redgate

52 TIPS & TRICKS TO BOOST .NET PERFORMANCE

This is a .NET community generated eBook with contributions from .NET experts everywhere

SRINIVASU ACHALLA

BARTOSZ ADAMCZEWSKI

DAVID BERRY

GIORGI DALAKISHVILI

JONATHAN DANYLKO

RAKESH NHALIL EDAVALATH

BEN EMMETT

SHUBHAJYOTI GHOSH

PAUL GLAVICH

NICK HARRISON

ZIJIAN HUANG

ROB KARATZAS

ANDRE KRAEMER

GAVIN LANATA

MEGHA MAHESHWARI

NIK MOLNAR

COSIMO NOBILE

JEFF OSIA

RAGHAVENDRA RENTALA

RICHIE RUMP

LOUIS SOMERS

MICHAEL SORENS

RAINER STROPEK

TUGBERK UGURLU

MATT WARREN

GREG YOUNG

What do you do when you have a .NET problem?

Your code is running slow. There's too much load on your database. Memory usage is causing performance problems.

You get the picture.

One solution is to sit back and ask the advice of fellow .NET experts. Developers who have had exactly the same problem at some point and spent hours finding the right solution.

That's what we did. We asked .NET developers everywhere for their tips on how to make .NET perform smarter, faster, and better.

And this eBook is the result.

All that knowledge distilled into an easily digestible form so that rather than taking hours to find the answer you need, it will take just seconds.

Thanks to the .NET community for contributing so many valuable tips. Thanks too to Paul Glavich and Ben Emmett who edited them.

Now it's over to you.

Enjoy the tips – and if you slave for hours over a problem we've not covered and come up with a beautifully elegant solution, let us know. We'll save it for the next book.

Carly Meichen
dotnetteam@red-gate.com

redgate

Contents

GENERAL PEFORMANCE ADVICE

1 If you must measure small time differences in your code, ensure you use the StopWatch class

Giorgi Dalakishvili
@GioDalakishvili
www.aboutmycode.com

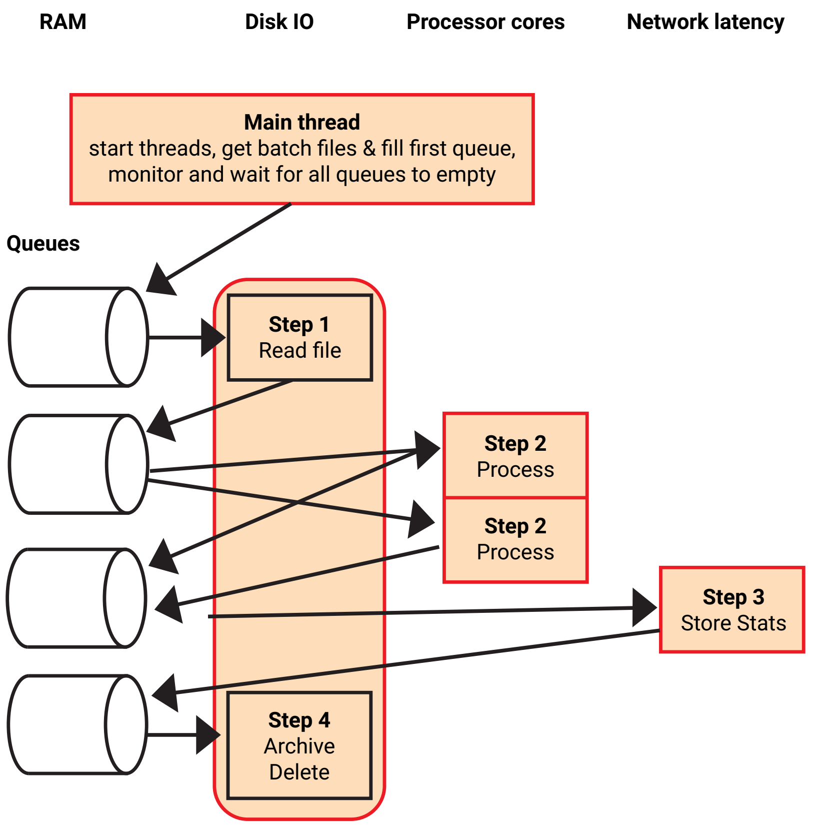
DateTime.UtcNow isn't designed for high-precision timing and will often have a resolution over 10ms, making it unsuitable for measuring small periods of time. The StopWatch class is designed for this purpose, although beware of ending up with an entire codebase with StopWatch instrumentation.

2 Fully utilize all bottlenecks

Louis Somers
@infotopie
www.infotopie.nl/blog

While Async-Await habits work well, an architectural pattern can switch into higher gears. When processing multiple small jobs, consider splitting job parts into resource-specific chunks. Then create separate threads for each resource, and place memory buffers in between. The goal is to fully utilize whatever resource is the bottleneck on any given machine.

In the diagram below, for example, an application has four steps. It first reads a file from disk, does some processing on it, stores results in a database, and finally deletes the file. Each red box is a separate thread, and each vertical 'swim-lane' represents a particular resource. The process flows from top to bottom and uses queues as buffers between each resource-switch.



3 Use defensive coding techniques such as performing null checks wherever applicable to avoid NullReferenceExceptions at runtime

Rob Karatzas
@ebiztutor

Exceptions can be slow and should only be used in exceptional circumstances, rather than for general control flow. Instead of assuming an object will not be null at runtime, utilize defensive coding to ensure your code only acts on objects that are not null. This will assist performance by throwing fewer exceptions, and ensure a more robust and reliable application.

For example, consider the Tester-Doer pattern explained here: https://msdn.microsoft.com/en-us/library/vstudio/ms229009(v=vs.100).aspx

4 Assume .NET problems are your fault, unless you have evidence to the contrary

Zijian Huang

It's tempting to blame system code, third party libraries, or even the .NET framework itself when there's a problem. But it's almost always the case that your application is misusing other people's code, so make sure you have really good evidence that a problem is elsewhere before blaming someone else.

5 Prepare performance optimization projects in advance

Rainer Stropek
@rstropek
www.software-architects.com

Good performance optimization projects need appropriate preparation in four key areas.

1. Create a test environment where you can reproduce consistent performance behavior. Ideally, you want an automated test that shows equal performance if you run it multiple times. Don't start optimizing without such a test environment.

2. Measure performance KPIs before you change anything to serve as a benchmark. Compare the performance against the benchmark after changing the code. If the changes make the code faster, the new test results are your new benchmark. If not, undo the changes and continue to analyze with a tool like ANTS Performance Profiler.

3. Don't change too much at the same time. Make small changes, measure, and decide if you want to keep your change. If you make multiple changes, one change can make your code faster, while the other one might destroy the positive effects.

4. Never guess, measure! Get a profiler like ANTS and use it to spot performance bottlenecks. In many cases, performance killers are where you don't expect them. Avoid optimizing code just because you have the feeling that it does not perform well.

6 Before optimizing code, profile the app so that you know where bottlenecks are

Giorgi Dalakishvili
@GioDalakishvili
www.aboutmycode.com

This is something that is said in many places but I want to repeat it once again because it's so important. Even using the limited profiling tools in Visual Studio to analyze performance issues in your application will help. With an expert profiling tool like ANTS Performance Profiler, you'll find out a lot more about the bottlenecks that are there – and be able to do a lot more, a lot faster.

7 Concurrent asynchronous I/O on server applications

Tugberk Ugurlu
Redgate
@tourismgeek www.tugberkugurlu.com

Concurrent I/O operations on server applications are tricky and hard to handle correctly. This is especially the case when applications are under high load. I would like to highlight two core principals to keep in mind:

Firstly, never perform concurrent I/O operations in a blocking fashion on your .NET server applications. Instead, perform then asynchronously, like the following code snippet:

```
[HttpGet]
public async Task<IEnumerable<Car>>
AllCarsInParallelNonBlockingAsync() {

  IEnumerable<Task<IEnumerable<Car>>> allTasks =
PayloadSources.Select(uri => GetCarsAsync(uri));
  IEnumerable<Car>[] allResults = await Task.
WhenAll(allTasks);

    return allResults.SelectMany(cars => cars);
}
```

When concurrent requests are made, the asynchronous method typically performs six times faster.

Secondly, remember to also perform load tests against your server applications based on your estimated consumption rates if you have any sort of multiple I/O operations.

For more information, see http://www.tugberkugurlu.com/archive/how-and-where-concurrent-asynchronous-io-with-asp-net-web-api

8 Consider String.Replace() for parsing, trimming, and replacing

Rob Karatzas
@ebiztutor

String.Replace() can be a good choice, particularly for individual case-sensitive replacements where it outperforms other algorithms. String Split/Join and String Split/Concat perform equally, while LINQ and RegEx have poor performance for parsing, trimming, and replacing.

For more information, see: http://blogs.msdn.com/b/debuggingtoolbox/archive/2008/04/02/comparing-regex-replace-string-replace-and-stringbuilder-replace-which-has-better-performance.aspx

9 Don’t use interop (WinFormsHost) with WPF when display performance is critical

Srinivasu Achalla
www.practicaldevblog.wordpress.com

When creating a Win32 window in particular, the display performance will fall by a few hundred milliseconds if there are several interop controls involved.

10 Eagerly create common object types to prevent constant evaluation

Paul Glavich
@glav
http://weblogs.asp.net/pglavich

Many of us use code frequently in applications. For example:

```
if (myobject.GetType() == typeof(bool)) { …. /* do
something */ }
  -- or --
if (myobject.GetType() == typeof()(MyCustomObject)) { ….
/* do something */ }
```

Instead, consider eagerly creating these type of instances:

```
public static class CommonTypes
{
public static readonly Type TypeOfBoolean = typeof(bool);
public static readonly Type TypeOfString =
typeof(string);
}
```

You can then do the following:

```
if (arg.GetType() == CommonTypes.TypeOfBoolean)
{
// Do something
}
```

Or:

```
if (arg.GetType() == CommonTypes.TypeOfString)
{
// Do something
}
```

And save the cost of performing the typeof(bool) or typeof(string) repeatedly in your code.

11 Make use of the Async await pattern

Gavin Lanata

Starting to introduce use of the Async await pattern on an established codebase can pose challenges which don't exist when using Async on a new project. If the code already makes use of background threads which do Thread.Sleep(), when you change to using Async methods, make sure you switch from Thread.Sleep() to await Task.Delay() instead. This achieves the same result of suspending that bit of work, but does it in a non-blocking way which can reduce the total resources needed by the application.

12 NGen your EntityFramework.dll to improve startup performance

Raghavendra Rentala
@vamosraghava

From Entity Framework version 6, core assemblies of Entity Framework are not part of the .NET framework and so a native image is not generated automatically. This despite the fact that a native image can improve memory when the assembly is shared among multiple processes.

Generating a native image of EntityFramework.dll and EntityFramework.SqlServer.dll will not only avoid multiple JIT compilations, but also improve the startup process significantly.

13 Remember that different ways of reading large files have different performance characteristics

Srinivasu Achalla
www.practicaldevblog.wordpress.com

Using File.ReadAllBytes() can sometimes be faster than reading a stream obtained by opening a large file such as XML. This can be influenced by factors like whether you need to access all of a file or just a small portion of it. If in doubt, use a tool like ANTS Performance Profiler to measure the impact of different methods of file access.

14 Remember that type casting and conversions incur performance penalties

Rob Karatzas
@ebiztutor

If you must perform type casting and conversions, try doing as little of them as possible. For example, rather than …

```
if (obj is SomeType) then { ((SomeType)obj) ……. };
```

… which causes two casts (one when performing the 'if' evaluation and one in the body of the If statement), use this:

```
var myObj = obj as SomeType;
if (myObj != null) { myObj …… };
```

That way, the cast is only done once with the 'as' parameter. If it fails, the value is NULL. If not, you can go ahead and use it, resulting in only one cast.

For more information, see: https://msdn.microsoft.com/en-us/library/ms173105.aspx

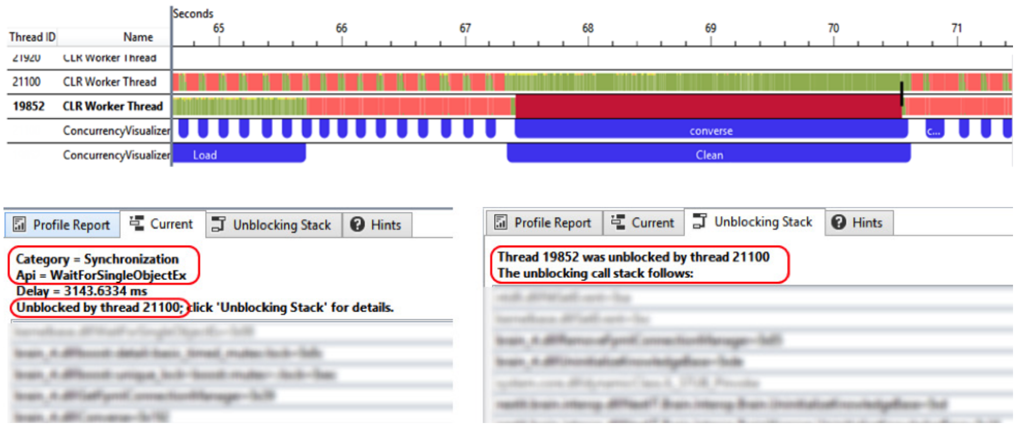
15 Spot potential issues in your code with Concurrency Visualizer

Michael Sorens
www.linkedin.com/in/michaelsorens

From Visual Studio 2013, Concurrency Visualizer became a plug-in rather than a standard feature. It is still a tremendously useful performance analysis tool, however, particularly when you use the SDK to instrument your code with flags, messages, alerts, and spans.

Concurrency Visualizer provides several views: utilization, threads, and cores. For me, the threads view is most useful. MSDN describes the threads view here, but that page does not do justice to the power of what you can see and find (however, they do offer another page within the docs that gives a bigger glimpse as to the power lurking beneath your fingertips).

The following Concurrency Visualizer timeline is a good example:



Category = Synchronization
Api = WaitForSingleObjectEx
Delay = 3143.6334 ms
Unblocked by thread 21100; click 'Unblocking Stack' for details.

Thread 19852 was unblocked by thread 21100
The unblocking call stack follows:

This is from a recent project where a multithreaded system has a large number of users making requests via a web interface and expecting prompt responses back. The system allows an administrator to hot swap pieces of the back end supposedly without interfering with the horde of eager users. But there was a delay of a few seconds now and then, and the source was not clear.

After instrumenting the likely sections of code for Concurrency Visualizer, starting a CV data collection, then running thousands of automated user inputs in parallel with dozens of automated admin inputs, a pattern quickly emerged in the graph.

There was a synchronization lock being held in the admin chunk of code (the “Clean” span) that blocked the end-user processing (the “converse” span) significantly. You can see in the illustration that most of the end-user time was waiting for that lock (the red bar).

I had only to visually scan the timeline to find my markers. Concurrency Visualizer revealed what was being blocked; when the lock was released on one thread and – via the vertical black line at the end of the red block – how that tied back to the other thread and allowed it to continue; and provided stack threads that told me exactly where in the code all this was happening.

16 Take advantage of spans

Michael Sorens
www.linkedin.com/in/michaelsorens

Spans – regions of code to explicitly demarcate on a timeline graph – are a great feature in Concurrency Visualizer. There are two big things to watch out for.

Firstly, when instrumenting spans, the documentation suggests you use:

```
var span = Markers.EnterSpan(description);
// do stuff here
span.Leave();
```

That pattern is more error prone than it needs to be – you must manually ensure and remember to “close” your span through all code pathways.

Secondly, you can enhance their disposable nature further with the using statement:

```
using (Markers.EnterSpan(description))
{
    // do stuff here
}
```

17

Perception is king

Ben Emmett
Redgate

If you can’t do everything quickly, sometimes it’s enough just to make a user think you’re quick.

From a user’s perspective, the most important time is often the delay until they can see something useful on a page, rather than every single detail being available. For example, if serving up a Tweet button adds 200ms to a page’s load time, consider using ajax to load the button asynchronously after the more important content is available.

This principle applies to desktop apps too. Ensuring that long-running tasks are done off the UI thread avoids the impression that an application has “locked up”, even if an action takes the same length of time to complete. Techniques like spinnies and progress indicators also have a role to play here, letting the user feel as though progress is being made.

18 Consider front-end performance issues as well as back-end performance issues

David Berry
@DavidCBerry13
www.buildingbettersoftware.blogspot.com

Many applications written today are web applications, so it is important to consider front-end performance issues as well as back-end performance issues. While the processing of a web page on the web server may only take a few hundred milliseconds, for example, it can take several seconds for the browser to load and render the page to the user.

Tools like Google PageSpeed Insights and Yahoo YSlow can analyze your pages for adherence to known front-end performance best practices. You not only get an overall score for your page, but a prioritized list of issues so you know exactly what items need to be addressed. Given how easy these tools are to use, every web developer should install them in their browser and analyze their pages on a regular basis.

19 Flush HTML responses early

Nik Molnar
@nikmd23
www.nikcodes.com/

By flushing HTML responses, the browser gets a jump start on downloading critical resources. It's also pretty easy to do in ASP. NET MVC with the CourtesyFlush NuGet package.

20 Consolidate your images with sprites

Jonathan Danylko
@jdanylko
www.danylkoweb.com/

If you have a standard collection of images with the same height and width (like social icons), use a sprite generator to consolidate the images into one file, and then use CSS to position the separate images on your website. Instead of multiple browser requests, you then make one request for the consolidated image, and the CSS positions the images for you.

21 Efficiently Streaming Large HTTP Responses With HttpClient

Tugberk Ugurlu
Redgate
@tourismgeek   www.tugberkugurlu.com

HttpClient in .NET has a default behaviour of buffering the response body when you make the call through GetAsync, PostAsync, PutAsync, etc. This is generally okay if you are dealing with small sized response bodies. However, if you wanted to download a large image and write it to a disk, you might end up consuming too much memory unnecessarily.

The following code, for example, will use up lots of memory if the response body is large:

```
static async Task HttpGetForLargeFileInWrongWay()
{
    using (HttpClient client = new HttpClient())
    {
        const string url = “https://github.com/
tugberkugurlu/ASPNETWebAPISamples/archive/master.zip”;
        using (HttpResponseMessage response = await
client.GetAsync(url))
        using (Stream streamToReadFrom = await response.
Content.ReadAsStreamAsync())
        {
            string fileToWriteTo = Path.GetTempFileName();
            using (Stream streamToWriteTo = File.
Open(fileToWriteTo, FileMode.Create))
            {
                await streamToReadFrom.
CopyToAsync(streamToWriteTo);
            }

            response.Content = null;
        }
    }
}
```

By calling GetAsync method directly, every single byte is loaded into memory.

A far better method is to only read the headers of the response and then get a handle for the network stream as below:

```
static async Task HttpGetForLargeFileInRightWay()
{
    using (HttpClient client = new HttpClient())
    {
        const string url = “https://github.com/
tugberkugurlu/ASPNETWebAPISamples/archive/master.zip”;
        using (HttpResponseMessage response =
await client.GetAsync(url, HttpCompletionOption.
ResponseHeadersRead))
        using (Stream streamToReadFrom = await response.
Content.ReadAsStreamAsync())
        {
            string fileToWriteTo = Path.GetTempFileName();
            using (Stream streamToWriteTo = File.
Open(fileToWriteTo, FileMode.Create))
            {
                await streamToReadFrom.
CopyToAsync(streamToWriteTo);
            }
        }
    }
}
```

The HttpCompletionOption.ResponseHeadersRead enumeration value reads the headers and returns the control back rather than buffering the response. The CopyTo Async method then streams the content rather than downloading it all to memory.

For more information, see http://www.tugberkugurlu.com/archive/efficiently-streaming-large-http-responses-with-httpclient

22 Move to ASP.NET MVC

Jonathan Danylko
@jdanylko
www.danylkoweb.com/

With the new release of Visual Studio 2015 and ASP.NET 5.0, there is no better time to move to ASP.NET MVC. With WebForm's ViewState taking up a hefty amount of space on the return trip to the browser (and possibly a mobile device), it could take a long time to render the content to the page. Particularly when web pages are now becoming larger and larger.

ASP.NET MVC makes your HTML more granular, returns a smaller HTML footprint to the browser, and provides a better separation of concerns for your development.

23 Use ASP.NET generic handlers instead of WebForms or MVC

Jeff Osia
www.linkedin.com/in/jeffosia

In ASP.NET applications, generic handlers, WebForms, and MVC all implement the IHttpHandler interface. The generic handler is normally the most lightweight option.

For more information, see: https://msdn.microsoft.com/en-us/library/bb398986.aspx

24 Cache your static content by directory

Jonathan Danylko
@jdanylko
www.danylkoweb.com/

If you have a directory of content like JavaScript, images, and CSS, place a web.config in that content directory to cache your static content.

For example, if you had a directory called 'Content' with an images, styles, and scripts directory underneath it, you would place a web.config (similar to below) in just the content directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
        <staticContent>
            <clientCache cacheControlMode="UseMaxAge"
cacheControlMaxAge="1.00:00:00" />
        </staticContent>
  </system.webServer>
</configuration>
```

You can include this web.config in any static directory to gain maximum caching web performance.

25 Keep an eye on your server-side code

Nik Molnar
@nikmd23
www.nikcodes.com/

Practice good performance hygiene by keeping an eye on your server-side code with a tool like Glimpse. Glimpse inspects web requests as they happen, providing insights and tooling that reduce debugging time and empower every developer to improve their web applications.

26 Send as little as possible to the web browser

Ben Emmett
Redgate

Web page load time is often limited by network latency, which can range in the hundreds of milliseconds, especially for mobile devices. The more files are transferred and the larger those files, the more round trips will be needed, so the greater the impact the latency will have. As a developer, there's rarely much you can do to reduce the latency, but you can cut down the number of times the round trip cost is incurred.

HTTP compression should always be turned on, and makes a particularly big impact for easily compressible content like html.

Minificafion and bundling of JavaScript & CSS files can be automatically handled by ASP.NET from v4.5 onwards. Just make sure you set BundleTable.EnableOptimizations = true;.

If you're including libraries like jQuery, serve it up from Google or Microsoft's free CDNs. Google's datacenters are probably better than yours, and it's very likely a user will already have a cached copy of the same library from having visited other websites using the same CDN, entirely eliminating their browser's need to re-download it. It also reduces your own server's load and bandwidth.

Clear out the cruft! A new ASP.NET project includes many libraries, not all of which are necessarily used, and if something's being unnecessarily sent to the client, it's incurring performance cost for no good reason. It can also be astonishing how many libraries get added to a project over time, but not removed when they stop being required. An occasional spring clean helps keep this in check.

DATABASE ACCESS

27 If your ORM is capable of handling multi-dataset returns, use this to reduce your database round-trips

Rob Karatzas
@ebiztutor

This is often called a ‘none-chatty’ design, where a single larger call is preferred over multiple smaller calls. Each call involves a degree of negotiation and overhead and the more calls you make, the more overhead is incurred. A single larger call reduces the overhead.

28 Minimize your database calls

Jonathan Danylko
@jdanylko
www.danylkoweb.com/

While Entity Framework can be a fantastic ORM tool, it can also be a major pain when you are executing a number of calls that are constantly hitting the database.

For simple database activities (SELECT, INSERT, UPDATE, or DELETE), use Entity Framework’s standard Add, SaveChanges, and Delete methods. For complex result sets, it can sometimes be more efficient to place them in a stored procedure and leave the heavy lifting to the database.

29 Confirm you are retrieving only the records you need

Jonathan Danylko
@jdanylko
www.danylkoweb.com/

If you are calling the database using Entity Framework, you may write this command:

```
return context.Products.ToList().FirstOrDefault();
```

While this essentially returns one record, it retrieves the entire Products table from the database and then returns you one record. A better command would be:

```
return context.Products.FirstOrDefault();
```

This will produce a SQL command equivalent to grabbing the TOP 1 record instead of the entire contents of the table.

30 Delayed execution in EF can trip you up

Nick Harrison
@neh123us
www.simple-talk.com/author/nick-harrison

If your Model exposes an IQueryable object, your View may actually run a query multiple times when it tries to read the object's data.

List properties in the Model should be of type IList to force the query to be executed once and only once.

It's easy to spot these additional queries with a tool like ANTS Performance Profiler, which shows you what queries are being run by your application.

31 Listen to generated SQL

Raghavendra Rentala
@vamosraghava

Entity Framework greatly simplifies database access for the developer, but it can also introduce problems when you are writing more complex LINQ to SQL queries.

This may generate poorly-performing SQL, so troubleshooting the generated SQL with tools like SQL Server Profiler or ANTS Performance Profiler is very important.

32 Avoid issues with string queries

Raghavendra Rentala
@vamosraghava

Instructing Entity Framework to use the right kind of string when generating queries can resolve datatype conversion issues. A simple solution is to introduce Column Annotation:

```
public class MyTable
{
     [Column(TypeName=”varchar”)]
     public string Property1 { get; set; }
}
```

This is particularly valid for string datatypes where .NET Strings are Unicode by default and are not the same in SQL Server.

For more information, see: http://www.simple-talk.com/dotnet/.net-tools/catching-performance-issues-in-development/

33 Don’t overlook ‘WHERE IN’ style LINQ to SQL Queries

Raghavendra Rentala
@vamosraghava

Entity Framework is smart enough to convert the Contains() operator on LINQ queries to WHERE IN (….) in SQL . But there is a hidden problem: Giving a data set length of greater than around 10,000 records in a WHERE IN (…) clause will significantly degrade the performance of the query generation and query execution:

```
var ids = new int[]{0,1, 2,3,4,5,6,7,8,9,10........99995,
99996,99997,99998,99999};

var matches = (from person in people
               where ids.Contains(person.Id)
            select person).ToArray();
```

The above statement generates the following fat SQL query:

```
SELECT * FROM PERSON WHERE ID IN
(0,1,2,3,4,5,6,7,8,9,10.....,99995,99996,99997,99998,999
99)
```

It is advisable therefore to send data in batches. 500 records per batch, for example, would yield a significant improvement in performance, but you should do benchmarking to see what works for you.

34 Beware hidden cursors

Raghavendra Rentala
@vamosraghava

If you're using an ORM like Entity Framework, when you declare an object mapped to another table with foreign keys, you automatically get references of those related entities. Unfortunately, there is a significant hidden cost when accessing the related entities, as separate queries can be run to retrieve details of each referenced row. This is commonly called the n+1 select problem.

For example, consider the following code where we fetch a list of schools from a database then filter that list. On line 1, a query is run to retrieve a list of n schools. On line 2, for every item in the schools list, a query is run to retrieve the number of pupils at that school, so in total n+1 queries are run.

```
List<School> schools = context.Schools.ToList();
List<School> filteredSchools = schools.Where(s =>
s.Pupils.Count > 1000).ToList();
```

Consider using Eager Loading to avoid this scenario if you need to access properties of Pupils later:

```
List<School> schools = context.Schools.Include(s =>
s.Pupils).ToList();
```

Or in this scenario, simply replace line 1 with:

```
List<School> schools = context.Schools.Where(s =>
s.Pupils.Count > 1000).ToList();
```

35 Use LINQ's 'let' keyword to tune emitted SQL

Raghavendra Rentala
@vamosraghava

Projecting a row into an object with a nested object has a big impact on the generated SQL. For example, here is an original query:

```
from s in Scholars
where s.ID == 2764
select new
{
    s.School.Address1,
    s.School.Address2,
    s.School.City,
    s.School.State,
    s.School.ZIP,
    s.School.PhoneNo,
    s.School.Email,
    Principal_FirstName = s.School.Leader.FirstName,
    Principal_LastName = s.School.Leader.LastName,
    Principal_Email = s.School.Leader.Email
}
```

This generates the following SQL:

```
SELECT
1 AS [C1],
[Extent2].[Address1] AS [Address1],
[Extent2].[Address2] AS [Address2],
[Extent2].[City] AS [City],
[Extent2].[State] AS [State],
[Extent2].[ZIP] AS [ZIP],
[Extent2].[PhoneNo] AS [PhoneNo],
[Extent2].[Email] AS [Email],
[Join2].[FirstName] AS [FirstName],
[Join4].[LastName] AS [LastName],
[Join6].[Email] AS [Email1]
FROM     [dbo].[Scholar] AS [Extent1]
LEFT OUTER JOIN [dbo].[School] AS [Extent2] ON [Extent1].
[SchoolID] = [Extent2].[ID]
LEFT OUTER JOIN (SELECT [Extent3].[ID] AS [ID1],
[Extent4].[FirstName] AS [FirstName]
    FROM  [dbo].[Staff] AS [Extent3]
    INNER JOIN [dbo].[Person] AS [Extent4] ON [Extent3].
[ID] = [Extent4].[ID] ) AS [Join2] ON [Extent2].
[LeaderStaffID] = [Join2].[ID1]
LEFT OUTER JOIN (SELECT [Extent5].[ID] AS [ID2],
[Extent6].[LastName] AS [LastName]
    FROM  [dbo].[Staff] AS [Extent5]
    INNER JOIN [dbo].[Person] AS [Extent6] ON [Extent5].
[ID] = [Extent6].[ID] ) AS [Join4] ON [Extent2].
[LeaderStaffID] = [Join4].[ID2]
LEFT OUTER JOIN  (SELECT [Extent7].[ID] AS [ID3],
[Extent8].[Email] AS [Email]
    FROM  [dbo].[Staff] AS [Extent7]
    INNER JOIN [dbo].[Person] AS [Extent8] ON [Extent7].
[ID] = [Extent8].[ID] ) AS [Join6] ON [Extent2].
[LeaderStaffID] = [Join6].[ID3]
WHERE 2764 = [Extent1].[ID]
```

Using the ‘let’ keyword allows us to define the navigation as an alias:

```
from s in Scholars
where s.ID == 2764
let leader = s.School.Leader
select new
{
    s.School.Address1,
    s.School.Address2,
    s.School.City,
    s.School.State,
    s.School.ZIP,
    s.School.PhoneNo,
    s.School.Email,
    Principal = new {
        leader.FirstName,
        leader.LastName,
        leader.Email
    }
}
```

This results in a much smaller query:

```
SELECT
1 AS [C1],
[Extent2].[Address1] AS [Address1],
[Extent2].[Address2] AS [Address2],
[Extent2].[City] AS [City],
[Extent2].[State] AS [State],
[Extent2].[ZIP] AS [ZIP],
[Extent2].[PhoneNo] AS [PhoneNo],
[Extent2].[Email] AS [Email],
[Join2].[FirstName] AS [FirstName],
[Join2].[LastName] AS [LastName],
[Join2].[Email] AS [Email1]
FROM   [dbo].[Scholar] AS [Extent1]
LEFT OUTER JOIN [dbo].[School] AS [Extent2] ON [Extent1].
[SchoolID] = [Extent2].[ID]
LEFT OUTER JOIN  (SELECT [Extent3].[ID] AS [ID1],
[Extent4].[FirstName] AS [FirstName], [Extent4].
[LastName] AS [LastName], [Extent4].[Email] AS [Email]
    FROM  [dbo].[Staff] AS [Extent3]
    INNER JOIN [dbo].[Person] AS [Extent4] ON [Extent3].
[ID] = [Extent4].[ID] ) AS [Join2] ON [Extent2].
[LeaderStaffID] = [Join2].[ID1]
WHERE 2764 = [Extent1].[ID]
```

Looking at the query execution plan in SSMS, the first query is roughly twice as expensive as the second, so not only is the query cleaner, but it performs and scales better as well.

A tool like ANTS Performance Profiler can be used here to discover the performance improvement because it lets you see the generated SQL from a LINQ to SQL/EF query.

36 Use the SQLBulkCopy class to load data into SQL Server from .NET

Richie Rump
@Jorriss
www.jorriss.net
www.awayfromthekeyboard.com

Using SQLBulkCopy can dramatically decrease the time it takes to load data into SQL Server. A test using SQL Server 2012 on a local machine loading a 100,000 row file had the following results:

- Using a stored procedure: 37 seconds
- Using concatenated inline SQL: 45 seconds
- Using Entity Framework: 45 minutes
- Using the SQLBulkCopy class: **4.5 seconds**

Let's say you need to load a webserver log into SQL Server. You would still need to load a file, read the file, parse the file, and load the data into objects. Then you would create a DataTable (you could also use the DataReader or an array of DataRow too):

```
DataTable table = new DataTable();
table.TableName = “LogBulkLoad”;

table.Columns.Add(“IpAddress”, typeof(string));
table.Columns.Add(“Identd”, typeof(string));
table.Columns.Add(“RemoteUser”, typeof(string));
table.Columns.Add(“LogDateTime”, typeof(System.
DateTimeOffset));
table.Columns.Add(“Method”, typeof(string));
table.Columns.Add(“Resource”, typeof(string));
table.Columns.Add(“Protocol”, typeof(string));
table.Columns.Add(“QueryString”, typeof(string));
table.Columns.Add(“StatusCode”, typeof(int));
table.Columns.Add(“Size”, typeof(long));
table.Columns.Add(“Referer”, typeof(string));
table.Columns.Add(“UserAgent”, typeof(string));
```

Next step would be to load the DataTable with data that you’ve parsed:

```
foreach (var log in logData)
{
      DataRow row = table.NewRow();

      row[“IpAddress”] = log.IpAddress;
      row[“Identd”] = log.Identd;
      row[“RemoteUser”] = log.RemoteUser;
      row[“LogDateTime”] = log.LogDateTime;
      row[“Method”] = log.Method;
      row[“Resource”] = log.Resource;
      row[“Protocol”] = log.Protocol;
      row[“QueryString”] = log.QueryString;
      row[“StatusCode”] = log.StatusCode;
      row[“Size”] = log.Size;
      row[“Referer”] = log.Referer;
      row[“UserAgent”] = log.UserAgent;

      table.Rows.Add(row);
}
```

Now you’re ready to use the SqlBulkCopy class. You will need an open SqlConnection object (this example pulls the connection string from the config file). Once you’ve created the SqlBulkCopy object you need to do two things: set the destination table name (the name of the table you will be loading); and call the WriteToServer function passing the DataTable.

This example also provides the column mappings from the DataTable to the table in SQL Server. If your DataTable columns and SQL server columns are in the same positions, then there will be no need to provide the mapping, but in this case the SQL Server table has an ID column and the DataTable does not need to explicitly map them:

```
using (SqlConnection conn = new SqlConnection(Configu
rationManager.ConnectionStrings["LogParserContext"].
ConnectionString))
{
      conn.Open();
      using (SqlBulkCopy s = new SqlBulkCopy(conn))
      {
      s.DestinationTableName = "LogBulkLoad";

      s.ColumnMappings.Add("IpAddress", "IpAddress");
      s.ColumnMappings.Add("Identd", "Identd");
      s.ColumnMappings.Add("RemoteUser", "RemoteUser");
      s.ColumnMappings.Add("LogDateTime", "LogDateTime");
      s.ColumnMappings.Add("Method", "Method");
      s.ColumnMappings.Add("Resource", "Resource");
      s.ColumnMappings.Add("Protocol", "Protocol");
      s.ColumnMappings.Add("QueryString", "QueryString");
      s.ColumnMappings.Add("StatusCode", "StatusCode");
      s.ColumnMappings.Add("Size", "Size");
      s.ColumnMappings.Add("Referer", "Referer");
      s.ColumnMappings.Add("UserAgent", "UserAgent");

      s.WriteToServer((DataTable)table);
            }
}
```

There are other features of the SqlBulkCopy class that are useful. The BatchSize property can control the number of rows in each batch sent to the server, and the NotifyAfter property allows an event to be fired after a specified number of rows, which is useful for updating the user on the progress of the load.

37 Use AsNoTracking when retrieving data for reading with Entity Framework

Andre Kraemer
@codemurai
www.andrekraemer.de/blog

In a lot of cases, data retrieved by the Entity Framework will not be modified within the scope of the same DBContext. Typical examples of this are ASP.NET MVC or ASP.NET MVC API action methods that just answer a get request. However, if you're using the default way of retrieving data, Entity Framework will prepare everything to be able to detect changes on your retrieved entities in order to persist those changes later in the database. This doesn't only add a performance penalty, but costs some memory, too.

A typical method of retrieving data is:

```
var products = db.Products.Where(p => p.InStock).ToList();
```

A better way uses the extension method AsNoTracking from the System.Data.Entity Namespace:

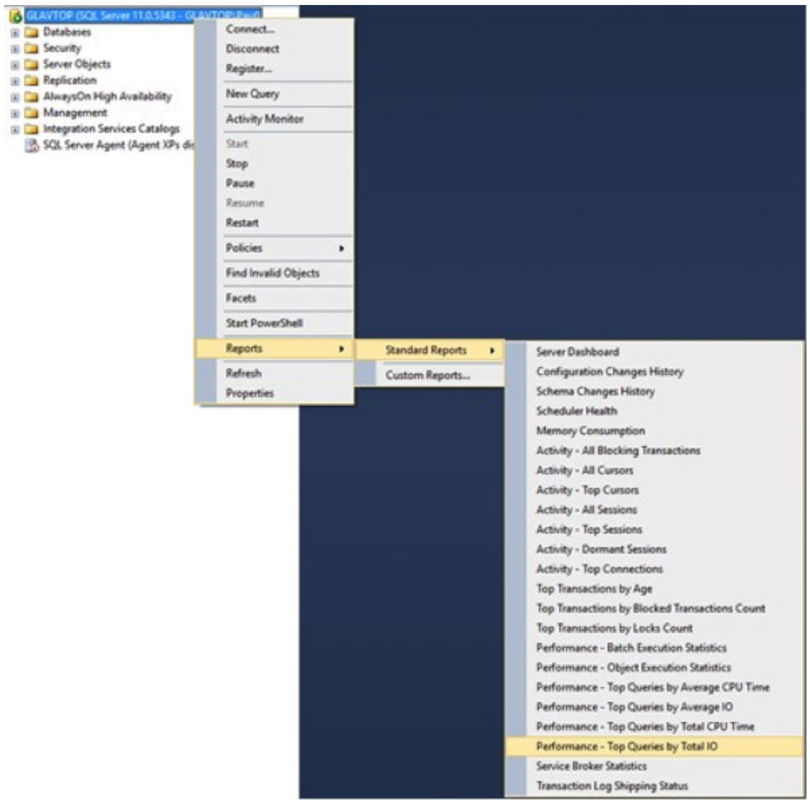
```
var products = db.Products.Where(p => p.InStock).
AsNoTracking().ToList();
```

38 Indexing tables is not an exact science

Rob Karatzas
@ebiztutor

Indexing tables requires some trial and error combined with lots of testing to get things right. Even then, performance metrics change over time as more and more data is added or becomes aged.

When you're using SQL Server, it's a good idea to regularly run and analyze the standard reports SQL Server provides that show index usage (such as top queries by total IO, top queries by average IO, etc). This can highlight unused indexes, and can also show queries that are using excessive IO which may require further indexing.


GLAVTOP (SQL Server 11.0.5343 - GLAVTOP\Paul)
Databases
Security
Server Objects
Replication
AlwaysOn High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs di
Connect...
Disconnect
Register...
New Query
Activity Monitor
Start
Stop
Pause
Resume
Restart
Policies
Find Invalid Objects
Facets
Start PowerShell
Reports
Refresh
Properties
Standard Reports
Custom Reports...
Server Dashboard
Configuration Changes History
Schema Changes History
Scheduler Health
Memory Consumption
Activity - All Blocking Transactions
Activity - All Cursors
Activity - Top Cursors
Activity - All Sessions
Activity - Top Sessions
Activity - Dormant Sessions
Activity - Top Connections
Top Transactions by Age
Top Transactions by Blocked Transactions Count
Top Transactions by Locks Count
Performance - Batch Execution Statistics
Performance - Object Execution Statistics
Performance - Top Queries by Average CPU Time
Performance - Top Queries by Average IO
Performance - Top Queries by Total CPU Time
Performance - Top Queries by Total IO
Service Broker Statistics
Transaction Log Shipping Status

For a deeper analysis of queries, you can also use a tool like ANTS Performance Profiler.

39 Use caching to reduce load on the database

Paul Glavich
@glav
http://weblogs.asp.net/pglavich

Accessing the database is often the slowest aspect of an application due to the physical nature of accessing the data from disk (database query caching not withstanding). Developing your application with an efficient caching mechanism in mind can relieve the need for your database to perform requests and let it devote its time where required.

Simple things like caching reference data or data that changes very infrequently can make easy gains and reduce load on your database. As you cache more and more, it’s important to ensure you invalidate cached data when it is updated using a common key, and this needs to be factored into the design.

For a headstart on caching and supporting multiple cache engines easily, try this library: https://bitbucket.org/glav/cacheadapter. This allows you to support ASP.NET web cache, memcached, Redis and the now defunct AppFabric on the same codebase via configuration only.

40 If you don't need all the columns from your table, don't select them

Andre Kraemer
@codemurai
www.andrekraemer.de/blog

Entity Framework is a great productivity booster for developers. Instead of writing tedious SQL statements, you just write code like this:

```
var products = db.Products.ToList();
```

A line like this is great if you only have a few columns in your products table or if you don't care much about performance or memory consumption. With this code, Entity Framework selects all the columns of the products table, but if your product table has 25 columns and you only need two of them, your database, network, and PC all run slower.

Do your users a favor and retrieve only the columns you need. If, for example, you just need the 'Id' and 'Name' fields, you could rewrite your statement like this:

```
var db.Products.Select(p => new {p.Id, p.Name}).ToList();
```

MEMORY USAGE

41 Use lists instead of arrays when the size is not known in advance

Megha Maheshwari
www.linkedin.com/in/formegha
www.quora.com/megha-maheshwari

When you want to add or remove data, use lists instead of arrays. Lists grow dynamically and don't need to reserve more space than is needed, whereas resizing arrays is expensive. This is particularly useful when you know what the pattern of growth is going to be, and what your future pattern of access will be.

42 Don't call GC.Collect() explicitly

Rakesh Nhalil Edavalath
@rakeshne
https://in.linkedin.com/pub/rakesh-nhalil-edavalath/85/402/588

The Garbage Collector is very good at working out appropriate times to run, influenced by factors like memory usage in the application and OS memory pressure. It's almost never necessary to call it explicitly.

Worse, running a Garbage Collection has an impact on application performance. The performance hit is proportional to the number of objects in memory which survive Garbage Collection, so running the Garbage Collector earlier or more frequently than necessary can seriously harm performance.

43 Learn about the .NET Garbage Collector (GC) and when it can cause pauses that slow your application down

Matt Warren
@matthewwarren
www.mattwarren.org

Over time the .NET GC has become more advanced (most notably the Background Server GC Mode in .NET 4.5), but there are still situations where the GC can have an adverse effect on your application's performance.

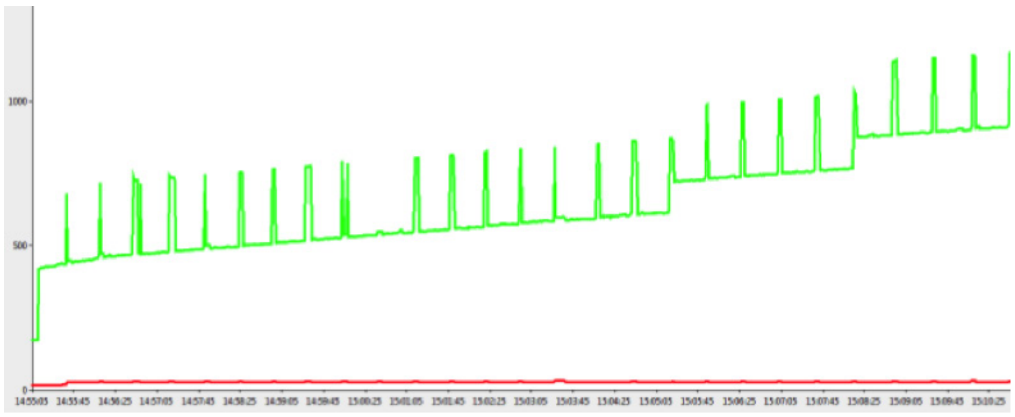
Understanding how to detect these situations and more importantly how to fix them is a useful skill. For example, in many applications there are some actions which are more performance-critical than others, and it would be preferable for Garbage Collection to run during the less critical periods. Setting a GCLatencyMode is a useful way of asking the Garbage Collector to be more conservative about choosing to run during these times.

44 Fix memory leaks to avoid performance issues

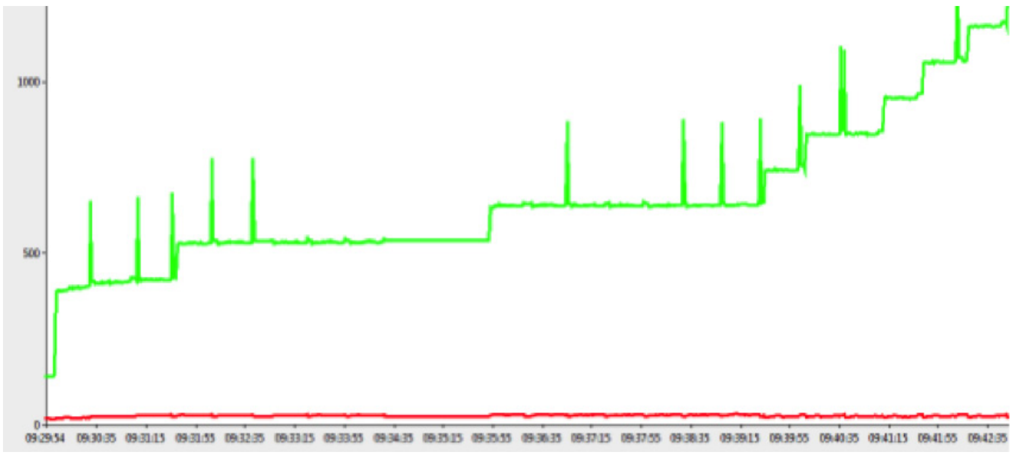
Cosimo Nobile
@cosimo_nobile

Memory leaks can have an impact on performance as well. When I suspect memory leaks, I often start my investigation by plotting managed vs unmanaged memory usage against the target process in Performance Monitor. I find this can often give some preliminary and qualitative indications about the number and nature of the memory leaks and the way they change with time or in response to external input.

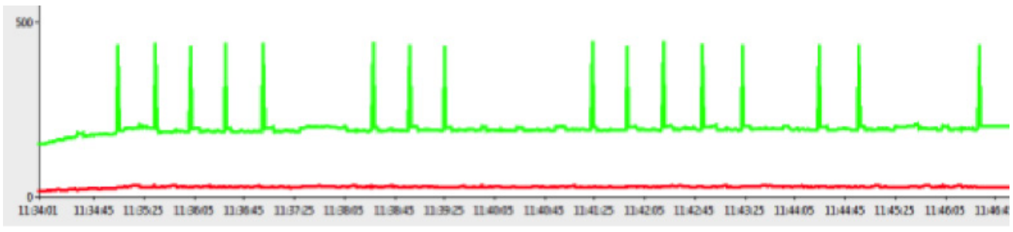
As an example, on one occasion I started with:



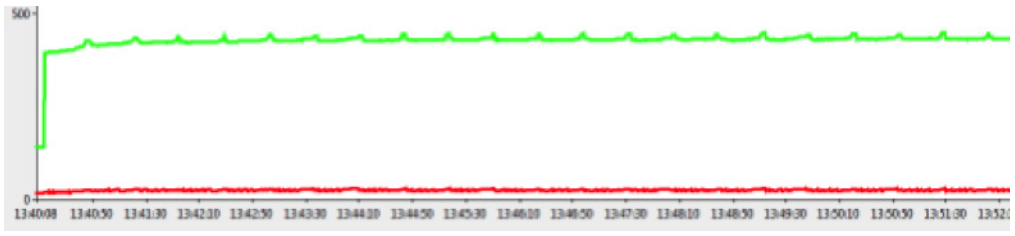
This confirmed the process was leaking unmanaged memory. It also gave me some initial indications that there could potentially be several causes because the graph shows some random steps in addition to the constant gradient slope. Fixing the cause of the constant leak resulted in the following:



The slope was clearly gone but the occasional random steps were still present. This was harder to find but it soon became clear what component was responsible for it because after disabling it I obtained a flat tracing with occasional spikes:



Finally, having located and fixed the leak in the identified component, the following graph provided me with the reassurance that all issues had been found and addressed:



45 Preallocate sizes on things if you can

Greg Young
@gregyoung

Many objects like memory stream, list, and dictionary will double their size as needed causing wasteful copying and allocations. If you know your list will have 100,000 items, initialize it as such to avoid problems in the future.

46 Avoid duplicate string reference issues Shubhajyoti Ghosh

Shubhajyoti Ghosh
@radcorpindia

String referencing duplication is one of the major memory hogging performance issues. String interning is a useful solution if you're generating a lot of strings at runtime that are likely to be the same. It calls IsInterned to see if an interned string exists as follows:

```
class Program
{
    static void Main()
    {
      // A.
      // String generated at runtime.
      // Is not unique in string pool
      string s1 = new StringBuilder().Append("cat").
Append(" and dog").ToString();

      // B.
      // Interned string added at runtime.
      // Is unique in string pool.
      string s2 = string.Intern(s1);
    }
}
```

My own benchmarking showed that string interning can improve performance by more than four times when the string comparison is always true.

47

Dispose of unmanaged resources

Rob Karatzas
@ebiztutor

File I/O, network resources, database connections, etc, should be disposed of once their usage is complete, rather than waiting for the Garbage Collector to handle them. This can take anything from milliseconds to minutes, depending on the rate you consume memory in your code. If you don't use a lot of, it will take a long time.

These types of resources typically implement the IDisposable interface so that unmanaged resources can be released once use of the object is complete.

Ideally, use a 'using' {..} block to automatically dispose the object once out of scope, or ensure you manually call Dispose().

For more information, see: https://msdn.microsoft.com/en-us/library/498928w2.aspx

48 Optimize allocations

Bartosz Adamczewski
@badamczewski01

Avoiding hidden allocations is a good way to improve performance but sometimes allocations have to be made. There are two major ways to optimize them:

a) Calculate on the stack and publish to the heap

Consider the following code:

```
public class Counter
 {
  public int Cnt;
 }
 ...
 Thread[] th = new Thread[8];
 for (int i = 0; i < th.Length; i++) {
   int n = i;
   th[i] = new Thread(new ThreadStart(() => {
         int cnt = 0;
         for (int k = 0; k < 100000000; k++)
         {
              cnt = cnt + 1;
         }
         p.arrCounter[n] = new Counter();
         p.arrCounter[n].Cnt = cnt;
      }));}
```

We first increment the counter on the stack, after which we publish the results to the heap.

b) Off heap allocate

Consider the following code:

```
public static unsafe byte* global;
...
unsafe { global = Unsafe.Allocate(65536); } //this uses
VirtualAllocEX
...
 Thread[] th = new Thread[8];
 for (int i = 0; i < th.Length; i++) {
    int n = i;
    th[i] = new Thread(new ThreadStart(() => {
       unsafe {
         int* ptr = (int*)(global);
         ptr = ptr + (n * 16);

         for (int k = 0; k < 100000000; k++) {
            *ptr = *ptr + 1;
         }}}));}
```

We allocate unmanaged memory pool and increment the counter and save it to that space.

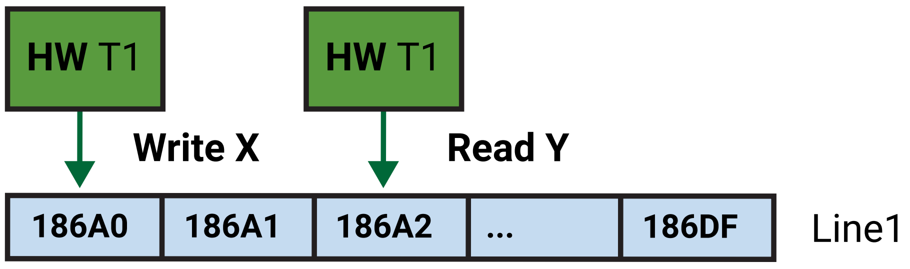
49

Avoid false sharing

Bartosz Adamczewski
@badamczewski01

False sharing is a performance degrading effect that happens when two or more hardware threads (meaning two cores or processors) use the same cache line in the following way:

- At least one writes to the cache line
- They modify different cells (otherwise it's sharing)



Consider this simple program:

```
public static int[] arr = new int[1024];
 ...
 Thread[] th = new Thread[4];

 for (int i = 0; i < th.Length; i++)
 {
   int n = i

    th[i] = new Thread(new ThreadStart(() => {
       for (int k = 0; k < 100000000; k++) {
          arr[n] = arr[n] + 1;
       }
    }));
  }
```

We create four threads and assign each thread to a specific array index and then increment the counter. This program has false sharing because every thread will use the same cache line. In order to eliminate false sharing, we need to assign each thread its own cache line by offsetting the array index given to each:

```
public static int[] arr = new int[1024];
  ...
  Thread[] th = new Thread[4];

  for (int i = 0; i < th.Length; i++)
  {
    int n = i;

    th[i] = new Thread(new ThreadStart(() => {
       for (int k = 0; k < 100000000; k++) {
         arr[n * 16] = arr[n * 16] + 1;
       }
     }));
    }
```

The performance gain of such a simple change is usually two or three times faster, depending on core and processor numbers.

50

Avoid hidden allocations

Bartosz Adamczewski
@badamczewski01

Hidden allocations are a problem because they often allocate a bunch of small stuff on the heap, which introduces memory pressure resulting in garbage collection. There are two major categories of hidden allocations.

a) Hidden boxing

Consider the following code:

```
public static void Sum(int a, int b)
{
 Console.WriteLine(“Sum of a {0} b {1} is”, a, b, a + b);
}
```

This code will compile to `void [mscorlib]System.Console::WriteLine(string, object, object, object)` which means that it will be boxed to objects and allocated on the heap.

A simple fix is to call to string on the ints. This will do string allocation but is still better than boxing.

```
public static void Sum(int a, int b)
{
  Console.WriteLine(“Sum of a {0} b {1} is”,
a.ToString(), b.ToString(), (a + b).ToString());
}
```

b) Lambdas

Lambdas are problematic because they create lots of hidden allocations and can also box values.

Consider the following code:

```
 public static Client FindClientByName(string name)
  {
    return clients.Where(x => x.Name == name).
FirstOrDefault();
  }
```

This simple lambda creates tons of allocations. First of all, it creates a new delegate in the Where clause, but that delegate uses the 'name' variable which is out of scope of the delegate so a helper (capture) class needs to be created to capture that variable.

There is also hidden boxing here because we're calling FirstOrDefault which operates on IEnumerable<TSource>, so we lose type information. In order to get the first element, we need to get its enumerator which is a struct, and thus we will box.

The code for FirstOrDefault looks as follows:

```
public static TSource FirstOrDefault<TSource>(IEnumerable
<TSource> source)
 {
  if (source == null)
  {
    throw Error.ArgumentNull(“source”);
   }
   IList<TSource> list = source as IList<TSource>;
   if (list != null)
            {
                if (list.Count > 0)
                {
                    return list[0];
                }
            }
            else
            {
                using (IEnumerator<TSource> enumerator =
source.GetEnumerator()) //We will box here!
                {
                    if (enumerator.MoveNext())
                    {
                        return enumerator.Current;
                    }
                }
            }
            return default(TSource);
                     }
```

The fix is simple:

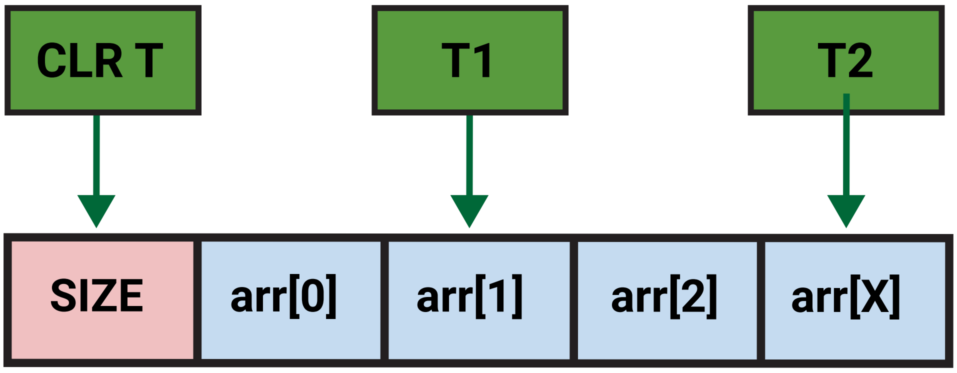
```
public static Client FindClientByName(string name)
        {
            foreach(Client c in clients)
            {
                if (c.Name == name)
                    return c;
            }

            return null;
         }
```

51 Avoid hidden false sharing

Bartosz Adamczewski
@badamczewski01

In .NET there are lots of hidden components doing things that could potentially lead to false sharing such as GC Compaction, Card Tables, Card Bundles, Brick Tables, and Intern Tables.

The previous code example that fixed false sharing still contains false sharing and it's related to .NET bounded check implementation. In order to check if we're not out of array bounds, the thread that writes (or reads) will need to check its metadata for size, the information for which is stored at the start of the array.



In our program, each thread actually uses not one but two cache lines and the first cache line is shared between all other threads. Since they only need to read from it, everything should be alright.

The sample code shows, however, that the first thread uses this line. There will therefore be one thread that will be writing while others are reading. In other words, false sharing.

To fix this, we need to guarantee that the cache line where metadata of the array is stored has its own cache line and there are no writes to it. We can satisfy this condition if we offset the first thread:

```
for (int i = 0; i < th.Length; i++)
  {
    int n = i + 1 // WE ADD ONE;

    th[i] = new Thread(new ThreadStart(() => {
       for (int k = 0; k < 100000000; k++) {
          arr[n * 16] = arr[n * 16] + 1;
       }
     }));
    }
```

The increase in performance is dependent on the number of threads but on average it will be 10-15%.

52 Optimize your code for cache prefetching

Bartosz Adamczewski
@badamczewski01

Cache prefetching can dramatically improve performance by taking advantage of the way modern processors read data. Consider this simple program:

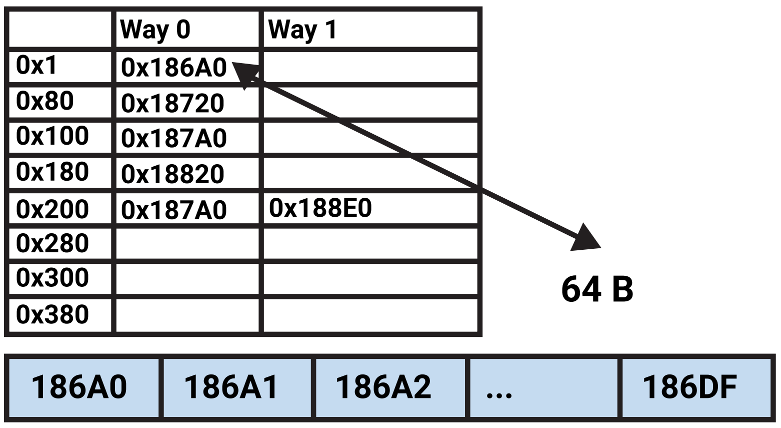
```
int n = 10000;
int m = 10000;
int[,] tab = new int[n, m];

for (int i = 0; i < n; i++)
{
for (int j = 0; j < m; j++)
{
tab[j, i] = 1; //j instead of i
}
}
```

This program will execute (depending on the machine) in around 1500 ms. Now let's swap the indexes of the array such that we will start from 'I' and move to 'j'.

```
int n = 10000;
int m = 10000;
int[,] tab = new int[n, m];

for (int i = 0; i < n; i++)
{
for (int j = 0; j < m; j++)
{
tab[i, j] = 1; //i instead of j
}
}
```

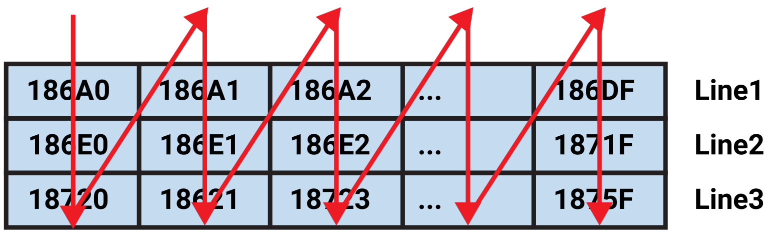
Now this program will execute in around 700 ms, so on average it will run at least two times faster. In terms of pure memory access, array indexing should not make a change since virtual memory addressing makes the memory seam linear and its access times should be more or less the same.

It happens because modern processors read data in portions called lines (usually 64 bytes). Processors also detect memory access patterns so if our memory access is predictable then all of the data needed to execute our loop will be loaded ahead of time (cache prefetching).



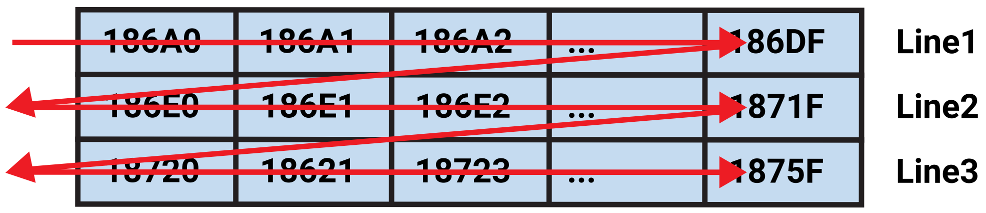
	Way 0	Way 1
0x1	0x186A0	
0x80	0x18720	
0x100	0x187A0	
0x180	0x18820	
0x200	0x187A0	0x188E0
0x280		
0x300		
0x380		

So the first example will load an entire line of continuous data, write to the first four bytes (the size of an int) then throw it into oblivion and load another one write and throw it away and so on. This is very inefficient and usually prevents cache prefetching since processors only prefetch data on linear data access patterns.



186A0	186A1	186A2	...	186DF	Line1
186E0	186E1	186E2	...	1871F	Line2
18720	18621	18723	...	1875F	Line3

The second example loads a cache line, modifies all data cells in that line and then loads the next one and so on, which is much faster.



The given example was doing writes only, but if it read from an already populated array the performance impact would be much greater because writes are always volatile and some cycles are wasted writing to main memory.

The takeaway is that you should access your data structures and arrays in a sequential way rather than jumping through the memory.

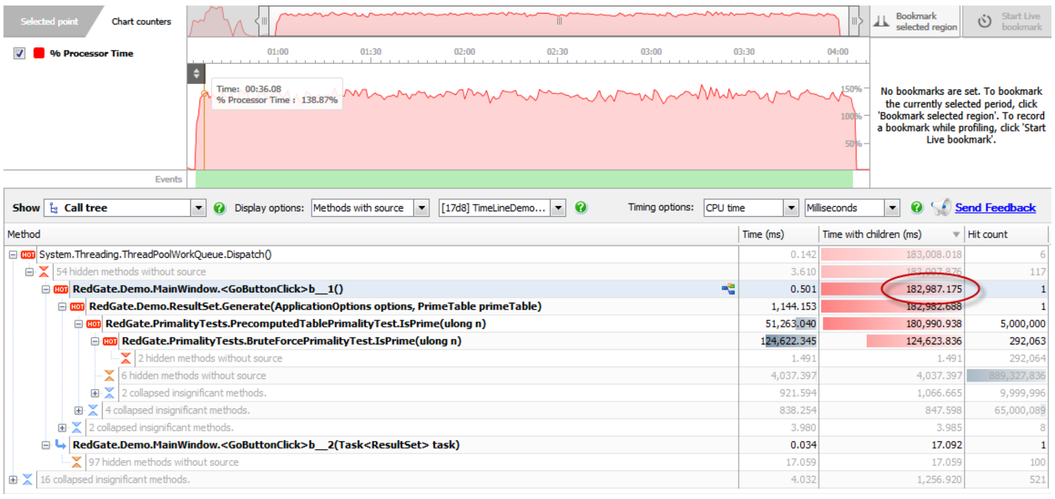
That's not the end of the story

We hope you've founds lots of tips and tricks to boost your .NET performance in this eBook. If you'd like more, or would like a detailed guide on .NET performance or memory management, take a look at the other eBooks we've published:

You don’t just need the right knowledge – you need the right tools

Redgate has developed two of the most popular .NET performance and memory profilers that help .NET professionals save time and fix the things that matter.

ANTS PERFORMANCE PROFILER



No bookmarks are set. To bookmark the currently selected period, click 'Bookmark selected region'. To record a bookmark while profiling, click 'Start Live bookmark'.

Show: Call tree — Display options: Methods with source — [17d8] TimeLineDemo... — Timing options: CPU time — Milliseconds — Send Feedback

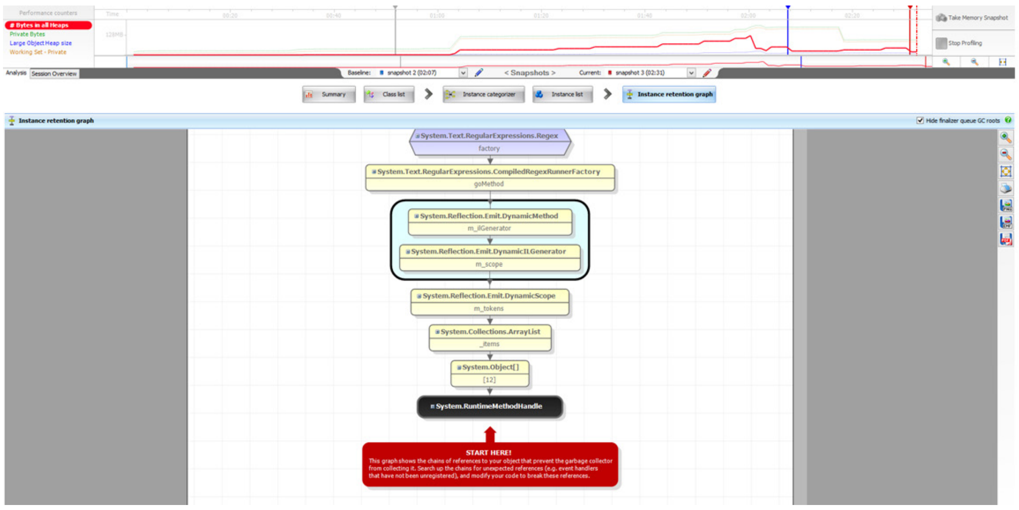
Method	Time (ms)	Time with children (ms)	Hit count
System.Threading.ThreadPoolWorkQueue.Dispatch()	0.142	183,008.018	6
54 hidden methods without source	3.610	183,007.876	117
RedGate.Demo.MainWindow.<GoButtonClick>b__1()	0.501	182,987.175	1
RedGate.Demo.ResultSet.Generate(ApplicationOptions options, PrimeTable primeTable)	1,144.153	182,982.688	1
RedGate.PrimalityTests.PrecomputedTablePrimalityTest.IsPrime(ulong n)	51,263.040	180,990.938	5,000,000
RedGate.PrimalityTests.BruteForcePrimalityTest.IsPrime(ulong n)	124,622.345	124,623.836	292,063
2 hidden methods without source	1.491	1.491	292,064
6 hidden methods without source	4,037.397	4,037.397	889,327,836
2 collapsed insignificant methods.	921.594	1,066.665	9,999,996
4 collapsed insignificant methods.	838.254	847.598	65,000,089
2 collapsed insignificant methods.	3.980	3.985	8
RedGate.Demo.MainWindow.<GoButtonClick>b__2(Task<ResultSet> task)	0.034	17.092	1
97 hidden methods without source	17.059	17.059	100
16 collapsed insignificant methods.	4.032	1,256.920	521

Profile and boost the performance of your .NET applications

- Go straight to the bottleneck with rich performance data and line-level timings
- Find performance bottlenecks in your database, .NET code or file I/O
- Profile SQL queries and see execution plans

Download your free 14-day trial.

ANTS
MEMORY
PROFILER


Performance counters
Bytes in all Heaps
Private Bytes
Large Object Heap size
Working Set - Private
Take Memory Snapshot
Stop Profiling
Analysis
Session Overview
Baseline: snapshot 2 (02:07)
< Snapshots >
Current: snapshot 3 (02:31)
Summary
Class list
Instance categorizer
Instance list
Instance retention graph
Hide finalizer queue GC roots
System.Text.RegularExpressions.Regex
factory
System.Text.RegularExpressions.CompiledRegexRunnerFactory
goMethod
System.Reflection.Emit.DynamicMethod
m_ilGenerator
System.Reflection.Emit.DynamicILGenerator
m_scope
System.Reflection.Emit.DynamicScope
m_tokens
System.Collections.ArrayList
_items
System.Object[]
[12]
System.RuntimeMethodHandle

START HERE!
This graph shows the chains of references to your object that prevent the garbage collector from collecting it. Search up the chains for unexpected references (e.g. event handlers that have not been unregistered), and modify your code to break these references.

Find memory leaks in minutes

- Optimize the memory usage of your C# and VB.NET code
- Profile your code’s use of unmanaged memory
- Create better performing, less resource-intensive applications

Download your free 14-day trial.