

SAMPLE CHAPTER

# Entity Framework 4 IN ACTION

Stefano Mostarda  
Marco De Sanctis  
Daniele Bochicchio

FOREWORD BY NOAM BEN-AMI





*Entity Framework 4 in Action*

by Stefano Mostarda, Marco De Sanctis,  
Daniele Bochicchio

**Chapter 19**

Copyright 2011 Manning Publications

# *brief contents*

---

|               |  |            |
|---------------|--|------------|
| <b>PART 1</b> | <b>REDEFINING YOUR DATA-ACCESS STRATEGY .....</b>  | <b>1</b>   |
| 1             | ■ Data access reloaded: Entity Framework           | 3          |
| 2             | ■ Getting started with Entity Framework            | 33         |
| <b>PART 2</b> | <b>GETTING STARTED WITH ENTITY FRAMEWORK .....</b> | <b>61</b>  |
| 3             | ■ Querying the object model: the basics            | 63         |
| 4             | ■ Querying with LINQ to Entities                   | 80         |
| 5             | ■ Domain model mapping                             | 119        |
| 6             | ■ Understanding the entity lifecycle               | 151        |
| 7             | ■ Persisting objects into the database             | 176        |
| 8             | ■ Handling concurrency and transactions            | 203        |
| <b>PART 3</b> | <b>MASTERING ENTITY FRAMEWORK.....</b>             | <b>225</b> |
| 9             | ■ An alternative way of querying: Entity SQL       | 227        |
| 10            | ■ Working with stored procedures                   | 253        |
| 11            | ■ Working with functions and views                 | 284        |
| 12            | ■ Exploring EDM metadata                           | 296        |
| 13            | ■ Customizing code and the designer                | 322        |

|               |   |            |
|---------------|---|------------|
| <b>PART 4</b> | <b>APPLIED ENTITY FRAMEWORK.....</b>                | <b>355</b> |
| 14            | ■ Designing the application around Entity Framework | 357        |
| 15            | ■ Entity Framework and ASP.NET                      | 378        |
| 16            | ■ Entity Framework and <i>n</i> -tier development   | 396        |
| 17            | ■ Entity Framework and Windows applications         | 423        |
| 18            | ■ Testing Entity Framework                          | 447        |
| 19            | ■ Keeping an eye on performance                     | 474        |

# 19

## *Keeping an eye on performance*

---

### ***This chapter covers***

- Comparing Entity Framework and ADO.NET performance
- Optimizing LINQ to Entities
- Optimizing Entity SQL
- Optimizing Entity Framework

When people first approach Entity Framework, one of their first questions is, “What about performance?” This is a great question that involves many aspects of Entity Framework.

Entity Framework is a layer in your application, and it’s not news that additional layers slow down performance. But that’s not always bad. Entity Framework simplifies development so much that the decreased performance can be a reasonable tradeoff. Naturally, you must still take care of performance. Fortunately, Entity Framework has several internal behaviors that you can use to gain much more performance benefit than you might imagine.

Optimizing SQL is only part of the game. For instance, LINQ to Entities queries can benefit from *query compilation*, whereas Entity SQL queries can be optimized by

enabling *plan caching*. Change tracking plays a role too. When you only read entities, disabling change tracking ensures better performance.

All these optimizations are the key to a well-performing data access layer, and in this chapter you'll learn how to use them. Furthermore, you'll learn how much Entity Framework slows down performance, compared with the classic ADO.NET approach, and you'll see how performance optimizations can make them extremely close in certain scenarios. By the end of the chapter, you'll be able not only to write robust applications, but also to make them perform smoothly.

Let's start by building a testing application that will demonstrate the performance of the Entity Framework and ADO.NET queries we'll look at.

## 19.1 Testing configuration and environment

The database we'll use for our tests is OrderIT, and it's installed on the same machine where the test application will be run. The machine has the following configuration:

- 8 GB of RAM
- T9600 dual core processor, 2.80 GHz
- Solid state disk, 250 MBps (read) and 220 MBps (write)
- Windows 7, 64-bit operating system
- SQL Server 2008

This is a good configuration for a desktop machine, but a server would be much more powerful. As a result, the performance numbers in this chapter are intended only for comparing ADO.NET and Entity Framework, or different techniques in Entity Framework itself. They aren't intended as absolute performance numbers.

To ensure an exact comparison, we'll follow these guidelines:

- *All queries must be performed 50 times.* To ensure that the context doesn't use state-manager caching to return objects, the context is created and destroyed for each query. This means a connection is opened and closed each time. In ADO.NET code, you do exactly the same thing, which ensures that ADO.NET and Entity Framework perform exactly the same tasks.
- *In ADO.NET, all fields must be iterated.* Entity Framework reads all data from a query to shape it into its internal format. When using ADO.NET, you have to read all columns to simulate the same behavior.
- *500 customers and suppliers are added in the scope of a transaction.* A good test can't ignore updates to the database. You insert 500 records in a single `SaveChanges` call, which opens a transaction, issues the commands to the database, and then commits it. The same steps are reproduced in ADO.NET.
- *An Entity Framework warm-up query must be executed before issuing the commands.* This is necessary to ensure that metadata and other stuff necessary to Entity Framework is already in place when running the test. Otherwise, the test might be influenced by initialization processes.

The tests are performed in a Windows Forms application that offers a nice way of seeing the results.

### 19.1.1 *The performance test visualizer*

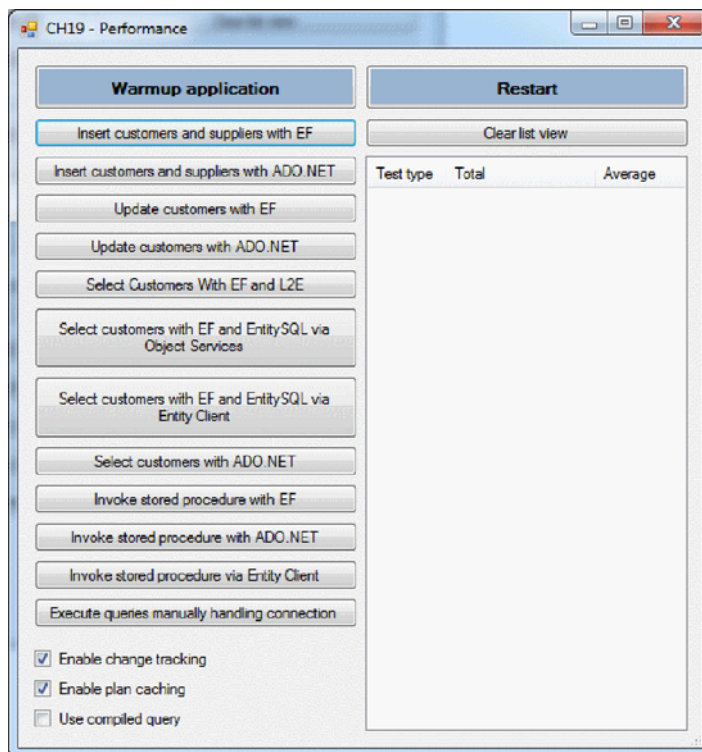
The test application is easy to understand. It contains a single form with a set of buttons on the left to start the tests, and a `ListView` on the right to show the results. The `ListView` shows the test type, the total execution time, the average command-execution time, the first command-execution time, and the execution time for all the other commands (each time is shown in milliseconds). This form is shown in figure 19.1.

It's particularly important to separate the first execution time from subsequent times because, as you'll discover, in Entity Framework the first command is the slowest and the others are much faster. You'll learn in this chapter how to mitigate this issue and how much it improves performance.

To test performance, you need to measure, and to measure you need a timer.

### 19.1.2 *Building the timer*

To build the timer, you create a class that wraps the `System.Diagnostics.Stopwatch` class: `Watch`. Its purpose isn't only measuring execution time, but also writing the result to the output `ListView` so that showing the result has minimal impact on the



**Figure 19.1** The testing application. On the left are the buttons to start the tests and check boxes to enable various optimizations. The right `ListView` shows the results of each test.

test code. (We could have used the Template Method pattern, but we opted for the “get in and out quick” approach). Here’s the `Watch` class.

### Listing 19.1 The custom timer

**C#**

```
public class Watch
{
    Stopwatch _sw;
    ListView _result;
    string _testType;
    List<long> _laps;

    public Watch(ListView result)
    {
        _result = result;
    }

    public void Start(string testType)
    {
        _laps = new List<long>();
        _sw = new Stopwatch();
        _testType = testType;
        _sw.Start();
    }

    public void SaveLap()
    {
        _laps.Add(_sw.ElapsedMilliseconds);
    }

    public void Stop()
    {
        _sw.Stop();
        if (_laps.Count > 0)
        {
            string[] localTimes = new string[_laps.Count-1];
            for (int i = 1; i < _laps.Count-1; i++)
            {
                localTimes[i-1] =
                    (_laps[i] - _laps[i - 1]).ToString();
            }
            _result.Items.Add(new ListViewItem(new[] {
                _testType,
                _sw.ElapsedMilliseconds.ToString(),
                ((double)_sw.ElapsedMilliseconds /
                    (double)_laps.Count).ToString(),
                _laps[0].ToString(),
                String.Join(", ", localTimes) }));
        }
        else
            _result.Items.Add(new ListViewItem(new[] {
                _testType,
                _sw.ElapsedMilliseconds.ToString(),
                _sw.ElapsedMilliseconds.ToString(),
                "0", String.Empty }));
    }
}
```

1 Stores output  
ListView

2 Sets up  
timer

3 Stores  
iteration length

4 Stops timer

5 Writes result  
in ListView for  
iterations

6 Writes result  
in ListView for  
single query

**VB**

```

Public Class Watch
    Private _sw As Stopwatch
    Private _result As ListView
    Private _testType As String
    Private _laps As List(Of Long)

    Public Sub New(ByVal result As ListView)
        _result = result
    End Sub

    Public Sub Start(ByVal testType As String)
        _laps = New List(Of Long)()
        _sw = New Stopwatch()
        _testType = testType
        _sw.Start()
    End Sub

    Public Sub SaveLap()
        _laps.Add(_sw.ElapsedMilliseconds)
    End Sub

    Public Sub [Stop]()
        _sw.Stop()
        If _laps.Count > 0 Then
            Dim localTimes As String() =
                New String(_laps.Count - 2)
            For i As Integer = 1 To _laps.Count - 2
                localTimes(i - 1) =
                    (_laps(i) - _laps(i - 1)).ToString()
            Next
            _result.Items.Add(New ListViewItem(New () {
                _testType,
                _sw.ElapsedMilliseconds.ToString(),
                (Cdbl(_sw.ElapsedMilliseconds) /
                    Cdbl(_laps.Count)).ToString(),
                _laps(0).ToString(),
                String.Join(", ", localTimes)})
        Else
            _result.Items.Add(New ListViewItem(New () {
                _testType,
                _sw.ElapsedMilliseconds.ToString(),
                _sw.ElapsedMilliseconds.ToString(),
                "0", String.Empty}))
        End If
    End Sub
End Class

```

**1** Stores output  
ListView

**2** Sets up  
timer

**3** Stores  
iteration length

**4** Stops timer

**5** Writes result  
in ListView for  
iterations

**6** Writes result  
in ListView for  
single query

Watch's constructor **1** takes as input the list view the result are written into when a test is finished.

The Start method resets the list view internal state, saves the test type that's passed as an argument, and then starts the timer **2**.

SaveLap saves in a list the number of milliseconds that have passed since Start was invoked **3**. Calling SaveLap at each iteration allows you to know how long each operation takes to execute.

The `Stop` method stops the timer ❹. If any laps are stored, it first calculates the exact execution time for each iteration and then reports it in the list view ❺. If no laps are stored, there's no calculation, and the data is reported as a single execution ❻.

Now you know what you're going to do, how you're going to do it, and how you'll measure it. It's time to see some real tests. Because you can't query empty tables, you'll start by using both Entity Framework and classic ADO.NET to pour data into a table. You'll then measure the performance of both approaches and compare them.

## 19.2 Database-writing comparison

Writing data to the database means adding, modifying, and deleting rows. From a database perspective, these operations affect performance in very different ways; but we're interested only in caller performance so we'll only use the `INSERT` command to make comparisons.

**NOTE** Because the test code is verbose and of no great interest, we'll only show the C# code in this chapter. The VB code is included in the book's source code.

The test of the `INSERT` commands creates 500 `Customer` and 500 `Supplier` objects, adds them to the context, and calls the `SaveChanges` method, as shown in the following listing.

### Listing 19.2 Inserting customers and suppliers using Entity Framework

```
using (OrderITEntities ctx = new OrderITEntities())
{
    for (int i = 0; i < 500; i++)
    {
        Customer c = CreateCustomer();
        ctx.Companies.AddObject(c);

        Supplier s = CreateSupplier();
        ctx.Companies.AddObject(s);
    }
    _watch.Start("Insert 500 customers and
    ➤ 500 suppliers with EF");
    ctx.SaveChanges();
    _watch.Stop();
}
```

This listing creates a customer and adds it to the context, creates a supplier and adds it to the context, and then measures persistence performance. The object-creation phase isn't timed. You only care about the `SaveChanges` duration, because that code performs the same work you'd perform when using ADO.NET: it opens a connection, starts a transaction, and issues 500 commands to insert customers and 500 to insert suppliers. The comparable ADO.NET code is shown in the following listing.

**Listing 19.3 Inserting customers and suppliers using ADO.NET**

```

_watch.Start("Insert 500 customers and
➡ 500 suppliers with ADO.NET");           ← Starts timer
using (var conn = new SqlConnection(connString)) ← Creates connection
{
    conn.Open();
    using (SqlTransaction tr = conn.BeginTransaction()) ← Starts transaction
    {
        try
        {
            for (int i = 0; i < 500; i++)
            {
                using (SqlCommand comm = new SqlCommand("", conn, tr))
                {
                    comm.CommandText = GetCustomerSQL();
                    comm.Parameters = GetCustomerParams();
                    comm.ExecuteNonQuery();           ← Saves customer
                }
            }
            for (int i = 0; i < 500; i++)
            {
                using (SqlCommand comm = new SqlCommand("", conn, tr))
                {
                    comm.CommandText = GetSupplierSQL();
                    comm.Parameters = GetSupplierParams();
                    comm.ExecuteNonQuery();         ← Saves supplier
                }
            }
            tr.Commit();
        }
        catch (Exception er)
        {
            tr.Rollback();
        }
    }
}
_watch.Stop();

```

The `GetCustomerSQL` and `GetSupplierSQL` methods return the SQL to insert a customer and a supplier, respectively. And as you may guess, `GetCustomerParams` and `GetSupplierParams` return the parameters for the SQL.

We executed both tests three times, but we wanted more. We also wanted to find out what would happen with lower and higher numbers of records to write to the table. Table 19.1 shows the results with 100, 5,000, and 10,000 records.

These results offer lots of information. First, the percentage difference between Entity Framework and ADO.NET when performing bulk inserts of a small number of objects the difference is negligible; but as number of objects grows, the difference increases to an unacceptable level. What's interesting is that the more objects there are, the more slowly the difference grows.

It's rare that an application created for human beings inserts a huge number of objects in a single transaction, so don't worry too much about the preceding statistics. For small numbers of inserts, you can hardly spot the difference.

**Table 19.1** Comparing the performance of Entity Framework and ADO.NET with different numbers of records

| Technology       | Items  | 1st attempt | 2nd attempt | 3rd       | Difference |
|------------------|--------|-------------|-------------|-----------|------------|
| ADO.NET          | 100    | 493 ms      | 491 ms      | 486 ms    | —          |
| Entity Framework | 100    | 530 ms      | 513 ms      | 521 ms    | +6.39%     |
| ADO.NET          | 1,000  | 4,626 ms    | 4,892 ms    | 4712 ms   | —          |
| Entity Framework | 1,000  | 5,386 ms    | 5,385 ms    | 5,660 ms  | +15.46%    |
| ADO.NET          | 5,000  | 24,906 ms   | 25,325 ms   | 24,968 ms | —          |
| Entity Framework | 5,000  | 30,536 ms   | 29,914 ms   | 29,535 ms | +19.66%    |
| ADO.NET          | 10,000 | 48,308 ms   | 47,496 ms   | 48,324 ms | —          |
| Entity Framework | 10,000 | 58,297 ms   | 57,659 ms   | 58,325 ms | +20.92%    |

In contrast, if you have to perform a bulk insert, Entity Framework and other O/RM tools are your enemy. O/RM tools simplify development, but using them for bulk transactions is a capital sin. You're better off relying on ADO.NET or database-specific tools.

When it comes to updates and deletes, the situation doesn't change. Updating 1,000 rows is slower than inserting them, but that's due to database internals. Both Entity Framework and ADO.NET suffer equally from this performance slowdown. Entity Framework is obviously slower than ADO.NET, but in both cases you can do nothing to improve performance for database updates.

Let's now move on to talk about queries.

### 19.3 Query comparisons in the default environment

To test queries, you'll first test them in the environment you created. Then, you'll start introducing the tweaks necessary to speed up things a little.

As we mentioned in section 19.1, the query you'll use returns customers whose names start with *C*, and it's performed 50 times. To embrace the spectrum of querying technologies, we've created tests for ADO.NET, LINQ to Entities, Entity SQL via the `ObjectContext`, and Entity SQL via Entity Client.

Unlike the insert test, you won't warm up the application for this test. Each test is isolated from the others; after each test, the application is restarted to make sure the test is performed in a clean environment. We wanted you to see the performance with the default configuration, and how much performance improves when you optimize some aspects.

In chapter 3, you learned that you can perform queries through the Object Services layer and through the Entity Client. In the following listing, you see the test code that uses Object Services. Later in this section, we'll show you the test code that uses Entity Client. Here's the test.

**Listing 19.4 Retrieving customers using Object Services**

```

private void SelectCustomersViaObjectServices(string testType,
    bool enableTracking, bool useCompiledQuery, bool enablePlanCaching,
    bool useEntitySQL)
{
    _watch.Start(testType);
    for (int i = 0; i < 50; i++)
    {
        using (OrderITEntities ctx = new OrderITEntities())
        {
            List<Customer> items;
            if (!enableTracking)
                ctx.Companies.MergeOption =
                    MergeOption.NoTracking;

            if (!useEntitySQL)
            {
                if (useCompiledQuery)
                {
                    var it = compQuery.Invoke(ctx, "C");
                    foreach (var item in it)
                        object o = item;
                }
                else
                {
                    string name = "C";
                    items = ctx.Companies.OfType<Customer>()
                        .Where(c => c.Name.StartsWith(name))
                        .ToList();
                }
            }
            else
            {
                var oq = ctx.CreateQuery<Customer>("SELECT VALUE c
FROM OFTYPE(OrderITEntities.Companies, OrderIT.Model.Customer)
AS c WHERE c.name LIKE @name");

                if (!enableTracking)
                    oq.MergeOption = MergeOption.NoTracking;

                if (!enablePlanCaching)
                    oq.EnablePlanCaching = false;

                oq.Parameters.Add(new ObjectParameter("name", "C%"));
                items = oq.ToList();
            }
        }
        _watch.SaveLap();
    }
    _watch.Stop();
}

```

**1 Sets change-tracking****2 Uses compiled LINQ to Entities query****3 Uses classic LINQ to Entities query****4 Uses EntitySQL query****1 Sets change-tracking****5 Sets plan caching**

This code tests the queries via Object Services using LINQ to Entities compiled queries **2** (more about that in the next section), LINQ to Entities **3**, and Entity SQL **4**. It also considers plan caching **5** and change-tracking **1** depending on the parameters.

The following listing performs queries via the Entity Client.

### Listing 19.5 Retrieving customers using Entity Client

```

string testType = "Select Customers using EntitySQL via Entity Client. ";
_watch.Start(testType);
for (int i = 0; i < 50; i++)
{
    using (var conn = new EntityConnection(connString) ← ❶ Opens connection
    {
        conn.Open();
        using (var comm = new EntityCommand(
            "SELECT VALUE c
            FROM OFTYPE(OrderITEntities.Companies,
            OrderITModel.Customer)
            AS c WHERE c.name LIKE @name", conn))
        {
            comm.EnablePlanCaching =
                enablePlanCaching.Checked;
            comm.Parameters.AddWithValue("name", "C%");
            using (EntityDataReader reader =
                comm.ExecuteReader(
                    CommandBehavior.SequentialAccess))
            {
                while (reader.Read())
                {
                    var x = reader.GetValue(0);
                    var x1 = reader.GetValue(1);
                    var x2 = reader.GetValue(2);
                    var x3 = reader.GetValue(3);
                    var x4 = reader.GetValue(4);
                }
            }
        }
    }
    _watch.SaveLap();
}
_watch.Stop();

```

❷ Creates command

❸ Sets plan caching

❹ Executes query

❺ Iterates over query result

This code tests queries via Entity Client. It first opens a connection ❶, and then it creates a command ❷, enables plan-caching depending on a check box in the form ❸, executes the query ❹, and iterates over the records returned by it ❺.

The code for ADO.NET is identical to listing 19.5, except that the `Entity*` objects are replaced by `Sql*` objects, and the connection string is in plain old ADO.NET format. You can find it in the book's source code.

Table 19.2 shows the performance comparison between the various techniques. Ouch! We knew Entity Framework was likely to lose, but we didn't expect LINQ to Entities to take *six* times longer than ADO.NET (LINQ to Entities is the most-adopted query technique).

Things get better when Entity SQL comes into play, because it's much easier to parse than LINQ to Entities. But when used via the `ObjectContext` class's `CreateQuery<T>` method, it takes almost *five* times longer than ADO.NET!

**Table 19.2** Comparing performance of Entity Framework and ADO.NET in querying customers whose names start with C

| Technology                     | Total    | Average  | Difference |
|--------------------------------|----------|----------|------------|
| ADO.NET                        | 171 ms   | 3.42 ms  | —          |
| LINQ to Entities               | 1,078 ms | 21.56 ms | +530%      |
| Entity SQL via Object Services | 843 ms   | 16.86 ms | +392%      |
| Entity SQL via Entity Client   | 420 ms   | 8.4 ms   | +145%      |

If you skip object generation and opt for the Entity Client, Entity Framework takes more than twice the time ADO.NET takes. This is because, as you learned in chapter 3, Entity Client creates a `DbDataReader` whose columns match the conceptual format instead of the database format. This is one of the heaviest tasks Entity Client performs in the query pipeline (remember that this task is performed even when using LINQ to Entities and Entity SQL via `ObjectContext`).

A couple of years ago, the ADO.NET team published an interesting blog post about Entity Framework performance: “Exploring the Performance of the ADO.NET Entity Framework, Part 1” (<http://mng.bz/27XQ>). What is great about this post is that it separates tasks performed in the query pipeline, showing how long each of them takes to execute. You’re going to use that information to make Entity Framework much faster.

## 19.4 Optimizing performance

The numbers shown in table 19.2 could make you think that Entity Framework isn’t worth the effort. You’ll be glad to know that there are four areas where you can do a lot to improve performance, and in this section we’ll look at them in detail:

- View generation
- LINQ to Entities query compilation
- Entity SQL plan caching
- Change tracking

If you optimize these points, you’ll be surprised by how much Entity Framework gains on ADO.NET. Let’s start with the first point, which is very important.

### 19.4.1 Pregenerating views

Views are Entity SQL statements that retrieve data from entity sets and association sets declared in the SSDL and CSDL. These commands are used internally by the Entity Client to generate the final SQL.

According to the blog post we mentioned earlier, generating views is the most expensive task in the query pipeline, taking 56% of overall time. Fortunately, views are generated once per `AppDomain`, so when they’re created, the other queries can reuse them, even if you use other contexts or the Entity Client.

As a result of view generation, the first query is tremendously slow. If you had warmed up the application by issuing a dummy query, the results in table 19.2 would have been much different because the views would have already been generated.

Table 19.3 shows the results after the warmup.

**Table 19.3 Comparing performance of Entity Framework and ADO.NET querying customers whose names start with C" after a warmup**

| Technology                     | Total  | Average  | Difference |
|--------------------------------|--------|----------|------------|
| ADO.NET                        | 171 ms | 3.42 ms  | —          |
| LINQ to Entities               | 975 ms | 19.5 ms  | +470%      |
| Entity SQL via Object Services | 788 ms | 15.76 ms | +360%      |
| Entity SQL via Entity Client   | 416 ms | 8.32 ms  | +143%      |

Queries executed via Object Services (rows 2 and 3 of table 19.3) are now faster, and Entity Framework gains on ADO.NET. In contrast, queries issued using Entity Client perform pretty much the same as before (compare table 19.3 with table 19.2).

But although issuing a dummy query works, it's a waste. What's worse, when you deal with large models, Entity Framework takes its time to generate views. You don't want users to wait a long time before starting to use an application. To improve on this, you can generate the views at design time using the *EdmGen* tool, as shown in the following listing, and then compile them into the project.

#### Listing 19.6 Using EdmGen to generate views

```
%windir%\Microsoft.NET\Framework\v4.XXXX\EdmGen.exe"
/nologo
/language:C#
/mode:ViewGeneration
"/inssdl:c:\OrderIT\model.ssd1"
"/incsd1:c:\OrderIT\model.csd1"
"/inmsl:c:\OrderIT\model.msl"
"/outviews:c:\OrderIT\School.Views.cs"
```

EdmGen needs the CSDL, SSDL, and MSL files, and it returns a C# or VB file, depending on the `/language` switch, containing the views. Now you just need to add the file to the project and compile everything.

The problem with this approach is that you need the three EDM files, but you only have the EDMX. You can let the designer generate the three files, but then you need to remember to reference them differently in the connection string and deploy them along with the rest of the application. This process is awkward.

There's another path you can follow:

- 1 Set the designer to generate the three files on build.
- 2 Build the application.

- 3 Launch EdmGen in a post-build event.
- 4 Reset the designer to embed the files in the application.
- 5 Build it again.

This is even worse than the previous option.

What you really want is a simpler way to generate the views at design time, without touching the designer. Fortunately, the solution is astonishingly simple: use a template.

#### PREGENERATING VIEWS VIA TEMPLATE

Although view generation happens internally, the APIs are public, which means you can generate views from code. You can write a template that reads the EDMX, extracts the three files, and invokes the API to generate the views. Listing 19.7 contains the main code of such a template.

#### Listing 19.7 The main part of the template that generates views

##### C#

```
using (StreamWriter writer = new StreamWriter(new MemoryStream()))
{
    XmlReader csdlReader = null;
    XmlReader mslReader = null;
    XmlReader ssdlReader = null;

    GetConceptualMappingAndStorageReaders(edmxFilePath,
        out csdlReader, out mslReader, out ssdlReader);
    var edmItems = new EdmItemCollection(
        new XmlReader[] { csdlReader });
    var storeItems = new StoreItemCollection(
        new XmlReader[] { ssdlReader });
    var mappingItems = new StorageMappingItemCollection(
        edmItems, storeItems,
        new XmlReader[] { mslReader });

    EntityViewGenerator viewGenerator =
        new EntityViewGenerator();
    viewGenerator.LanguageOption =
        LanguageOption.GenerateCSharpCode;
    IList<EdmSchemaError> errors =
        viewGenerator.GenerateViews(mappingItems, writer);

    foreach (EdmSchemaError e in errors)
        this.Error(e.Message);

    MemoryStream memStream =
        writer.BaseStream as MemoryStream;
    this.WriteLine(
        Encoding.UTF8.GetString(memStream.ToArray()));
}
```

1 Creates EDM files from EDMX

2 Generates view code

3 Enumerates errors

4 Writes view code in file

##### VB

```
Using writer As New StreamWriter(New MemoryStream())
    Dim csdlReader As XmlReader = Nothing
    Dim mslReader As XmlReader = Nothing
    Dim ssdlReader As XmlReader = Nothing
```

```

GetConceptualMappingAndStorageReaders(
    edmxFilePath, csdlReader, mslReader, ssdlReader)

Dim edmItems As New EdmItemCollection(
    New XmlReader() {csdlReader})
Dim storeItems As New StoreItemCollection(
    New XmlReader() {ssdlReader})
Dim mappingItems As
    New StorageMappingItemCollection(edmItems,
        storeItems, New XmlReader() {mslReader})

Dim viewGenerator As New EntityViewGenerator()
viewGenerator.LanguageOption =
    LanguageOption.GenerateVBCode
Dim errors As IList(Of EdmSchemaError) =
    viewGenerator.GenerateViews(mappingItems, writer)

For Each e As EdmSchemaError In errors
    Me.Error(e.Message)
Next

Dim memStream =
    TryCast(writer.BaseStream, MemoryStream)
Me.WriteLine(
    Encoding.UTF8.GetString(memStream.ToArray()))
End Using

```

1 Creates EDM files from EDMX

2 Generates view code

3 Enumerates errors

4 Writes view code in file

In the first part of this code, a stream for each EDM file is extracted from the EDMX, and the item collections are instantiated using those streams ①. When that's done, the `EntityViewGenerator` class (situated in the `System.Data.Entity.Design` namespace inside the `System.Data.Entity.Design` assembly) is instantiated, and its `GenerateView` method is invoked, passing the item collections joined into one, and the writer that the code will be written into ②. Finally, any errors are sent to the output ③, and the stream containing the file is serialized as a string and is written to the output file ④.

This template has already been created and has been made available for download by the Entity Framework team through a blog entry entitled, “How to use a T4 template for View Generation” (<http://mng.bz/8ZNK>).

We strongly recommend pregenerating views via a template. You'll gain all the runtime benefits of the already-compiled views without any design-time drawbacks.

Naturally, pregeneration speeds up things only at startup. Let's now look at a mechanism every LINQ to Entities query can benefit from: *query compilation*.

### 19.4.2 Compiling LINQ to Entities queries

LINQ to Entities queries are parsed at runtime by the Object Services layer. The more complex a query is, the more time the parsing process takes. The Entity Framework team knows that, and they've added the option to compile queries. By compiling queries, you tell Object Services to parse the query and cache the generated command tree. The next time the query is executed, the Object Services layer doesn't have to parse the query again; it uses the cached command instead.

**Table 19.4** Comparing performance of Entity Framework and ADO.NET with and without LINQ to Entities query compilation

| Technology                          | 1st query | Other query average |
|-------------------------------------|-----------|---------------------|
| ADO.NET                             | 4 ms      | 2.59 ms             |
| LINQ to Entities (no compilation)   | 378 ms    | 12.38 ms            |
| LINQ to Entities (with compilation) | 378 ms    | 10.01 ms            |

Obviously, the first time you execute a particular query, you get no benefits. The benefits become evident from the second execution on, as you see in table 19.4.

As you can see, compiling queries helps hugely in improving performance, but it requires a different style of coding. You must choose during development which queries must be compiled and which must not; introducing this feature later means changing a lot of code.

#### WRITING A COMPILED QUERY

At first sight, creating a compiled query may look odd because it involves the `CompiledQuery` class in association with a predicate. Don't worry—after a bit of practice, it will look familiar.

To create a compiled query, you have to write a predicate. It must accept the context and any parameters you need for the query, and it must return an `IQueryable<Customer>` instance. To instantiate the predicate, you have to invoke the `CompiledQuery` class's `Compile` method, passing in the LINQ to Entities query. The last thing you need to do is make the predicate static.

In this example, the predicate accepts the context plus a string containing the first letter of the customers' names you're looking for in the query. (Remember, in the previous examples you looked for customers whose names start with *C*.) Here's an example of a compiled query.

#### Listing 19.8 The compiled query code

##### C#

```
static Func<OrderITEntities, string, IQueryable<Customer>> compQuery =
    CompiledQuery.Compile<OrderITEntities, string, IQueryable<Customer>>(
        (ctx, name) => ctx.Companies
            .OfType<Customer>()
            .Where(c => c.Name.StartsWith(name)
        )
    );
```

##### VB

```
Shared compQuery As Func(Of OrderITEntities, String,
    IQueryable(Of Customer)) =
    CompiledQuery.Compile(Of OrderITEntities, String,
    IQueryable(Of Customer))
        (Function(ctx, name) ctx.Companies
            .OfType(Of Customer) ()
```

```

        .Where(Function(c) c.Name.StartsWith(name)
    )
)

```

The first generic parameter of the predicate *must* be the context, and the last represents the return type. Between those two, you can specify any parameters you need.

To use the query you just need to invoke the predicate, passing in the context and the string "C":

**C#**

```
var result = compQuery.Invoke(ctx, "C").ToList();
```

**VB**

```
Dim result = compQuery.Invoke(ctx, "C").ToList()
```

That's all you need to do to compile a LINQ to Entities query. As you can see, it's pretty simple. The Object Services layer takes care of compiling the query and reusing the compiled version after the first execution. When the Object Services layer looks for an existing compiled query, it uses the parameters too. This means that if you search for customers whose names start with *C* and then for customers whose names start with *A*, Object Services caches two command trees.

Despite their simplicity, compiled queries have some nasty internals that are worth mentioning.

**COMPILED QUERY INTERNALS**

The first thing to know about compiled queries is that nothing happens until they're executed. `CompiledQuery.Invoke` does nothing unless a `ToList`, `ToArray`, `foreach`, or `Execute` forces the query to execute.

Another aspect of compiled queries is that if you combine them with other LINQ to Entities operators, you lose all benefits: the entire query is recompiled. For instance, if you use the query in listing 19.8 and then attach the `First` operator, as shown here, the entire command is reparsed and nothing is reused:

**C#**

```
var result = compQuery.Invoke(ctx, "C").First();
```

**VB**

```
Dim result = compQuery.Invoke(ctx, "C").First()
```

To mitigate this problem, you can call the `AsEnumerable` method, as in the following snippet. It fetches data from the database and then uses LINQ to Object operators or creates a brand-new compiled query:

**C#**

```
var result = compQuery.Invoke(ctx, "C").AsEnumerable().First();
```

**VB**

```
Dim result = compQuery.Invoke(ctx, "C").AsEnumerable().First()
```

Things get trickier when `MergeOption` comes into play, because `MergeOption` is set at query level. This means that when the query is compiled, the merge option is saved

along with the compiled version. After it's created, you can't change the merge option of a compiled query. The next piece of code shows what problems you might face:

**C#**

```
ctx1.Companies.MergeOption = MergeOption.NoTracking;
var c = compiledQuery(ctx1, "C").AsEnumerable().First();

ctx2.Companies.MergeOption = MergeOption.AppendOnly;
var c = compiledQuery(ctx2, "C").AsEnumerable().First();
```

**VB**

```
ctx1.Companies.MergeOption = MergeOption.NoTracking
var c = compiledQuery(ctx1, "C").AsEnumerable().First()

ctx2.Companies.MergeOption = MergeOption.AppendOnly
var c = compiledQuery(ctx2, "C").AsEnumerable().First()
```

In the first case, `MergeOption` is set to `NoTracking`, so the object is in the `Detached` state. In the second case, even if the merge option has changed, the state of the object remains `Detached` because the query was compiled with the `NoTracking` option. If you need to perform the same query with different merge options, it's best to create two separate predicates.

This is a nasty behavior, and it's likely to change in future versions. For the moment, you need to be aware of it so you can avoid the pitfalls.

### 19.4.3 **Enabling plan caching for Entity SQL**

When you execute an Entity SQL query, the Entity Client parses it and then saves it in a cache along with its counterpart in native SQL. The second time the query is executed, before it parses the query again, the Entity Client checks whether a copy already exists in the cache.

The check is based on the Entity SQL string and query parameters, and it's case sensitive. "SELECT VALUE ..." is different from "Select Value ...". If you create two queries like those, you'll end up with two different entries for the same query, and that's a waste of resources.

We strongly recommend choosing a convention so you can avoid that problem. For example, use uppercase for keywords and write object names using Pascal case, as in the following query:

```
SELECT VALUE c FROM OrderITEntities.Companies AS c
```

Plan caching is enabled by default, so you have it for free. Should you need to disable it, you can do so via the `EnablePlanCaching` property of the `ObjectSet<T>` and `EntityCommand` classes. Table 19.5 shows the benefits of having plan caching enabled.

**Table 19.5 Entity SQL performance with plan caching enabled and disabled**

| Technology                     | Plan caching enabled | Plan caching disabled |
|--------------------------------|----------------------|-----------------------|
| Entity SQL via Object Services | 9.66 ms              | 11.66 ms              |
| Entity SQL via Entity Client   | 3.82 ms              | 4.84 ms               |

There's one last optimization to keep in mind, and it regards change tracking. Often you read objects but don't need to update them. Let's see how this scenario can be optimized.

#### 19.4.4 Disabling tracking when it's not needed

When you display customers in a grid, you probably don't allow them to be modified. In such a scenario, change tracking is unnecessary, so you can disable it by setting the `MergeOption` of a query property to `MergeOption.NoTracking`.

This optimization lets object generation in the context skip many steps, and it has a dramatic impact on performance, as you can see in table 19.6.

**Table 19.6 Performance with change tracking enabled and disabled (average)**

| Technology                          | Enabled  | Disabled | Difference |
|-------------------------------------|----------|----------|------------|
| ADO.NET                             | 2.59     | 2.59 ms  | —          |
| LINQ to Entities (with compilation) | 10.21 ms | 3.44 ms  | +32%       |
| Entity SQL via Object Services      | 9.62 ms  | 3.33 ms  | +28%       |

As you can see, disabling change tracking makes object generation much quicker. We strongly recommend removing change tracking from your application wherever it's unnecessary.

You have optimized a lot. For LINQ to Entities queries, you went from +530% to +32% greater time, as compared with ADO.NET. This is why we always suggest checking not only your SQL code but your Entity Framework code too. We have encountered many cases where the problem wasn't the SQL produced by Entity Framework, but the total, or sometimes partial, lack of code optimization.

Another area where you can make some optimizations is in stored-procedure execution. These types of queries are interesting, because they involve a different materialization process.

#### 19.4.5 Optimizing stored procedures

You know that invoking a stored procedure triggers a different materialization mechanism. To cover the various cases, we created a stored procedure that returns all order details and invoked it from Object Services, Entity Client, and ADO.NET so we could compare the results, which are shown in table 19.7.

**Table 19.7 Stored-procedure execution performance**

| Technology      | Total | Average | Difference |
|-----------------|-------|---------|------------|
| ADO.NET         | 11 ms | 0.22 ms | —          |
| Object Services | 23 ms | 0.46 ms | +109%      |
| Entity Client   | 19 ms | 0.38 ms | +72%       |

As you may have expected, ADO.NET is faster than any other technique. What's interesting here is that Entity Framework isn't that much slower. If you compare these results with those in table 19.3, you'll see that the materialization mechanism used by stored procedures is much faster than that used for queries. We've used a table that contains less data here, but that's not the point. What we wanted to stress is the performance of the different materialization mechanisms.

You know that the methods added to the context for calling the stored procedures hide the internal call to the context's `ExecuteFunction<T>` method. This method has two overloads: the first accepts the function name and the parameters, and the second accepts both of those plus the `MergeOption`. The methods added to the context for calling stored procedures internally use only the first overload, so by default you have the `AppendOnly` behavior.

If you don't need change tracking for entities returned by the stored procedure, you can directly invoke the `ExecuteFunction<T>` method using the second overload and passing the `MergeOption.NoTracking` value, as shown in the next example:

**C#**

```
var parameters = CreateParameters();
var details = ctx.ExecuteFunction<OrderDetail>("GetDetails",
    MergeOption.NoTracking, parameters);
```

**VB**

```
Dim parameters = CreateParameters()
Dim details = ctx.ExecuteFunction(Of OrderDetail)("GetDetails",
    MergeOption.NoTracking, parameters)
```

Even better, you can modify the template that generates the context to create two overloads for each stored procedure, so you don't have to manually invoke the `ExecuteFunction<T>` method. (This book's source code contains such a template.)

There's nothing more to add about performance. Now you know what the bottlenecks are, what you can do to avoid common pitfalls, and, most important, what you can do to make database access via Entity Framework much quicker. It's not as fast as ADO.NET, but it surely makes your life easier.

## 19.5 Summary

Performance is a key point of almost any data-driven application. If you're developing a simple catalog for your DVD collection, performance probably won't even cross your mind; but in enterprise applications with lots of clients and lots of queries, you can't ignore it.

Entity Framework was built with one eye always on performance. Almost every aspect of it has been optimized or made a target for an opt-in optimization. Plan caching and LINQ to Entities query compilation are great examples of opt-in optimizations. Unfortunately, generated SQL isn't always as good as you might like, but you can resort to stored procedures and live happily.

Always remember these golden rules: check the SQL with a profiler, use compiled LINQ to Entities queries, and disable change tracking as much as possible. That's your take-away.

Congratulations! You've reached the end of the book. It's been a long journey, but now you have a solid understanding of Entity Framework. You can develop an application using Entity Framework and solve most of the problems you're likely to encounter.

# Entity Framework 4 IN ACTION

Mostarda • De Sanctis • Bochicchio



Entity Framework builds on the ADO.NET persistence model and the language features of LINQ to create a powerful persistence mechanism that bridges the gap between relational databases and object-oriented languages.

**Entity Framework 4 in Action** is an example-rich tutorial that helps .NET developers learn and master the subject. It begins by explaining object/relational mapping and then shows how you can easily transition to EF from ADO.NET. Through numerous focused examples and two larger case studies, the book unfolds the EF story in a clear, easy-to-follow style. Infrastructure and inner workings of EF are discussed when you need them to understand a particular feature.

## What's Inside

- Full coverage of EF 4 features
- Layer separation, Data Layer, and Domain Model
- Best practices

This book is written for .NET developers. Knowledge of ADO.NET is helpful but not required.

**Stefano Mostarda**, **Marco De Sanctis**, and **Daniele Bochicchio** are all ASP.NET MVPs and core members of ASPItalia.com, Italy's largest .NET community. Collectively, they have over 25 years of .NET development experience.

For access to the book's forum and a free eBook for owners of this book, go to [manning.com/EntityFramework4inAction](http://manning.com/EntityFramework4inAction)

"Answers questions you didn't even know you had."

—From the Foreword by  
Noam Ben-Ami, Microsoft

"Finally, real insight into the Framework."

—Christian Siegers, Capgemini

".NET database development at the next level."

—Berndt Hamboeck, pmOne

"Easy to read, comprehend, and apply."

—David Barkol, Neudesic

"All you need to conquer the ORM beast."

—Jonas Bandi, TechTalk

ISBN 13: 978-1-935182-18-4  
ISBN 10: 1-935182-18-8



9 781935 182184