# SQL Server Connection Pooling (ADO.NET)

**.NET Framework 4.5** 16 out of 23 rated this helpful

Connecting to a database server typically consists of several time-consuming steps. A physical channel such as a socket or a named pipe must be established, the initial handshake with the server must occur, the connection string information must be parsed, the connection must be authenticated by the server, checks must be run for enlisting in the current transaction, and so on.

In practice, most applications use only one or a few different configurations for connections. This means that during application execution, many identical connections will be repeatedly opened and closed. To minimize the cost of opening connections, ADO.NET uses an optimization technique called *connection pooling*.

Connection pooling reduces the number of times that new connections must be opened. The *pooler* maintains ownership of the physical connection. It manages connections by keeping alive a set of active connections for each given connection configuration. Whenever a user calls **Open** on a connection, the pooler looks for an available connection in the pool. If a pooled connection is available, it returns it to the caller instead of opening a new connection. When the application calls **Close** on the connection, the pooler returns it to the pooled set of active connections instead of closing it. Once the connection is returned to the pool, it is ready to be reused on the next **Open** call.

Only connections with the same configuration can be pooled. ADO.NET keeps several pools at the same time, one for each configuration. Connections are separated into pools by connection string, and by Windows identity when integrated security is used. Connections are also pooled based on whether they are enlisted in a transaction. When using ChangePassword, the SqlCredential instance affects the connection pool. Different instances of SqlCredential will use different connection pools, even if the user ID and password are the same.

Pooling connections can significantly enhance the performance and scalability of your application. By default, connection pooling is enabled in ADO.NET. Unless you explicitly disable it, the pooler optimizes the connections as they are opened and closed in your application. You can also supply several connection string modifiers to control connection pooling behavior. For more information, see "Controlling Connection Pooling with Connection String Keywords" later in this topic.

**Note**

When connection pooling is enabled, and if a timeout error or other login error occurs, an exception will be thrown and subsequent connection attempts will fail for the next five seconds, the "blocking period". If the application attempts to connect within the blocking period, the first exception will be thrown again. Subsequent failures after a blocking period ends will result in a new blocking periods that is twice as long as the previous blocking period, up to a maximum of one minute.

## Pool Creation and Assignment

When a connection is first opened, a connection pool is created based on an exact matching algorithm that associates the pool with the connection string in the connection. Each connection pool is associated with a distinct connection string. When a new connection is opened, if the connection string is not an exact match to an existing pool, a new pool is created. Connections are pooled per process, per application domain, per connection string and when integrated security is used, per Windows identity. Connection strings must also be an exact match; keywords supplied in a different order for the same connection will be pooled separately.

In the following C# example, three new SqlConnection objects are created, but only two connection pools are required to manage them. Note that the first and second connection strings differ by the value assigned for Initial Catalog.

```
using (SqlConnection connection = new SqlConnection(
  "Integrated Security=SSPI;Initial Catalog=Northwind"))
    {
        connection.Open();
        // Pool A is created.
    }

using (SqlConnection connection = new SqlConnection(
  "Integrated Security=SSPI;Initial Catalog=pubs"))
    {
        connection.Open();
        // Pool B is created because the connection strings differ.
    }

using (SqlConnection connection = new SqlConnection(
  "Integrated Security=SSPI;Initial Catalog=Northwind"))
    {
        connection.Open();
        // The connection string matches pool A.
    }
```

If **MinPoolSize** is either not specified in the connection string or is specified as zero, the connections in the pool will be closed after a period of inactivity. However, if the specified **MinPoolSize** is greater than zero, the connection pool is not destroyed until the **AppDomain** is unloaded and the process ends. Maintenance of inactive or empty pools involves minimal system overhead.

**Note**

The pool is automatically cleared when a fatal error occurs, such as a failover.

## Adding Connections

A connection pool is created for each unique connection string. When a pool is created, multiple connection objects are created and added to the pool so that the minimum pool size requirement is satisfied. Connections are added to the pool as needed, up to the maximum pool size specified (100 is the default). Connections are released back into the pool when they are closed or disposed.

When a SqlConnection object is requested, it is obtained from the pool if a usable connection is available. To be usable, a connection must be unused, have a matching transaction context or be unassociated with any transaction context, and have a valid link to the server.

The connection pooler satisfies requests for connections by reallocating connections as they are released back into the pool. If the maximum pool size has been reached and no usable connection is available, the request is queued. The pooler then tries to reclaim any connections until the time-out is reached (the default is 15 seconds). If the pooler cannot satisfy the request before the connection times out, an exception is thrown.

**Caution**

We strongly recommend that you always close the connection when you are finished using it so that the connection will be returned to the pool. You can do this using either the **Close** or **Dispose** methods of the **Connection** object, or by opening all connections inside a **using** statement in C#, or a **Using** statement in Visual Basic. Connections that are not explicitly closed might not be added or returned to the pool. For more information, see using Statement (C# Reference) or How to: Dispose of a System Resource (Visual Basic) for Visual Basic.

**Note**

Do not call **Close** or **Dispose** on a **Connection**, a **DataReader**, or any other managed object in the **Finalize** method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a **Finalize** method in your class definition. For more information, see Garbage Collection.

**Note**

Login and logout events will not be raised on the server when a connection is fetched from or returned to the connection pool. This is because the connection is not actually closed when it is returned to the connection pool. For more information, see Audit Login Event Class and Audit Logout Event Class in SQL Server Books Online.

## Removing Connections

The connection pooler removes a connection from the pool after it has been idle for approximately 4-8 minutes, or if the pooler detects that the connection with the server has been severed. Note that a severed connection can be detected only after attempting to communicate with the server. If a connection is found that is no longer connected to the server, it is marked as invalid. Invalid connections are removed from the connection pool only when they are closed or reclaimed.

If a connection exists to a server that has disappeared, this connection can be drawn from the pool even if the connection pooler has not detected the severed connection and marked it as invalid. This is the case because the overhead of checking that the connection is still valid would eliminate the benefits of having a pooler by causing another round trip to the server to occur. When this occurs, the first attempt to use the connection will detect that the connection has been severed, and an exception is thrown.

## Clearing the Pool

ADO.NET 2.0 introduced two new methods to clear the pool: ClearAllPools and ClearPool. **ClearAllPools** clears the connection pools for a given provider, and **ClearPool** clears the connection pool that is associated with a specific connection. If there are connections being used at the time of the call, they are marked appropriately. When they are closed, they are discarded instead of being returned to the pool.

## Transaction Support

Connections are drawn from the pool and assigned based on transaction context. Unless Enlist=false is specified in the connection string, the connection pool makes sure that the connection is enlisted in the Current context. When a connection is closed and returned to the pool with an enlisted **System.Transactions** transaction, it is set aside in such a way that the next request for that connection pool with the same **System.Transactions** transaction will return the same connection if it is available. If such a request is issued, and there are no pooled connections available, a connection is drawn from the non-transacted part of the pool and enlisted. If no connections are available in either area of the pool, a new connection is created and enlisted.

When a connection is closed, it is released back into the pool and into the appropriate subdivision based on its transaction context. Therefore, you can close the connection without generating an error, even though a distributed transaction is still pending. This allows you to commit or abort the distributed transaction later.

## Controlling Connection Pooling with Connection String Keywords

The **ConnectionString** property of the SqlConnection object supports connection string key/value pairs that can be used to adjust the behavior of the connection pooling logic. For more information, see ConnectionString.

## Pool Fragmentation

Pool fragmentation is a common problem in many Web applications where the application can create a large number of pools that are not freed until the process exits. This leaves a large number of connections open and consuming memory, which results in poor performance.

### Pool Fragmentation Due to Integrated Security

Connections are pooled according to the connection string plus the user identity. Therefore, if you use Basic authentication or Windows Authentication on the Web site and an integrated security login, you get one pool per user. Although this improves the performance of subsequent database requests for a single user, that user cannot take advantage of connections made by other users. It also results in at least one connection per user to the database server. This is a side effect of a particular Web application architecture that developers must weigh against security and auditing requirements.

### Pool Fragmentation Due to Many Databases

Many Internet service providers host several Web sites on a single server. They may use a single database to confirm a Forms authentication login and then open a connection to a specific database for that user or group of users. The connection to the authentication database is pooled and used by everyone. However, there is a separate pool of connections to each database, which increase the number of connections to the server.

This is also a side-effect of the application design. There is a relatively simple way to avoid this side effect without compromising security when you connect to SQL Server. Instead of connecting to a separate database for each user or group, connect to the same database on the server and then execute the Transact-SQL USE statement to change to the desired database. The following code fragment demonstrates creating an initial connection to the **master** database and then switching to the desired database specified in the databaseName string variable.

**C#**

```
// Assumes that command is a SqlCommand object and that
// connectionString connects to master.
command.Text = "USE DatabaseName";
using (SqlConnection connection = new SqlConnection(
  connectionString))
  {
    connection.Open();
    command.ExecuteNonQuery();
  }
```

## Application Roles and Connection Pooling

After a SQL Server application role has been activated by calling the **sp_setapprole** system stored procedure, the security context of that connection cannot be reset. However, if pooling is enabled, the connection is returned to the pool, and an error occurs when the pooled connection is reused. For more information, see the Knowledge Base article, "SQL application role errors with OLE DB resource pooling."

### Application Role Alternatives

We recommend that you take advantage of security mechanisms that you can use instead of application roles. For more information, see Creating Application Roles in SQL Server.

## See Also

Concepts
Performance Counters [from BPUEDev11]
Other Resources
Connection Pooling
SQL Server and ADO.NET
ADO.NET Managed Providers and DataSet Developer Center