.NET Framework Developer's Guide

**Connection Pooling (ADO.NET)**

Connecting to a data source can be time consuming. To minimize the cost of opening connections, ADO.NET uses an optimization technique called *connection pooling*, which minimizes the cost of repeatedly opening and closing connections. Connection pooling is handled differently for the .NET Framework data providers.

http://i.msdn.microsoft.com/Global/Images/clear.gif In This Section

[SQL Server Connection Pooling (ADO.NET)](http://msdn.microsoft.com/en-us/library/8xx3tyca.aspx)

Provides an overview of connection pooling and describes how connection pooling works in SQL Server.

[OLE DB, ODBC, and Oracle Connection Pooling (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms254502.aspx)

Describes connection pooling for the .NET Framework Data Provider for OLE DB, the .NET Framework Data Provider for ODBC, and the .NET Framework Data Provider for Oracle.

.NET Framework Developer's Guide

**SQL Server Connection Pooling (ADO.NET)**

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.

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.

http://i.msdn.microsoft.com/Global/Images/clear.gif 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](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) 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.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl07other');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl07other');)

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.

|  |
| --- |
| **8xx3tyca.alert_note(en-us,VS.90).gifNote:** |
| The pool is automatically cleared when a fatal error occurs, such as a failover. |

http://i.msdn.microsoft.com/Global/Images/clear.gif 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](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) 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.

|  |
| --- |
| **8xx3tyca.alert_caution(en-us,VS.90).gifCaution:** |
| 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)](http://msdn.microsoft.com/en-us/library/yh598w02.aspx) or [How to: Dispose of a System Resource](http://msdn.microsoft.com/en-us/library/wydd5hkd.aspx) for Visual Basic. |
| **8xx3tyca.alert_note(en-us,VS.90).gifNote:** |
| 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](http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx). |

|  |
| --- |
| **8xx3tyca.alert_note(en-us,VS.90).gifNote:** |
| 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](http://msdn2.microsoft.com/en-us/library/ms190260.aspx) and [Audit Logout Event Class](http://msdn2.microsoft.com/en-us/library/ms175827.aspx) in SQL Server Books Online. |

http://i.msdn.microsoft.com/Global/Images/clear.gif Removing Connections

The connection pooler removes a connection from the pool after it has been idle for a long time, 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.

http://i.msdn.microsoft.com/Global/Images/clear.gif Clearing the Pool

ADO.NET 2.0 introduced two new methods to clear the pool: [ClearAllPools](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.clearallpools.aspx) and [ClearPool](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.clearpool.aspx). **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.

http://i.msdn.microsoft.com/Global/Images/clear.gif 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](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.current.aspx) 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.

For more information on transactions and connection pooling, see [System.Transactions Integration with SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms172070.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Controlling Connection Pooling with Connection String Keywords

The **ConnectionString** property of the [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) 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](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif 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.

Visual Basic

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl53VisualBasic');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl53VisualBasic');)

' Assumes that command is a valid SqlCommand object and that

' connectionString connects to master.

command.Text = "USE DatabaseName"

Using connection As New SqlConnection(connectionString)

connection.Open()

command.ExecuteNonQuery()

End Using

C#

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl54CSharp');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl54CSharp');)

// 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();

}

http://i.msdn.microsoft.com/Global/Images/clear.gif 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](http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q229564)."

**Application Role Alternatives**

If you are using SQL Server 2005, we recommend that you take advantage of the new security mechanisms that you can use instead of application roles. For more information, see [Creating Application Roles in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669062.aspx).

.NET Framework Developer's Guide

**OLE DB, ODBC, and Oracle Connection Pooling (ADO.NET)**

Pooling connections can significantly enhance the performance and scalability of your application. This section discusses connection pooling for the .NET Framework data providers for OLE DB, ODBC and Oracle.

http://i.msdn.microsoft.com/Global/Images/clear.gif Connection Pooling for OleDb

The .NET Framework Data Provider for OLE DB automatically pools connections using OLE DB session pooling. Connection string arguments can be used to enable or disable OLE DB services including pooling. For example, the following connection string disables OLE DB session pooling and automatic transaction enlistment.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl05other');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl05other');)

Provider=SQLOLEDB;OLE DB Services=-4;Data Source=localhost;Integrated Security=SSPI;

We recommend that you always close or dispose of a connection when you are finished using it in order to return the connection to the pool. Connections that are not explicitly closed may not get returned to the pool. For example, a connection that has gone out of scope but that has not been explicitly closed will only be returned to the connection pool if the maximum pool size has been reached and the connection is still valid.

For more information about OLE DB session or resource pooling, as well as how to disable pooling by overriding OLE DB provider service defaults, see the [OLE DB Programmer's Guide](http://go.microsoft.com/fwlink/?linkid=45232) in the MSDN library.

http://i.msdn.microsoft.com/Global/Images/clear.gif Connection Pooling for Odbc

Connection pooling for the .NET Framework Data Provider for ODBC is managed by the ODBC Driver Manager that is used for the connection, and is not affected by the .NET Framework Data Provider for ODBC.

To enable or disable connection pooling, open ODBC Data Source Administrator in the Administrative Tools folder of Control Panel. The Connection Pooling tab allows you to specify connection pooling parameters for each ODBC driver installed. Note that connection pooling changes for a specific ODBC driver affect all applications that use that ODBC driver.

For more information on ODBC connection pooling, see [INFO: Frequently Asked Questions About ODBC Connection Pooling](http://support.microsoft.com/kb/169470) in the MSDN library.

http://i.msdn.microsoft.com/Global/Images/clear.gif Connection Pooling for OracleClient

The .NET Framework Data Provider for Oracle provides connection pooling automatically for your ADO.NET client application. You can also supply several connection string modifiers to control connection pooling behavior (see "Controlling Connection Pooling with Connection String Keywords," later in this topic).

**Pool Creation and Assignment**

When a connection is 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.

Once created, connection pools are not destroyed until the active process ends. Maintaining inactive or empty pools uses very few system resources.

**Connection Addition**

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.

When an [OracleConnection](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracleconnection.aspx) object is requested, it is obtained from the pool if a usable connection is available. To be usable, the connection must currently be unused, have a matching transaction context or not be associated with any transaction context, and have a valid link to the server.

If the maximum pool size has been reached and no usable connection is available, the request is queued. The connection pooler satisfies these requests by reallocating connections as they are released back into the pool. Connections are released back into the pool when they are closed or disposed.

**Connection Removal**

The connection pooler removes a connection from the pool after it has been idle for an extended period of time, or if the pooler detects that the connection with the server has been severed. Note that this 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. The connection pooler periodically scans connection pools looking for objects that have been released to the pool and are marked as invalid. These connections are then permanently removed.

If a connection exists to a server that has disappeared, this connection can be drawn from the pool if the connection pooler has not detected the severed connection and marked it as invalid. When this occurs, an exception is generated. However, you must still close the connection in order to release it back into the pool.

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](http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx).

**Transaction Support**

Connections are drawn from the pool and assigned based on transaction context. The context of the requesting thread and the assigned connection must match. Therefore, each connection pool is actually subdivided into connections with no transaction context associated with them, and into N subdivisions that each contain connections with a particular transaction context.

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 at a later time.

**Controlling Connection Pooling with Connection String Keywords**

The [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracleconnection.connectionstring.aspx) property of the [OracleConnection](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracleconnection.aspx) object supports connection string key/value pairs that can be used to adjust the behavior of the connection pooling logic.

The following table describes the [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracleconnection.connectionstring.aspx) values you can use to adjust connection pooling behavior.

|  |  |  |
| --- | --- | --- |
| **Name** | **Default** | **Description** |
| **Connection Lifetime** | 0 | When a connection is returned to the pool, its creation time is compared with the current time, and the connection is destroyed if that time span (in seconds) exceeds the value specified by **Connection Lifetime**. This is useful in clustered configurations to force load balancing between a running server and a server just brought online.  A value of zero (0) will cause pooled connections to have the maximum time-out. |
| **Enlist** | 'true' | When **true**, the pooler automatically enlists the connection in the current transaction context of the creation thread if a transaction context exists. |
| **Max Pool Size** | 100 | The maximum number of connections allowed in the pool. |
| **Min Pool Size** | 0 | The minimum number of connections maintained in the pool. |
| **Pooling** | 'true' | When **true**, the connection is drawn from the appropriate pool, or if necessary, created and added to the appropriate pool. |

.NET Framework Developer's Guide

**Performance Counters (ADO.NET)**

ADO.NET 2.0 introduced expanded support for performance counters that includes support for both [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx) and [System.Data.OracleClient](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.aspx). The [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx) performance counters available in previous versions of ADO.NET have been deprecated and replaced with the new performance counters discussed in this topic. You can use ADO.NET performance counters to monitor the status of your application and the connection resources that it uses. Performance counters can be monitored by using Windows Performance Monitor or can be accessed programmatically using the [PerformanceCounter](http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx) class in the [System.Diagnostics](http://msdn.microsoft.com/en-us/library/system.diagnostics.aspx) namespace.

http://i.msdn.microsoft.com/Global/Images/clear.gif Available Performance Counters

Currently there are 14 different performance counters available for [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx) and [System.Data.OracleClient](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.aspx) as described in the following table. Note that the names for the individual counters are not localized across regional versions of the Microsoft .NET Framework.

|  |  |
| --- | --- |
| **Performance counter** | **Description** |
| **HardConnectsPerSecond** | The number of connections per second that are being made to a database server. |
| **HardDisconnectsPerSecond** | The number of disconnects per second that are being made to a database server. |
| **NumberOfActiveConnectionPoolGroups** | The number of unique connection pool groups that are active. This counter is controlled by the number of unique connection strings that are found in the AppDomain. |
| **NumberOfActiveConnectionPools** | The total number of connection pools. |
| **NumberOfActiveConnections** | The number of active connections that are currently in use.  ms254503.alert_note(en-us,VS.90).gif**Note:**  This performance counter is not enabled by default. To enable this performance counter, see [Activating Off-By-Default Counters](http://msdn.microsoft.com/en-us/library/ms254503.aspx#ActivatingOffByDefault). |
| **NumberOfFreeConnections** | The number of connections available for use in the connection pools.  ms254503.alert_note(en-us,VS.90).gif**Note:**  This performance counter is not enabled by default. To enable this performance counter, see [Activating Off-By-Default Counters](http://msdn.microsoft.com/en-us/library/ms254503.aspx#ActivatingOffByDefault). |
| **NumberOfInactiveConnectionPoolGroups** | The number of unique connection pool groups that are marked for pruning. This counter is controlled by the number of unique connection strings that are found in the AppDomain. |
| **NumberOfInactiveConnectionPools** | The number of inactive connection pools that have not had any recent activity and are waiting to be disposed. |
| **NumberOfNonPooledConnections** | The number of active connections that are not pooled. |
| **NumberOfPooledConnections** | The number of active connections that are being managed by the connection pooling infrastructure. |
| **NumberOfReclaimedConnections** | The number of connections that have been reclaimed through garbage collection where **Close** or **Dispose** was not called by the application. Not explicitly closing or disposing connections hurts performance. |
| **NumberOfStasisConnections** | The number of connections currently awaiting completion of an action and which are therefore unavailable for use by your application. |
| **SoftConnectsPerSecond** | The number of active connections being pulled from the connection pool.  ms254503.alert_note(en-us,VS.90).gif**Note:**  This performance counter is not enabled by default. To enable this performance counter, see [Activating Off-By-Default Counters](http://msdn.microsoft.com/en-us/library/ms254503.aspx#ActivatingOffByDefault). |
| **SoftDisconnectsPerSecond** | The number of active connections that are being returned to the connection pool.  ms254503.alert_note(en-us,VS.90).gif**Note:**  This performance counter is not enabled by default. To enable this performance counter, see [Activating Off-By-Default Counters](http://msdn.microsoft.com/en-us/library/ms254503.aspx#ActivatingOffByDefault). |

**Connection Pool Groups and Connection Pools**

When using Windows Authentication (integrated security), you must monitor both the **NumberOfActiveConnectionPoolGroups** and **NumberOfActiveConnectionPools** performance counters. The reason is that connection pool groups map to unique connection strings. When integrated security is used, connection pools map to connection strings and additionally create separate pools for individual Windows identities. For example, if Fred and Julie, each within the same AppDomain, both use the connection string "Data Source=MySqlServer;Integrated Security=true", a connection pool group is created for the connection string, and two additional pools are created, one for Fred and one for Julie. If John and Martha use a connection string with an identical SQL Server login, "Data Source=MySqlServer;User Id=lowPrivUser;Password=Strong?Password", then only a single pool is created for the **lowPrivUser** identity.

**Activating Off-By-Default Counters**

The performance counters **NumberOfFreeConnections**, **NumberOfActiveConnections**, **SoftDisconnectsPerSecond**, and **SoftConnectsPerSecond** are off by default. Add the following information to the application's configuration file to enable them:

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl29other');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl29other');)

<system.diagnostics>

<switches>

<add name="ConnectionPoolPerformanceCounterDetail"

value="4"/>

</switches>

</system.diagnostics>

http://i.msdn.microsoft.com/Global/Images/clear.gif Retrieving Performance Counter Values

The following console application shows how to retrieve performance counter values in your application. Connections must be open and active for information to be returned for all of the ADO.NET performance counters.

|  |
| --- |
| **ms254503.alert_note(en-us,VS.90).gifNote:** |
| This example uses the sample **AdventureWorks** database included with SQL Server 2005. The connection strings provided in the sample code assume that the database is installed and available on the local computer with an instance name of SqlExpress, and that you have created SQL Server logins that match those supplied in the connection strings. You may need to enable SQL Server logins if your server is configured using the default security settings which allow only Windows Authentication. Modify the connection strings as necessary to suit your environment. |

**Example**

Visual Basic

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl36VisualBasic');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl36VisualBasic');)

Option Explicit On

Option Strict On

Imports System.Data.SqlClient

Imports System.Diagnostics

Imports System.Runtime.InteropServices

Class Program

Private PerfCounters(9) As PerformanceCounter

Private connection As SqlConnection = New SqlConnection

Public Shared Sub Main()

Dim prog As Program = New Program

' Open a connection and create the performance counters.

prog.connection.ConnectionString = \_

GetIntegratedSecurityConnectionString()

prog.SetUpPerformanceCounters()

Console.WriteLine("Available Performance Counters:")

' Create the connections and display the results.

prog.CreateConnections()

Console.WriteLine("Press Enter to finish.")

Console.ReadLine()

End Sub

Private Sub CreateConnections()

' List the Performance counters.

WritePerformanceCounters()

' Create 4 connections and display counter information.

Dim connection1 As SqlConnection = New SqlConnection( \_

GetIntegratedSecurityConnectionString)

connection1.Open()

Console.WriteLine("Opened the 1st Connection:")

WritePerformanceCounters()

Dim connection2 As SqlConnection = New SqlConnection( \_

GetSqlConnectionStringDifferent)

connection2.Open()

Console.WriteLine("Opened the 2nd Connection:")

WritePerformanceCounters()

Console.WriteLine("Opened the 3rd Connection:")

Dim connection3 As SqlConnection = New SqlConnection( \_

GetSqlConnectionString)

connection3.Open()

WritePerformanceCounters()

Dim connection4 As SqlConnection = New SqlConnection( \_

GetSqlConnectionString)

connection4.Open()

Console.WriteLine("Opened the 4th Connection:")

WritePerformanceCounters()

connection1.Close()

Console.WriteLine("Closed the 1st Connection:")

WritePerformanceCounters()

connection2.Close()

Console.WriteLine("Closed the 2nd Connection:")

WritePerformanceCounters()

connection3.Close()

Console.WriteLine("Closed the 3rd Connection:")

WritePerformanceCounters()

connection4.Close()

Console.WriteLine("Closed the 4th Connection:")

WritePerformanceCounters()

End Sub

Private Enum ADO\_Net\_Performance\_Counters

NumberOfActiveConnectionPools

NumberOfReclaimedConnections

HardConnectsPerSecond

HardDisconnectsPerSecond

NumberOfActiveConnectionPoolGroups

NumberOfInactiveConnectionPoolGroups

NumberOfInactiveConnectionPools

NumberOfNonPooledConnections

NumberOfPooledConnections

NumberOfStasisConnections

' The following performance counters are more expensive to track.

' Enable ConnectionPoolPerformanceCounterDetail in your config file.

' SoftConnectsPerSecond

' SoftDisconnectsPerSecond

' NumberOfActiveConnections

' NumberOfFreeConnections

End Enum

Private Sub SetUpPerformanceCounters()

connection.Close()

Me.PerfCounters(9) = New PerformanceCounter()

Dim instanceName As String = GetInstanceName()

Dim apc As Type = GetType(ADO\_Net\_Performance\_Counters)

Dim i As Integer = 0

Dim s As String = ""

For Each s In [Enum].GetNames(apc)

Me.PerfCounters(i) = New PerformanceCounter()

Me.PerfCounters(i).CategoryName = ".NET Data Provider for SqlServer"

Me.PerfCounters(i).CounterName = s

Me.PerfCounters(i).InstanceName = instanceName

i = (i + 1)

Next

End Sub

Private Declare Function GetCurrentProcessId Lib "kernel32.dll" () As Integer

Private Function GetInstanceName() As String

'This works for Winforms apps.

Dim instanceName As String = \_

System.Reflection.Assembly.GetEntryAssembly.GetName.Name

' Must replace special characters like (, ), #, /, \\

Dim instanceName2 As String = \_

AppDomain.CurrentDomain.FriendlyName.ToString.Replace("(", "[") \_

.Replace(")", "]").Replace("#", "\_").Replace("/", "\_").Replace("\\", "\_")

'For ASP.NET applications your instanceName will be your CurrentDomain's

'FriendlyName. Replace the line above that sets the instanceName with this:

'instanceName = AppDomain.CurrentDomain.FriendlyName.ToString.Replace("(", "[") \_

' .Replace(")", "]").Replace("#", "\_").Replace("/", "\_").Replace("\\", "\_")

Dim pid As String = GetCurrentProcessId.ToString

instanceName = (instanceName + ("[" & (pid & "]")))

Console.WriteLine("Instance Name: {0}", instanceName)

Console.WriteLine("---------------------------")

Return instanceName

End Function

Private Sub WritePerformanceCounters()

Console.WriteLine("---------------------------")

For Each p As PerformanceCounter In Me.PerfCounters

Console.WriteLine("{0} = {1}", p.CounterName, p.NextValue)

Next

Console.WriteLine("---------------------------")

End Sub

Private Shared Function GetIntegratedSecurityConnectionString() As String

' To avoid storing the connection string in your code,

' you can retrive it from a configuration file.

Return ("Data Source=.\SqlExpress;Integrated Security=True;" &

"Initial Catalog=AdventureWorks")

End Function

Private Shared Function GetSqlConnectionString() As String

' To avoid storing the connection string in your code,

' you can retrive it from a configuration file.

Return ("Data Source=.\SqlExpress;User Id=LowPriv;Password=Data!05;" &

"Initial Catalog=AdventureWorks")

End Function

Private Shared Function GetSqlConnectionStringDifferent() As String

' To avoid storing the connection string in your code,

' you can retrive it from a configuration file.

Return ("Initial Catalog=AdventureWorks;Data Source=.\SqlExpress;" & \_

"User Id=LowPriv;Password=Data!05;")

End Function

End Class

C#

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_mainContentContainer_ctl37CSharp');)Copy Code](javascript:CopyCode('ctl00_mainContentContainer_ctl37CSharp');)

using System;

using System.Data.SqlClient;

using System.Diagnostics;

using System.Runtime.InteropServices;

class Program

{

PerformanceCounter[] PerfCounters = new PerformanceCounter[10];

SqlConnection connection = new SqlConnection();

static void Main()

{

Program prog = new Program();

// Open a connection and create the performance counters.

prog.connection.ConnectionString =

GetIntegratedSecurityConnectionString();

prog.SetUpPerformanceCounters();

Console.WriteLine("Available Performance Counters:");

// Create the connections and display the results.

prog.CreateConnections();

Console.WriteLine("Press Enter to finish.");

Console.ReadLine();

}

private void CreateConnections()

{

// List the Performance counters.

WritePerformanceCounters();

// Create 4 connections and display counter information.

SqlConnection connection1 = new SqlConnection(

GetIntegratedSecurityConnectionString());

connection1.Open();

Console.WriteLine("Opened the 1st Connection:");

WritePerformanceCounters();

SqlConnection connection2 = new SqlConnection(

GetSqlConnectionStringDifferent());

connection2.Open();

Console.WriteLine("Opened the 2nd Connection:");

WritePerformanceCounters();

SqlConnection connection3 = new SqlConnection(

GetSqlConnectionString());

connection3.Open();

Console.WriteLine("Opened the 3rd Connection:");

WritePerformanceCounters();

SqlConnection connection4 = new SqlConnection(

GetSqlConnectionString());

connection4.Open();

Console.WriteLine("Opened the 4th Connection:");

WritePerformanceCounters();

connection1.Close();

Console.WriteLine("Closed the 1st Connection:");

WritePerformanceCounters();

connection2.Close();

Console.WriteLine("Closed the 2nd Connection:");

WritePerformanceCounters();

connection3.Close();

Console.WriteLine("Closed the 3rd Connection:");

WritePerformanceCounters();

connection4.Close();

Console.WriteLine("Closed the 4th Connection:");

WritePerformanceCounters();

}

private enum ADO\_Net\_Performance\_Counters

{

NumberOfActiveConnectionPools,

NumberOfReclaimedConnections,

HardConnectsPerSecond,

HardDisconnectsPerSecond,

NumberOfActiveConnectionPoolGroups,

NumberOfInactiveConnectionPoolGroups,

NumberOfInactiveConnectionPools,

NumberOfNonPooledConnections,

NumberOfPooledConnections,

NumberOfStasisConnections

// The following performance counters are more expensive to track.

// Enable ConnectionPoolPerformanceCounterDetail in your config file.

// SoftConnectsPerSecond

// SoftDisconnectsPerSecond

// NumberOfActiveConnections

// NumberOfFreeConnections

}

private void SetUpPerformanceCounters()

{

connection.Close();

this.PerfCounters = new PerformanceCounter[10];

string instanceName = GetInstanceName();

Type apc = typeof(ADO\_Net\_Performance\_Counters);

int i = 0;

foreach (string s in Enum.GetNames(apc))

{

this.PerfCounters[i] = new PerformanceCounter();

this.PerfCounters[i].CategoryName = ".NET Data Provider for SqlServer";

this.PerfCounters[i].CounterName = s;

this.PerfCounters[i].InstanceName = instanceName;

i++;

}

}

[DllImport("kernel32.dll", SetLastError = true)]

static extern int GetCurrentProcessId();

private string GetInstanceName()

{

//This works for Winforms apps.

string instanceName =

System.Reflection.Assembly.GetEntryAssembly().GetName().Name;

// Must replace special characters like (, ), #, /, \\

string instanceName2 =

AppDomain.CurrentDomain.FriendlyName.ToString().Replace('(', '[')

.Replace(')', ']').Replace('#', '\_').Replace('/', '\_').Replace('\\', '\_');

// For ASP.NET applications your instanceName will be your CurrentDomain's

// FriendlyName. Replace the line above that sets the instanceName with this:

// instanceName = AppDomain.CurrentDomain.FriendlyName.ToString().Replace('(','[')

// .Replace(')',']').Replace('#','\_').Replace('/','\_').Replace('\\','\_');

string pid = GetCurrentProcessId().ToString();

instanceName = instanceName + "[" + pid + "]";

Console.WriteLine("Instance Name: {0}", instanceName);

Console.WriteLine("---------------------------");

return instanceName;

}

private void WritePerformanceCounters()

{

Console.WriteLine("---------------------------");

foreach (PerformanceCounter p in this.PerfCounters)

{

Console.WriteLine("{0} = {1}", p.CounterName, p.NextValue());

}

Console.WriteLine("---------------------------");

}

private static string GetIntegratedSecurityConnectionString()

{

// To avoid storing the connection string in your code,

// you can retrive it from a configuration file.

return @"Data Source=.\SqlExpress;Integrated Security=True;" +

"Initial Catalog=AdventureWorks";

}

private static string GetSqlConnectionString()

{

// To avoid storing the connection string in your code,

// you can retrive it from a configuration file.

return @"Data Source=.\SqlExpress;User Id=LowPriv;Password=Data!05;" +

// "Initial Catalog=AdventureWorks";

}

private static string GetSqlConnectionStringDifferent()

{

// To avoid storing the connection string in your code,

// you can retrive it from a configuration file.

return @"Initial Catalog=AdventureWorks;Data Source=.\SqlExpress;" +

"User Id=LowPriv;Password=Data!05;";

}

}

**What's Connection Pooling?**  
Connection pooling is the ability of re-use your connection to the Database. This means if you enable Connection pooling in the connection object, actually you enable the re-use of the connection to more than one user.  
  
The connection pooling is enabled by default in the connection object. If you disable the connection pooling, this means the connection object which you create will not be re-used to any other user than who create that object.

**Shall I Enable/Disable Connection pool?**

Let's do an example to use what the time has required if we enable/disable the connection pool in an application.

***Sample 1(Connection Pooling is enabled):***  
Create a console application and put the following lines of code to Main Method:

SqlConnection testConnection = new SqlConnection(@"Data Source=(local)\SQLEXPRESS;Initial Catalog=DEMO;Integrated Security=SSPI;");

long startTicks = DateTime.Now.Ticks;

for (int i = 1; i <= 100; i++)

{

testConnection.Open();

testConnection.Close();

}

long endTicks = DateTime.Now.Ticks;

Console.WriteLine("Time taken : " + (endTicks - startTicks) + " ticks.");

testConnection.Dispose();

Run the application, on my machine the difference in time is:  937626 ticks

***Sample 2(Connection Pooling is disabled):***  
  
Just add ***Pooling=false*** in the connection string.  
  
Run the application, on my machine the difference in time is:  3906500 ticks

If you measure the difference you will see the time required by disabling the connection polling is 4 times greater than using connection pooling.  
  
On of the good practices when using your connection object is enclose your code by try {..} catch {} finally {} blocks.

On finally block you have to call Conn.Close(); or Conn.Dispose();   
  
To remove all resources attached to that connection.

One of you asked what's the difference of calling Close or Dispose, the answer don't use both of them, Dispose method actually call close method internally plus remove all allocated resource for that object to be garbage collected and at the same time the underlying connection object can be pooled.

**Conclusion**  
  
Use connection pooling in your applications to maximize the use of physical connection with the Database and your application.