



Introduction

Suppose you wrote an application which uses the Entity Framework and you have to create a new feature which imports a large amount of data. Simple, right? Just iterate through the data you need to import, create the necessary entities, add them to theObjectContext and call SaveChanges().

This approach might work fine, until you start to import a lot of data. As the amount of data increases, the performance of your application will suffer. If you want to do a bulk insert of data then the Entity Framework (or any other ORM) might not be the way to go.

Sometimes a bit of direct ADO.NET is required. The .NET Framework 2.0 introduced the [SqlBulkCopy class](#) which lets you bulk load a SQL Server table with data. Let's see how we can combine the Entity Framework with the SqlBulkCopy class.

Table Of Contents

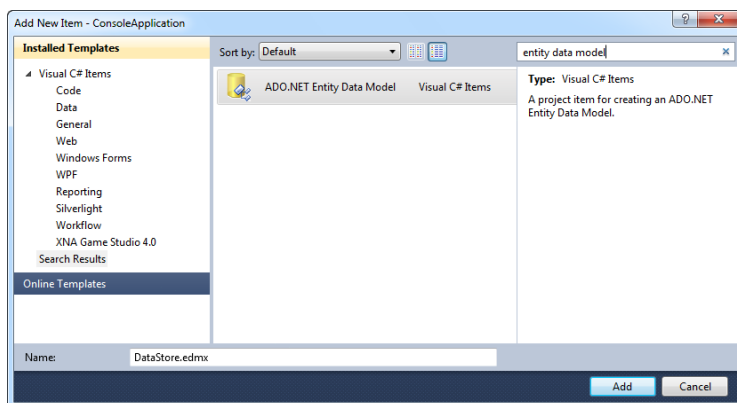
- [Introduction](#)
- [Entity Framework](#)
- [LINQ Entity Data Reader](#)

Entity Framework

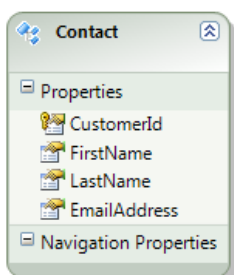
Let's first demonstrate the performance using the Entity Framework. For this exercise, I created a simple SQL Server 2008 database called MyDataStore. It contains one table named Contact.

| | Column Name | Data Type | Allow Nulls |
|---|--------------|------------------|--------------------------|
| 🔑 | CustomerId | uniqueidentifier | <input type="checkbox"/> |
| | FirstName | nvarchar(50) | <input type="checkbox"/> |
| | LastName | nvarchar(50) | <input type="checkbox"/> |
| | EmailAddress | nvarchar(50) | <input type="checkbox"/> |

Next I created a blank solution ("BulkCopy") in Visual Studio and added a Console Application project to which I added an ADO.NET Entity Data Model called DataStore.



To make things easy I generated the model from the database. After creating the model it contains one simple entity.



I created the following method which inserts a thousand contacts into the Contact table. When it is done it returns the time ([TimeSpan](#))

it took to insert the records.

```
static TimeSpan AddAThousandContacts()
{
    using (var context = new MyDataStoreEntities())
    {
        var stopwatch = new Stopwatch();
        stopwatch.Start();
        for (int i=0; i<1000; i++)
        {
            var entity = new Contact
            {
                CustomerId = Guid.NewGuid(),
                FirstName = "Ruben",
                LastName = "Geers",
                EmailAddress = "geersch@gmail.com"
            };

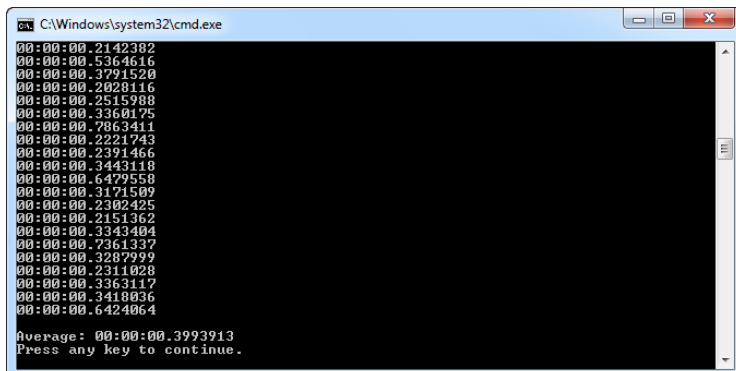
            context.Contacts.AddObject(entity);
        }
        context.SaveChanges();
        stopwatch.Stop();
        return stopwatch.Elapsed;
    }
}
```

The Console Application executes this method one hundred times and calculates the average time it takes to insert a thousand contacts.

```
var total = new TimeSpan();
for (int i = 0; i < 100; i++)
{
    var timeTaken = AddAThousandContacts();
    Console.WriteLine(timeTaken);

    total += timeTaken;
}
Console.WriteLine();
var average = new TimeSpan(total.Ticks / 100);
Console.WriteLine(String.Format("Average: {0}", average));
```

When I executed this sample application I got an average time of 0.3993913s. Ok, it didn't take seconds, minutes or hours...but keep in mind this is just a small application which inserts data in a simple table.



```
C:\Windows\system32\cmd.exe
00:00:00.2142302
00:00:00.5364616
00:00:00.3791520
00:00:00.2028116
00:00:00.2515988
00:00:00.3360175
00:00:00.7863411
00:00:00.2221743
00:00:00.2391466
00:00:00.3443118
00:00:00.6479558
00:00:00.3171509
00:00:00.2302425
00:00:00.2151362
00:00:00.3343404
00:00:00.7361337
00:00:00.3287999
00:00:00.2311028
00:00:00.3363117
00:00:00.3418036
00:00:00.6424064
Average: 00:00:00.3993913
Press any key to continue.
```

When the SaveChanges method is called, the Entity Framework generates and executes commands that perform the insert, update or delete statements. The Entity Framework does not support batch inserts so this will result in degraded performance. If you want to accomplish this kind of bulk insert you need to write some code against the lower layers of ADO.NET directly.

[Top of page](#)

Let's improve the performance by using the SqlBulkCopy class. This class contains a method called [WriteToServer\(...\)](#) which has the following overloads:

- [WriteToServer\(DataRow\(\)\)](#)
- [WriteToServer\(DataTable\)](#)
- [WriteToServer\(IDataReader\)](#)
- [WriteToServer\(DataTable, DataRowState\)](#)

In order to use the SqlBulkCopy class we need to be able to convert our collection of Contact entities to an array of DataRow instances, a DataTable or an IDataReader.

Luckily, [David Browne](#) already created an IDataReader implementation which allows you to read a collection of entities. You can download it here:

[LINQ Entity Data Reader](#)

Now we can create a new version of the AddAThousandContacts() method using the LINQ Entity Data Reader.

```
static TimeSpan AddAThousandContactsUsingSqlBulkCopy()
{
    var stopwatch = new Stopwatch();
    stopwatch.Start();

    var contacts = new List<Contact>();
    for (int i = 0; i < 1000; i++)
    {
        var entity = new Contact
        {
            CustomerId = Guid.NewGuid(),
            FirstName = "Ruben",
            LastName = "Geers",
            EmailAddress = "geersch@gmail.com"
        };

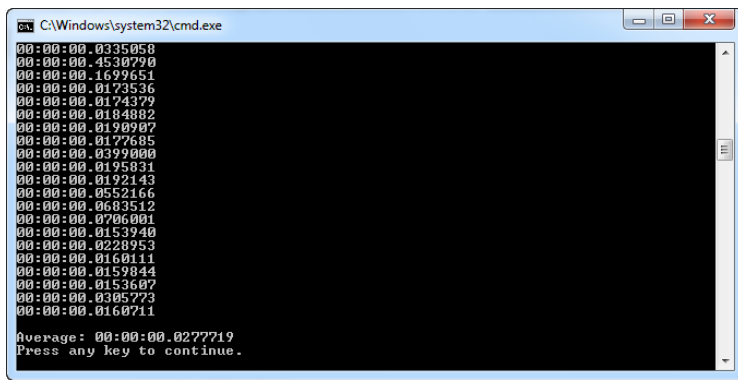
        contacts.Add(entity);
    }

    var connectionString =
        ConfigurationManager.ConnectionStrings["MyDataStore"].ConnectionString;
    var bulkCopy = new SqlBulkCopy(connectionString);
    bulkCopy.DestinationTableName = "Contact";
    bulkCopy.WriteToServer(contacts.AsDataReader());

    stopwatch.Stop();
    return stopwatch.Elapsed;
}
```

Instead of adding each new Contact entity to theObjectContext I add them to a generic list. When a thousand contacts have been added the AsDataReader extension method provided by the LINQ Entity Data Reader is used to bulk load the contacts into the database using the SqlBulkCopy class.

If you run this method a hundred times and calculate an average time you should see a drastic improvement.



```
C:\Windows\system32\cmd.exe
00:00:00.00335058
00:00:00.04330770
00:00:00.1699651
00:00:00.01723536
00:00:00.01743779
00:00:00.0184882
00:00:00.0190907
00:00:00.0177605
00:00:00.0399000
00:00:00.0195031
00:00:00.0192143
00:00:00.0552166
00:00:00.0603512
00:00:00.0706001
00:00:00.0153940
00:00:00.0228953
00:00:00.0160111
00:00:00.0159044
00:00:00.0153607
00:00:00.0305773
00:00:00.0160711
Average: 00:00:00.0277719
Press any key to continue.
```

An average time of 0.0277719s. This is 14.38x faster!

Remark: Before you start using this technique be aware that there are a few gotcha's:

- The LINQ Entity Data Reader does not work with POCO's
- You can only add data to one table at a time using the SqlBulkCopy
- The SqlBulkCopy only works with SQL Server
- The order of the scalar properties in your entity should have the same order as the fields in your table definition

There might be other gotcha's, but these are the ones I know off the top of my head. You can find the source code for this article on the [download page](#) of this blog.

[Top of page](#)