


[Database](#) » [Database](#) » [ADO.NET](#)
License: [The Code Project Open License \(CPOL\)](#)

C#, .NET (.NET 3.5), ADO.NET, Architect, Dev

Performance and the Entity Framework

By [Perry Marchant](#)Version: **11 (See All)**Posted: **12 Aug 2009**Updated: **12 Aug 2009**Views: **1,299**Bookmarked: **13 times**

An article describing the best practices for Entity Framework performance

12 votes for this article.

Popularity: 5.28 Rating: **4.89** out of 5

Introduction

If you are using the Entity Framework (EF), then you need to understand the best practices for improving its performance, or you will suffer the consequences!

Background

My team has spent almost 2 years with the first version of the Entity Framework in an enterprise application (including the beta). Our application has a Service Oriented Architecture (SOA) that uses the .NET 3.5 Framework SP1, SQL Server 2008 and IIS 7.0 on Windows Server 2008. During development we ran into significant issues with performance which we overcame by finding and following several best practices. The intent of this article is to explain these best practices. This article assumes that you are familiar with what the EF is and how to use it. If you need more of an introductory overview, check out the [Introduction to the Entity Framework](#) article.

Best Practices

Our application has a database with 400+ tables, some of which are very inter-connected. All of our Entity Framework calls are contained in our web service projects. Many of our EF calls took 8.5 seconds to complete on first time calls. By making certain changes (see the chart below) to our web service projects, we were able to improve these calls to 3 seconds. Second EF calls became almost instant. The chart below gives an indication of the overall performance impact with respect to the EF and the level of refactoring it took us to implement the change.

Best Practice	Impact	Required Refactoring
Pre-generated Your View	Major	Minor
Use Small EDMX Files	Minor	Moderate
Create Smart Connection Strings	Minor	Minor
Disable Change Tracking	Minor	Minor
Use Compiled Queries	Moderate	Moderate
Be Careful With Includes	Moderate	Major

The following sections in this article will explain these best practices in more detail- so read on!

Pre-generated Your View

The most expensive operation (it takes over 50% of the initial query) is *View Generation*. A big part of creating an abstracted view of the database is providing the actual view for queries and updates in the store's native language. During View Generation, the store views are created. Fortunately we can eliminate

the expense of building the in-memory view by running the EDM generator (*EdmGen.exe*) command line tool with the view generation command parameter `/mode:ViewGeneration` (see below).

Running this tool will create a code file (either `.cs` or `.vb`) with a pre-generated view that can be included and compiled in your project. Having the view pre-generated reduces the startup time considerably. The first time the EF `ObjectContext` is created the view must be generated, so pre-generating the view and deploying it with your application is important. The disadvantage of doing this is that you must keep the generated views synchronized with any changes you make to the data model.

```
"%windir%\Microsoft.NET\Framework\v3.5\EdmGen.exe"  
  /mode:ViewGeneration  
  /language:CSharp  
  /nologo  
  "/inssdl:MyEntityModel.ssd1"  
  "/incsd1:MyEntityModel.csd1"  
  "/inmsl:MyEntityModel.msl"  
  "/outviews:MyEntityModel.Views.cs"
```

The *EdmGen.exe* tool is installed in the .NET Framework directory. In many cases, this is located in `C:\windows\Microsoft.NET\Framework\v3.5`. For 64-bit systems, this is located in `C:\windows\Microsoft.NET\Framework64\v3.5`. You can also access the *EdmGen.exe* tool from the Visual Studio command prompt (Click Start, point to All Programs, point to Microsoft Visual Studio 2008, point to Visual Studio Tools, and then click Visual Studio 2008 Command Prompt). In order to get the input `.ssdl`, `.csdl` and `.msl` files, you need to use Visual Studio to do the following:

1. Open the project that contains your Entity Framework project.
2. Open the EDMX file in the designer.
3. Click the background of the model to open the model's Properties window.
4. Change the Metadata Artifact Processing property to: Copy to Output Directory.
5. Save the project- this will create the `.ssdl`, `.csdl`, and `.msl` files.

Another way to do this is by running the EDM generator command line tool with the full generation command parameter `/mode:FullGeneration` (see below). *EdmGen.exe* requires only a valid connection string to your database to generate the conceptual model (`.csdl`), storage model (`.ssdl`), and mapping (`.msl`) files, as well as the pre-generated view file. All options shown are required:

```
"%windir%\Microsoft.NET\Framework\v3.5\EdmGen.exe"  
  /mode:FullGeneration  
  /c:"Data Source=localhost;Initial Catalog=MyDatabase; Integrated Security=true"  
  /nologo  
  /language:CSharp  
  /entitycontainer:MyEntities  
  /namespace:MyModel  
  "/outviews:MyEntityModel.Views.cs"  
  "/outssdl:MyEntityModel.ssd1"  
  "/outcsdl:MyEntityModel.csd1"  
  "/outmsl:MyEntityModel.msl"  
  "/outobjectlayer:MyEntities.ObjectLayer.cs"
```

Use Smaller EDMX Files

The Entity Framework is capable of handling large entity data models but you can run into performance problems if the data model is very inter-connected. In general you should start thinking about breaking up a data model (into multiple `.edmx` files) when it has reached an order of 100 entities.

The size of the `.xml` schema files is somewhat proportional to the number of tables in the database that you generated the model from. As the size of the schema files increase, the time it takes to parse and create an in-memory model for this metadata will also increase. As stated earlier, this is a one-time cost incurred per `ObjectContext` instance and can be shortened by pre-generating the view. The Visual Studio Designer also starts to suffer from poor performance when the number of entities gets too large. The disadvantage of having several smaller data models is that you must keep the individual `.edmx` files synchronized when you make changes to the database.

Smart Connection Strings

The `ObjectContext` object retrieves connection information automatically from the application `config` file when creating object queries. You can supply this named connection string instead of relying on the `connectionString` parameter in the `.config` file when instantiating the `ObjectContext` class. The `Metadata` property in the `connectionString` parameter contains a list of locations for the `EntityClient` provider to search for EDM mapping and metadata files (the `.ssdl`, `.csdl`, and `.msl` files).

Mapping and metadata files are often deployed in the same directory as the application executable file. These mapping files can also be included as an embedded resource in the application which will improve performance. In order to embed the EDM mapping files in an assembly, you need to use Visual Studio to do the following:

1. Open the project that contains your Entity Framework project.
2. Open the EDMX file in the designer.
3. Click the background of the model to open the model's Properties window.
4. Change the Metadata Artifact Processing property to: Embed in Output Assembly.
5. Rebuild project- this will build a `.dll` file with the `.ssdl`, `.csdl`, and `.msl` information.

Embedded resources are specified as follows: `Metadata=res://<assemblyFullName>/<resourceName>`. Note when you use wildcard for the `assemblyFullName` (*), the Entity Framework has to look through the calling, referenced, and side-by-side assemblies for resources with the correct name. To improve performance, always specify the assembly name instead of the wildcard. Below is an excerpt from a `web.config` file that contains the `connectionString` parameter. You can see the `metadata` property is specifying the assembly name `MyApp.MyService.BusinessEntities.dll`:

```
<connectionStrings>
  <add name="MyEntities"
        connectionString="metadata=res://MyApp.ServiceRequest.BusinessEntities,
        Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=null;/provider=System.Data.SqlClient;
        provider connection string="Data Source=localhost;Initial Catalog=MyDatabase;
        Integrated Security=True;MultipleActiveResultSets=True"
        providerName="System.Data.EntityClient" />
</connectionStrings>
```

Disable Change Tracking

Once the View Generation cost is eliminated, the most expensive operation is *Object Materialization*. This operation eats up 75% of your query time because it has to read from the `DbDataReader` object and create an object. When you are using the Entity Framework, you have objects that represent the tables in your database. These objects are created by an internal process called object materialization. This process takes the returned data and builds the relevant objects for you. The object can be `EntityObject` derived objects, anonymous types, or `DbDataRecord` `DbDataRecord`.

The `ObjectContext` object will create an `ObjectStateEntry` object to help track changes made to related entities. Objects are tracked when queried, added, or attached to the cached references inside this class. The tracking behavior is specified using the `MergeOption` enumeration. When updates to properties of the tracked objects occur, the properties are marked as modified and the original values are kept for performing updates back to the database. This enables users to write code against the objects themselves and call `SaveChanges`.

We can minimize the overhead of change tracking by using the `MergeOption.NoTracking` option. Doing so will increase the performance of your system in most situations. The loss of change tracking is irrelevant if you are sending your data across the network via a web service because this feature will not work in a "disconnected" mode. Even if you are not disconnected, you can use this option in a page where there are no updates to the database. Take a look at the code snippet below for one example of how to disable change tracking:

```

public List<ServiceRequest> GetBySubmissionDate(DateTime startDate, DateTime endDate)
{
    using (new Tracer("Trace"))
    {
        using (MyEntities db = new MyEntities())
        {
            ObjectQuery<ServiceRequest> query =
                (ObjectQuery<ServiceRequest>)CompiledQueries.GetBySubmissionDate
                    (db, startDate, endDate);
            // The .Execute call is using the NoTracking option
            List<ServiceRequest> serviceRequests =
                query.Execute(MergeOption.NoTracking).ToList();
            return serviceRequests;
        }
    }
}

```

It's unclear how much the `MergeOption.NoTracking` option helps performance because every query is different. In general, I would say it's certainly worth trying. Note however the following caveats exist when using the `MergeOption.NoTracking` option:

- There is no identity resolution so, depending on your query, it's possible that you will get entities with the same primary key that are referentially distinct.
- Foreign keys are only loaded if both entities are loaded. If the other end of the association is not included, then the foreign key will not be loaded either.
- Tracked objects are cached, so subsequent calls for that object will not hit the database. If you use `NoTracking` and try to load the same object multiple times, the database will be queried each time.

Use Compiled Queries

Query Creation also takes up a good chunk of time, about 10% of your query after the EF is warmed up. Some parts of the query are cached so that subsequent queries are faster than the first. However not all parts of the query are cached leaving some parts needing to be rebuilt each time the query is executed (unless you are using `EntitySql`). We can eliminate the need for rebuilding query plans by using Compiled Queries. To compile a query for later, you can use the `CompiledQuery.Compile` method which uses a delegate:

```

public static readonly Func<MyEntities, DateTime,
    DateTime, IQueryable<ServiceRequest>> GetBySubmissionDate =
    CompiledQuery.Compile<MyEntities, DateTime, DateTime, IQueryable<ServiceRequest>>(
        (db, startDate, endDate) => (from s in db.ServiceRequest
            .Include("ServiceRequestType")
            .Include("ServiceRequestStatus")
            where s.SubmissionDate >= startDate
                && s.SubmissionDate <= endDate
            orderby s.SubmissionDate ascending
            select s);

```

Note the following caveats exist when using the `CompiledQuery.Compile` option:

- You cannot use `CompiledQueries` with parametrized Includes- so you have to use `EntitySQL` instead.
- Compiling a query for reuse takes at least double the time of simply executing it without caching.

Be Careful With Includes

You can direct the framework to do eager loading of relationships by using an `.Include` statement. With an eager load, the framework will construct a join query and the relationship data will be fetched along with the original entity data. The `.Include` statement can result in a large amount of data being brought back, it can also result in complex queries because of the need to use many `Join` statements at the data store.

```
public List<ServiceRequest> GetBySubmissionDate(DateTime startDate, DateTime endDate)
{
    using (new Tracer("Trace"))
    {
        using (MyEntities db = new MyEntities())
        {
            // The compiled query uses .Include to eager
            // load ServiceRequestType and ServiceRequestStatus
            ObjectQuery<ServiceRequest> query = (ObjectQuery<ServiceRequest>)
                CompiledQueries.GetBySubmissionDate(db, startDate, endDate);
            List<ServiceRequest> serviceRequests = query.Execute
                (MergeOption.NoTracking).ToList();

            return serviceRequests;
        }
    }
}
```

This can be avoided by using an `.Load` statement to lazy load the relationship. With lazy loading, an additional query is constructed and run for each call to load to fetch the relationship information. If you want to lazy load only certain relationships, you may choose instead to have code like that shown below:

```
public List<ServiceRequest> GetBySubmissionDate(DateTime startDate, DateTime endDate)
{
    using (new Tracer("Trace"))
    {
        using (MyEntities db = new MyEntities())
        {
            ObjectQuery<ServiceRequest> query =
                (ObjectQuery<ServiceRequest>)CompiledQueries.GetBySubmissionDate
                (db, startDate, endDate);
            List<ServiceRequest> serviceRequests =
                query.Execute(MergeOption.NoTracking).ToList();
            // Use the .Load statement to lazy load the ServiceRequestType
            foreach (ServiceRequest item in serviceRequests)
            {
                if (!item.ServiceRequestTypeReference.IsLoaded)
                    item.ServiceRequestTypeReference.Load(MergeOption.NoTracking);
            }
            return serviceRequests;
        }
    }
}
```

Note that in certain instances, eager loading is better while in others lazy loading is a better approach. In the code below, every table that has a relationship to the specified `entity` parameter is lazy loaded:

```
public static void LoadRelated(EntityObject entity, MergeOption mergeOption)
{
    System.Reflection.PropertyInfo[] properties = entity.GetType().GetProperties();

    foreach (System.Reflection.PropertyInfo propInfo in properties)
    {
        if (IsEntityReference(propInfo))
        {
            EntityReference er = (EntityReference)propInfo.GetValue(entity, null);
            if (!er.IsLoaded)
                er.Load(mergeOption);
        }

        if (IsEntityCollection(propInfo))
        {
            //The actual stored value of the EntityCollection is a RelatedEnd object.
            System.Data.Objects.DataClasses.RelatedEnd end =
                (System.Data.Objects.DataClasses.RelatedEnd)propInfo.GetValue(entity, null);

            if (!end.IsLoaded)
            {
                end.Load(mergeOption);

                //Get the enumerator off the RelatedEnd object so we can cycle
                //through items in EntityCollection without knowing the type
                System.Collections.IEnumerator enumerator = end.GetEnumerator();

                //Cycle through items in EntityCollection and add them to the list.
                while (enumerator.MoveNext())
                {
                    LoadChildren(enumerator.Current as EntityObject);
                }
            }
        }
    }
}
```

You may want to take a look at the series of articles on [Transparent Lazy Loading for Entity Framework](#) for a more readable and simple version of lazy loading using the [EFLazyLoading](#) library.

Points of Interest

There are several things to keep in mind with regard to performance when you work with the Entity Framework:

- Initial creation of the `ObjectContext` includes the cost of loading and validating the metadata.
- Initial execution of any query includes the costs of building up a query cache to enable faster execution of subsequent queries.
- Compiled LINQ queries are faster than non-compiled LINQ queries.
- Queries executed with a the `NoTracking` merge option works well when changes and relationships do not need to be tracked, such as data sent across the network.
- You may want to check out the [EdmGen2.exe](#) tool which can be used as a replacement for the original `EdmGen.exe` tool and is capable of reading and writing EDMX files.
- If you try to navigate a many-to-one or one-to-one relationship that is not loaded, you will get a `NullReferenceException`.

Acknowledgments

Special thanks to Alex Creech for crunching the numbers in the Visual Studio Profiler and writing much of our Entity Framework helper code!

Revision History

- 11th August, 2009 - Initial revision

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author

Perry Marchant




A results oriented professional building on 10 years of progressive accomplishments in Software Development and Information Technology, Perry joined TrayGames in 2005 to deliver an innovative online games platform for the consumer market. He is an MCSD and has a B.S. Degree in Computer & Information Science from Brooklyn College.

Occupation: Software Developer (Senior)

Location:  United States

Member

Discussions and Feedback

 **4 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/database/PerfEntityFramework.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)
Last Updated: 12 Aug 2009
Editor: [Deeksha Shenoy](#)

Copyright 2009 by Perry Marchant
Everything else Copyright © [CodeProject](#), 1999-2009
Web20 | [Advertise on the Code Project](#)