName="org.postgresql.Driver"
```


Copyright © 2004 Paul Ferraro All rights reserved.

```
            url="jdbc:postgresql://server1/database"/>

  <Resource name="jdbc/database2" type="javax.sql.DataSource"
            username="postgres" password="password"
driverClassName="org.postgresql.Driver"
            url="jdbc:postgresql://server2/database"/>

  <Resource name="jdbc/cluster" type="javax.sql.DataSource"
            factory="net.sf.hajdbc.sql.DataSourceFactory"
            cluster="cluster2" config="file:///path/to/ha-jdbc-{0}.xml"/>
  <!-- ... -->
</Context>
```

web.xml:

```
<web-app>
  <!-- ... -->
  <resource-env-ref>
    <resource-env-ref-name>jdbc/database1</resource-env-ref-name>
    <resource-env-ref-type>javax.sql.DataSource</resource-env-ref-type>
  </resource-env-ref>
  <resource-env-ref>
    <resource-env-ref-name>jdbc/database2</resource-env-ref-name>
    <resource-env-ref-type>javax.sql.DataSource</resource-env-ref-type>
  </resource-env-ref>
  <resource-env-ref>
    <resource-env-ref-name>jdbc/cluster</resource-env-ref-name>
    <resource-env-ref-type>javax.sql.DataSource</resource-env-ref-type>
  </resource-env-ref>
  <!-- ... -->
</web-app>
```

4.2.2. Explicit Configuration

e.g.

The following is a sample HA-JDBC configuration that uses explicitly defines DataSource facilities to obtain connections. This setup has the advantage of not exposing underlying data sources to your application and is less verbose.

```
<ha-jdbc>
  <!-- ... -->
  <cluster balancer="..." dialect="PostgreSQL" default-sync="..."
transaction-mode="...">
    <datasource id="database1">
      <name>org.postgresql.ds.PGSimpleDataSource</name>
      <property name="serverName">server1</property>
      <property name="portNumber">5432</property>
      <property name="databaseName">database</property>
    </datasource>
    <datasource id="database2">
      <name>org.postgresql.ds.PGSimpleDataSource</name>
```


Copyright © 2004 Paul Ferraro All rights reserved.

```
      <property name="serverName">server2</property>
      <property name="portNumber">5432</property>
      <property name="databaseName">database</property>
    </datasource>
  </cluster>
</ha-jdbc>
```

The corresponding Tomcat configuration might look like the following:

server.xml:

```
<Context>
  <!-- ... -->
  <Resource name="jdbc/cluster" type="javax.sql.DataSource"
            factory="net.sf.hajdbc.sql.DataSourceFactory"
            cluster="cluster2" config="file:///path/to/ha-jdbc-{0}.xml"/>
  <!-- ... -->
</Context>
```

web.xml:

```
<web-app>
  <!-- ... -->
  <resource-env-ref>
    <resource-env-ref-name>jdbc/cluster</resource-env-ref-name>
    <resource-env-ref-type>javax.sql.DataSource</resource-env-ref-type>
  </resource-env-ref>
  <!-- ... -->
</web-app>
```

4.3. XADataSource-based access

An HA-JDBC cluster can also wrap one or more XADataSources. You can define underlying data sources either by JNDI names to which they are already bound, or explicitly in your configuration file.

Warning:

Typically, when configuring an XADataSource for your application server, only the DataSource exposed to the application is bound to JNDI, not the XADataSource itself. It is crucial that you configure HA-JDBC to proxy the XADataSource, and NOT the application-facing DataSource directly. HA-JDBC must be able to detect the boundaries of a transaction, which, in the case of distributed transactions, are visible to the XAConnections created by the XADataSource, but not the Connections created by the application facing DataSource.

e.g.

The following is a sample HA-JDBC configuration that uses explicitly defines XADataSource facilities to obtain connections.

```
<ha-jdbc>
  <!-- ... -->
```


Copyright © 2004 Paul Ferraro All rights reserved.

```
  <cluster balancer="..." dialect="PostgreSQL" default-sync="..."
transaction-mode="...">
    <xa-datasource id="database1">
      <name> org.postgresql.xa.PGXADataSource</name>
      <property name="serverName">host1</property>
       <property name="portNumber">5432</property>
       <property name="databaseName">mydatabase</property>
    </xa-datasource>
    <xa-datasource id="database2">
      <name> org.postgresql.xa.PGXADataSource</name>
      <property name="serverName">host1</property>
       <property name="portNumber">5432</property>
       <property name="databaseName">mydatabase</property>
    </xa-datasource>
  </cluster>
</ha-jdbc>
```

The corresponding JBoss configuration might look like the following:

```
<datasources>
  <xa-datasource>
    <jndi-name>haDS</jndi-name>
<xa-datasource-class>net.sf.hajdbc.xa.XADataSource</xa-datasource-class>
    <xa-datasource-property
name="Cluster">mycluster</xa-datasource-property>
    <track-connection-by-tx/>
  </xa-datasource>
</datasources>
```

Connections are made to the HA-JDBC cluster via the following code:

```
Context context = new InitialContext();
DataSource dataSource = (DataSource) context.lookup("java:/haDS");
Connection connection = dataSource.getConnection();
```

4.4. Unique identifier generation

As of version 2.0, HA-JDBC now supports database sequences and identity (i.e. auto-incrementing) columns.

It is important to note the performance implications when using sequences and/or identity columns in conjunction with HA-JDBC. Both algorithms introduce per statement regular expression matching and mutex costs in HA-JDBC, the latter being particularly costly for distributed environments. Because of their performance impact, support for both sequences and identity columns can be disabled via the detect-sequences and detect-identity-columns cluster attributes, respectively. Fortunately, the performance penalty for sequences can be mitigated via what Hibernate calls a Sequence-HiLo algorithm.

For best performance, HA-JDBC recommends using a table-based high-low or UUID algorithm so that statement parsing and locking costs can be avoided. Object-relation

Copyright © 2004 Paul Ferraro All rights reserved.

mapping (ORM) frameworks (e.g. Hibernate, OJB, Cayenne) typically include implementations of these mechanisms.

4.5. Failed Database Nodes

A database node may fail for any number of reasons:

- Network outage
- Hardware failure
- Operating System crash
- Database application crash
- Out of disk space
- No more free connections
- etc.

Failed database nodes are detected after an SQLException is thrown when executing a given database operation. A database is determined to have failed if it fails to respond to a trivial query (e.g. SELECT 1). The query used to validate that a database is alive is defined by the configured dialect.

This query may be executed manually, via the `isAlive(String)` method on the management interface.

If HA-JDBC determines that a given database has failed:

1. An ERROR message is logged.
2. The database is removed from the internal registry of active databases. No more requests will be sent to this database.
3. If the cluster was configured to be *distributable*, other servers are notified of the deactivation.

Databases can also be manually deactivated via the JMX management interface.

Optionally, you can configure HA-JDBC to proactively detect database failures via the `failure-detect-schedule` attribute. The value of this attribute defines a cron expression, which specifies the schedule a database cluster will detect failed databases and deactivate them.

e.g.

```
<ha-jdbc>
  <!-- ... -->
  <!-- Failure detection will run every minute -->
  <cluster ... failure-detect-schedule="0 * * ? * *">
    <!-- ... -->
```


Copyright © 2004 Paul Ferraro All rights reserved.

```
  </cluster>
</ha-jdbc>
```

4.6. Database Cluster Administration

HA-JDBC database clusters are administered via one of two JMX interfaces:

- net.sf.hajdbc.sql.DriverDatabaseClusterMBean
- net.sf.hajdbc.sql.DataSourceDatabaseClusterMBean

Most of the management operations are defined the common super interface, net.sf.hajdbc.DatabaseClusterMBean.

A database cluster mbean is registered using the following object-name:

net.sf.ha-jdbc:cluster=*database-cluster-id*

JMX operations can be executed from the JConsole interface packaged with JDK 1.5+.

As an alternative to JConsole (namely, for Java 1.4 deployments), you can use any of several 3rd-party JMX clients:

- MC4J
- jManage
- EJTools JMX Browser
- Panoptes
- Eclipse-JMX

HA-JDBC database clusters may also be administered programatically:

e.g.

```
String clusterId = "cluster1";
String databaseId = "database1";

MBeanServer server = ManagementFactory.getPlatformMBeanServer();
ObjectName name = ObjectName.getInstance("net.sf.hajdbc", "cluster",
clusterId);

// There are 2 ways to programatically invoke methods on an mbean:

// 1. Generic invoke
Object[] parameterValues = new Object[] { databaseId };
String[] parameterTypes = new String[] { String.class.getName() };
server.invoke(name, "activate", parameterValues, parameterTypes);

// 2. Dynamic proxy
DriverDatabaseClusterMBean cluster = JMX.newMBeanProxy(server, name,
DriverDatabaseClusterMBean.class);
cluster.activate(databaseId);
```


Copyright © 2004 Paul Ferraro All rights reserved.

4.6.1. Activating a Database

Database nodes are activated by executing one of the following methods on the database cluster mbean:

`activate(String databaseId)`
Synchronizes, using the default synchronization strategy, and activates the specified database.

`activate(String databaseId, String strategyId)`
Synchronizes, using the specified synchronization strategy, and activates the specified database.

In general, database synchronization is an intensive and intrusive task. To maintain database consistency, each database node in the cluster is read locked (i.e. writes are blocked) until synchronization completes. Since synchronization may take anywhere from seconds to hours (depending on the size of your database and synchronization strategy employed), if your database cluster is used in a high write volume environment, it is recommended that activation only be performed during off-peak hours.

As of version 1.1, HA-JDBC includes a useful database cluster option: "auto-activate-schedule". If specified, HA-JDBC will automatically attempt to activate database nodes that are inactive, but alive, according to the specified cron schedule.

e.g.

```
<ha-jdbc>
  <!-- ... -->
  <!-- Auto-activation will run every day at 2:00 AM -->
  <cluster ... auto-activate-schedule="0 0 2 ? * *">
    <!-- ... -->
  </cluster>
</ha-jdbc>
```

Note:
In distributable mode, listening servers are automatically notified of any activated databases.

4.6.2. Deactivating a Database

There are a number of reasons why you might want to deactivate a database node manually - most commonly, to perform some maintenance on the database or the machine itself. Database nodes are deactivated by executing the following method on the database cluster mbean:

`deactivate(String databaseId)`

Copyright © 2004 Paul Ferraro All rights reserved.

Deactivates the specified database.

Note:
In distributable mode, listening servers are automatically notified of any deactivated databases.

4.6.3. Adding a Database

In HA-JDBC 1.0, the database nodes in a cluster were static. They could not be altered while the application/server was running.

As of HA-JDBC 1.1, database nodes can be added, updated, or removed during runtime.

net.sf.hajdbc.sql.DriverDatabaseClusterMBean:

- `add(String databaseId, String driver, String url)`
- `remove(String databaseId)`

net.sf.hajdbc.sql.DataSourceDatabaseClusterMBean:

- `DataSourceDatabaseClusterMBean.add(String databaseId, String name)`
- `remove(String databaseId)`

When a new database is added, it is initially inactive. Database nodes may also be removed from the cluster. A database node must first be inactive before it can be removed.

HA-JDBC 1.1 also adds mbean interfaces for individual database nodes:

- `net.sf.hajdbc.sql.InactiveDriverDatabaseMBean`
- `net.sf.hajdbc.sql.InactiveDataSourceDatabaseMBean`
- `net.sf.hajdbc.sql.ActiveDriverDatabaseMBean`
- `net.sf.hajdbc.sql.ActiveDataSourceDatabaseMBean`

Database mbeans are registered using the following object-name:

net.sf.ha-jdbc:cluster=*cluster-id*,database=*database-id*

While inactive, a database node's connection properties (e.g. user, password, etc.) may be modified. When active, a database node's connection properties are read-only.

When a database node is activated and it's configuration has changed, or when a database node is removed, the configuration is saved to it's original URL.

Warning:
In distributable mode, listening servers are *not* notified of database node additions, updates, and removals. These cluster modifications must be made on each server.

Copyright © 2004 Paul Ferraro All rights reserved.

5. Limitations

- HA-JDBC does not safely support stored procedures that update sequences or insert rows containing identity columns.

6. Migrating from HA-JDBC 1.1

6.1. Environment

- HA-JDBC is now compatible with JDBC 4.0 found in Java 1.6.
- HA-JDBC now supports sequences and identity columns. Users should be aware of the performance implications of these mechanisms.
- HA-JDBC now fully supports large objects implemented as SQL locators.

6.2. Configuration

- Prior to version 2.0, the ha-jdbc.xml file could contain many `<cluster>s` distinguished by an `id` attribute. As of version 2.0, each cluster is defined in its own XML file, named according to the pattern: ha-jdbc-{0}.xml.
- The `<cluster>` element no longer defines an `id` attribute.
- The `transaction-mode="parallel|serial"` attribute replaces the `transactions="local|xa"` attribute of `<cluster>`
- New required meta-data-cache="none|lazy|eager" cluster attribute.
- `<distributable>` mode now uses JGroups channel multiplexing. The old `protocol` attribute is superceded by the new `config` and `stack` attributes.
- `dialect="default"` is renamed `dialect="standard"`.
- Clusters of Firebird, Mckoi, or Ingres databases should use the respective dialects, instead of the standard dialect.

6.3. JMX

- HA-JDBC no longer quotes identifiers when constructing the names for cluster and database mbeans.

Copyright © 2004 Paul Ferraro All rights reserved.