c3p0 - JDBC3 Connection and Statement Pooling

version 0.13.0

by Steve Waldman <swaldman@mchange.com>

- API docs for c3p0 are here.
- Looking for the definition of configuration properties?
- Looking for advice in using c3p0 with hibernate?

- Follow or fork c3p0 on GitHub.
- Follow tech.interfluidity.com.
- This may not be the most recent version of c3p0. See the current CHANGELOG.

Maven coordinates

You'll find this version on the Maven Central repository — **`com.mchange:c3p0:0.13.0`**

For support of asynchrony via Java 21 "loom" virtual threads, use instead — **`com.mchange:c3p0-loom:0.13.0`**

Security Note

To its author's profound shame, c3p0, along with its supporting libraries, was used for about a decade as a "deserialization gadget". If an attacker is able to replace and maliciously recraft a `javax.naming.Reference` or Java-serialized object that an application will decode, c3p0's libraries could be misused to expand that access into execution of arbitrary malicious code.

c3p0-0.12.0, along with its dependency mchange-commons-java-0.4.0, includes mitigations that lock down the functionality misused as gadget chains.

Although it remains possible to resurrect and make use of the dangerous functionality, it requires new, affirmative configuration, and very few contemporary applications should do so.

Most installations will not, but if you experience breaking changes in c3p0-0.12.0, you may need to customize security configuration for your deployment. Please see Configuring Security below for information on how, and for more background on the security issues.

c3p0-0.13.0, with mchange-commons-java-0.5.0, eliminates all use of Java serialization in resolving `Reference`s, definitively ending any possibility of misuse of c3p0-related JNDI utilities to construct deserialization gadgets.

Many thanks to David Pollak of Spice Labs for a very detailed report about this issue.

See also Warning: c3p0 trusts its CLASSPATH and configuration.

Contents

1. Contents
2. Quickstart
3. What is c3p0?
4. Prerequisites
5. Installation
6. Using c3p0
   i. Using ComboPooledDataSource
   ii. Using the DataSouces factory class
      - Box: Overriding authentication information (from non-c3p0 DataSources)
   iii. Querying Pool Status
      - Box: Using C3P0Registry to find a reference to a DataSource
   iv. Cleaning Up Pool Resources
   v. Advanced: Building Your Own PoolBackedDataSource
   vi. Advanced: Raw Connection and Statement Operations
7. Configuration
   i. Introduction
      - Warning: c3p0 trusts its CLASSPATH and configuration!
   ii. Basic Pool Configuration
   iii. Managing Pool Size and Connection Age

iv. Configuring Connection Testing
  - Box: Simple advice on Connection testing
v. Configuring Statement Pooling
vi. Configuring Recovery From Database Outages
vii. Managing Connection Lifecycles with Connection Customizers
viii. Configuring Unresolved Transaction Handling
ix. Configuring To Debug and Workaround Broken Client Applications
x. Configuring To Avoid Memory Leaks On Redeploy
xi. Configuring Threading
xii. Other DataSource Configuration
xiii. Configuring and Managing c3p0 via JMX
xiv. Configuring Logging
xv. Configuring Security
xvi. Named configurations
xvii. Per-user configurations
xviii. User extensions to configuration
xix. Mixing named, per-user, and user-defined configuration extensions

8. Performance
9. Known shortcomings
10. Feedback and support
11. Appendix A: Configuration Properties
12. Appendix B: Configuation Files, etc.
    i. Overriding c3p0 defaults with a c3p0.properties file
    ii. Overriding c3p0 defaults with "HOCON" (typesafe-config) configuration files
    iii. Overriding c3p0 defaults with System properties
    iv. Named and Per-User configuration: Overriding c3p0 defaults via c3p0-config.xml
    v. Precedence of Configuration Settings
13. Appendix C: Hibernate-specific notes
14. Appendix D: Configuring c3p0 pooled DataSources for Apache Tomcat
15. Appendix E: JBoss-specific notes
16. Appendix F: Oracle-specific API: createTemporaryBLOB() and createTemporaryCLOB()
17. Appendix G: Legacy, configuring Connection testing with a ConnectionTester

(See also the API Documentation here)

Quickstart ⇑

c3p0 was designed to be butt-simple to use.

Just bring Maven dependency `com.mchange:c3p0:0.13.0` into your application's effective `CLASSPATH` (which should bring along its one transitive dependency, `mchange-commons-java`). Then make a DataSource like this:

```
import com.mchange.v2.c3p0.*;

...

ComboPooledDataSource cpds = new ComboPooledDataSource();
cpds.setDriverClass( "org.postgresql.Driver" ); //loads the jdbc driver
cpds.setJdbcUrl( "jdbc:postgresql://localhost/testdb" );
cpds.setUser("dbuser");
cpds.setPassword("dbpassword");
```

[Optional] If you want to turn on PreparedStatement pooling, you must also set `maxStatements` and/or `maxStatementsPerConnection` (both default to 0):

```
cpds.setMaxStatements( 180 );
```

Do whatever you want with your DataSource, which will be backed by a Connection pool set up with default parameters. You can bind the DataSource to a JNDI name service, or use it directly, as you prefer.

When you are done, you can clean up the DataSource you've created like this:

```
cpds.close();
```

That's it! The rest is detail.

What is c3p0?

c3p0 is an easy-to-use library for making traditional JDBC drivers "enterprise-ready" by augmenting them with functionality defined by the jdbc3 spec and the optional extensions to jdbc2. c3p0 now also fully supports the jdbc4.

In particular, c3p0 provides several useful services:

- A class which adapts traditional DriverManager-based JDBC drivers to the newer `javax.sql.DataSource` scheme for acquiring database Connections.
- Transparent pooling of Connection and PreparedStatements behind DataSources which can "wrap" around traditional drivers or arbitrary unpooled DataSources.

The library tries hard to get the details right:

- c3p0 DataSources are both `Referenceable` and `Serializable`, and are thus suitable for binding to a wide-variety of JNDI-based naming services.
- Statement and ResultSets are carefully cleaned up when pooled Connections and Statements are checked in, to prevent resource-exhaustion when clients use the lazy but common resource-management strategy of only cleaning up their Connections. (Don't be naughty.)
- The library adopts the approach defined by the JDBC 2 and 3 specification (even where these conflict with the library author's preferences). DataSources are written in the JavaBean style, offering all the required and most of the optional properties (as well as some non-standard ones), and no-arg constructors. All JDBC-defined internal interfaces are implemented (ConnectionPoolDataSource, PooledConnection, ConnectionEvent-generating Connections, etc.) You can mix c3p0 classes with compliant third-party implementations (although not all c3p0 features will work with external implementations of `ConnectionPoolDataSource`).

c3p0 hopes to provide DataSource implementations more than suitable for use by high-volume "J2EE enterprise applications". Please provide feedback, bug-fixes, etc!

Prerequisites

c3p0-0.13.0 requires a level 1.7.x or above Java Runtime Environment.

Installation

There is no installation beyond accessing managed Maven dependency **`com.mchange:c3p0:0.13.0`**.

If you wish to make use of Java 21 ("loom") virtual threading, use **`com.mchange:c3p0-loom:0.13.0`** instead.

If you want to install c3p0 by hand, just place the files `c3p0-0.13.0.jar` and `mchange-commons-java-0.5.0.jar` somewhere in your CLASSPATH (or any other place where your application's classloader will find it). For Java 21 "loom" support, also include the jar `c3p0-loom-0.13.0.jar`.

Using c3p0

From a users' perspective, c3p0 simply provides standard jdbc DataSource objects. When creating these DataSources, users can control pooling-related, naming-related, and other properties. (See Appendix A for a comprehensive list of configuration properties.) All pooling is entirely transparent to users once a DataSource has been created.

There are three ways of acquiring c3p0 pool-backed DataSources: 1) directly instantiate and configure a `ComboPooledDataSource` bean; 2) use the `DataSources` factory class; or 3) "build your own" pool-backed DataSource by directly instantiating `PoolBackedDataSource` and setting its `ConectionPoolDataSource`. Most users will probably find instantiating `ComboPooledDataSource` to be the most convenient approach. Once instantiated, c3p0 DataSources can be bound to nearly any JNDI-compliant name service.

Regardless of how you create your DataSource, c3p0 will use defaults for any configuration parameters that you do not specify programmatically. c3p0 has built-in, hard-coded defaults, but you can override these with configuration files, placed as top-level resources in the same `CLASSPATH` (or ClassLoader) that loads c3p0's jar file.

c3p0 DataSources may be configured via simple `java.util.Properties` files called `c3p0.properties`, via more advanced HOCON configuration files (e.g. `application.conf`, `application.json`), or in an XML format, `c3p0-config.xml`. See Configuration below.

Instantiating and Configuring a ComboPooledDataSource

Perhaps the most straightforward way to create a c3p0 pooling DataSource is to instantiate an instance of `com.mchange.v2.c3p0.ComboPooledDataSource`. This is a JavaBean-style class with a public, no-arg constructor, but before you use the DataSource, you'll have to be sure to set at least the property `jdbcUrl`. You may also want to set `user` and `password`, and, if you use an old-style JDBC driver that you will not externally preload, you should set the `driverClass`.

```
ComboPooledDataSource cpds = new ComboPooledDataSource();
cpds.setDriverClass( "org.postgresql.Driver" ); //loads the jdbc driver
cpds.setJdbcUrl( "jdbc:postgresql://localhost/testdb" );
```

```
cpds.setUser("swaldman");
cpds.setPassword("test-password");

// the settings below are optional -- c3p0 can work with defaults
cpds.setMinPoolSize(5);
cpds.setAcquireIncrement(5);
cpds.setMaxPoolSize(20);

// The DataSource cpds is now a fully configured and usable pooled DataSource

...
```

The default values of any c3p0 DataSource property are determined by configuration you supply (e.g. `c3p0.properties`), or else revert to hard-coded defaults [see configuration properties]. c3p0 supports named configurations so that you can configure multiple DataSources. If you wish to use a named configuration, construct your `com.mchange.v2.c3p0.ComboPooledDataSource` with the configuration name as a constructor agument:

```
ComboPooledDataSource cpds = new ComboPooledDataSource("intergalactoApp");
```

Of course, you can still override any configuration properties programmatically, as above.

Using the DataSources factory class

⇑

Alternatively, you can use the static factory class `com.mchange.v2.c3p0.DataSources` to build unpooled DataSources from traditional JDBC drivers, and to build pooled DataSources from unpooled DataSources:

```
DataSource ds_unpooled = DataSources.unpooledDataSource("jdbc:postgresql://localhost/testdb",
                                                        "swaldman",
                                                        "test-password");
DataSource ds_pooled = DataSources.pooledDataSource( ds_unpooled );

// The DataSource ds_pooled is now a fully configured and usable pooled DataSource.
// The DataSource is using a default pool configuration, and Postgres' JDBC driver
// is presumed to have already been loaded via the jdbc.drivers system property or an
// explicit call to Class.forName("org.postgresql.Driver") elsewhere.

...
```

If you use the DataSources factory class, and you want to programmatically override default configuration parameters, you can supply a map of override properties:

```
DataSource ds_unpooled = DataSources.unpooledDataSource("jdbc:postgresql://localhost/testdb",
                                                        "swaldman",
                                                        "test-password");

Map overrides = new HashMap();
overrides.put("maxStatements", "200");         //Stringified property values work
overrides.put("maxPoolSize", new Integer(50)); //"boxed primitives" also work

// create the PooledDataSource using the default configuration and our overrides
ds_pooled = DataSources.pooledDataSource( ds_unpooled, overrides );

// The DataSource ds_pooled is now a fully configured and usable pooled DataSource,
// with Statement caching enabled for a maximum of up to 200 statements and a maximum
// of 50 Connections.

...
```

If you are using named configurations, you can specify the configuration that defines the default configuration for your DataSource:

```
// create the PooledDataSource using the a named configuration and specified overrides
// "intergalactoApp" is a named configuration
ds_pooled = DataSources.pooledDataSource( ds_unpooled, "intergalactoApp", overrides );
```

RARE: Forcing authentication information, regardless of (mis)configuration of the underlying (unpooled) DataSource

You can wrap any DataSouce using `DataSource.pooledDataSource( ... )`, usually with no problem whatsoever. DataSources are supposed to indicate the username and password associated by default with Connections via standard properties `user` and `password`. Some DataSource implementations do not offer these properties. Usually this is not at all a problem. `c3p0` works around this by acquiring "default" Connections from the DataSource if it can't find default authentication information, and a client has not specified the authentification information via `getConnection( user, password )`.

However, in rare circumstances, non-c3p0 unpooled DataSources provide a `user` property, but not a password property, or you have access to a DataSource that you wish to wrap behind a pool, but you wish to override its build-in authentification defaults without actually modifying the `user` or `password` properties.

`c3p0` provides configuation properties `overrideDefaultUser` and `overrideDefaultPassword`. If you set these properties, programmatically as above, or via any of c3p0's configuration mechanisms, `c3p0` PooledDataSources will ignore the user and password property associated with the underlying DataSource, and use the specified overrides instead.

Querying a PooledDataSource's current status

⇑

c3p0 DataSources backed by a pool, which include implementations of `ComboPooledDataSource` and the objects returned by `DataSources.pooledDataSource( ... )`, all implement the interface `com.mchange.v2.c3p0.PooledDataSource`, which makes available a number of methods for querying the status of DataSource Connection pools. Below is sample code that queries a DataSource for its status:

```
// fetch a JNDI-bound DataSource
InitialContext ictx = new InitialContext();
DataSource ds = (DataSource) ictx.lookup( "java:comp/env/jdbc/myDataSource" );

// make sure it's a c3p0 PooledDataSource
if ( ds instanceof PooledDataSource)
{
  PooledDataSource pds = (PooledDataSource) ds;
  System.err.println("num_connections: "      + pds.getNumConnectionsDefaultUser());
  System.err.println("num_busy_connections: " + pds.getNumBusyConnectionsDefaultUser());
  System.err.println("num_idle_connections: " + pds.getNumIdleConnectionsDefaultUser());
  System.err.println();
}
else
  System.err.println("Not a c3p0 PooledDataSource!");
```

The status querying methods all come in three overloaded forms, such as:

- `public int getNumConnectionsDefaultUser()`
- `public int getNumConnections(String username, String password)`
- `public int getNumConnectionsAllUsers()`

c3p0 maintains separate pools for Connections with distinct authentications. The various methods let you query the status of pools individually, or aggregate statistics for all authentifications for which your DataSource is maintaining pools. *Note that pool configuration parmeters such as `maxPoolSize` are enforced on a per-authentification basis!* For example, if you have set `maxPoolSize` to 20, and if the DataSource is managing connections under two username-password pairs [the default, and one other pair established via a call to `getConnection(user, password)`, you should expect to see as many as 40 Connections from `getNumConnectionsAllUsers()`.

Most applications only acquire default-authenticated Connections from DataSources, and can typically just use the `getXXXDefaultUser()` to gather Connection statistics.

As well as Connection pool related statistics, you can retrieve status information about each DataSource's Thread pool. Please see `PooledDataSource` for a complete list of available operations.

Using C3P0Registry to get a reference to a DataSource

If it's inconvenient or impossible to get a reference to your DataSource via JNDI or some other means, you can find all live c3p0 DataSources using the `C3P0Registry` class, which includes three static methods to help you out:

- `public static Set getPooledDataSources()`
- `public static Set pooledDataSourcesByName( String dataSourceName )`
- `public static PooledDataSource pooledDataSourceByName( String dataSourceName )`

The first method will hand you the Set of all live c3p0 PooledDataSources. If you are sure your application only makes only one PooledDataSources/a>, or you can distinguish between the DataSources by their configuration properties (inspected via "getters"), the first method may be sufficient. Because this will not always be the case, c3p0 PooledDataSources have a special property called `dataSourceName`. You can set the `dataSourceName` property directly when you construct your DataSource, or dataSourceName can be set like any other property in a named or the default config. Otherwise, `dataSourceName` will default to either 1) the name of your DataSource's configuration, if you constructed it with a named configuration; or 2) a unique (but unpredictable) name if you are using the default configuration.

There is no guarantee that a `dataSourceName` will be unique. For example, if two c3p0 DataSources share the same named configuration, and you have not set the `dataSourceName` programmatically, the two data sources will both share the name of the configuration. To get all of the DataSources with a particular `dataSourceName`, use `pooledDataSourcesByName( ... )`. If you've ensured that your DataSource's name is unique (as you will generally want to do, if you intend to use `C3P0Registry` to find your DataSources), you can use the convenience method `pooledDataSourceByName( ... )`, which will return your DataSource directly, or `null` if no DataSource with that name is available. If you use `pooledDataSourceByName( ... )` and more than one DataSource shares the name supplied, which one it will return is undefined.

Cleaning up after c3p0 PooledDataSources

⇑

The easy way to clean up after c3p0-created DataSources is to use the static destroy method defined by the class DataSources. Only `PooledDataSource`s need to be cleaned up, but DataSources.destroy(...) does no harm if it is called on an unpooled or non-c3p0 DataSource.

```
DataSource ds_pooled   = null;
```

```
try
{
  DataSource ds_unpooled = DataSources.unpooledDataSource("jdbc:postgresql://localhost/testdb",
                                                          "swaldman",
                                                          "test-password");
  ds_pooled = DataSources.pooledDataSource( ds_unpooled );

  // do all kinds of stuff with that sweet pooled DataSource...
}
finally
{
  DataSources.destroy( ds_pooled );
}
```

Alternatively, c3p0's PooledDataSource interface contains a `close()` method that you can call when you know you are finished with a DataSource. So, you can cast a c3p0 derived DataSource to a `PooledDataSource` and close it:

```
static void cleanup(DataSource ds) throws SQLException
{
  // make sure it's a c3p0 PooledDataSource
  if ( ds instanceof PooledDataSource)
  {
    PooledDataSource pds = (PooledDataSource) ds;
    pds.close();
  }
  else
    System.err.println("Not a c3p0 PooledDataSource!");
}
```

ComboPooledDataSource is an instance of PooledDataSource, and can be closed directly via its `close()` method. `PooledDataSource` implements `java.lang.AutoCloseable`, so they may be managed by Java 7+ try-with-resources blocks.

Advanced: Building your own PoolBackedDataSource

There is little reason for most programmers to do this, but you can build a PooledDataSource in a step-by-step way by instantiating and configuring an unpooled DriverManagerDataSource, instantiating a WrapperConnectionPoolDataSource and setting the unpooled DataSource as its `nestedDataSource` property, and then using that to set the `connectionPoolDataSource` property of a new PoolBackedDataSource.

This sequence of events is primarily interesting if your driver offers an implementation of ConnectionPoolDataSource, and you'd like c3p0 to use that. Rather than using c3p0's WrapperConnectionPoolDataSource, you can create a PoolBackedDataSource and set its `connectionPoolDataSource` property. Statement pooling, ConnectionCustomizers, and many c3p0-specific properties are unsupported with third party implementations of `ConnectionPoolDataSource`. (Third-party `DataSource` implementations can be substituted for c3p0's DriverManagerDataSource with no significant loss of functionality.)

Advanced: Raw Connection and Statement Operations

Note: c3p0 version 0.9.5 and above supports the standard JDBC4 `unwrap()` methods for looking through proxies. Note that if you use the `unwrap()` methods, c3p0 can not clean up any `Statement` or `ResultSet` objects you may generate from raw Connections or Statements. Users must take care to clean up these objects directly. Further, users should take care not to modify underlying `Connection`s in some manner that would render them no longer interchangeable with other `Connection`s, as they must be to remain suitable for pooling.

JDBC drivers sometimes define vendor-specific, non-standard API on Connection and Statement implementations. C3P0 wraps these Objects behind a proxies, so you cannot cast C3P0-returned Connections or Statements to the vendor-specific implementation classes. C3P0 does not provide any means of accessing the raw Connections and Statements directly, because C3P0 needs to keep track of Statements and ResultSets created in order to prevent resource leaks and pool corruption.

C3P0 does provide an API that allows you to invoke non-standard methods reflectively on an underlying Connection. To use it, first cast the returned Connection to a C3P0ProxyConnection. Then call the method `rawConnectionOperation`, supplying the `java.lang.reflect.Method` object for the non-standard method you wish to call as an argument. The `Method` you supply will be invoked on the target you provide on the second argument (null for static methods), and using the arguments you supply in the third argument to that function. For the target, and for any of the method arguments, you can supply the special token `C3P0ProxyConnection.RAW_CONNECTION`, which will be replaced with the underlying vendor-specific Connection object before the `Method` is invoked.

C3P0ProxyStatement offers an exactly analogous API.

Any Statements (including Prepared and CallableStatements) and ResultSets returned by raw operations will be c3p0-managed, and will be properly cleaned-up on `close()` of the parent proxy Connection. Users must take care to clean up any non-standard resources returned by a vendor-specific method.

Here's an example of using Oracle-specific API to call a static method on a raw Connection:

```
C3P0ProxyConnection castCon = (C3P0ProxyConnection) c3p0DataSource.getConnection();
Method m = CLOB.class.getMethod("createTemporary", new Class[]{Connection.class, boolean.class, int.class});
Object[] args = new Object[] {C3P0ProxyConnection.RAW_CONNECTION, Boolean.valueOf( true ), new Integer( 10 )};
```

```
CLOB oracleCLOB = (CLOB) castCon.rawConnectionOperation(m, null, args);
```

Configuration

Introduction

While c3p0 does not *require* very much configuration, it is very, very tweakable. Most of the interesting knobs and dials are represented as JavaBean properties. Following JavaBean conventions, we note that if an Object has a property of type `T` called `foo`, it will have methods that look like...

```
public T getFoo();
public void setFoo(T foo);
```

...or both, depending upon whether the property is read-only, write-only, or read-writable.

There are several ways to modify c3p0 properties: You can directly alter the property values associated with a particular DataSource in your code, or you can configure c3p0 externally...

- via a simple Java properties file
- via HOCON (typesafe-config) files (if and only if you bundle the typesafe-config library with your application)
- via an XML configuration file
- via System properties

Configuration files are normally looked up under standard names (`c3p0.properties` or `c3p0-config.xml` or HOCON variations) at the top level of an application's classpath. The XML configuration may be placed elsewhere in an application's file system or classpath, if the system property `com.mchange.v2.c3p0.cfg.xml` is set.

Warning: c3p0 trusts its CLASSPATH and configuration!

c3p0 trusts the information it discovers from any of the above sources.

The library presumes that any `c3p0.properties`, `c3p0-config.xml`, or HOCON variations (`application.{conf|properties|json}`, `reference.{conf|properties|json}`, `c3p0.{conf|properties|json}`) discovered in the `CLASSPATH` are safe and trustworthy. System properties, which can override most config, are also presumed secure.

In general, c3p0 requires the application `CLASSPATH` to be secure. If it is not, an adversary could interfere with internal c3p0 resources, not to mention class bytecode, and cause a wide variety of mischief.

A reviewer has noted that an adversary that controls c3p0 configuration properties could perform SQL injection attacks via legacy configuration parameters `automaticTestTable` and `preferredTestQuery`.

c3p0 now adds a guard to reject dangerous values of `automaticTestTable`, but there is no suitable guard for `preferredTestQuery`. Creative and unusual values of `preferredTestQuery`, including calls to stored procedures, are sometimes desirable.

An adversary that can 1) add classes to the application `CLASSPATH` and 2) redefine any of config parameters `connectionCustomizerClassName`, `connectionTesterClassName`, `taskRunnerFactoryClassName`, or `com.mchange.v2.naming.nameGuardClassName` can cause c3p0 to execute whatever arbitrary code a malicious plug-in implementation might supply.

c3p0 trusts its configuration — in its `CLASSPATH`, its System properties, and in any XML file located at System property `com.mchange.v2.c3p0.cfg.xml`. If an adversary compromises and can alter that configuration, c3p0 will not be secure.

DataSources are usually configured before they are used, either during or immediately following their construction. c3p0 does support property modifications midstream, however.

If you obtain a DataSource by instantiating a `ComboPooledDataSource`, configure it by simply calling appropriate setter methods offered by that class before attempting a call to `getConnection()`. See the example above.

If you obtain a DataSource by using factory methods of the utility class `com.mchange.v2.c3p0.DataSources`, and wish to use a non-default configuration, you can supply a Map of property names (beginning with lower-case letters) to property values (either as Strings or "boxed" Java primitives like Integer or Boolean).

You can also set multiple configuration properties from a Map after your DataSource has been constructed. Use `DataSources.overwriteJavaBeanProperties(...)`.

After a DataSource has been constructed, you can overwrite multiple configuration properties from a `Properties` object, containing c3p0.properties-like c3p0-prefixed configuration properties. This is useful for allowing applications to take c3p0 configuration from nonstandard sources. So if an application has properties-based configuration like

```
myapp.title=It's My App
myapp.api.port=9245
myapp.twistiness=severe
mail.smtp.user=admin@myapp.com
mail.smtp.password=big.secret.09
c3p0.jdbcUrl=jdbc:postgresql://localhost:5432/myapp
c3p0.user=myapp
c3p0.password=please.dont.tell
```

```
c3p0.maxPoolSize=20
```

You could load the properties object and call `DataSources.overwriteC3P0PrefixedProperties(...)`.

The properties not prefixed with `c3p0.` would be ignored, while properties `jdbcUrl`, `user`, `password`, and `maxPoolSode` would be overwritten by this configuration.

All tweakable properties are documented for reference in Appendix A. Most c3p0 configuration topics are discussed in detail below.

Basic Pool Configuration

c3p0 Connection pools are very easy to configure via the following basic parameters:

- `acquireIncrement`
- `initialPoolSize`
- `maxPoolSize`
- `maxIdleTime`
- `minPoolSize`

`initialPoolSize`, **`minPoolSize`**, **`maxPoolSize`** define the number of Connections that will be pooled. Please ensure that `minPoolSize <= maxPoolSize`. Unreasonable values of **`initialPoolSize`** will be ignored, and **`minPoolSize`** will be used instead.

Within the range between **`minPoolSize`** and **`maxPoolSize`**, the number of Connections in a pool varies according to usage patterns. The number of Connections increases whenever a Connection is requested by a user, no Connections are available, and the pool has not yet reached **`maxPoolSize`** in the number of Connections managed.

Since Connection acquisition is very slow, it is almost always useful to increase the number of Connections eagerly, in batches, rather than forcing each client to wait for a new Connection to provoke a single acquisition when the load is increasing. **`acquireIncrement`** determines how many Connections a c3p0 pool will attempt to acquire when the pool has run out of Connections. (Regardless of **`acquireIncrement`**, the pool will never allow **`maxPoolSize`** to be exceeded.)

The number of Connections in a pool decreases whenever a pool tests a Connection and finds it to be broken (see Configuring Connection Testing below), or when a Connection is expired by the pool after sitting idle for a period of time, or for being too old (See Managing Pool Size and Connection Age.)

Managing Pool Size and Connection Age

Different applications have different needs with regard to trade-offs between performance, footprint, and reliability. C3P0 offers a wide variety of options for controlling how quickly pools that have grown large under load revert to `minPoolSize`, and whether "old" Connections in the pool should be proactively replaced to maintain their reliablity.

- `maxConnectionAge`
- `maxIdleTime`
- `maxIdleTimeExcessConnections`

By default, pools will never expire Connections. If you wish Connections to be expired over time in order to maintain "freshness", set **`maxIdleTime`** and/or **`maxConnectionAge`**. **`maxIdleTime`** defines how many seconds a Connection should be permitted to go unused before being culled from the pool. **`maxConnectionAge`** forces the pool to cull any Connections that were acquired from the database more than the set number of seconds in the past.

`maxIdleTimeExcessConnections` is about minimizing the number of Connections held by c3p0 pools when the pool is not under load. By default, c3p0 pools grow under load, but only shrink if Connections fail a Connection test or are expired away via the parameters described above. Some users want their pools to quickly release unnecessary Connections after a spike in usage that forces a large pool size. You can achieve this by setting **`maxIdleTimeExcessConnections`** to a value much shorter than **`maxIdleTime`**, forcing Connections beyond your set minimum size to be released if they sit idle for more than a short period of time.

Some general advice about all of these timeout parameters: Slow down! The point of Connection pooling is to bear the cost of acquiring a Connection only once, and then to reuse the Connection many, many times. Most databases support Connections that remain open for hours at a time. There's no need to churn through all your Connections every few seconds or minutes. Setting **`maxConnectionAge`** or **`maxIdleTime`** to 1800 (30 minutes) is quite aggressive. For most databases, several hours may be more appropriate. You can ensure the reliability of your Connections by testing them, rather than by tossing them. (see Configuring Connection Testing.) The only one of these parameters that should generally be set to a few minutes or less is **`maxIdleTimeExcessConnections`**.

Configuring Connection Testing

By default, c3p0 does no Connection testing at all. Usually you will want some Connection testing. In the modern world, most users need concern themselves with a very few, very simple Connection-testing related configuration parameters.

> *If you are not of the modern world — if you are using a Java 5 or older JDBC driver — then you may have to revert to c3p0's more complicated legacy config, including parameters like `automaticTestTable`, `connectionTesterClassName`, and `preferredTestQuery`. We* [shiver] *won't talks about that stuff here. Click the link, if you dare.*

c3p0 provides users a great deal of flexibility in testing Connections, via the following configuration parameters:

- connectionIsValidTimeout
- idleConnectionTestPeriod
- testConnectionOnCheckin
- testConnectionOnCheckout

idleConnectionTestPeriod, **testConnectionOnCheckout**, and **testConnectionOnCheckin** control when Connections will be tested. **connectionIsValidTimeout** affects how they will be tested.

The most reliable time to test Connections is on check-out. But this is also means clients experience a (usually very modest!) performance hit from the test.

Most applications should just set `testConnectionOnCheckout` to `true` and be done with it!

However, if you really wish to minimize client latency, most applications should work reliably using a combination of **idleConnectionTestPeriod** and **testConnectionOnCheckin**. Both the idle test and the check-in test are performed asynchronously, which can lead to better performance, both perceived and actual.

For some applications, high performance is more important than the risk of an occasional database exception. In its default configuration, c3p0 does no Connection testing at all. Setting a fairly long **idleConnectionTestPeriod**, and not testing on checkout and check-in at all is another high-performance approach.

Simple advice on Connection testing

If you don't know what to do, try this:

1. Begin by setting `testConnectionOnCheckout` to `true` and get your application to run correctly and stably. If you are happy with your application's performance, *you can stop here!* This is the simplest, most reliable form of Connection-testing, but it does have a client-visible performance cost.
2. If you'd like to improve performance by eliminating Connection testing from clients' code path:
    a. Set `testConnectionOnCheckout` to `false`
    b. Set `testConnectionOnCheckin` to `true`
    c. Set `idleConnectionTestPeriod` to `30`, fire up you application and observe. This is a pretty robust setting, all Connections will tested on check-in and every 30 seconds thereafter while in the pool. Your application should experience broken or stale Connections only very rarely, and the pool should recover from a database shutdown and restart quickly. But there is some overhead associated with all that Connection testing.
    d. If database restarts will be rare so quick recovery is not an issue, consider reducing the frequency of tests by `idleConnectionTestPeriod` to, say, `300`, and see whether clients are troubled by stale or broken Connections. If not, stick with `300`, or try an even bigger number. Consider setting `testConnectionOnCheckin` back to false to avoid unnecessary tests on checkin. Alternatively, if your application does encounter bad Connections, consider reducing `idleConnectionTestPeriod` and set `testConnectionOnCheckin` back to `true`. There are no correct or incorrect values for these parameters: you are trading off overhead for reliability in deciding how frequently to test. The exact numbers are not so critical. It's usually easy to find configurations that perform well. It's rarely worth spending time in pursuit of "optimal" values here.

So, when should you stick with simple and reliable (Step 1 above), and when is it worth going for better performance (Step 2)? In general, it depends on how much work clients typically do with Connections once they check them out. If clients usually make complex queries and/or perform multiple operations, adding the extra cost of one fast test per checkout will not much affect performance. But if your application typically checks out a Connection and performs one simple query with it, throwing in an additional test can really slow things down.

That's nice in theory, but often people don't really have a good sense of how much work clients perform on average. You can give Step 2 a try, see if it helps (however you measure performance), see if it hurts (is your application troubled by broken Connections? does it recover from database restarts well enough?), and then decide. You can always go back to simple, slow, and robust. Just set `testConnectionOnCheckout` to `true`, `testConnectionOnCheckin` to `false`, and set `idleConnectionTestPeriod` to `0`.

Configuring Statement Pooling

⇑

c3p0 implements transparent PreparedStatement pooling as defined by the JDBC spec. Under some circumstances, statement pooling can dramatically improve application performance. Under other circumstances, the overhead of statement pooling can slightly harm performance. Whether and how much statement pooling will help depends on how much parsing, planning, and optimizing of queries your databases does when the statements are prepared. Databases (and JDBC drivers) vary widely in this respect. It's a good idea to benchmark your application with and without statement pooling to see if and how much it helps.

You configure statement pooling in c3p0 via the following configuration parameters:

- maxStatements
- maxStatementsPerConnection
- statementCacheNumDeferredCloseThreads

maxStatements is JDBC's standard parameter for controlling statement pooling. **maxStatements** defines the total number `PreparedStatements` a DataSource will cache. The pool will destroy the least-recently-used PreparedStatement when it hits this limit. This sounds simple, but it's actually a strange approach, because cached statements conceptually belong to individual Connections; they are not global resources. To figure out a size for **maxStatements** that does not "churn" cached statements, you need to consider the number of *frequently used* PreparedStatements in your application, and multiply that by the number of Connections you expect in the pool (`maxPoolSize` in a busy application).

maxStatementsPerConnection is a non-standard configuration parameter that makes a bit more sense conceptually. It defines how many statements each pooled Connection is allowed to own. You can set this to a bit more than the number of `PreparedStatements` your application *frequently* uses, to avoid churning.

If either of these parameters are greater than zero, statement pooling will be enabled. If both parameters are greater than zero, both limits will be enforced. If only one is greater than zero, statement pooling will be enabled, but only one limit will be enforced.

If **`statementCacheNumDeferredCloseThreads`** is greater than zero, the Statement pool will defer physically close()ing cached Statements until its parent Connection is not in use by any client or internally (in e.g. a test) by the pool itself. For some JDBC drivers (especially Oracle), attempts to close a Statement freeze if the parent Connection is in use. This parameter defaults to 0. Set it to a positive value if you observe "APPARENT DEADLOCKS" related to Connection close tasks. If you are using custom threading rather than c3p0's default Thread pool, look for other evidence of Connection close tasks freezing.

Almost always, the value of **`statementCacheNumDeferredCloseThreads`** should be either its default value of zero, or just one. If you need more than one Thread dedicated solely to Statement destruction, you probably should set **`maxStatements`** and/or **`maxStatementsPerConnection`** to higher values so you don't churn through cached Statements so quickly.

Note: If you enable statement caching and see log mesages like "Problem with checked-in Statement, discarding.", and then stack traces indicating a Statement is unexpectedly closed, your JDBC driver may be automatically closing unclosed Statements at the end of marked user sessions. This behavior interferes with Statement caching. Try setting markSessionBoundaries to `if-no-statement-cache` or `never` to address the problem.

Configuring Recovery From Database Outages

c3p0 DataSources are designed (and configured by default) to recover from temporary database outages, such as those which occur during a database restart or brief loss of network connectivity. You can affect how c3p0 handles errors in acquiring Connections via the following configurable properties:

- acquireRetryAttempts
- acquireRetryDelay
- breakAfterAcquireFailure

When a c3p0 DataSource attempts and fails to acquire a Connection, it will retry up to **`acquireRetryAttempts`** times, with a delay of **`acquireRetryDelay`** between each attempt. If all attempts fail, any clients waiting for Connections from the DataSource will see an Exception, indicating that a Connection could not be acquired. Note that clients do not see any Exception until a full round of attempts fail, which may be some time after the initial Connection attempt. If **`acquireRetryAttempts`** is set to a value less than 0, c3p0 will attempt to acquire new Connections indefinitely, and calls to `getConnection()` may block indefinitely waiting for a successful acquisition.

Once a full round of acquisition attempts fails, there are two possible policies. By default, the c3p0 DataSource will remain active, and will try again to acquire Connections in response to future requests for Connections. If you set **`breakAfterAcquireFailure`** to `true`, the DataSource will consider itself broken after a failed round of Connection attempts, and future client requests will fail immediately.

Note that if a database restart occurs, a pool may contain previously acquired but now stale Connections. By default, these stale Connections will only be detected and purged lazily, when an application attempts to use them, and sees an Exception. Setting `maxIdleTime` or `maxConnectionAge` can help speed up the replacement of broken Connections. (See Managing ConnectionAge.) If you wish to avoid application Exceptions entirely, you must adopt a connection testing strategy that is likely to detect stale Connections prior to their delivery to clients. (See "Configuring Connection Testing".) Even with active Connection testing (`testConnectionOnCheckout` set to `true`, or `testConnectionOnCheckin` and a short `idleConnectionTestPeriod`), your application may see occasional Exceptions on database restart, for example if the restart occurs after a Connection to the database has already been checked out.

Managing Connection Lifecycles with Connection Customizer

Application frequently wish to set up Connections in some standard, reusable way immediately after Connection acquisitions. Examples of this include setting-up character encodings, or date and time related behavior, using vendor-specific APIs or non-standard SQL statement executions. Occasionally it is useful to override the default values of standard Connection properties such as `transactionIsolation`, `holdability`, or `readOnly`. c3p0 provides a "hook" interface that you can implement, which gives you the opportunity to modify or track Connections just after they are checked out from the database, immediately just prior to being handed to clients on checkout, just prior to being returned to the pool on check-in, and just prior to final destruction by the pool. The Connections handed to ConnectionCustomizers are raw, physical Connections, with all vendor-specific API accessible. See the API docs for `ConnectionCustomizer`.

To install a `ConnectionCustomizer` just implement the interface, make your class accessible to c3p0's ClassLoader, and set the configuration parameter below:

- connectionCustomizerClassName

ConnectionCustomizers are required to be immutable classes with public no argument constructors. They shouldn't store any state. For (rare) applications that wish to track the behavior of individual DataSources with ConnectionCustomizers, the lifecycle methods each accept a DataSource-specific "identityToken", which is unique to each PooledDataSource. ConnectionCustomizers can be configured via user-defined configuration extensions.

Below is a sample `ConnectionCustomizer`. Implementations that do not need to override all four `ConnectionCustomizer` methods can extend `AbstractConnectionCustomizer` to inherit no-op implementations of all methods.

```
import com.mchange.v2.c3p0.*;
import java.sql.Connection;

public class VerboseConnectionCustomizer
{
    public void onAcquire( Connection c, String pdsIdt )
    {
       System.err.println("Acquired " + c + " [" + pdsIdt + "]");

       // override the default transaction isolation of
       // newly acquired Connections
```

```
        c.setTransactionIsolation( Connection.REPEATABLE_READ );
    }

    public void onDestroy( Connection c, String pdsIdt )
    { System.err.println("Destroying " + c + " [" + pdsIdt + "]"); }

    public void onCheckOut( Connection c, String pdsIdt )
    { System.err.println("Checked out " + c + " [" + pdsIdt + "]"); }

    public void onCheckIn( Connection c, String pdsIdt )
    { System.err.println("Checking in " + c + " [" + pdsIdt + "]"); }
}
```

For an example `ConnectionCustomizer` that employs user-defined configuration properties, see below.

Configuring Unresolved Transaction Handling

Connections checked into a pool cannot have any unresolved transactional work associated with them. If users have set `autoCommit` to `false` on a Connection, and c3p0 cannot guarantee that there is no pending transactional work, c3p0 must either `rollback()` or `commit()` on check-in (when a user calls `close()`). The JDBC spec is (unforgivably) silent on the question of whether unresolved work should be committed or rolled back on Connection close. By default, c3p0 rolls back unresolved transactional work when a user calls `close()`.

You can adjust this behavior via the following configuration properties:

- `autoCommitOnClose`
- `forceIgnoreUnresolvedTransactions`

If you wish c3p0 to allow unresolved transactional work to commit on checkin, set **`autoCommitOnClose`** to true. If you wish c3p0 to leave transaction management to you, and neither commit nor rollback (nor modify the state of Connection `autoCommit`), you may set **`forceIgnoreUnresolvedTransactions`** to true. Setting **`forceIgnoreUnresolvedTransactions`** is strongly discouraged, because if clients are not careful to commit or rollback themselves prior to close(), or do not set Connection `autoCommit` consistently, bizarre unreproduceable behavior and database lockups can occur.

Configuring to Debug and Workaround Broken Client Applications

Sometimes client applications are sloppy about close()ing all Connections they check out. Eventually, the pool grows to `maxPoolSize`, and then runs out of Connections, because of these bad clients.

The right way to address this problem is to fix the client application. `c3p0` can help you debug, by letting you know where Connections are checked out that occasionally don't get checked in. In rare and unfortunate situations, development of the client application is closed, and even though it is buggy, you cannot fix it. c3p0 can help you work around the broken application, preventing it from exhausting the pool.

The following parameters can help you debug or workaround broken client applications.

- `debugUnreturnedConnectionStackTraces`
- `unreturnedConnectionTimeout`

`unreturnedConnectionTimeout` defines a limit (in seconds) to how long a Connection may remain checked out. If set to a nozero value, unreturned, checked-out Connections that exceed this limit will be summarily destroyed, and then replaced in the pool. Obviously, you must take care to set this parameter to a value large enough that all intended operations on checked out Connections have time to complete. You can use this parameter to merely workaround unreliable client apps that fail to close() Connections.

Much better than working-around is fixing. If, *in addition to setting* `unreturnedConnectionTimeout`, you set **`debugUnreturnedConnectionStackTraces`** to `true`, then a stack trace will be captured each time a Connection is checked-out. Whenever an unreturned Connection times out, that stack trace will be logged, revealing where a Connection was checked out that was not checked in promptly. **`debugUnreturnedConnectionStackTraces`** is intended to be used only for debugging, as capturing a stack trace can slow down Connection check-out.

Configuring To Avoid Memory Leaks On Hot Redeploy Of Clients

c3p0 spawns a variety of Threads (helper threads, `java.util.Timer` threads), and does so lazily in response to the first client request experienced by a PooledDataSource. By default, the Threads spawned by c3p0 inherit a `java.security.AccessControlContext` and a `contextClassLoader` property from this first-calling Thread. If that Thread came from a client that may need to be hot-undeployed, references to these objects may prevent the undeployed application, often partitioned into a `ClassLoader`, from being garbage collected. (See for example this description of Tomcat memory leaks on redeployment.)

c3p0 provides two configuration parameters that can help with this:

- `contextClassLoaderSource`
- `privilegeSpawnedThreads`

`contextClassLoaderSource` should be set to one of `caller`, `library`, or `none`. The default (which yields the default behavior described above) is `caller`. Set this to `library` to use c3p0's ClassLoader, so that no reference is maintained to a client that may need to be redeployed.

`privilegeSpawnedThreads` is a boolean, `false` by default. Set this to `true` so that c3p0's Threads use the the c3p0 library's `AccessControlContext`, rather than an `AccessControlContext` that may be associated with the client application and prevent its garbage collection.

Note: If users take control over c3p0's threading by providing a non-default `taskRunnerFactoryClassName`, these properties may or may not be honored. See the documentation or the implementation of your `TaskRunnerFactory`.

Configuring threading

See Appendix A for information about the following configuration properties:

- maxAdministrativeTaskTime
- numHelperThreads
- taskRunnerFactoryClassName

numHelperThreads and **maxAdministrativeTaskTime** help to configure the behavior of DataSource thread pools. By default, each DataSource has only three associated helper threads. If performance seems to drag under heavy load, or if you observe via JMX or direct inspection of a PooledDataSource, that the number of "pending tasks" is usually greater than zero, try increasing **numHelperThreads**. **maxAdministrativeTaskTime** may be useful for users experiencing tasks that hang indefinitely and "APPARENT DEADLOCK" messages. (See Appendix A for more.)

Alternatively, you can take full control over what kind of threading or thread pool your c3p0 DataSource uses, by providing a non-default value for taskRunnerFactoryClassName. This should be the fully qualified class name of an immutable class with a public no-arg constructor, which implements the interface TaskRunnerFactory. It becomes up to the implementor of this class whether and how **numHelperThreads**, **maxAdministrativeTaskTime**, **contextClassLoaderSource**, and **privilegeSpawnedThreads** are supported.

Most users will want to leave **taskRunnerFactoryClassName** at c3p0's default, or use else published alternative implementations. Out of the box c3p0 includes

- com.mchange.v2.c3p0.impl.DefaultTaskRunnerFactory
- com.mchange.v2.c3p0.FixedThreadPoolExecutorTaskRunnerFactory

If you have brought in support for Java 21 "loom" virtual threads, you can also use

- com.mchange.v2.c3p0.loom.UninstrumentedVirtualThreadPerTaskTaskRunnerFactory
- com.mchange.v2.c3p0.loom.VirtualThreadPerTaskExecutorTaskRunnerFactory

But of course you can roll your own! Implement the interface TaskRunnerFactory with an immutable class with a public, no-arg constructor, put it in your application's CLASSPATH, and supply its fully qualified name as **taskRunnerFactoryClassName**.

c3p0 includes special support for delegating threading to a java.util.concurrent.Executor. Your PooledDataSource may "own" the Executor, so it will be shut down when your DataSource is closed, or may share (not own) it, in which case the lifecycle of the Executor will be independent of the lifecycle of your DataSource. Just extend the abstract class AbstractExecutorTaskRunnerFactory. A useful utility will be TaskRunnerThreadFactory, an implementation of java.util.concurrent.ThreadFactory that handles support of contextClassLoaderSource and privilegeSpawnedThreads.

AbstractExecutorTaskRunnerFactory defers its "findCreate" of its executor until it receives its first asynchronous request. You can safely stand-up a shared Executor in your application initialization, then have your implementation of Executor findCreateExecutor(TaskRunnerInit init) look up it up after application start. If you are using a shared executor, just be sure your override of boolean taskRunnerOwnsExecutor() returns false.

com.mchange.v2.c3p0.FixedThreadPoolExecutorTaskRunnerFactory provides a a decent example implementation of AbstractExecutorTaskRunnerFactory. It provides its own Executor, rather than look up a shared pool.

```
public final class FixedThreadPoolExecutorTaskRunnerFactory extends AbstractExecutorTaskRunnerFactory
{
    // for lazy initialization, called only on first-use
    protected Executor findCreateExecutor( TaskRunnerInit init )
    {
        ThreadFactory tf = new TaskRunnerThreadFactory( init.contextClassLoaderSourceIfSupported, init.privilege_spawned_threads_if_supported, init.threadLa
        return Executors.newFixedThreadPool( init.num_threads_if_supported, tf );
    }

    protected boolean taskRunnerOwnsExecutor() { return true; }

    protected ThreadPoolReportingAsynchronousRunner createTaskRunner( TaskRunnerInit init, Timer timer )
    { return new FixedThreadPoolExecutorAsynchronousRunner( init, timer ); }

    protected final class FixedThreadPoolExecutorAsynchronousRunner extends AbstractExecutorAsynchronousRunner
    {
        protected FixedThreadPoolExecutorAsynchronousRunner( TaskRunnerInit init, Timer timer )
        { super( init, timer ); }

        public int getThreadCount() { return init.num_threads_if_supported; }
        public int getIdleCount()   { return getThreadCount() - getActiveCount(); }
    }
}
```

See also com.mchange.v2.c3p0.loom.VirtualThreadPerTaskExecutorTaskRunnerFactory.

Use of c3p0-loom's `UninstrumentedVirtualThreadPerTaskTaskRunnerFactory` can reduce monitor contention

c3p0's asynchrony is old-school, built around object monitors ("locks") and synchronized blocks, Object.wait() and Object.notifyAll.

c3p0 is extremely careful to ensure that Threads never block while holding a lock, that locking is reserved for fast in-memory applications. Nevertheless, under very high concurrent load, Threads will occasionally block to await monitor release.

For all the excitement about newer forms of lockless concurrency, every concurrency management scheme imposes overhead for managing shared access to mutable state, and the actual time cost of lock contention under c3p0 is usually negligible when amortized over the total number of `contending_threads x client_requests`.

Nevertheless.

By far the most contended lock under c3p0 is the `BasicResourcePool`, the actual Connection pool that tracks Connections, whether they're checked out, how long they've been idle, etc. The only way to reduce contention for this resource is to balance your load over multiple pools.

The next most contended resource is the Thread pool. c3p0's default Thread pool, `com.mchange.v2.async.ThreadPoolAsynchronousRunner`, provides very rich instrumentation (via JMX) and great information in your logs about deadlocks (usually due to database locking, pool exhaustion, or JDBC driver quirks), including full stack traces of active threads and the list of pending tasks.

But management of all that does lead to a small amount of monitor contention in practice. If you don't need any of that, replacing the threadpool with `com.mchange.v2.c3p0.loom.UninstrumentedVirtualThreadPerTaskTaskRunnerFactory` is an alternative that involves no locking or lock contention at all, beyond whatever happens in the infrastructure beneath Java 21 virtual threads.

Other DataSource Configuration

See Appendix A for information about the following configuration properties:

- `attemptResurrectOnCheckin`
- `cancelAutomaticallyClosedStatements`
- `checkoutTimeout`
- `factoryClassLocation`
- `forceSynchronousCheckins`
- `markSessionBoundaries`

When Connections throw Exceptions in client sessions, c3p0 tests them, and if they fail, marks them for destruction rather than return to the pool. If **`attemptResurrectOnCheckin`** is set to `true`, however, the library tests them once again at client check-in, and if that test succeeds, they are retained and reintegrated into the pool. This can prevent unnecessary churn through Connection, especially under databases (like Postgres) whose Connections may seem broken inside of invalidated transactions, but become functional again following a `rollback()` or failed attempt to `commit()`.

c3p0 automatically calls `close()` on Statements when (i) a parent Connection is closed (returned to the pool) while clients have failed to `close()` statements they have created or prepared and; (ii) when statement pooling is configured and the pool need to expire a cached statement (that from a client's perspective may have been closed), in order to conform to limits set by `maxStatements` and `maxStatementsPerConnection`. Sometimes case (i) will occur because a Statement has gone haywire and does not complete. Clients may eventually close the parent Connection, or an `unreturnedConnectionTimeout` may force the close. In this case, under some jdbc drivers and DBMSes, it may be desirable for the `cancel()` method of Statement to be called prior to the call to `close()`, to encourage the DBMS to kill the wayward query. This is the effect of setting **`cancelAutomaticallyClosedStatements`** to `true`. This is a course-grained setting: Statements that c3p0 closes may well not be in progress, because a client simply forgot to call `close()` or because the Statement cache is managing it. Some drivers may well throw an Exception in this case, and not all drivers even support `cancel()`. If **`cancelAutomaticallyClosedStatements`** is set to `true`, c3p0 will make a "best-effort" attempt to `cancel()` the Statement prior to destroying it with `close()`, and will catch and ignore any Exception that results from the attempted cancellation.

`checkoutTimeout` limits how long a client will wait for a Connection, if all Connections are checked out and one cannot be supplied immediately.

`factoryClassLocation` can be used to indicate where a URL from which c3p0 classes can be downloaded, if c3p0 DataSources will be retrieved as References from a JNDI DataSource by clients who do not have c3p0 locally installed. *Note: As of c3p0-0.12.0, remote classloading is now disabled by default.* See Configuring Security and com.mchange.v2.naming.supportReferenceRemoteFactoryClassLocation.

Ordinarily check-ins are performed asynchronously so that clients do not experience the overhead of on-check-in Connection tests and/or operations specified in `ConnectionCustomizer.onCheckIn(...)`. However, asynchronous checkins add to Thread pool congestion. Under loads so large that it is impractical to expand `numHelperThreads` to reduce congestion, you can set **`forceSynchronousCheckins`**. This cause client Threads to perform the checkin operations directly, reducing the load on the Thread pool and precluding any delays in termination of checkin due to Thread pool congestion. As long as you neither perform Connection tests on check-in (see `testConnectionOnCheckin`) nor perform database operations or other slow work in `ConnectionCustomizer.onCheckIn(...)`, this setting is likely to improve performance. However, if Connections *are* tested on check-in, or custom work is performed, setting **`forceSynchronousCheckins`** will cause clients to experience delays associated with that work when they call `Connection.close()`.

`markSessionBoundaries` can be used to disable c3p0's default behavior, which is to call the `beginRequest` method on checkout and `endRequest` method on check-in for drivers that support the JDBC 4.3 methods. These methods are hints to the JDBC driver, marking the beginning and end of a single client session. However, some drivers react to these hints in ways you may not want. In particular, some close unclosed statements at the end of a user-session, breaking c3p0-level `Statement` caching. Legal values for `markSessionBoundaries` are `always`, `never`, and `if-no-statement-cache`. The first two values are self-explanatory. The last will mark statement boundaries if and only if statement-caching is not configured, that is if both `maxStatements` and `maxStatementsPerConnection` are left at their default of zero.

Configuring and Managing c3p0 via JMX

If JMX libraries and a JMX MBeanServer are available in your environment (they are included in JDK 1.5 and above), you can inspect and configure your c3p0 datasources via a JMX administration tool (such as jconsole, bundled with jdk 1.5). You will find that c3p0 registers MBeans under the domain `com.mchange.v2.c3p0`, one with statistics about the library as a whole (called `C3P0Registry`), and an MBean for each `PooledDataSource` you deploy. You can view and modify your DataSource's configuration properties, track the activity of Connection, Statement, and Thread pools, and reset pools and DataSources via the `PooledDataSource` MBean. (You may wish to view the API docs of `PooledDataSource` for documentation of the available operations.)

Configuring JMX Names

Each `PooledDataSource` within your application may have the following attributes embedded within its `ObjectName`:

- type
- identityToken
- name

The type will always be PooledDataSource. The identityToken is a unique String associated with each c3p0 DataSource. The name will be the value of the property dataSourceName, which you can set yourself to ensure that semanically equivalent data sources are identifiable across application restarts. If you do not set a dataSourceName, the name attribute may not be defined at all, or it may take some default value.

For example, the following might be the full JMX ObjectName, in String format, of a c3p0 DataSource whose dataSourceName is intergalactoApp:

```
com.mchange.v2.c3p0:type=PooledDataSource,identityToken=2rvy139515ecj0rkwntk|16adc251,name=intergalactoApp
```

c3p0 prefers to include identity tokens in JMX ObjectNames to ensure that every ObjectName is unique. If you can, stick with c3p0's default behavior. But if you really need to, you can configure c3p0 to exclude the identityToken attribute from ObjectNames, so that your PooledDataSources have predictable, reproducible names. Set the following, as a System property, in c3p0.properties, or in HOCON config:

```
com.mchange.v2.c3p0.management.ExcludeIdentityToken=true
```

This will lead to names missing the long identity token, names like

```
com.mchange.v2.c3p0:type=PooledDataSource,name=intergalactoApp
```

If you exclude identity tokens from JMX names **you must ensure that each PooledDataSource always has a unique dataSourceName value!** Otherwise, only one of the PooledDataSources with identical names will be accessible by JMX, and which one will be undefined. Excluding identity tokens from JMX names is particularly hazardous if you will initialize multiple DataSource from the same named configuration. By default, dataSourceName takes the value of the configuration name. **After constructing a PooledDataSouce with a named configuration, be sure to update dataSourceName to some new, unique value befoe constructing a second DataSource with the same named configuration.**

The singleton C3P0Registry is also repesented by an MBean. It may have the following attributes embedded within its JMX ObjectName:

- type
- name

The value of the name attribute is determined by the following property, which may be set as a System property, in c3p0.properties, or in HOCON config.

```
com.mchange.v2.c3p0.management.RegistryName=CoolC3P0Registry
```

With the RegistryName shown above, the full JMX ObjectName, in String format, would be

```
com.mchange.v2.c3p0:type=C3P0Registry,name=CoolC3P0Registry
```

If you do not explicitly set a RegistryName, no default value is used, no name attribute is embedded. The full JMX ObjectName in String format would be

```
com.mchange.v2.c3p0:type=C3P0Registry
```

Disabling JMX Support

If you do not want c3p0 to register MBeans with your JMX environment, you can suppress JMX support entirely. Set the following, as a System property, in c3p0.properties, or in HOCON config:

```
com.mchange.v2.c3p0.management.ManagementCoordinator=com.mchange.v2.c3p0.management.NullManagementCoordinator
```

Configuring Logging

⇑

c3p0 uses a custom logging library similar to jakarta commons-logging. Log messages can be directed to the to the popular slf4j (with its logback backend), to the venerable log4j library, the more recent log4j2 library, to the standard logging facility introduced with jdk1.4, or to System.err. Nearly all configuration should be done at the level of your preferred logging library. There are a very few configuration options specific to c3p0's logging, and usually the defaults will be fine. Logging-related parameters may be placed in your c3p0.properties file, in HOCON configuration files, in a file called mchange-log.properties at the top-level of your classpath, or they may be defined as System properties. (The logging properties defined below may not be defined in c3p0-config.xml!) See the box below.

c3p0's logging behavior is affected by certain build-time options. If build-option c3p0.debug is set to false, all messages at a logging level below INFO will be suppressed. Build-option c3p0.trace controls how fine-grained c3p0's below INFO level reporting will be. For the moment, distributed c3p0 binaries are compiled with debug set to true and trace set to its maximum level of 10. But binaries may eventually be distributed with debug

set to `false`. (For the moment, the performance impact of the logging level-checks seems very small, and it's most flexible to compile in all the messages, and let your logging library control which are emitted.) When c3p0 starts up, it emits the build-time values of debug and trace, along with the version and build time.

`com.mchange.v2.log.MLog`

Determines which library c3p0 will output log messages to. By default, if slf4j is available, it will use that library, otherwise log4j if available, otherwise log4j2 if available, otherwise jdk1.4 logging apis, and if all are unavailable, it will fallback to logging via `System.err`. If you want to directly control which library is used, you may set this property to one of:

- `com.mchange.v2.log.slf4j.Slf4jMLog`
- `com.mchange.v2.log.log4j.Log4jMLog`
- `com.mchange.v2.log.log4j2.Log4j2MLog`
- `com.mchange.v2.log.jdk14logging.Jdk14MLog`
- `com.mchange.v2.log.FallbackMLog`

Alternatively, the following abbreviations are supported:

- `slf4j`
- `log4j`
- `log4j2`
- `jul, jdk14, java.util.logging`
- `fallback`

You may also set this property to a comma separated list of any mix the above alternatives, to define an order of preference among logging libraries.

`com.mchange.v2.log.MLog.useRedirectableLoggers`

Using API in `com.mchange.v2.log.MLog`, it is possible to change at runtime what library c3p0 output logs to, or to switch midstream to the standard-error based fallback logger. However, for this to be useful, loggers already constructed against the original configuration need to be sensitive to the change. Setting this value to `true` will cause c3p0 to use slightly less performant loggers that can be redirected between libraries at runtime. This setting is `false` by default.

`com.mchange.v2.log.jdk14logging.suppressStackWalk`

Under JDK standard logging, the logging library may inspect stack traces to determine the class and method from which a log message was generated. That can be helpful, but it is also slow. Setting this configuration parameter to true will suppress this stack walk, and reduce the overhead of logging. **This property now defaults to `true`, and logger names are logged in place of class names.** To return to the original slower but more informative approach, explicitly set the property to false.

`com.mchange.v2.log.NameTransformer`

By default, c3p0 uses very fine-grained logging, in general with one logger for each c3p0 class. For a variety of reasons, some users may prefer fewer, more global loggers. You may opt for one-logger-per-package by setting `com.mchange.v2.log.NameTransformer` to the value `com.mchange.v2.log.PackageNames`. Advanced users can also define other strategies for organizing the number and names of loggers by setting this variable to the fully-qualified class name of a custom implementation of the `com.mchange.v2.log.NameTransformer` interface.

`com.mchange.v2.log.FallbackMLog.DEFAULT_CUTOFF_LEVEL`

If, whether by choice or by necessity, you are using c3p0's `System.err` fallback logger, you can use this parameter to control how detailed c3p0's logging should be. Any of the following values (taken from the jdk1.4 logging library) are acceptable:

- `OFF`
- `SEVERE`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`
- `ALL`

This property defaults to `INFO`.

Configuring Security

⇑

c3p0 is an old library, originally released in 2002. In those days, people were very excited about the fact that JVM applications could download code on the fly and incorporate it into running applications. Obviously, that is a practice with security, um, implications. But in the early days of Java, inspired by browser-bound Java applets and their "sandboxing", the imagined security model was that code would be annotated with the location from which it was downloaded and would be signed. The JVM would enforce constraints on running code based on its provenance.

That is a retrofuture now. All that's left of it are certain relics, especially the venerable and now deprecated `java.lang.SecurityManager`. Tools still remain in the JVM to download and execute code into live applications (e.g. `java.net.URLClassLoader`), but under most circumstances, that code can then do what it wishes. Responsibility for preventing execution of malicious remote code now lies with application deployers and library developers, rather than with security barriers enforced by the JVM.

c3p0's main dependency, mchange-commons-java contains an independent implementation of `javax.naming.Reference` resolution, including support for remote `factoryClassLocation` from which the code for referenced objects could be downloaded via `URLClassLoader`. This functionality continues to exist. But as of the c3p0-0.12.0, it is now disabled by default.

Two other potentially dangerous JNDI capabilities — JNDI-lookups against apparently remote name services and support of JNDI contexts configured by a deserialized environment — are also disabled by default.

Objects looked up via JNDI get reconstituted by a `javax.naming.spi.ObjectFactory` instance, which gets constructed via Java reflection. `c3p0` now requires and sets a whitelist of allowable `ObjectFactory` fully-qualified classname.

For information on the vulnerabilities against which these changes defend, see e.g. here and here. Other vulnerabilities documented in those resources, like the Java-serialization-based `userOverridesAsString` parameter, are eliminated, but in ways that require no new configuration. (For

example, `userOverridesAsString` is now stored as CSV, rather than in a Java-serialization-based format.)

The following properties can be configured as System properties, in `c3p0.properties`, or in HOCON configuration files. The `boolean` properties are *false biased*. Their values are `false` by default, if the properties do not appear in config or as System properties. Any values provided other than `true` get interpreted as `false`. If any of these properties are set as `false` in System properties, that will override any potentially conflicting configuration elsewhere.

Most applications should not change these properties from their restrictive defaults!

`com.mchange.v2.naming.acceptDeserializedInitialContextEnvironment`

`false` by default. If reset to `true`, objects serialized as JNDI references may also include a serialized "environment", that is, a serialized mapping of key-value pairs that configure the lookup. That in turn can configure several dangerous parameters, including `java.naming.factory.initial`, which specifies a class that will be reflexively instantiated to handle the lookup, and `java.naming.provider.url` which may specify the URL an unsafe or malicious name service.

`com.mchange.v2.naming.generateSerializedObjectBinaryRefAddr`

`false` by default. If reset to `true`, when `javax.naming.Reference` instances are constructed, if they contain properties that cannot otherwise be represented or serialized, Java serialization may be used to store the value of these properties. **`c3p0` no longer uses or relies upon this capablility.** Only users building custom extensions with unusual references would ever have use for this capability, and they should avoid it given how dangerous a vector for malware Java serialization has proven to be.

If you do wish to construct `Reference` instances containing Serialized objects, keep in mind that c3p0's (and mchange-commons-java's) ObjectFactory instances no longer *deserialize* these properties by default. You'd need to

1. Subclass `com.mchange.v2.c3p0.impl.C3P0JavaBeanObjectFactory` or `com.mchange.v2.naming.JavaBeanObjectFactory`
2. Override `protected void handleDeserializeBinaryRefAddressContent(String propertyName, Map out, byte[] content) throws ClassNotFoundException, IOException` with a call to `dangerousDeserializeBinaryRefAddressContent(propertyName, out, content)`, skipping the default implementation, which is to fail with a warning.
3. Make sure your setting for `com.mchange.v2.naming.objectFactoryWhitelist` includes the fully-qualified-name of your new `ObjectFactory` implementation. If you will be referencing/derefencing c3p0 DataSources, your whitelist must also include `com.mchange.v2.c3p0.impl.C3P0JavaBeanObjectFactory`.

`com.mchange.v2.naming.nameGuardClassName`

`null` by default. If left as `null`, the default behavior is to restrict to apparently local names, names beginning with `java:` if Strings, or whose first component is nonempty and begins with `java:`, if a `javax.naming.Name`. If this property is reset, it should be reset to the fully qualified class name of an implementation of `com.mchange.v2.naming.NameGuard`, which defines what JNDI names this library will consider acceptable. These implementations must include a public no-arg constructor for reflective instantiation. You can define your own implementations of `NameGuard`, but four implementations are already provided:

- `com.mchange.v2.naming.ApparentlyLocalNameGuard` — implements the default behavior, accepts names beginning with `java:` if Strings, or whose first component is nonempty and begins with `java:`, if a `javax.naming.Name`.
- `com.mchange.v2.naming.FirstComponentIsJavaIdentifierNameGuard` — accepts names whose first component, or whose substring prior to first '/', would be a valid Java identifier. This allows names like `jdbc/myDataSource` for applications that place these names at the top-level rather than in the recommended `java:comp/env` context. Names that would likely be remote, for example `ldap://my.server/myName`, remain forbidden, as `ldap:` is not a valid Java identifier.
- `com.mchange.v2.naming.ApparentlyLocalOrFirstComponentIsJavaIdentifierNameGuard` — accepts names that would be accepted by EITHER of the prior two `NameGuard` implementations.
- `com.mchange.v2.naming.AnyNameNameGuard` — all names are accepted, effectively disabling name-based security restrictions. Please use only with great care.

`com.mchange.v2.naming.objectFactoryWhitelist`

By default it's `com.mchange.v2.c3p0.impl.C3P0JavaBeanObjectFactory`, the `ObjectFactory` c3p0 uses when constructing its own references. In general, it's comma separated list of fully qualified class name expected to be implementations of `javax.naming.spi.ObjectFactory`.

(Note: If this property is defined distinctly in System properties and in other config, only the intersection of the two provided whitelists will be permitted. Under c3p0, System properties are incorporated into and override other library config, so differences between the System properties value and the library config value should be very rare. Such differences might result if System properties are changed at runtime, after c3p0 config has been read and cached.)

`com.mchange.v2.naming.supportReferenceRemoteFactoryClassLocation`

`false` by default. If reset to `true`, lookups of `javax.naming.Reference` that download remote code will be permitted. This is very dangerous.

Named configurations

⇑

You can define *named configurations* which augment and override the default configuration that you define. When you instantiate a c3p0 `PooledDataSource`, whether via the `ComboPooledDataSource` constructor or via the `DataSources` factory class, you can supply a configuration name. For example, using `ComboPooledDataSource`:

```
ComboPooledDataSource cpds = new ComboPooledDataSource("intergalactoApp");
```

Or using the `DataSources` factory class:

```
DataSource ds_pooled = DataSources.pooledDataSource( ds_unpooled, "intergalactoApp" );
```

To *define* named configurations...

In a properties-style config file...

```
# define default-config param values
c3p0.maxPoolSize=30
c3p0.minPoolSize=10

# define params for a named config called intergalactoApp
c3p0.named-configs.intergalactoApp.maxPoolSize=1000
c3p0.named-configs.intergalactoApp.minPoolSize=100
c3p0.named-configs.intergalactoApp.numHelperThreads=50

# define params for a named config called littleTeenyApp
c3p0.named-configs.littleTeenyApp.maxPoolSize=5
c3p0.named-configs.littleTeenyApp.minPoolSize=2
```

In a HOCON config file...

```
c3p0 {
  maxPoolSize=30
  minPoolSize=10

  named-configs {
    intergalactoApp {
      maxPoolSize=1000
      minPoolSize=100
      numHelperThreads=50
    }
    littleTeenyApp {
      maxPoolSize=5
      minPoolSize=2
    }
  }
}
```

In an XML config file...

```
<c3p0-config>

  <default-config>
    <property name="maxPoolSize">30</property>
    <property name="minPoolSize">10</property>
  </default-config>

  <named-config name="intergalactoApp">
    <property name="maxPoolSize">1000</property>
    <property name="minPoolSize">100</property>
    <property name="numHelperThreads">50</property>
  </named-config>

  <named-config name="littleTeenyApp">
    <property name="maxPoolSize">5</property>
    <property name="minPoolSize">2</property>
  </named-config>

</c3p0-config>
```

Per-user configurations

⇑

You can define overrides of default or named configurations that apply only to pools of Connections authenticated for a particular user. Not all configuration parameters support per-user overrides. See Appendix A for details.

To define per-user configurations...

In a properties-style config file...

```
# define default-config param values
c3p0.maxPoolSize=30
c3p0.minPoolSize=10

# define params for a user called 'steve'
c3p0.user-overrides.steve.maxPoolSize=15
c3p0.user-overrides.steve.minPoolSize=5

# define params for a user called 'ramona'
c3p0.user-overrides.ramona.maxPoolSize=50
c3p0.user-overrides.ramona.minPoolSize=20
```

In a HOCON config file...

```
c3p0 {
  maxPoolSize=30
  minPoolSize=10

  user-overrides {
    steve {
      maxPoolSize=15
      minPoolSize=5
    }
    ramona {
      maxPoolSize=50
      minPoolSize=20
    }
  }
}
```

In an XML config file...

```
<c3p0-config>

  <default-config>

    <property name="maxPoolSize">30</property>
    <property name="minPoolSize">10</property>

    <user-overrides user="steve">
      <property name="maxPoolSize">15</property>
      <property name="minPoolSize">5</property>
    </user-overrides>

    <user-overrides user="ramona">
      <property name="maxPoolSize">50</property>
      <property name="minPoolSize">20</property>
    </user-overrides>

  </default-config>

</c3p0-config>
```

User extensions to configuration

⇑

Users can add their own configuration information, usually to customize the behavior of `ConnectionCustomizers`. User configuration is stored as a `Map` containing `String` keys and values, stored under the following configuration parameter:

- `extensions`

The extensions `Map` can be set programatically like any other configuration parameter. However, there is special support for defining keys and values for the extensions `Map` in configuration files. In a properties-style config file...

```
c3p0.extensions.initSql=SET SCHEMA 'foo'
c3p0.extensions.timezone=PDT
...
```

In a HOCON config file...

```
c3p0 {
  extensions {
     initSql=SET SCHEMA 'foo'
     timezone=PDT
  }
}
```

In an XML config file...

```
<c3p0-config>
  <default-config>
    <extensions>
      <property name="initSql">SET SCHEMA 'foo'</property>
      <property name="timezone">PDT</property>
    </extensions>
```

```
  </default-config>
</c3p0-config>
```

To find the extensions defined for a `PooledDataSource`, you must have access to its `identityToken`, which is supplied as an argument to all `ConnectionCustomizer` methods. Given an `identityToken`, you can use the method `C3P0Registry.extensionsForToken(...)` to access the extensions `Map`.

Because extensions are primary designed to be used within `ConnectionCustomizer` implementations, the `AbstractConnectionCustomizer` class also defines a protected extensionsForToken(...) method as a convenience.

Here is an example `ConnectionCustomizer` implementation that makes use of user-defined configuration extensions. It defines an `initSql` extension, whose value should be a `String` containing SQL that should be executed when a `Connection` is checked out from the pool:

```
package mypkg;

import java.sql.*;
import com.mchange.v2.c3p0.AbstractConnectionCustomizer;

public class InitSqlConnectionCustomizer extends AbstractConnectionCustomizer
{
    private String getInitSql( String parentDataSourceIdentityToken )
    { return (String) extensionsForToken( parentDataSourceIdentityToken ).get ( "initSql" ); }

    public void onCheckOut( Connection c, String parentDataSourceIdentityToken  ) throws Exception
    {
        String initSql = getInitSql( parentDataSourceIdentityToken );
        if ( initSql != null )
        {
            Statement stmt = null;
            try
            {
                stmt = c.createStatement();
                stmt.executeUpdate( initSql );
            }
            finally
            { if ( stmt != null ) stmt.close(); }
        }
    }
}
```

Note: There's no need to implement your own `ConnectionCustomizer` if you just want `initSql`. This example, plus a bit of extra TRACE-level logging, is implemented within the library as `com.mchange.v2.c3p0.example.InitSqlConnectionCustomizer`.

Mixing named, per-user, and user-defined configuration extensions

⇑

Named configurations, per-user overrides, and user-defined configuration extensions can easily be mixed.

In a properties-style config file...

```
c3p0.maxPoolSize=30
c3p0.extensions.initSql=SET SCHEMA 'default'

c3p0.named-configs.intergalactoApp.maxPoolSize=1000
c3p0.named-configs.intergalactoApp.extensions.initSql=SET SCHEMA 'intergalacto'
c3p0.named-configs.user-overrides.steve.maxPoolSize=20
```

In a HOCON config file...

```
c3p0 {
  maxPoolSize=30
  extensions {
    initSql=SET SCHEMA 'default'
  }
  named-configs {
    intergalactoApp {
      maxPoolSize=1000
      user-overrides {
        steve {
          maxPoolSize=20
        }
      }
      extensions {
        initSql=SET SCHEMA 'intergalacto'
      }
    }
  }
}
```

In an XML config file...

```
<c3p0-config>

  <default-config>
    <property name="maxPoolSize">30</property>
    <extensions>
      <property name="initSql">SET SCHEMA 'default'</property>
    </extensions>
  </default-config>

  <named-config name="intergalactoApp">
    <property name="maxPoolSize">1000</property>
    <user-overrides name="steve">
      <property name="maxPoolSize">20</property>
    </user-overrides>
    <extensions>
      <property name="initSql">SET SCHEMA 'intergalacto'</property>
    </extensions>
  </named-config>

</c3p0-config>
```

Performance

Enhanced performance is the purpose of Connection and Statement pooling, and a major goal of the c3p0 library. For most applications, Connection pooling will provide a significant performance gain, especially if you are acquiring an unpooled Connection for each client access. If you are letting a single, shared Connection serve many clients to avoid Connection acquisition overhead, you may suffer performance issues and problems managing transactions when your Connection is under concurrent load; Connection pooling will enable you to switch to a one Connection-per-client model with little or no cost. If you are writing Enterprise Java Beans, you may be tempted to acquire a Connection once and not return it until the bean is about to be destroyed or passivated. But this can be resource-costly, as dormant pooled beans needlessly hold the Connection's network and database resources. Connection pooling permits beans to only "own" a Connection while they are using it.

But, there are performance costs to c3p0 as well. In order to implement automatic cleanup of unclosed `ResultSets` and `Statements` when parent resources are returned to pools, all client-visible `Connections`, `ResultSets`, `Statements` are really wrappers around objects provided by an underlying unpooled DataSource or "traditional" JDBC driver. Thus, there is some extra overhead to all JDBC calls.

Some attention has been paid to minimizing the "wrapper" overhead of c3p0. In my environment, the wrapper overhead amounts from several hundreths to several thousandths of the cost of Connection acquisition, so unless you are making many, many JDBC calls in fast succession, there will be a net gain in performance and resource-utilization efficiency. Significantly, the overhead associated with ResultSet operations (where one might iterate through a table with thousands of records) appears to be negligibly small.

Known Shortcomings

- Connections and Statements are pooled on a per-authentication basis. So, if one pool-backed DataSource is used to acquire Connections both for [`user`=alice, `password`=secret1] and [`user`=bob, `password`=secret2], there will be two distinct pools, and the DataSource might in the worst case manage twice the number of Connections specified by the `maxPoolSize` property.

  This fact is a natural consequence of the definition of the DataSource spec (which allows Connections to be acquired with multiple user authentications), and the requirement that all Connections in a single pool be functionally identical. This "issue" will not be changed or fixed. It's noted here just so you understand what's going on.

- The overhead of Statement pooling is too high. For drivers that do not perform significant preprocessing of PreparedStatements, the pooling overhead outweighs any savings. Statement pooling is thus turned off by default. If your driver does preprocess `PreparedStatements`, especially if it does so via IPC with the RDBMS, you will probably see a significant performance gain by turning Statement pooling on. (Do this by setting the configuration property `maxStatements` or `maxStatementsPerConnection` to a value greater than zero.).

Feedback and Support

Please provide any and all feedback to <swaldman@mchange.com>! Also, feel free to join and ask questions on the `c3p0-users` mailing list. Sign up at http://sourceforge.net/projects/c3p0/

Thank you for using c3p0!!!

Appendix A: Configuration Properties

c3p0 configuration properties can be divided into JavaBeans-style Properties and Other Properties.

JavaBeans-style Properties

The following properties can be set directly in code as JavaBeans properties, via a System properties or a `c3p0.properties` file (with `c3p0.` prepended to the property name), in HOCON (typesafe-config) files, or in a `c3p0-config.xml` file. See the section on Configuration above. Click on the property name for a full description.

acquireIncrement	extensions	minPoolSize
acquireRetryAttempts	factoryClassLocation	numHelperThreads
acquireRetryDelay	forceIgnoreUnresolvedTransactions	overrideDefaultUser
attemptResurrectOnCheckin	forceSynchronousCheckins	overrideDefaultPassword
autoCommitOnClose	forceUseNamedDriverClass	password
automaticTestTable	idleConnectionTestPeriod	preferredTestQuery
breakAfterAcquireFailure	initialPoolSize	privilegeSpawnedThreads
cancelAutomaticallyClosedConnections	jdbcUrl	propertyCycle
checkoutTimeout	markSessionBoundaries	statementCacheNumDeferredCloseThreads
connectionCustomizerClassName	maxAdministrativeTaskTime	taskRunnerFactoryClassName
connectionIsValidTimeout	maxConnectionAge	testConnectionOnCheckin
connectionTesterClassName	maxIdleTime	testConnectionOnCheckout
contextClassLoaderSource	maxIdleTimeExcessConnections	unreturnedConnectionTimeout
dataSourceName	maxPoolSize	user
debugUnreturnedConnectionStackTraces	maxStatements	
driverClass	maxStatementsPerConnection	

`acquireIncrement`

Default: 3

Determines how many connections at a time c3p0 will try to acquire when the pool is exhausted. [See "Basic Pool Configuration"]

`acquireRetryAttempts`

Default: 30

Defines how many times c3p0 will retry after a failure to acquire a new Connection from the database before giving up. If this value is less than zero, c3p0 will keep trying to fetch a Connection indefinitely. [See "Configuring Recovery From Database Outages"]

`acquireRetryDelay`

Default: 1000

Milliseconds, time c3p0 will wait between acquire attempts. [See "Configuring Recovery From Database Outages"]

`attemptResurrectOnCheckin`

Default: false

If `attemptResurrectOnCheckin` is `true`, c3p0 will perform an extra Connection test at checkin on Connections it has determined may be broken, and reintegrate any Connections that pass the test.

When a Connection (or its subsidiary resources) throws an Exception within a client session, c3p0 silently retests the Connection to try to learn whether it was an application Exception or a sign thet the Connection is invalid. If this Connection test fails, c3p0 marks the Connection for exclusion from the pool. When the client checks in (calls `close()` on) the excluded Connection, c3p0 closes the physical Connection and if necessary replaces it in the pool.

Under some databases and circumstances, an `Exception` can break a session in ways that client applications can recover from. For example, once an operation within a transaction has failed, some databases signal failure on any new operations, including Connection tests, until a `rollback()` or `commit()` (which might fail, but still rehabilitate the session). In these circumstances it can be wasteful, and for some application, disruptive to destroy and replace Connections that have been restored to good health. [See "Other DataSource Configuration"]

`autoCommitOnClose`

Default: false

The JDBC spec is unforgivably silent on what should happen to unresolved, pending transactions on Connection close. C3P0's default policy is to rollback any uncommitted, pending work. (I think this is absolutely, undeniably the right policy, but there is no consensus among JDBC driver vendors.) Setting `autoCommitOnClose` to true causes uncommitted pending work to be committed, rather than rolled back on Connection close. [*Note: Since the spec is absurdly unclear on this question, application authors who wish to avoid bugs and inconsistent behavior should ensure that all transactions are explicitly either committed or rolled-back before close is called.*] [See "Configuring Unresolved Transaction Handling"]

`automaticTestTable`

Default: null

If provided, c3p0 will create an empty table of the specified name, and use queries against that table to test the Connection. [See "Configuring Connection Testing"]

This parameter can be useful only rarely, when working with very old JDBC drivers that do not support Java 6 `Connection.isValid(int timeout)`. When using recent JDBC drivers, please leave this parameter, `preferredTestQuery`, and `connectionTesterClassName` at their default values of `null`, which will allow the more modern and efficient `isValid(...)` to be used for `Connection` tests.

If `automaticTestTable` is provided, c3p0 will generate its own test query, therefore any `preferredTestQuery` set will be ignored. You should not work with the named table after c3p0 creates it; it should be strictly for c3p0's use in testing your Connection. (If you define your own ConnectionTester, it must implement the QueryConnectionTester interface for this parameter to be useful.)

`breakAfterAcquireFailure`

Default: false

If true, a pooled DataSource will declare itself broken and be permanently closed if a Connection cannot be obtained from the database after making `acquireRetryAttempts` to acquire one. If false, failure to obtain a Connection will cause all Threads waiting for the pool to acquire a Connection to throw an Exception, but the DataSource will remain valid, and will attempt to acquire again following a call to `getConnection()`. [See "Configuring Recovery From Database Outages"]

`cancelAutomaticallyClosedStatements`

Default: false

If true, whenever c3p0 will automatically call `close()` on a Statement — because a client has forgotten to close the Statement, because the Statement cache is expiring the Statement, or because an `unreturnedConnectionTimeout` has forced a Connection to close while it still has open statements — c3p0 will make a best-effort attempt to call `Statement.cancel()` prior to calling `close()`. If the call to `cancel()` fails, it will be ignored. The Statement will be closed regardless. [See "Other DataSource Configuration"]

`checkoutTimeout`

Default: 0

The number of milliseconds a client calling getConnection() will wait for a Connection to be checked-in or acquired when the pool is exhausted. Zero means wait indefinitely. Setting any positive value will cause the getConnection() call to time-out and break with an `SQLException` after the specified number of milliseconds. [See "Other DataSource Configuration"]

`connectionCustomizerClassName`

Default: null

The fully qualified class-name of an implememtation of the `ConnectionCustomizer` interface, which users can implement to set up Connections when they are acquired from the database, or on check-out, and potentially to clean things up on check-in and Connection destruction. If standard Connection properties (holdability, readOnly, or transactionIsolation) are set in the ConnectionCustomizer's onAcquire() method, these will override the Connection default values.

`connectionIsValidTimeout`

Default: 0, or the value of `com.mchange.v2.c3p0.impl.DefaultConnectionTester.isValidTimeout` if set

Seconds, length of time during which the default `Connection` test — a call to the `Connection.isValid(int timeout)` method — will wait for a response before giving up and returning `false`. [See "Configuring Connection Testing"]

Note: *This parameter does not apply if you override the default `Connection` test by supply a non-null value for `connectionTesterClassName`. This parameter is equivalent to setting `com.mchange.v2.c3p0.impl.DefaultConnectionTester.isValidTimeout` under prior versions' when the default ConnectionTester was in use. In order not to break old configs, if `com.mchange.v2.c3p0.impl.DefaultConnectionTester.isValidTimeout` is set, we'll pick it up as the default, but with a warning recommend transitioning from the legacy config.*

`connectionTesterClassName`

Default: null

If not null (this should almost always be null!), the fully qualified class-name of an implememtation of the `ConnectionTester` interface, or `QueryConnectionTester` if you would like instances to have access to a user-configured `preferredTestQuery` or `automaticTestTable`. [See "Configuring Connection Testing"]

This parameter is primarily useful for supporting ancient JDBC drivers that do not offer the `Connection.isValid(int timeout)` method introduced with Java 6. Connection-testing used to be complicated, but now it is standardized and simplified. If you are using an ancient driver, you can set this this to the old-school `com.mchange.v2.c3p0.impl.DefaultConnectionTester`. Or you can just set `preferredTestQuery` or `automaticTestTable`, which will conjure that ghost if `connectionTesterClassName` is left as null. And, of course, you are free to roll your own.

`contextClassLoaderSource`

Default: caller

Must be one of `caller`, `library`, or `none`. Determines how the `contextClassLoader` (see `java.lang.Thread`) of c3p0-spawned Threads is determined. If `caller`, c3p0-spawned Threads (helper threads, `java.util.Timer` threads) inherit their `contextClassLoader` from the client Thread that provokes initialization of the pool. If `library`, the `contextClassLoader` will be the class that loaded c3p0 classes. If `none`, no `contextClassLoader` will be set (the property will be `null`), which in practice means the system ClassLoader will be used. The default setting of `caller` is sometimes a problem when client applications will be hot redeployed by an app-server. When c3p0's Threads hold a reference to a `contextClassLoader` from the first client that hits them, it may be impossible to garbage collect a `ClassLoader` associated with that client when it is undeployed in a running VM. Setting this to `library` can resolve these issues. [See "Configuring To Avoid Memory Leaks On Hot Redeploy Of Client"]

Does Not Support Per-User Overrides.

`dataSourceName`

Default: if configured with a named config, the config name, otherwise the pool's "identity token"

Every c3p0 pooled data source is given a `dataSourceName`, which serves two purposes. It helps users find DataSources via C3P0Registry, and it is included in the name of JMX mBeans in order to help track and distinguish between multiple c3p0 DataSources even across application or JVM restarts. `dataSourceName` defaults to the pool's configuration name, if a named config was used, or else to an "identity token" (an opaque, guaranteed unique String associated with every c3p0 DataSource). You may update this property to any name you find convenient. `dataSourceName` is *not* guaranteed to be unique — for example, multiple DataSource created from the same named configuration will share the same `dataSourceName`. But if you are going to make use of `dataSourceName`, you will probably want to ensure that all pooled DataSources within your JVM do have unique names.

`debugUnreturnedConnectionStackTraces`

Default: false

If true, and if `unreturnedConnectionTimeout` is set to a positive value, then the pool will capture the stack trace (via an Exception) of all Connection checkouts, and the stack traces will be printed when unreturned checked-out Connections timeout. This is intended to debug applications with Connection leaks, that is applications that occasionally fail to return Connections, leading to pool growth, and eventually exhaustion (when the pool hits `maxPoolSize` with all Connections checked-out and lost). This parameter should only be set while debugging,

as capturing the stack trace will slow down every Connection check-out. [See "Configuring to Debug and Workaround Broken Client Applications"]

`driverClass`

Default: null

The fully-qualified class name of the JDBC driverClass that is expected to provide Connections. c3p0 will preload any class specified here to ensure that appropriate URLs may be resolved to an instance of the driver by `java.sql.DriverManager`. If you wish to skip `DriverManager` resolution entirely and ensure that an instance of the specified class is used to provide Connections, use `driverClass` in combination with `forceUseNamedDriverClass`. [See also `jdbcUrl`.]

Does Not Support Per-User Overrides.

`extensions`

Default: an empty `java.util.Map`

A `java.util.Map` (raw type) containing the values of any user-defined configuration extensions defined for this DataSource.

Does Not Support Per-User Overrides.

`factoryClassLocation`

Default: null

DataSources that will be bound by JNDI and use that API's Referenceable interface to store themselves may specify a URL from which the class capable of dereferencing a them may be loaded. If (as is usually the case) the c3p0 libraries will be locally available to the JNDI service, leave this set as null. **Note: As of c3p0-0.12.0, remote classloading is now disabled by default.** See Configuring Security and com.mchange.v2.naming.supportReferenceRemoteFactoryClassLocation.

Does Not Support Per-User Overrides.

`forceIgnoreUnresolvedTransactions`

Default: false

Strongly disrecommended. Setting this to `true` may lead to subtle and bizarre bugs. This is a terrible setting, leave it alone unless absolutely necessary. It is here to workaround broken databases / JDBC drivers that do not properly support transactions, but that allow Connections' `autoCommit` flags to go to false regardless. If you are using a database that supports transactions "partially" (this is oxymoronic, as the whole point of transactions is to perform operations reliably and completely, but nonetheless such databases are out there), if you feel comfortable ignoring the fact that Connections with `autoCommit == false` may be in the middle of transactions and may hold locks and other resources, you may turn off c3p0's wise default behavior, which is to protect itself, as well as the usability and consistency of the database, by either rolling back (default) or committing (see `c3p0.autoCommitOnClose` *above*) unresolved transactions. **This should only be set to true when you are sure you are using a database that allows Connections' autoCommit flag to go to false, but offers no other meaningful support of transactions. Otherwise setting this to true is just a bad idea.** [See "Configuring Unresolved Transaction Handling"]

`forceSynchronousCheckins`

Default: false

Setting this to `true` forces Connections to be checked-in synchronously, which under some circumstances may improve performance. Ordinarily Connections are checked-in asynchronously so that clients avoid any overhead of testing or custom check-in logic. However, asynchronous check-in contributes to thread pool congestion, and very busy pools might find clients delayed waiting for check-ins to complete. Expanding `numHelperThreads` can help manage Thread pool congestion, but memory footprint and switching costs put limits on practical thread pool size. To reduce thread pool load, you can set `forceSynchronousCheckins` to `true`. Synchronous check-ins are likely to improve overall performance when `testConnectionOnCheckin` is set to false and no slow work is performed in a `ConnectionCustomizer`'s `onCheckIn(...)` method. If Connections are tested or other slow work is performed on check-in, then this setting will cause clients to experience the overhead of that work on `Connection.close()`, which you must trade-off against any improvements in pool performance. [See "Other DataSource Configuration"]

`forceUseNamedDriverClass`

Default: false

Setting the parameter driverClass causes that class to preload and register with `java.sql.DriverManager`. However, it does not on its own ensure that the driver used will be an instance of driverClass, as `DriverManager` may (in unusual cases) know of other driver classes which can handle the specified jdbcUrl. Setting this parameter to true causes c3p0 to ignore `DriverManager` and simply instantiate driverClass directly.

Does Not Support Per-User Overrides.

`idleConnectionTestPeriod`

Default: 0

If this is a number greater than 0, c3p0 will test all idle, pooled but unchecked-out connections, every this number of seconds. [See "Configuring Connection Testing"]

`initialPoolSize`

Default: 3

Number of Connections a pool will try to acquire upon startup. Should be between `minPoolSize` and `maxPoolSize`. [See "Basic Pool Configuration"]

`jdbcUrl`

Default: null

The JDBC URL of the database from which Connections can and should be acquired. Should resolve via `java.sql.DriverManager` to an appropriate JDBC Driver (which you can ensure will be loaded and available by setting `driverClass`), or if you wish to specify which driver to use directly (and avoid `DriverManager` resolution), you may specify driverClass in combination with forceUseNamedDriverClass. Unless you are supplying your own unpooled DataSource, a jdbcUrl must always be provided and appropriate for the JDBC driver, however it is resolved.

Does Not Support Per-User Overrides.

markSessionBoundaries

Default: always

One of `always`, `never`, or `if-no-statement-cache`. Defines whether session boundaries will be marked by calling the `beginRequest` and `endRequest` methods of `Connection` on connection checkout and checkin respectively. These methods are optional hints to the JDBC driver, helping them manage user session. Almost always the default of, um, `always` will be best, but occasionally JDBC drivers respond to these hints undesirably. In particular, some JDBC drivers close unclosed `Statement` objects when `endRequest` is called. Setting this parameter to `never` disables session-boundary marking entirely. Setting this parameter to `if-no-statement-cache` enables session-boundary marking if and only if no statement cache is configured (that is if both maxStatements and maxStatementsPerConnection are configured to their defaults of zero). [See "Other DataSource Configuration"]

---

maxAdministrativeTaskTime

Default: 0

Seconds before c3p0's thread pool will try to interrupt an apparently hung task. Rarely useful. Many of c3p0's functions are not performed by client threads, but asynchronously by an internal thread pool. c3p0's asynchrony enhances client performance directly, and minimizes the length of time that critical locks are held by ensuring that slow jdbc operations are performed in non-lock-holding threads. If, however, some of these tasks "hang", that is they neither succeed nor fail with an Exception for a prolonged period of time, c3p0's thread pool can become exhausted and administrative tasks backed up. If the tasks are simply slow, the best way to resolve the problem is to increase the number of threads, via numHelperThreads. But if tasks sometimes hang indefinitely, you can use this parameter to force a call to the task thread's `interrupt()` method if a task exceeds a set time limit. [c3p0 will eventually recover from hung tasks anyway by signalling an "APPARENT DEADLOCK" (you'll see it as a warning in the logs), replacing the thread pool task threads, and interrupt()ing the original threads. But letting the pool go into APPARENT DEADLOCK and then recover means that for some periods, c3p0's performance will be impaired. So if you're seeing these messages, increasing numHelperThreads and setting `maxAdministrativeTaskTime` might help. `maxAdministrativeTaskTime` should be large enough that any resonable attempt to acquire a Connection from the database, to test a Connection, or to destroy a Connection, would be expected to succeed or fail within the time set. Zero (the default) means tasks are never interrupted, which is the best and safest policy under most circumstances. If tasks are just slow, allocate more threads. If tasks are hanging forever, try to figure out why, and maybe setting `maxAdministrativeTaskTime` can help in the meantime. [See "Configuring Threading"]

Does Not Support Per-User Overrides.

---

maxConnectionAge

Default: 0

Seconds, effectively a time to live. A Connection older than `maxConnectionAge` will be destroyed and purged from the pool. This differs from `maxIdleTime` in that it refers to absolute age. Even a Connection which has not been much idle will be purged from the pool if it exceeds `maxConnectionAge`. Zero means no maximum absolute age is enforced. [See Managing Pool Size and Connection Age]

---

maxIdleTime

Default: 0

Seconds a Connection can remain pooled but unused before being discarded. Zero means idle connections never expire. [See "Basic Pool Configuration"]

---

maxIdleTimeExcessConnections

Default: 0

Number of seconds that Connections in excess of `minPoolSize` should be permitted to remain idle in the pool before being culled. Intended for applications that wish to aggressively minimize the number of open Connections, shrinking the pool back towards minPoolSize if, following a spike, the load level diminishes and Connections acquired are no longer needed. If `maxIdleTime` is set, `maxIdleTimeExcessConnections` should be smaller if the parameter is to have any effect. Zero means no enforcement, excess Connections are not idled out. [See Managing Pool Size and Connection Age]

---

maxPoolSize

Default: 15

Maximum number of Connections a pool will maintain at any given time. [See "Basic Pool Configuration"]

---

maxStatements

Default: 0

The size of c3p0's global PreparedStatement cache. If both `maxStatements` and `maxStatementsPerConnection` are zero, statement caching will not be enabled. If `maxStatements` is zero but `maxStatementsPerConnection` is a non-zero value, statement caching will be enabled, but no global limit will be enforced, only the per-connection maximum. `maxStatements` controls the total number of Statements cached, for all Connections. If set, it should be a fairly large number, as each pooled Connection requires its own, distinct flock of cached statements. As a guide, consider how many distinct PreparedStatements are used *frequently* in your application, and multiply that number by `maxPoolSize` to arrive at an appropriate value. Though `maxStatements` is the JDBC standard parameter for controlling statement caching, users may find c3p0's alternative `maxStatementsPerConnection` more intuitive to use. [See "Configuring Statement Pooling"]

---

maxStatementsPerConnection

Default: 0

The number of PreparedStatements c3p0 will cache for a single pooled Connection. If both `maxStatements` and `maxStatementsPerConnection` are zero, statement caching will not be enabled. If `maxStatementsPerConnection` is zero but `maxStatements` is a non-zero value, statement caching will be enabled, and a global limit enforced, but otherwise no limit will be set on the number of cached statements for a single Connection. If set, maxStatementsPerConnection should be set to about the number distinct PreparedStatements that are used *frequently* in your application, plus two or three extra so infrequently statements don't force the more common cached statements to be culled. Though `maxStatements` is the JDBC standard parameter for controlling statement caching, users may find `maxStatementsPerConnection` more intuitive to use. [See "Configuring Statement Pooling"]

---

minPoolSize

Default: 3

Minimum number of Connections a pool will maintain at any given time. [See "Basic Pool Configuration"]

---

numHelperThreads

Default: 3

c3p0 is very asynchronous. Slow JDBC operations are generally performed by helper threads that don't hold contended locks. Spreading these operations over multiple threads can significantly improve performance by allowing multiple operations to be performed simultaneously.

Does Not Support Per-User Overrides.

`overrideDefaultUser`

Default: null

Forces the username that should by PooledDataSources when a user calls the default getConnection() method. This is primarily useful when applications are pooling Connections from a non-c3p0 unpooled DataSource. Applications that use `ComboPooledDataSource`, or that wrap any c3p0-implemented unpooled DataSource can use the simple user property.

Does Not Support Per-User Overrides.

`overrideDefaultPassword`

Default: null

Forces the password that should by PooledDataSources when a user calls the default getConnection() method. This is primarily useful when applications are pooling Connections from a non-c3p0 unpooled DataSource. Applications that use `ComboPooledDataSource`, or that wrap any c3p0-implemented unpooled DataSource can use the simple password property.

Does Not Support Per-User Overrides.

`password`

Default: null

For applications using `ComboPooledDataSource` or any c3p0-implemented unpooled DataSources — `DriverManagerDataSource` or the DataSource returned by `DataSources.unpooledDataSource( ... )` — defines the password that will be used for the DataSource's default `getConnection()` method. (See also user.)

Does Not Support Per-User Overrides.

`preferredTestQuery`

Default: null

Defines the query that will be executed for all connection tests, if the default ConnectionTester (or some other implementation of QueryConnectionTester, or better yet FullQueryConnectionTester) is being used. [See "Configuring Connection Testing"]

This is rarely useful, and should be left as null except when using very old (pre Java 6) or broken JDBC drivers. More recent drivers define a `Connection.isValid(int timeout)` method for efficient Connection tests, which will be used unless this parameter, `automaticTestTable`, and/or `connectionTesterClassName` are set to nondefault values. When working with very old JDBC drivers (or drivers who `Connection.isValid(...)` method is poorly implemented), defining a `preferredTestQuery` that will execute quickly in your database may dramatically speed up Connection tests.

`privilegeSpawnedThreads`

Default: false

If `true`, c3p0-spawned Threads will have the `java.security.AccessControlContext` associated with c3p0 library classes. By default, c3p0-spawned Threads (helper threads, `java.util.Timer` threads) inherit their `AccessControlContext` from the client Thread that provokes initialization of the pool. This can sometimes be a problem, especially in application servers that support hot redeployment of client apps. If c3p0's Threads hold a reference to an `AccessControlContext` from the first client that hits them, it may be impossible to garbage collect a `ClassLoader` associated with that client when it is undeployed in a running VM. Also, it is possible client Threads might lack sufficient permission to perform operations that c3p0 requires. Setting this to `true` can resolve these issues. [See "Configuring To Avoid Memory Leaks On Hot Redeploy Of Client"]

Does Not Support Per-User Overrides.

`propertyCycle`

Default: 0

Maximum time in seconds before user configuration constraints are enforced. Determines how frequently `maxConnectionAge`, `maxIdleTime`, `maxIdleTimeExcessConnections`, `unreturnedConnectionTimeout` are enforced. c3p0 periodically checks the age of Connections to see whether they've timed out. This parameter determines the period. Zero means automatic: A suitable period will be determined by c3p0. [You can call `getEffectivePropertyCycle...()` methods on a c3p0 PooledDataSource to find the period automatically chosen.]

`statementCacheNumDeferredCloseThreads`

Default: 0

If set to a value greater than 0, the statement cache will track when Connections are in use, and only destroy Statements when their parent Connections are not otherwise in use. Although closing of a Statement while the parent Connection is in use is formally within spec, some databases and/or JDBC drivers, most notably Oracle, do not handle the case well and freeze, leading to deadlocks. Setting this parameter to a positive value should eliminate the issue. This parameter should only be set if you observe that attempts by c3p0 to close() cached statements freeze (usually you'll see `APPARENT DEADLOCKS` in your logs). If set, this parameter should almost always be set to `1`. Basically, if you need more than one Thread dedicated solely to destroying cached Statements, you should set `maxStatements` and/or `maxStatementsPerConnection` so that you don't churn through Statements so quickly. [See "Configuring Statement Pooling"]

Does Not Support Per-User Overrides.

`taskRunnerFactoryClassName`

Default: `com.mchange.v2.c3p0.impl.DefaultTaskRunnerFactory`

Fully qualified class name of an implementation of `com.mchange.v2.c3p0.TaskRunnerFactory` which determines how c3p0's many, many asynchronous operations are posted and divided among threads. Most users will want to leave this at c3p0's default, or use published alternative implementations rather than roll your own. But of course you can roll your own! [See "Configuring Threading"]

Does Not Support Per-User Overrides.

`testConnectionOnCheckin`

Default: false

If true, an operation will be performed asynchronously at every connection checkin to verify that the connection is valid. Use in combination with `idleConnectionTestPeriod` for quite reliable, always asynchronous Connection testing. Also, setting an `automaticTestTable` or `preferredTestQuery` will usually speed up all connection tests. [See "Configuring Connection Testing"]

`testConnectionOnCheckout`

Default: false

If true, an operation will be performed at every connection checkout to verify that the connection is valid. ***Be sure to set an efficient*** `preferredTestQuery` ***or*** `automaticTestTable` ***if you set this to*** `true`. ***Performing the (expensive) default Connection test on every client checkout will harm client performance.*** Testing Connections in checkout is the simplest and most reliable form of Connection testing, but for better performance, consider verifying connections periodically using `idleConnectionTestPeriod`. [See "Configuring Connection Testing"]

`unreturnedConnectionTimeout`

Default: 0

Seconds. If set, if an application checks out but then fails to check-in [i.e. close()] a Connection within the specified period of time, the pool will unceremoniously destroy() the Connection. This permits applications with occasional Connection leaks to survive, rather than eventually exhausting the Connection pool. And that's a shame. Zero means no timeout, applications are expected to close() their own Connections. Obviously, if a non-zero value is set, it should be to a value longer than any Connection should reasonably be checked-out. Otherwise, the pool will occasionally kill Connections in active use, which is bad. [See "Configuring to Debug and Workaround Broken Client Applications"]

Setting `unreturnedConnectionTimeout` to an appropriate value may increase resiliency of your application. It can be a useful backstop. But it's better to be neurotic about closing your Connections in the first place. Now that we have try-with-resources, it's not so hard to get robust resource cleanup right! If you do set this property, please monitor your logs to observe and fix any Connection leaks. (The message "A checked-out resource is overdue, and will be destroyed" will appear at `INFO` in the log for logger `com.mchange.v2.resourcepool.BasicResourcePool`.) Use this setting in temporary combination with `debugUnreturnedConnectionStackTraces` to figure out where Connections are being checked-out that don't make it back into the pool!

`user`

Default: null

For applications using `ComboPooledDataSource` or any c3p0-implemented unpooled DataSources — `DriverManagerDataSource` or the DataSource returned by `DataSources.unpooledDataSource()` — defines the username that will be used for the DataSource's default `getConnection()` method. (See also password.)

Does Not Support Per-User Overrides.

Other Properties

⇑

The following configuration properties affect the behavior of the c3p0 library as a whole. They may be set as system properties, in a `c3p0.properties` file, or in a HOCON (typesafe-config) file, but not in XML config files.

Locating and Resolving Configuration Information

Normally, c3p0's configuration information is placed in a either a c3p0-config.xml, c3p0.properties, or a HOCON file at the top-level of an application's CLASSPATH. However, if you wish to place configuration information elsewhere, you may place c3p0 configuration information (in the XML file format only!) anywhere you'd like in the filesystem visible to your application. Just set the following property to the full, absolute path of the XML config file:

```
com.mchange.v2.c3p0.cfg.xml
```

If you set this property to a value beginning with "`classloader:`", c3p0 will search for an XML config file as a ClassLoader resource, that is, in any location you specify under your classpath, including jar-file `META-INF` directories.

Due to security concerns surrounding liberal parsing of XML references, c3p0 now parses XML files extremely restrictively by default. Among other things, it *no longer expands entity references in XML config files. Entity references may be silently ignored!* It also no longer support xml includes, and tries to disable any use of inline document type definitions if the JVM's underlying XML library supports that. In the *very rare cases* where configurations intentionally rely upon entity reference expansion, DTDs, XML include, or other dangerous features, you can turn permissive parsing back on, restoring c3p0's traditional behavior. *Be sure you understand the security concerns, and trust your control over and the integrity of your XML config file, before restoring the old, permissive behavior.* Then, if you wish, you may set the following property to `true`:

```
com.mchange.v2.c3p0.cfg.xml.usePermissiveParser
```

This will restore the traditional, liberal parsing of XML config files that c3p0 utilized by default in versions prior to c3p0-0.9.5.3. (As of version 0.9.5.4, `com.mchange.v2.c3p0.cfg.xml.usePermissiveParser` replaces the now-deprecated `com.mchange.v2.c3p0.cfg.xml.expandEntityReferences` introduced in 0.9.5.3, because much more than entity reference expansion is now restricted.)

Logging-related properties

The following properties affect c3p0's logging behavior. Please see Configuring Logging above for specific information.

```
com.mchange.v2.log.MLog
com.mchange.v2.log.jdk14logging.suppressStackWalk
com.mchange.v2.log.NameTransformer
com.mchange.v2.log.FallbackMLog.DEFAULT_CUTOFF_LEVEL
```

Security-related properties

The following properties help lock c3p0 down and prevent misuse of its libraries. Please see Configuring Security above for specific information.

```
com.mchange.v2.naming.supportReferenceRemoteFactoryClassLocation
com.mchange.v2.naming.acceptDeserializedInitialContextEnvironment
com.mchange.v2.naming.objectFactoryWhitelist
com.mchange.v2.naming.nameGuardClassName
```

Configuring JMX

The following properties affect c3p0's JMX management interface. Plese see Configuring and Managing c3p0 via JMX above for more information.

```
com.mchange.v2.c3p0.management.ExcludeIdentityToken
com.mchange.v2.c3p0.management.RegistryName
com.mchange.v2.c3p0.management.ManagementCoordinator
```

Configuring the VMID

Is it better to be beautiful or correct? Beginning with c3p0-0.9.1, c3p0 opts somewhat reluctantly for correctness.

Here's the deal. Every c3p0 DataSource is allocated a unique "identity token", which is used to ensure that multiple JNDI lookups of the same PooledDataSource always return the same instance, even if the JNDI name-server stores a Serialized or Referenced instance. Previously, c3p0 was happy for generated IDs to be unique within a single VM (and it didn't even get that quite right, before c3p0-0.9.1). But in theory, one VM might look up two different DataSources, generated by two different VMs, that by unlikely coincidence have the same "identity token", leading to errors as one of the two DataSources sneakily substitutes for the second. Though this hypothetical issue has never been reported in practice, c3p0 resolves it by prepending a VMID to its identity tokens. This makes them long and ugly, but correct.

If you don't like the long and ugly VMID, you can set your own, or you can turn off this solution to a hypothetical non-problem entirely with the following property:

```
com.mchange.v2.c3p0.VMID
```

Set it to `NONE` to turn off the VMID, set it to `AUTO` to let c3p0 generate a VMID, or provide any other String to set the VMID that will be used directly. The default is `AUTO`.

Configuring legacy DefaultConnectionTester.isValidTimeout

Note: *As of c3p0-0.10.0 and above, this is no longer relevant to most users, who should just use the ordinary config property `connectionIsValidTimeout` instead. It remains useful only for users who explicitly set a `preferredTestQuery` or `automaticTestTable`, or else who explcitly set `connectionTesterClassName` to `com.mchange.v2.c3p0.impl.DefaultConnectionTester`.*

Under circumstances when the JDBC 4+ `isValid(...)` test will be used by c3p0's `com.mchange.v2.c3p0.impl.DefaultConnectionTester` (see below), by default the test will never time out. If you would the test to timeout and fail, set the following key

```
com.mchange.v2.c3p0.impl.DefaultConnectionTester.isValidTimeout
```

to the desired timeout, in seconds.

Configuring legacy DefaultConnectionTester.QuerylessTestRunner

Note: *As of c3p0-0.10.0 and above, this is no longer relevant to most users. It remains has an effect only for users who explicitly set a `preferredTestQuery` or `automaticTestTable`, or else who explcitly set `connectionTesterClassName` to `com.mchange.v2.c3p0.impl.DefaultConnectionTester`.*

c3p0's built-in DefaultConnectionTester does the right and obvious thing when you've provided a preferredTestQuery or automaticTestTable parameter. But when it has no user-determined query to run to test a Connection, it's less clear what c3p0 should do. In the JDBC 3 API, there was no straightforward, reliable way to test a JDBC Connection. c3p0's DefaultConnectionTester adopted the very conservative technique, using the Connection to query DatabaseMetaData, since this represents a live query to the database that can be executed without any knowledge of a database's schema. Unfortunately, this technique is often very, very slow.

Fortunately, as of version 0.9.5, c3p0 supports the JDBC 4 API's testing using the new `Connection.isValid()` method. This is a fast, reliable test specified and implemented by the JDBC Driver provider.

Although **it will very rarely be necessary**, users can now specify exactly how DefaultConnectionTester will behave if no preferredTestQuery or automaticTestTable has been set via the following property:

```
com.mchange.v2.c3p0.impl.DefaultConnectionTester.querylessTestRunner
```

Possible values of this property include

`METADATA_TABLESEARCH`
: This is c3p0's very slow, but very reliable, traditional default Connection test. It will work even with very old JDBC drivers.

`IS_VALID`

This uses the new JDBC 4 API to perform a driver-defined Connection test. An `AbstractMethodError` will be provoked, however, if it is used with an old JDBC 3 driver.

`SWITCH`
: This first attempts the new JDBC 4 Connection test (like `IS_VALID`), but catches any `AbstractMethodError` and falls back onto `METADATA_TABLESEARCH` if necessary.

`THREAD_LOCAL`
: This checks whether the new `Connection.isValid()` method is available for any implementation of `Connection` that it tests, and stores a `ThreadLocal WeakHashMap` to track which Connection implementations support the method. It then consults the map and runs the fast `IS_VALID` test or the universal `METADATA_TABLESEARCH` test as appropriate.

You can also provide the fully-qualified classname of an implementation of the `DefaultConnectionTester.QuerylessTestRunner` interface and define your own behavior, whatever you'd like to do. Your class should be public, have a public no argument constructor, and be Thread-safe and sharable. (A c3p0 pool typically uses just one ConnectionTester to test all of its Connections, often concurrently.) For examples, see the built-in implementations in c3p0's source code.

The default value is `SWITCH`, which should be fine almost always.

Really, you should almost never bother to set this property. If you are using an old JDBC driver and want to eliminate the small overhead of trying the `isValid()` method and then catching an `AbstractMethodError`, you can do so by setting its value to `METADATA_TABLESEARCH`. But why bother, when a *much, much faster* approach is to set a preferredTestQuery, and avoid the queryless test entirely? If you want to do something totally different, you can implement your own `DefaultConnectionTester.QuerylessTestRunner`. Or you can just implement `ConnectionTester` directly, and set the parameter `connectionTesterClassName`.

Appendix B: Configuration Files, etc.

c3p0 configuration parameters can be set directly in Java code, via a simple Java properties file, via a Typesafe "HOCON" file, via an XML configuration file, or via System properties. Any which way is fine. Choose whatever works best for you.

Overriding c3p0 defaults via `c3p0.properties`

To override the library's built-in defaults, create a file called `c3p0.properties` and place it at the "root" of your classpath or classloader. For a typical standalone application, that means place the file in a directory named in your `CLASSPATH` environment variable. For a typical web-application, the file should be placed in `WEB-INF/classes`. In general, the file must be available as a classloader resource under the name `/c3p0.properties`, in the classloader that loaded c3p0's jar file. Review the API docs (especilly `getResource...` methods) of `java.lang.Class`, `java.lang.ClassLoader`, and `java.util.ResourceBundle` if this is unfamiliar.

The format of `c3p0.properties` should be a normal Java Properties file format, whose keys are c3p0 configurable properties. See Appendix A. for the specifics. An example `c3p0.properties` file is produced below:

```
# turn on statement pooling
c3p0.maxStatements=150

# close pooled Connections that go unused for
# more than half an hour
c3p0.maxIdleTime=1800
```

Overriding c3p0 defaults with "HOCON" (typesafe-config) configuration files

Typesafe has defined a very nice configuration file format called "HOCON".

c3p0 does not include Typesafe's library, but if you bundle it with your application, c3p0 will support configuration in this format. You may place c3p0 configuration in the standard `/application.conf` or `/reference.conf` files, or you may use a special `/c3p0.conf` file. (These files must be placed as top-level ClassLoader resources, see above.) `/application.json`, `/application.properties`, and `c3p0.json` are also supported, and substitutions from `/c3p0.properties` will be resolved. The Typesafe config library can be downloaded from the Maven Central Repository.

Remember, HOCON configuration in only supported if you explicitly bundle the Typesafe config library with your application. The library is not included with c3p0's binary distribution, nor is it downloaded with c3p0 as a transitive dependency!

Here are some examples of setting c3p0 configuration in HOCON:

```
c3p0 {
  # turn on statement pooling
  maxStatements=150

  # close pooled Connections that go unused for
  # more than half an hour
  maxIdleTime=1800
```

```
}
```

Note that you *must* specify ordinary config params explicitly inside a c3p0 scope one way or another, even in a `c3p0.conf` file. "Dot notation" can be used equivalently to scopes:

```
# equivalent to the example above, and
# identical to the properties file format.

c3p0.maxStatements=150
c3p0.maxIdleTime=1800
```

Also, the following two specifications are equivalent:

```
# properties-file-ish specification
com.mchange.v2.log.MLog=com.mchange.v2.log.log4j.Log4jMLog

# scoped specification of the same
com {
  mchange {
    v2 {
      log {
        MLog="com.mchange.v2.log.log4j.Log4jMLog"
      }
    }
  }
}
```

Overriding c3p0 defaults with System properties

⇑

c3p0 properties can also be defined as System properties, using the same "c3p0." prefix for properties specified in a `c3p0.properties` file.

```
swaldman% java -Dc3p0.maxStatements=150 -Dc3p0.maxIdleTime=1800 example.MyC3P0App
```

System properties override settings in c3p0.properties. Please see Precedence of Configuration Settings for more information.

Any key that would be legal in a c3p0.properties file is legal as a System property, including keys that might have defined named, per-user, or user-defined-extension parameters.

Overriding c3p0 defaults via `c3p0-config.xml`

⇑

You can use the XML config file for all c3p0 configuration, including configuration of defaults, named configurations, per-user overrides, and configuration extensions.

By default, c3p0 will look for an XML configuration file in its classloader's resource path under the name "/c3p0-config.xml". That means the XML file should be placed in a directly or jar file directly named in your applications CLASSPATH, in WEB-INF/classes, or some similar location.

If you prefer not to bundle your configuration with your code, you can specify an ordinary filesystem location for c3p0's configuration file via the system property `com.mchange.v2.c3p0.cfg.xml`. (You can also use this property to specify an alternative location in the ClassLoader resource path, e.g. `META-INF`. See Locating Configuration Information.)

Here is an example `c3p0-config.xml` file:

```
<c3p0-config>
  <default-config>
    <property name="automaticTestTable">con_test</property>
    <property name="checkoutTimeout">30000</property>
    <property name="idleConnectionTestPeriod">30</property>
    <property name="initialPoolSize">10</property>
    <property name="maxIdleTime">30</property>
    <property name="maxPoolSize">100</property>
    <property name="minPoolSize">10</property>
    <property name="maxStatements">200</property>

    <user-overrides user="test-user">
      <property name="maxPoolSize">10</property>
      <property name="minPoolSize">1</property>
      <property name="maxStatements">0</property>
```

```
    </user-overrides>

  </default-config>

  <!-- This app is massive! -->
  <named-config name="intergalactoApp">
    <property name="acquireIncrement">50</property>
    <property name="initialPoolSize">100</property>
    <property name="minPoolSize">50</property>
    <property name="maxPoolSize">1000</property>

    <!-- intergalactoApp adopts a different approach to configuring statement caching -->
    <property name="maxStatements">0</property>
    <property name="maxStatementsPerConnection">5</property>

    <!-- he's important, but there's only one of him -->
    <user-overrides user="master-of-the-universe">
      <property name="acquireIncrement">1</property>
      <property name="initialPoolSize">1</property>
      <property name="minPoolSize">1</property>
      <property name="maxPoolSize">5</property>
      <property name="maxStatementsPerConnection">50</property>
    </user-overrides>
  </named-config>
</c3p0-config>
```

Precedence of Configuration Settings

c3p0 now permits configuration parameters to be set in a number of different ways and places. Fortunately, there is a clear order of precedence that determines which configuration will "take" in the event of conflicting settings. Conceptually, c3p0 goes down this list from top to bottom, using the first setting it finds.

Most applications will never use per-user or named configurations. For these applications, we present a simplified precedence hierarchy:

1. Configuration values programmatically set.
2. Configuration values set in system properties.
3. Configuration values taken from the default configuration of a `c3p0-config.xml` file.
4. Configuration values specified in a `c3p0.properties` file
5. Configuration values in a `c3p0.json` file, if and only if Typesafe config libraries are available.
6. Configuration values in a `c3p0.conf` file, if and only if Typesafe config libraries are available.
7. Configuration values in a `application.properties` file, if and only if Typesafe config libraries are available.
8. Configuration values in a `application.json` file, if and only if Typesafe config libraries are available.
9. Configuration values in a `application.conf` file, if and only if Typesafe config libraries are available.
10. Configuration values in a `reference.conf` file, if and only if Typesafe config libraries are available.
11. c3p0's hard-coded default values.

For applications that do use named and per-user configurations, here is the complete, normative precedence hierarchy:

1. User-specific overrides programmatically set via:
   - `ComboPooledDataSource.setUserOverridesAsString()`
   - `WrapperConnectionPoolDataSource.setUserOverridesAsString()`

   Note that programmatically setting user-specific overrides replaces all user-specific configuration taken from other sources. If you want to merge programmatic changes with preconfigured overrides, you'll have to use `getUserOverridesAsString()` and modify the original settings before replacing.
2. User-specific overrides taken from a DataSource's named configuration (specified in `c3p0-config.xml`)
3. User-specific overrides taken from the default configuration (specified in `c3p0-config.xml`)
4. Non-user-specific values programmatically set.
5. Non-user-specific values taken from a DataSource's named configuration (specified in `c3p0-config.xml`)
6. System property setting of configuration value.
7. Non-user-specific values taken from the default configuration (specified in `c3p0-config.xml`)
8. Configuration values in a `c3p0.conf` file, if and only if Typesafe config libraries are available.
9. Configuration values specified in a `c3p0.properties` file
10. Configuration values in a `application.conf` file, if and only if Typesafe config libraries are available.
11. Configuration values in a `reference.conf` file, if and only if Typesafe config libraries are available.
12. c3p0's hard-coded default values.

Appendix C: Hibernate-specific notes

Hibernate's C3P0ConnectionProvider renames 7 c3p0 configuration properties, which, if set in your hibernate configuration, **will override** any configuration you may have set in a `c3p0.properties` file:

c3p0-native property name	hibernate configuration key

c3p0.acquireIncrement	hibernate.c3p0.acquire_increment
c3p0.idleConnectionTestPeriod	hibernate.c3p0.idle_test_period
c3p0.initialPoolSize	*not available -- uses minimum size*
c3p0.maxIdleTime	hibernate.c3p0.timeout
c3p0.maxPoolSize	hibernate.c3p0.max_size
c3p0.maxStatements	hibernate.c3p0.max_statements
c3p0.minPoolSize	hibernate.c3p0.min_size
c3p0.testConnectionOnCheckout	hibernate.c3p0.validate *hibernate 2.x only!*

You can set any c3p0 properties in your hibernate config using the prefix `hibernate.c3p0`. For example

```
hibernate.c3p0.unreturnedConnectionTimeout=30
hibernate.c3p0.debugUnreturnedConnectionStackTraces=true
```

might be set to help debug Connection leaks.

You can always set c3p0 config in a `c3p0.properties` or `c3p0-config.xml` file (see "Overriding c3p0 defaults via c3p0.properties", "Overriding c3p0 defaults via c3p0-config.xml"), but *any configuration set in Hibernate config files will override c3p0-native configuation!*

Appendix D: Configuring c3p0 DataSources in Tomcat

Note:
Please see "Configuring To Avoid Memory Leaks On Hot Redeploy Of Client" if you experience memory leaks on Tomcat.

TL; DR: Set `privilegeSpawnedThreads` to `true` and set `contextClassLoaderSource` to `library`.

You can easily configure Apache's Tomcat web application server to use c3p0 pooled DataSources. Below is a Tomcat 5.0 sample config to get you started. It's a fragment of Tomcat's `conf/server.xml` file, which should be modified to suit and placed inside a `<Context>` element.

```
<Resource name="jdbc/pooledDS" auth="Container" type="com.mchange.v2.c3p0.ComboPooledDataSource" />
<ResourceParams name="jdbc/pooledDS">
  <parameter>
    <name>factory</name>
    <value>org.apache.naming.factory.BeanFactory</value>
  </parameter>
  <parameter>
    <name>driverClass</name>
    <value>org.postgresql.Driver</value>
  </parameter>
  <parameter>
    <name>jdbcUrl</name>
    <value>jdbc:postgresql://localhost/c3p0-test</value>
  </parameter>
  <parameter>
    <name>user</name>
    <value>swaldman</value>
  </parameter>
  <parameter>
    <name>password</name>
    <value>test</value>
  </parameter>
  <parameter>
    <name>minPoolSize</name>
    <value>5</value>
  </parameter>
  <parameter>
    <name>maxPoolSize</name>
    <value>15</value>
  </parameter>
  <parameter>
    <name>acquireIncrement</name>
    <value>5</value>
  </parameter>
</ResourceParams>
```

For Tomcat 5.5, try something like the following (thanks to Carl F. Hall for the sample!):

```
        <Resource auth="Container"
                  description="DB Connection"
                  driverClass="com.mysql.jdbc.Driver"
                  maxPoolSize="4"
                  minPoolSize="2"
                  acquireIncrement="1"
                  name="jdbc/TestDB"
                  user="test"
                  password="ready2go"
```

```
                  factory="org.apache.naming.factory.BeanFactory"
                  type="com.mchange.v2.c3p0.ComboPooledDataSource"
                  jdbcUrl="jdbc:mysql://localhost:3306/test?autoReconnect=true" />
```

The rest is standard J2EE stuff: You'll need to declare your DataSource reference in your `web.xml` file:

```
<resource-ref>
  <res-ref-name>jdbc/pooledDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

And you can access your DataSource from code within your web application like this:

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("java:comp/env/jdbc/pooledDS");
```

That's it!

Appendix E: JBoss-specific notes

To use c3p0 with JBoss:

1. Place c3p0's jar file in the `lib` directory of your jboss server instance (e.g. `${JBOSS_HOME}/server/default/lib`)
2. Modify the file below, and save it as `c3p0-service.xml` in the `deploy` directory of your jboss server (e.g. `${JBOSS_HOME}/server/default/deploy`). Note that parameters must be capitalized in this file, but otherwise they are defined as described above.

Note: Users of c3p0 jboss support prior to c3p0-0.9.1 please click here!

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE server>

<server>

   <mbean code="com.mchange.v2.c3p0.jboss.C3P0PooledDataSource"
          name="jboss:service=C3P0PooledDataSource">

      <attribute name="JndiName">java:PooledDS</attribute>
      <attribute name="JdbcUrl">jdbc:postgresql://localhost/c3p0-test</attribute>
      <attribute name="DriverClass">org.postgresql.Driver</attribute>
      <attribute name="User">swaldman</attribute>
      <attribute name="Password">test</attribute>

      <!-- Uncomment and set any of the optional parameters below -->
      <!-- See c3p0's docs for more info.                         -->

      <!-- <attribute name="AcquireIncrement">3</attribute>                        -->
      <!-- <attribute name="AcquireRetryAttempts">30</attribute>                   -->
      <!-- <attribute name="AcquireRetryDelay">1000</attribute>                    -->
      <!-- <attribute name="AutoCommitOnClose">false</attribute>                   -->
      <!-- <attribute name="AutomaticTestTable"></attribute>                       -->
      <!-- <attribute name="BreakAfterAcquireFailure">false</attribute>            -->
      <!-- <attribute name="CheckoutTimeout">0</attribute>                         -->
      <!-- <attribute name="ConnectionCustomizerClassName"></attribute>            -->
      <!-- <attribute name="ConnectionTesterClassName"></attribute>                -->
      <!-- <attribute name="Description">A pooled c3p0 DataSource</attribute>      -->
      <!-- <attribute name="DebugUnreturnedConnectionStackTraces">false</attribute> -->
      <!-- <attribute name="FactoryClassLocation"></attribute>                     -->
      <!-- <attribute name="ForceIgnoreUnresolvedTransactions">false</attribute>   -->
      <!-- <attribute name="IdleConnectionTestPeriod">0</attribute>                -->
      <!-- <attribute name="InitialPoolSize">3</attribute>                         -->
      <!-- <attribute name="MaxAdministrativeTaskTime">0</attribute>               -->
      <!-- <attribute name="MaxConnectionAge">0</attribute>                        -->
      <!-- <attribute name="MaxIdleTime">0</attribute>                             -->
      <!-- <attribute name="MaxIdleTimeExcessConnections">0</attribute>            -->
      <!-- <attribute name="MaxPoolSize">15</attribute>                            -->
      <!-- <attribute name="MaxStatements">0</attribute>                           -->
      <!-- <attribute name="MaxStatementsPerConnection">0</attribute>              -->
      <!-- <attribute name="MinPoolSize">0</attribute>                             -->
      <!-- <attribute name="NumHelperThreads">3</attribute>                        -->
      <!-- <attribute name="PreferredTestQuery"></attribute>                       -->
      <!-- <attribute name="TestConnectionOnCheckin">false</attribute>             -->
      <!-- <attribute name="TestConnectionOnCheckout">false</attribute>            -->
      <!-- <attribute name="UnreturnedConnectionTimeout">0</attribute>             -->

      <depends>jboss:service=Naming</depends>
   </mbean>
```

```
</server>
```

Appendix F: Oracle-specific API: createTemporaryBLOB() and createTemporaryCLOB()

These utilities are no longer supported. Please use `Connection.unwrap(...)` to access Oracle-specific APIs.

~~The Oracle thin JDBC driver provides a non-standard API for creating temporary BLOBs and CLOBs that requires users to call methods on the raw, Oracle-specific Connection implementation. Advanced users might use the raw connection operations described above to access this functionality, but a convenience class is available in a separate jar file (`c3p0-oracle-thin-extras-0.13.0.jar`) for easier access to this functionality. Please see the API docs for `com.mchange.v2.c3p0.dbms.OracleUtils` for details.~~

Appendix G: Legacy, configuring Connection Testing with a ConnectionTester

As of c3p0 0.10.0, c3p0's config property `connectionTesterClassName` defaults to `null`. When this property is `null`, c3p0 just uses the Java 6+ `Connection.isValid(int timeout)` method to test Connections.

But c3p0 is an old library, first published uner Java 1.3. Back in the day, we didn't have no stinking standard `Connection.isValid(int timeout)` method, and had to roll our own Connection tests. This Appendix documents that old but still supported style of Connection test management. To make this section relevant again, just 1) set `connectionTesterClassName` to its old default of `com.mchange.v2.c3p0.DefaultConnectionTester`; 2) set `connectionTesterClassName` to the name of a custom `ConnectionTester` implementation with a public no-arg constructor; 3) set a non-null `preferredTestQuery`; or 4) set a non-null `automaticTestTable`. If any `connectionTesterClassName` is explicitly set, the provided `ConnectionTester` implementation will be used. If `preferredTestQuery` or `automaticTestTable` are set, but no `connectionTesterClassName` is provided, an instance of `com.mchange.v2.c3p0.DefaultConnectionTester` will be used to perform Connection tests.

If any of these conditions apply, the following section documents how Connection testing with a ConnectionTester works in c3p0.

---

c3p0 can be configured to test the Connections that it pools in a variety of ways, to minimize the likelihood that your application will see broken or "stale" Connections. Pooled Connections can go bad for a variety of reasons -- some JDBC drivers intentionally "time-out" long-lasting database Connections; back-end databases or networks sometimes go down "stranding" pooled Connections; and Connections can simply become corrupted over time and use due to resource leaks, driver bugs, or other causes.

c3p0 provides users a great deal of flexibility in testing Connections, via the following configuration parameters:

- `automaticTestTable`
- `connectionTesterClassName`
- `idleConnectionTestPeriod`
- `preferredTestQuery`
- `testConnectionOnCheckin`
- `testConnectionOnCheckout`

`idleConnectionTestPeriod`, `testConnectionOnCheckout`, and `testConnectionOnCheckin` control when Connections will be tested. `automaticTestTable`, `connectionTesterClassName`, and `preferredTestQuery` control how they will be tested.

When configuring Connection testing, first try to minimize the cost of each test. If you are using a JDBC driver that you are certain supports the new(ish) jdbc4 API — and if you are using c3p0-0.9.5 or higher! — let your driver handle this for you. jdbc4 Connections include a method called `isValid()` that should be implemented as a fast, reliable Connection test. By default, c3p0 will use that method if it is present.

However, if your driver does not support this new-ish API, c3p0's default behavior is to test Connections by calling the `getTables()` method on a Connection's associated `DatabaseMetaData` object. This has the advantage of being very robust and working with any database, regardless of the database schema. However, a call to `DatabaseMetaData.getTables()` is often much slower than a simple database query, and using this test may significantly impair your pool's performance.

The simplest way to speed up Connection testing under a JDBC 3 driver (or a pre-0.9.5 version of c3p0) is to define a test query with the `preferredTestQuery` parameter. Be careful, however. Setting `preferredTestQuery` will lead to errors as Connection tests fail if the query target table does not exist in your database *prior to initialization of your DataSource*. Depending on your database and JDBC driver, a table-independent query like `SELECT 1` may (or may not) be sufficient to verify the Connection. If a table-independent query is not sufficient, instead of `preferredTestQuery`, you can set the parameter `automaticTestTable`. Using the name you provide, c3p0 will create an empty table, and make a simple query against it to test the database.

The most reliable time to test Connections is on check-out. But this is also the most costly choice from a client-performance perspective. Most applications should work quite reliably using a combination of `idleConnectionTestPeriod` and `testConnectionOnCheckin`. Both the idle test and the check-in test are performed asynchronously, which can lead to better performance, both perceived and actual.

For some applications, high performance is more important than the risk of an occasional database exception. In its default configuration, c3p0 does no Connection testing at all. Setting a fairly long `idleConnectionTestPeriod`, and not testing on checkout and check-in at all is an excellent, high-performance approach.

It is possible to customize how c3p0's `DefaultConnectionTester` tests when no `preferredTestQuery` or `automaticTestTable` are available. Please see Configuring DefaultConnectionTester.isValidTimeout and Configuring DefaultConnectionTester.QuerylessTestRunner.

Advanced users may define any kind of Connection testing they wish, by implementing a ConnectionTester and supplying the fully qualified name of the class as `connectionTesterClassName`. If you'd like your custom ConnectionTesters to honor and support the `preferredTestQuery` and `automaticTestTable` parameters, implement UnifiedConnectionTester, most conveniently by extending AbstractConnectionTester. See the api docs for more information.

If you know you want to use the jdbc4 `Connection.isValid()` method, but you want to set a timeout, consider writing a trivial extension of IsValidConnectionTester.

```
package com.mchange.v2.c3p0.example;

import com.mchange.v2.c3p0.util.IsValidOnlyConnectionTester;

public final class IsValidOnlyConnectionTester30 extends IsValidOnlyConnectionTester
{
  protected int getIsValidTimeout() { return 30; }
}
```

See? These really are trivial to write.

Simple advice on Connection testing

If you don't know what to do, try this:

1. If you know your driver supports the JDBC 4 `Connection.isValid(...)` method and you are using c3p0-0.9.5 or above, don't set a `preferredTestQuery`. If your driver does not support this method (or if you are not sure), try **`SELECT 1`** for your `preferredTestQuery`, if you are running MySQL or Postgres. For other databases, look for suggestions here. Leave `automatedTestTable` undefined.
2. Begin by setting `testConnectionOnCheckout` to `true` and get your application to run correctly and stably. If you are happy with your application's performance, *you can stop here!* This is the simplest, most reliable form of Connection-testing, but it does have a client-visible performance cost.
3. If you'd like to improve performance by eliminating Connection testing from clients' code path:
    a. Set `testConnectionOnCheckout` to `false`
    b. Set `testConnectionOnCheckin` to `true`
    c. Set `idleConnectionTestPeriod` to `30`, fire up you application and observe. This is a pretty robust setting, all Connections will tested on check-in and every 30 seconds thereafter while in the pool. Your application should experience broken or stale Connections only very rarely, and the pool should recover from a database shutdown and restart quickly. But there is some overhead associated with all that Connection testing.
    d. If database restarts will be rare so quick recovery is not an issue, consider reducing the frequency of tests by `idleConnectionTestPeriod` to, say, `300`, and see whether clients are troubled by stale or broken Connections. If not, stick with `300`, or try an even bigger number. Consider setting `testConnectionOnCheckin` back to false to avoid unnecessary tests on checkin. Alternatively, if your application does encounter bad Connections, consider reducing `idleConnectionTestPeriod` and set `testConnectionOnCheckin` back to `true`. There are no correct or incorrect values for these parameters: you are trading off overhead for reliability in deciding how frequently to test. The exact numbers are not so critical. It's usually easy to find configurations that perform well. It's rarely worth spending time in pursuit of "optimal" values here.

So, when should you stick with simple and reliable (Step 2 above), and when is it worth going for better performance (Step 3)? In general, it depends on how much work clients typically do with Connections once they check them out. If clients usually make complex queries and/or perform multiple operations, adding the extra cost of one fast test per checkout will not much affect performance. But if your application typically checks out a Connection and performs one simple query with it, throwing in an additional test can really slow things down.

That's nice in theory, but often people don't really have a good sense of how much work clients perform on average. The best thing to do is usually to try Step 3, see if it helps (however you measure performance), see if it hurts (is your application troubled by broken Connections? does it recover from database restarts well enough?), and then decide. You can always go back to simple, slow, and robust. Just set `testConnectionOnCheckout` to `true`, `testConnectionOnCheckin` to `false`, and set `idleConnectionTestPeriod` to `0`.

But do, always, be sure that your tests themselves are performant, either because your JDBC driver supports `Connection.isValid(...)` *or because you have set an efficient* `preferredTestQuery` *!!!*

Back to Contents