Java Connection Pooling

The addition of JDBC connection pooling to your application usually involves little or no code modification and can often **provide** significant **benefits** in terms of application **performance**, **concurrency** and **scalability**. Improvements such as these can become **especially important** when your application is tasked with servicing many **concurrent** users within the requirements of sub second response time. By adhering to a small number of relatively simple connection pooling best practices your application can quickly and easily take effective advantage of connection pooling.

How It Works?

Connection pooling **creates** connections **minimum size** which is specified in the configuration settings of the pool. When necessary to increase the connection size to serve multiple users at the same time, the size of the connection pooling **increased to maximum** connection size specified in the configuration settings of the pool. Connection pool simply **creates** connections to database and **never closes them except as required**. Therefore, there **wont be performance degradation**. Connection pool is generally **created** at the **initialization** of a web application or at the **main** method of a **desktop** application as shown in the following codes:

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.ConnectionPoolDataSource;
import javax.sql.PooledConnection;

public class MainClass {
  public static void main(String[] args) {
    Connection connection = null;
    Statement statement = null;
    ResultSet resultSet = null;

    try {
      connection = getConnection();
      // Do work with connection
      statement = connection.createStatement();
      String selectEmployeesSQL = "SELECT * FROM employees";
      resultSet = statement.executeQuery(selectEmployeesSQL);

      while (resultSet.next()) {
        printEmployee(resultSet);
      }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (resultSet != null) {
        try {
          resultSet.close();
        } catch (SQLException e) {
        } // nothing we can do
      }
      if (statement != null) {
        try {
          statement.close();
        } catch (SQLException e) {
        } // nothing we can do
      }
      if (connection != null) {
        try {
          connection.close();
        } catch (SQLException e) {
        } // nothing we can do
      }
    }
  }

  private static Connection getConnection() throws NamingException, SQLException {
    InitialContext initCtx = createContext();
    String jndiName = "HrDS";
    ConnectionPoolDataSource dataSource = (ConnectionPoolDataSource) initCtx.lookup(jndiName);
    PooledConnection pooledConnection = dataSource.getPooledConnection();
    return pooledConnection.getConnection(); // Obtain connection from pool
  }

  private static InitialContext createContext() throws NamingException {
    Properties env = new Properties();
    env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.RegistryContextFactory");
    env.put(Context.PROVIDER_URL, "rmi://localhost:1099 (rmi://localhost:1099) ");
    InitialContext context = new InitialContext(env);
    return context;
  }

  private static void printEmployee(ResultSet resultSet) throws SQLException {
    System.out.print(resultSet.getInt("employee_id")+", ");
    System.out.print(resultSet.getString("last_name")+", ");
    System.out.print(resultSet.getString("first_name")+", ");
    System.out.println(resultSet.getString("email"));
  }

}
```

Software Object Pooling

There are many scenarios in software architecture where some **type** of **object** pooling is employed as a technique to improve application performance. Object **pooling** is **effective** for two simple **reasons**:

First, the run time creation of new software objects is often more expensive in terms of performance and memory than the reuse of previously created objects.

Second, garbage collection is an expensive process so when we reduce the number of objects to clean up we generally reduce the garbage collection load.

As the saying goes, there is no such thing as a free lunch and this maxim is also **true** with **object** pooling. Object pooling does **require** additional overhead for such tasks as managing the state of the **object** pool, issuing objects to the application and recycling used objects. Therefore objects that don't have **short** lifetimes in your application may not be good choices for **object** pooling since their low rate of reuse may not warrant the overhead of pooling.

However, objects that do have **short** lifetimes are often excellent candidates for pooling. In a pooling scenario your application first creates an **object** pool that can both cache pooled objects and issue objects that are not in use back to the application. For example, pooled objects could be database connections, process threads, server sockets or any other kind of **object** that may be expensive to create from scratch. As your application first starts asking the pool for objects they will be newly created but when the application has finished with the **object** it is returned to the pool rather than destroyed. At this point the benefits of **object** pooling will be realized since, now as the application needs more objects, the pool will be able to issue recycled objects that have previously been returned by the application.

JDBC Connection Pooling

JDBC connection pooling is conceptually similar to any other form of **object** pooling. Database connections are often expensive to create because of the overhead of establishing a network connection and initializing a database connection session in the back end database. In turn, connection session initialization often requires time consuming processing to perform user authentication, establish transactional contexts and establish other aspects of the session that are required for subsequent database usage.

Additionally, the database's ongoing management of all of its connection sessions can impose a major limiting factor on the scalability of your application. Valuable database resources such as locks, memory, cursors, transaction logs, statement handles and temporary tables all tend to increase based on the number of concurrent connection sessions.

All in all, JDBC database connections are both expensive to initially create and then maintain over time. Therefore, as we shall see, they are an ideal resource to pool.

If **your application** runs within a **J2EE** environment and acquires **JDBC** connections from an appserver defined datasource then **your application** is probably **already using** connection **pooling**. This fact also illustrates an important characteristic of a best practices pooling implementation -- your application is not even aware it's using it! Your **J2EE** application simply acquires **JDBC** connections from the datasource, does some work on the connection then closes the connection. Your application's use of connection pooling is transparent. The characteristics of the connection pool can be tweaked and tuned by your appserver's administrator without the application ever needing to know.

If your **application** is **not J2EE** based then you may need to investigate using a **standalone connection pool manager**. Connection pool implementations are available from **JDBC** driver vendors and a number of other sources.

JDBC Connection Scope

How should your application manage the life cycle of JDBC connections? Asked another way, this question really asks - what is the scope of the JDBC connection **object** within your application? Let's consider a servlet that performs JDBC access. One possibility is to define the connection with servlet scope as follows.

```java
import java.sql.*;

public class JDBCServlet extends HttpServlet {

    private Connection connection;

    public void init(ServletConfig c) throws ServletException {
      //Open the connection here
    }

    public void destroy() {
     //Close the connection here
    }

    public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {
      //Use the connection here
      Statement stmt = connection.createStatement();
      ..<do JDBC work>..
  }
}
```

Using this approach the servlet creates a JDBC connection when it is loaded and destroys it when it is unloaded. The **doGet()** method has immediate access to the connection since it has servlet scope. **However** the database **connection** is kept **open** for the **entire lifetime** of the servlet and that the database will have to **retain** an **open** connection for **every** user that is connected to your application. If your application supports a **large** number of concurrent **users** its **scalability** will be **severely limited!**

Method Scope Connections

To **avoid** the **long** life time of the JDBC connection in the above example we can change the connection to have method scope as follows.

```java
public class JDBCServlet extends HttpServlet {

  private Connection getConnection() throws SQLException {
    ..<create a JDBC connection>..
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {
    try {
      Connection connection = getConnection();
      ..<do JDBC work>..
      connection.close();
    }
    catch (SQLException sqlException) {
      sqlException.printStackTrace();
    }
  }
}
```

This approach represents a **significant improvement** over our first example **because** now the connection's life time is reduced to the time it takes to execute **doGet()**. The number of connections to the back end database at any instant is reduced to the number of users who are concurrently executing **doGet()**. **However** this example will **create** and **destroy** a **lot more connections** than the first example and this could easily become a performance problem.

In order to **retain** the advantages of a method scoped connection **but reduce** the performance hit of **creating** and **destroying** a large number of connections we **now utilize connection pooling** to arrive at our finished example that illustrates the best practices of connecting pool usage.

```java
import java.sql.*;
import javax.sql.*;

public class JDBCServlet extends HttpServlet {

  private DataSource datasource;

  public void init(ServletConfig config) throws ServletException {
    try {
      // Look up the JNDI data source only once at init time
      Context envCtx = (Context) new InitialContext().lookup("java:comp/env");
      datasource = (DataSource) envCtx.lookup("jdbc/MyDataSource");
    }
    catch (NamingException e) {
      e.printStackTrace();
    }
  }

  private Connection getConnection() throws SQLException {
    return datasource.getConnection();
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {
    Connection connection=null;
    try {
      connection = getConnection();
      ..<do JDBC work>..
    }
    catch (SQLException sqlException) {
      sqlException.printStackTrace();
    }
    finally {
      if (connection != null)
        try {connection.close();} catch (SQLException e) {}
      }
    }
  }
}
```

This **approach** uses the connection only for the minimum time the servlet requires it and also **avoids creating and destroying** a large number of physical database connections. The connection best practices that we have used are:

a. JNDI datasource is used as a factory for connections. The **JNDI** datasource is instantiated only once in **init()** since **JNDI** lookup can also be slow. **JNDI** should be configured so that the bound datasource **implements** connecting pooling. Connections issued from the pooling datasource will be returned to the pool when closed.

b. We have moved the **connection.close()** into a **finally** block to **ensure** that the connection is closed **even if** an **exception** occurs during the **doGet()** JDBC processing. This practice is **essential** essential when using a **connection pool**. If a **connection** is **not closed** it will **never** never be **returned** to the connection pool and **not** become **available for reuse**. A **finally** block can also guarantee the closure of resources attached to JDBC statements and result sets when unexpected exceptions occur. Just call **close()** on these objects also.

Connection Pool Tuning

One of the major **advantages** of using a connection pool is that characteristics of the pool can be changed without affecting the application. If your application confines itself to using generic JDBC you could even point it at a different vendor's database without changing any code! Different pool implementations will provide different settable properties to tune the connection pool. Typical properties include the number of initial connections, the minimum and maximum number of connections that can be present at any time and a mechanism to purge connections that have been idle for a specific period of time.

In general, **optimal performance** is attained when the pool in its **steady state** contains just enough connections to service all concurrent connection requests without having to create new physical database connections. If the pooling implementation supports **purging idle** connections it can optimize its size over time to accommodate **varying application loads** over the course of a day. For example, **scaling up** the number of connections cached in the pool **during business hours** then dynamically **reducing** the pool **size** after business hours.

Benefits:

It allows you to have **many** connections **open**, so when a thread needs one it can **reverse** the connection for use, when this thread finish he return the connection so it can be used in other threads (**without** without having to **open** and **close** the connection **every time**). In this way if you have a connection available a thread does not has to wait until another finish. As you can see is up to your app needs, I would advise you to analyze what you need first and then you can choose if you will use a connection pool or not.

Also it depends if you are going to do two or more queries into the database at the same time, in this case you should use the connection pool. Although the application only makes one query at the same time and does not **require** to be used for **long** periods of time, you should not worry about it, and just keep the connection open until you close the app.

Using a pool connection you will have multiples connection that you can assign to a thread, if a thread is using one of the connections this connection can't be used in another thread, however since you have multiple connections you can assign another one to this thread, whenever you are done whit the connection you just "close it" (which it does not really close it, the connection it's just returned to the pool, so you can assign it to another thread)

If you decide to use just one connection, you won't be able to use this connection in more than one thread.

```
1. How does we make sure that connection pool doesn't return the same object which is already in use?
```

Because if a connection has been borrowed from the pool and not returned yet, it's just not in the pool and can't be assigned to another client of the pool (resources are removed from the pool until they are returned).

```
2.How happens if client closed the connection after taking it out from Connection pool?
```

The connection a client gets from a pool is not really a `java.sql.Connection` (http://java.sun.com/javase/6/docs/api/java/sql/Connection.html) , it's a wrapper (a proxy) for a `java.sql.Connection` (http://java.sun.com/javase/6/docs/api/java/sql/Connection.html) that customizes the behavior of some methods. The `close()` method is one of them and does **not** close the `Connection` instance but returns it to the pool.

JDBC Connection Pool Settings

For optimum performance of database-intensive applications, tune the JDBC Connection Pools managed by the Application Server. These connection pools maintain numerous live database connections that can be reused to reduce the overhead of opening and closing database connections. This section describes how to tune JDBC Connection Pools to improve performance.

J2EE applications use JDBC Resources to obtain connections that are maintained by the JDBC Connection Pool. More than one JDBC Resource is allowed to refer to the same JDBC Connection Pool. In such a case, the physical connection pool is shared by all the resources.

Monitoring JDBC Connection Pools

As the creation of JDBC connections are expensive and frequently cause performance bottlenecks in applications, it is crucial to monitor how a JDBC connection pool is releasing and creating new connections, and how many threads are waiting to retrieve a connection from a particular pool. Statistics-gathering is enabled by **default** for JDBC Connection Pools. The following attributes are monitored:

averageConnWaitTime (count): Average wait time of connections for successful connection request attempts to the connector connection pool.

connectionRequestWaitTime (**range):** The longest and shortest wait times of connection requests.

numConnAcquired (count): Number of logical connections acquired from the pool.

numConnCreated (count): Number of physical connections created since the last reset.

numConnDestroyed (count): Number of physical connections destroyed since the last reset.

numConnFailedValidation (count): Number of connections that failed validation.

numConnFree (count): Number of free connections in the pool.

numConnReleased (count): Number of logical connections released to the pool.

numConnTimedOut (bounded range): Number of connections in the pool that have timed out.

numConnUsed (**range):** Number of connections that have been used.

waitQueueLength (count): Number of connection requests in the queue waiting to be serviced.

To get the statistics, use these commands:

```
asadmin get --monitor=true
serverInstance.resources.jdbc-connection-pool.*asadmin get
--monitor=true serverInstance.resources.jdbc-connection-pool. poolName.* *
```

Tuning JDBC Connection Pools

Set JDBC Connection Pool attributes with the Admin Console under Resources > JDBC > Connection Pools > **PoolName.** The following attributes affect performance:

Pool Size Settings

Timeout Settings

Isolation Level Settings

Connection Validation Settings

Pool Size Settings

The following settings control the size of the connection pool:

Initial and Mimimum Pool Size

Size of the pool when created, and its minimum allowable size.

Maximum Pool Size

Upper limit of size of the pool.

Pool Resize Quantity

Number of connections to be removed when the idle timeout expires. Connections that have idled for longer than the timeout are candidates for removal. When the pool size reaches the initial and minimum pool size, removal of connections stops.

The following table summarizes pros and cons to consider when sizing connection pools.

Connection pool	Pros	Cons
Small Connection pool	Faster access on the connection table	May not have enough connections to satisfy requests. Requests may spend more time in the queue
Large Connection pool	More connections to fulfill requests Requests will spend less (or no) time in the queue	Slower access on the connection table.

Timeout Settings

There are two timeout settings:

Max Wait Time: Amount of time the caller (the code requesting a connection) will wait before getting a connection timeout. The **default** is 60 seconds. A value of **zero** forces caller to wait **indefinitely**.

To improve **performance** set Max Wait Time to **zero(0)**. This essentially blocks the caller thread until a connection becomes available. Also, this allows the server to alleviate the task of tracking the elapsed wait time for each request and increases performance.

Idle Timeout: Maximum time in seconds that a connection can remain idle in the pool. After this time, the pool can close this connection. This property does not control connection timeouts on the database server.

Keep this timeout shorter than the database server timeout (if such timeouts are configured on the database), to prevent accumulation of unusable connection in Application Server.
For best **performance**, set idle timeout to **zero(0)** seconds, so that idle connections will **not be removed**. This ensures that there is normally no penalty in creating new connections and disables the idle monitor thread. However, there is a **risk** that the database server will **reset** a connection that is **unused for too long**.

Isolation Level Settings

Two settings control the connection pool's transaction isolation level on the database server:

Transaction Isolation Level: specifies the transaction isolation level of the pooled database connections. If this parameter is unspecified, the pool uses the **default** isolation level provided by the JDBC Driver.

Isolation Level Guaranteed: Guarantees that every connection obtained from the pool has the isolation specified by the Transaction Isolation Level parameter. Applicable only when the Transaction Isolation Level is specified. The **default** value is true.

This setting can have some performance impact on some JDBC drivers. Set to **false** when certain that the application does not change the isolation level before returning the connection.

Avoid specifying Transaction Isolation Level. If that is not possible, consider setting Isolation Level Guaranteed to **false** and make sure applications do not programmatically alter the connections' isolation level.

If you must specify isolation level, specify the best-performing level possible. The isolation levels listed from best performance to worst are:

1. READ_UNCOMMITTED
2. READ_COMMITTED
3. REPEATABLE_READ
4. SERIALIZABLE

Choose the isolation level that provides the best performance, yet still meets the concurrency and consistency needs of the application.

Connection Validation Settings

The following settings determine whether and how the pool performs connection validation.

Connection Validation Required

If **true**, the pool **validates** connections (checks to find out if they are **usable**) before providing them to an application.

If possible, **keep** the **default** value, **false**. Requiring connection **validation forces** the server to apply the validation algorithm **every time** the pool returns a connection, which adds **overhead** to the latency of **getConnection()**. If the database connectivity is **reliable**, you can **omit validation**.

Validation Method

Type of connection validation to perform. Must be one of:

auto-commit: attempt to perform an auto-commit on the connection.

metadata: attempt to get metadata from the connection.

table (performing a query on a specified table). Must also set Table Name. You may have to use this method if the JDBC driver caches calls to **setAutoCommit()** and **getMetaData().**

Not: Because many JDBC drivers cache the results of these calls, they do not always provide reliable validations. Check with the driver vendor to determine whether these calls are cached or not.

Table Name

Table name to query when Validation Method is "table."

Close All Connections On Any Failure

Whether to close all connections in the pool if a single validation check fails. The **default** is false. One attempt will be made to re-establish failed connections.