# SQL SERVER: UNDERSTANDING AND CONTROLLING CONNECTION

*by Mohammed Mawla* | October 7, 2008

*Posted in:* *Technical Track*

*Tags:* *Microsoft SQL Server*

I got the idea for this article when one of our clients complained that their server's performance was degrading during business hours. They thought it was weird that at the same time, SQL Server would list more than 1200 connections on SQL server Activity Monitor.

The server hosts more than 50 databases that serve an ASP.NET application hosted on some servers in a web farm. These servers issue connections to the databases in a distributed manner to balance the web application load. I tried to discover what these connections were doing and to what databases they were connected. Connections

grouped by database:

```
select  db_name(dbid) , count(*) 'connections count'
from master..sysprocesses
where spid > 50 and spid != @@spid
group by  db_name(dbid)
order by count(*) desc
```

This showed some databases having more than 300 connections associated with them.

What about logins used?

```
select  loginame , nt_username, count(*) 'Connections
count'
from master..sysprocesses
where spid > 50 and spid != @@spid
group by  loginame , nt_username
order by count(*) desc
```

This showed a mix of windows domain accounts (those with values in `column nt_username`, e.g: domain\user) beside SQL authentication accounts (those with column `nt_username` empty, e.g: "sa").

In order to reduce the number of times that new connections must be opened, applications may use connection pooling. This was clearly not the case here, and all these connections resulted in what is known as "pool fragmentation".

So what's connection pooling? how does it work, what can cause pool fragmentation and how can we avoid/reduce it?

## Connection pooling defined

A connection pool is a set of idle, open, and reusable database connections maintained by the database server so that the connections can be reused when the database receives future requests for data, instead of exclusively opening a new connection. In our case, this connection pool is maintained by ADO.NET and is used to communicate with SQL Server.

The benefit of connection pooling, is that connections placed in the pool

Also, opening and closing of connections to the database becomes less resource-expensive in terms of CPU and memory.

## How connection pooling works

When a connection is first opened, a connection pool is created based on matching criteria that associates the pool with the connection string in the connection. Each connection pool is associated with a distinct connection string. If the connection string is not an exact match to an existing pool when a new connection is opened, 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.

Whenever a user calls “open” on a connection with a matched connection string, the “pooler” looks for an available connection in the pool that corresponds to that connection string. If a pooled connection is available, it returns it to the caller, otherwise it will add a new connection to the pool, up to the maximum pool size specified (100 is the default). When the application calls “close” on the connection, instead of closing the connection, the pooler returns it to the pooled set of active connections. Once the connection is returned to the pool, it is ready to be reused on the next “open” call.

When the connection is closed or disposed, it is returned to the pool and remains idle until a request for a new connection comes in. The connection pooler removes a connection from the pool after it has been idle for a long time.

So how does it go in ADO.net? In the following C# code, we will open five connections to SQL Server, but only three connection pools are required to manage them.

```
SqlConnection sqlcon = new SqlConnection();

    string constring = "Data Source=.\\SQL2k8x;Connect
Timeout=5;";

    try
```

```
    {
        sqlcon.ConnectionString = constring + "database =
DB1;Integrated Security=true";
        sqlcon.Open();   // Pool 1 is created.
        sqlcon.Close();  // connection is closed ,
returned to pool 1

        sqlcon.ConnectionString = constring + "database =
DB2;Integrated Security=true";
        sqlcon.Open();   // Pool 2 is created , another
database is used
        sqlcon.Close();  // connection is closed ,
returned to pool 2

        sqlcon.ConnectionString = constring + "database =
DB1;Integrated Security=true";
        sqlcon.Open();   // Pool 1 is Used , same
connection string as when pool 1 was created.
        sqlcon.Close();  // connection is closed ,
returned to pool 1

        sqlcon.ConnectionString = constring + "database =
DB1;Uid=sa ;Pwd=password";

        sqlcon.Open();   // Pool 3 is created , SQL
authentication account is used even if same database as
Pool 1.
        sqlcon.Close();  // connection is closed ,
returned to pool 3

        sqlcon.ConnectionString = constring + "database =
DB2;Integrated Security=true";
        sqlcon.Open();   // Pool 2 is Used , same
connection string as when pool 2 was created.
        sqlcon.Close();  // connection is closed ,
returned to pool 2

    }
```

```
    catch (Exception ex)
    {
        MessageBox.Show("Error connecting to SQL
server.Message : " + Environment.NewLine +
                        ex.Message.ToString(), "Error
connecting to SQl instance");
        return;
    }
```

If we query the SQL Server instance to get the number of connections opened, the result returned will be three connections where the `program_name` column is ".Net SqlClient Data provider"

```
select * from master..sysprocesses where spid > 50 and
spid  @@spid
```

But, if we are connecting to the same instance and closing the connection directly after using it, why there isn't one connection pool created?

## What causes pool fragmentation?

Since connection pooling is created for each distinct connection string, there will be as many different connection pools as the number of connection strings used, increasing the number of connections opened and number of pools managed.

This can happen under two conditions:

1. Integrated security is used.Connections are pooled according to the connection string plus the user identity. Thus, if integrated security is used to connect to SQL server, 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 trade-off between manageability and logging/auditing, since you may like to use different accounts to connect to the database server to audit activity on the sever based on the account used in the connection.

2. The instance has many databases.This is significantly the case here. If the application extracts data from many databases and makes explicit calls to them, then there will be a separate pool of connections to each database, thus increasing the number of connections to the server. Fortunately, there is a relatively simple way to avoid this side effect. I'll get to that soon.

Now I will show a demo of an application that causes pool fragmentation and possible ways to avoid it. I have written it in C# using Visual Studio 2008.

In order to use the application, we need to create ten databases on the instance, each database containing once table. Here's the code:

```
DeclARE @i int , @sql nvarchar (500)

select @i=0

while @i < 10

begin

exec ( 'CREATE database pooling' + @i )

select @sql = '
 use pooling'+cast (@i as varchar(20))

select @sql = @sql + '

create table pooltest'+cast (@i as varchar(20))+' ( col1
int)

'
--print @sql
exec sp_executesql @sql

set @i= @i + 1

end
```

The application connects to the databases and returns list of tables in each database, besides returning ADO.net connection pooling counters. These counters can also be captured with performance monitor (perfmon) using object ".net data provider for Sqlserver". The old ADO.NET 1.1 object is called ".NET CLR Data". Note that counters for the object are created only after making at least one connection, so if you attempted to add them without making any connections, you will find them dimmed.

For more information about these counters, look here: Using ADO.NET Performance Counters at MSDN's Visual Studio 2005 Developer Center. I have used some of the code at this URL inside the application to display counters.

The application loops and makes exclusive connections to each database

Here is the code that makes connections attempts:

```
string strSQL = "select db_name() + '  -------  ' + name
'table' from sys.tables";

for (int i = 0; i < 10; i++)
{

 sqlcon.ConnectionString = "Data Source=" +
server.Text.Trim() + ";DataBase=pooling" + i.ToString() +
";" + "Integrated Security=true;Connect Timeout=5";

 if (checkBox1.Checked == true) // SQL autentication used
    {
        sqlcon.ConnectionString = "Data Source=" +
server.Text.Trim() + ";DataBase=pooling" + i.ToString() +
";" + "Uid=" + username.Text.Trim() + ";Pwd=" +
password.Text.Trim() + ";Connect Timeout=5";
    }

    try
    {
        sqlcon.Open();
```

```
        SqlCommand DBCmd = new SqlCommand(strSQL,
sqlcon);
    }
```

We will connect to SQL server once using integrated security and once using SQL authentication:

**Running using integrated security**

Here we are going to use integrated security to connect to the instance and make an explicit connection to each database, creating 10 connection pools and 10 pooled connections. See the following image.

**Running using SQL authentication**

Here application produces a similar result, except for **NumberOfActiveConnectionPoolGroups** because it is cumulative. We get 10 connection pools and 10 pooled connections, but 20 connection pool groups.

Running . . .

```
select program_name , hostprocess , nt_username  from
master..sysprocesses where spid >50 and spid @@spid
```

. . . returns 20 connections where the `program_name` column is ".Net SqlClient Data provider" for which each instance of the application is responsible for 10 connections.

Note that you can get different counter based on the connections already established on the instance.

This data can also be shown from performance monitor, but you must

add the counters after making the connections and the application should be left open. The performance monitor distinguishes among different instances of the same application by using process Id (**pid** in task manager processes tab), The process id is logged in the application main window.

Here's how it looks in performance monitor:

**Explanation**

Running the first instance of the application with windows security produced 10 pooled connections, 10 active connection pools, and 10 active pool groups. (This happens at point 1 in the above image.)

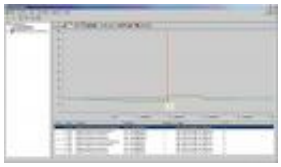
Running the second instance of the application with SQL authentication (and then adding the counter) shows 10 pooled connections, 10 active pools, but 20 active pool groups. (Point 2.)

Changing the first instance of the application to use SQL authentication while the application is still open sent the number of connection pools and pooled connection for this application with process id [5504] to rise to 20. They are running side to side that you can't see the yellow line for activeconnectionpools. (Point 3.)

The same happened when changing the second instance of the application to use integrated security while the application is still open (both instances exchanged the connection method). This also sent the number of connection pools and pooled connections for this application instance (804) to rise to 20. (Point 4.)

Note that number of *active* connections is always *zero*, as we close the connection directly after retrieving data. This is a best practice.

Leaving the application open for a while but without activity results in connection pools and connections being closed because they are inactive for a time. When I closed both application instances, all counters dropped to zero and shortly the red perfmon cursor stopped, point 1 in following image.



Now we know what pool fragmentation is and demonstrated it.

## How can we reduce or prevent pool fragmentation?

We can address pool fragmentation for the two previously outlined causes as follows.

**Pool fragmentation due to integrated security**

If it is possible to reduce the number of integrated security accounts used with applications, that in turn will reduce the number of connection pools and thus the overhead of managing them, increasing the performance of SQL Server.

It is also to be preferred when possible to use Windows Authentication mode, as it uses the Kerberos security protocol, which provides password policy enforcement with regard to complexity validation for strong passwords, and provides support for account lockout and password expiration.

**Pool fragmentation due to many databases**

There is a tiny and easy to apply trick here: 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 the context to the desired database.

For example, instead of this code:

```
string strSQL = "select db_name() + '  -------  ' + name
'table' from sys.tables";
sqlcon.ConnectionString = "Data Source=" +
server.Text.Trim() + ";DataBase=pooling" + i.ToString() +
";" + "Integrated Security=true;Connect Timeout=5";
sqlcon.Open();
SqlCommand DBCmd = new SqlCommand(strSQL, sqlcon);
```

we can use this:

```
string strSQL = " use pooling" + i.ToString() + " ;
select db_name() + '  -------  ' + name 'table' from
sys.tables";

sqlcon.ConnectionString = "Data Source=" +
server.Text.Trim() + ";DataBase=master;" + "Integrated
Security=true;Connect Timeout=5";

SqlCommand DBCmd = new SqlCommand(strSQL, sqlcon);
```

We will connect always to the master database and then use the USE keyword to change context to the user databases.

Now let's test this. From the demo application, we will just check the combo-box with the label **Use one connection and "USE" keyword to query databases**. This will set the connection string to connect to the master database and later to branch to other databases using the USE keyword.

Data from the performance monitor shows that the number of connection pools and the number of pooled connections didn't exceed 1 at initial run, and increased to 2 only when changing connection credentials and the "open connections" button is pressed again. You can verify the number of connections by querying "sysprocesses" in master database.

Here is how it looks in performance monitor:

**Explanation**

When running the first instance of the application with windows security, we had 1 connection pool, 1 pooled connection, and 10 active pool groups. (This happens at point 1 in the above image.)

Running the second instance of the application with SQL authentication (and then adding the counter in perfmon) shows 1 pooled connection, 1 active pools, but 20 active pool groups (they are cumulative). (Point 2.)

Changing the first instance of the application to use SQL authentication

while the application is still open sent number of connection pools and pooled connection for this application instance (pid:4808) only to 2. (Point 3.)

The same happened when changing the second instance of the application to use integrated security while the application was still open (both instances exchanged connection method). This also sent number of connection pools and pooled connection for this application instance (pid:5800) only to 2. (Point 4.)

Obviously there is improvement, and with environments with heavy connection loads and more databases, this improvement can be very significant — reduced time to fetch data and fewer resources to manage connections. Opening a connection from scratch is an expensive process.

To cut a long story short, we can summarize this post in few points:

1. Connection pooling is a great feature to increase your application performance and scalability.
2. Open connections as late as possible and close them as early as possible to release resources as fast as possible.
3. In order to reduce overhead of managing connections to server, avoid connection pooling fragmentation as much as you can.
4. Ensure that there is at least one connection open in the pool to maintain the connection pool, this can be done using the “Min Pool Size” property in ConnectionString. For more info, see SqlConnection.ConnectionString Property at msdn’s .NET Framework Developer Center.

That’s all for now. Here is a zip file containing a .EXE compiled from the C# code above: Conn.Pooling.zip.

I hope you enjoy a healthy SQL Server environment.

## Related Posts:

1. **Webinar: Applying the supply management promise to IT**
2. **Log Buffer #317, A Carnival of the Vanities for DBAs**
3. **SQL Server 2014 CTP2 – Memory Optimization Advisor**
4. **SQL Server Integrity Check – A Necessary Routine**

**Interested in working with Mohammed? Schedule a tech call.**

Mohammed Mawla

## 20 Comments. Leave new

**meriem** October 7, 2008 4:47 pm

congratulations mawla

Reply

**HÃ¥kan Nilsson** October 16, 2008 3:35 pm

Very good article, Mawla. We are using integrated security with just one account per application calling the sql server. Now we will set the max pool size and stress test the apps. I do believe we will get some better performance after this.

Reply

**IDisposable and unmanaged memory « Ben Biddington**

June 15, 2009 4:58 pm

[...] Database server connection pooling Possibly related posts: (automatically generated)IDisposable reloadedIDisposable Best Practicesthread/weakHashMap [...]

Reply



**vineet** November 23, 2009 1:32 am

This article is very usefull for me.You have explained alot of abount connection polling.
It give me a greate information.
Thanks a Lot.

But if my application uses only one database server not destributed then whether I should
use Pooling=True Or not?

Reply

**Mohammed Mawla** November 24, 2009 5:30 pm

Hey Vineet,

The blog mentions a scenario of multiple **web** servers and single database server.

In all cases , connection pooling should be enabled even if you use only one application server as it will help somehow.

HTH

Reply

**Pooled Connections « Systems Engineering and RDBMS**

May 31, 2010 12:39 pm

[...] Article at Pythian – excellent article discussing connection pooling – here. [...]

Reply

**Links: 2010-October « Sheen Space** October 22, 2010 5:34 am

[...] SQL Server: Understanding and Controlling Connection-Pooling Fragmentation [...]

Reply

**Rasmus Striib** February 2, 2011 8:10 am

This is a very good explanation of the nature of SQL Server connection pools. Saved me a lot of time. Thanks.

Reply

**Mayur** April 8, 2011 11:41 pm

I connect SQL server 2008 with Microsoft SQL server Management studio it was connect with using database engine server authentication it was connect. I was use this database connection in local site it was working perfect but when we use this same site online it was some time database connected and sometime not connected.
Please help me to solve this problem..

Reply

**Tiago Sumita** July 5, 2011 10:51 am

Great article!

Reply

**Hashim Abbas** October 7, 2011 1:41 am

Excellent information :) (Y)

Reply

**Amit** October 13, 2011 12:53 am

If I use the change database statement of SQL Connection object then for which database the pool wil be maintained ?

For eg: In my connection string I have master database. I have

opened the connection for master database, then I am using SQLConnection.changedatabase(“northwind”). Then I am executing query for a table which exists on Northwind.

Please let me know for which the connection pool will be maintained. For master or for Northwind.

Reply

**Mohammed Mawla** October 13, 2011 8:00 pm

Amit,
The SQLConnection.changedatabase does change database context for an **open connection** , thus the connection initial DB was master so the connection pooling is still for master

Please refer to

http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.changedatabase%28v=VS.85%29.aspx

The connection pooler is not influenced by ChangeDatabase as long you do not set Connection Reset to false. As the connection comes out of the pool the connection is reset with the server moving back to the login time database. There are no new connections created or reauthentications. If you set Connection Reset to false, connections in the pool to different databases might result.

Reply

**Doug Brown** March 26, 2013 12:41 pm

Great article! Found this while looking for a way to manage multiple user connection with multiple databases with a client server application (no .NET).

Reply

**Asim Suvedi** March 5, 2014 10:54 pm

There fist and second queries have syntax errors.
Thank you for the article.

Reply

**Mohammed Mawla** March 6, 2014 6:24 am

That was some formatting issue displaying “” , easy to fix

Reply

**clintm** March 21, 2014 2:34 am

Connection pooling in simple explanation

http://net-informations.com/faq/ado/connection-pooling.htm

clint

Reply

**SteveC** June 5, 2014 7:09 pm

As an admin rather than programmer, I thank you greatly for providing the compiled EXE. I see how the “use poolingN; select …” reduces Connections and ConnectionsPools, but don’t understand why PoolGroups is so high. The MS doc explains how multiple identities cause multiple pools within a PoolGroup, but I don’t see how it fits in this example. Can you please explain more clearly just what a PoolGroup is and why there can be a higher count for groups than for members?

Reply

**Golam Kabir** September 1, 2014 12:38 pm

Very helpful guide. Thanks and keep helping the community.

Reply

**Gilberto Ribeiro** October 3, 2014 7:28 am

Very helpful guide. Thanks and keep helping the community. [2]

Reply

## Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

POST COMMENT

Notify me of follow-up comments by email.

Notify me of new posts by email.

# Innovate today with Pythian.

## ABOUT PYTHIAN

Pythian is a global IT services company that specializes in designing, implementing, and managing systems that directly contribute to revenue and business success. We help companies adopt disruptive technologies to advance innovation and increase agility. Our highly skilled technical teams work as an integrated extension of our clients' organizations to deliver continuous transformation and uninterrupted operational excellence.

## LATEST TWEETS

Pythian Named One of Canada's Top Small & Medium Employers for 2016 https://t.co/E4GDigLiks,
1 hour ago

Discover challenges, solutions, and findings uncovered while preforming DataStax upgrade 4.1 to 5.1 for Cassandra https://t.co/Be1RTUI5mt,
24 hours ago

## CONTACT US

Phone: +1-866-798-4426

Email: info@pythian.com