Worked example - Profiling an ASP.NET application

This worked example describes how to profile a sample ASP.NET website called NerdDinner. You can download the original ASP.NET MVC source code for NerdDinner from CodePlex: in this example, the NerdDinner code has been modified to illustrate a performance problem.

In this example, NerdDinner is installed on the same computer as the one being used to profile it, and can be accessed in a web-browser at *http://127.0.0.1:8010* .

Imagine that the problem with NerdDinner is that it is slow to return results for a search involving a SQL query. Imagine you want to know whether you can do anything in the application's .NET code to improve the site's performance, before you investigate performance on the database, or spend money to improve the hardware the site runs on.

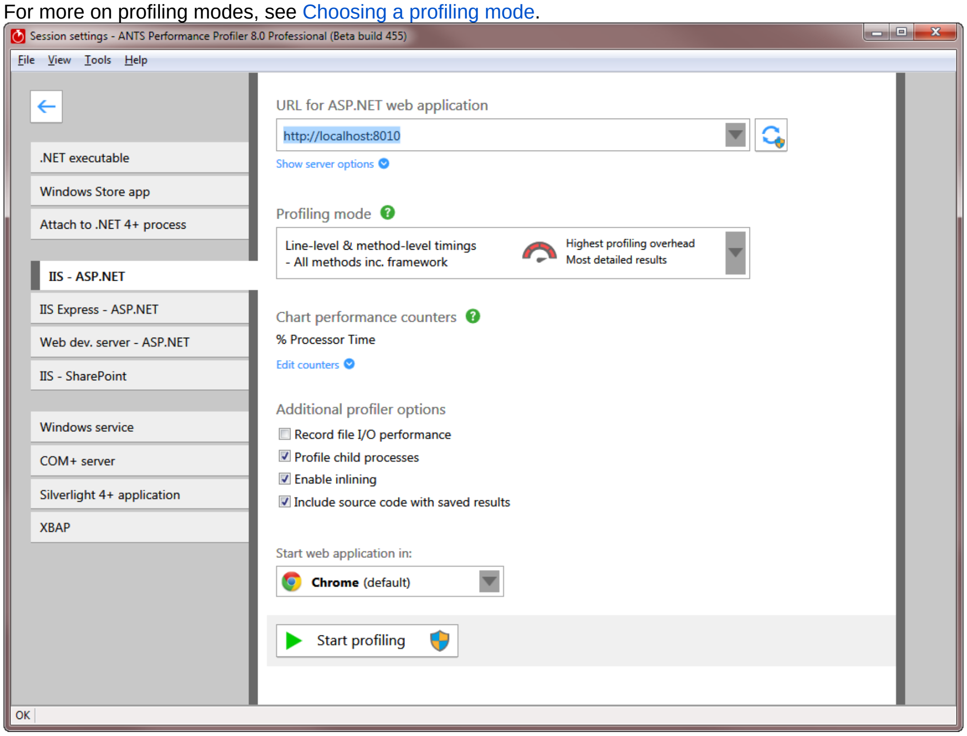
There are three main steps:

1. Set up ANTS Performance Profiler
2. Use NerdDinner
3. Analyze the profiler's results

Setting up ANTS Performance Profiler

To set up ANTS Performance Profiler:

1. In the ANTS Performance Profiler settings, on the Application Settings tab, select **ASP.NET web application (IIS).**
2. In the **ASP.NET web application (URL)** dropdown menu, select the site's URL: *http://127.0.0.1:8010.*
3. Choose the required profiling mode. Here, we'll use line-level and method-level timings.
   For more on profiling modes, see Choosing a profiling mode.



Session settings - ANTS Performance Profiler 8.0 Professional (Beta build 455)

URL for ASP.NET web application: http://localhost:8010

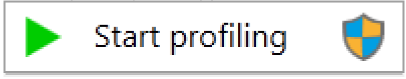
Profiling mode: Line-level & method-level timings - All methods inc. framework (Highest profiling overhead, Most detailed results)

Chart performance counters: % Processor Time

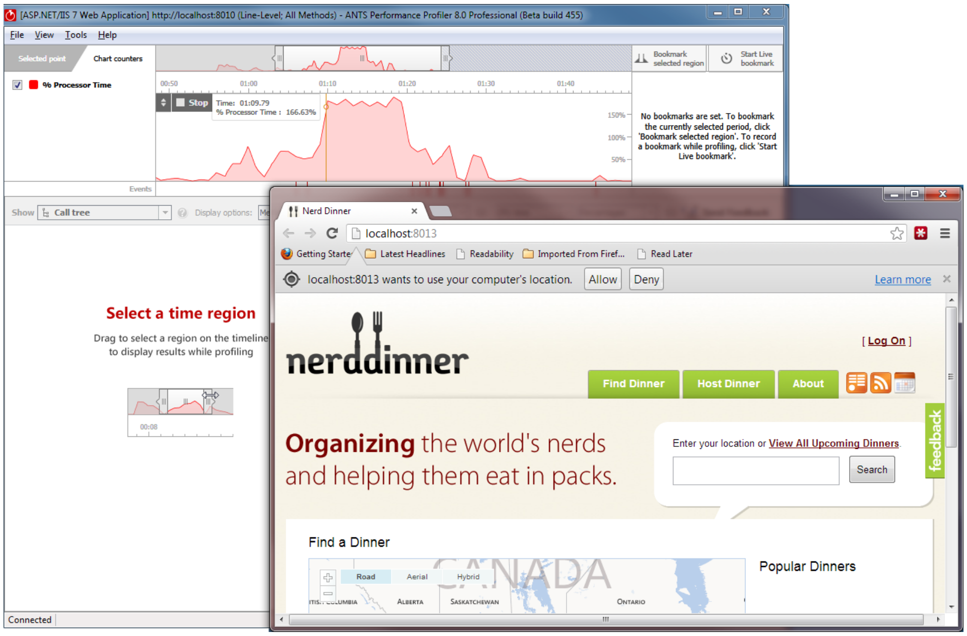
Additional profiler options:
- ☐ Record file I/O performance
- ☑ Profile child processes
- ☑ Enable inlining
- ☑ Include source code with saved results

Start web application in: Chrome (default)

4. Select the port on which to profile your application:
   - If you are using IIS 6, 7 or 8, select **Unused port** and choose a port that is not used by IIS.
     IIS will start a new instance of your application on the specified port.
     Note that this will not work if your application's code binds to a specific port.
   - If you are using IIS 5, or if you are using IIS 6+ and your application binds to a specific port, select **Original port**.
     IIS will restart so that the profiler can attach to the port.
     Note that restarting IIS stops IIS and restarts only the application that you are profiling. Other websites on the same IIS instance will not be present when IIS restarts.

   The port where the application will be profiled is displayed at the bottom of the ANTS Performance Profiler Settings dialog box.
   For more on profiling web applications in IIS, see Profiling ASP.NET applications running on IIS and Troubleshooting IIS profiling.
5. Click  .

6. The web browser you selected launches and loads NerdDinner.



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

Select a time region

Drag to select a region on the timeline to display results while profiling

localhost:8013 wants to use your computer's location.

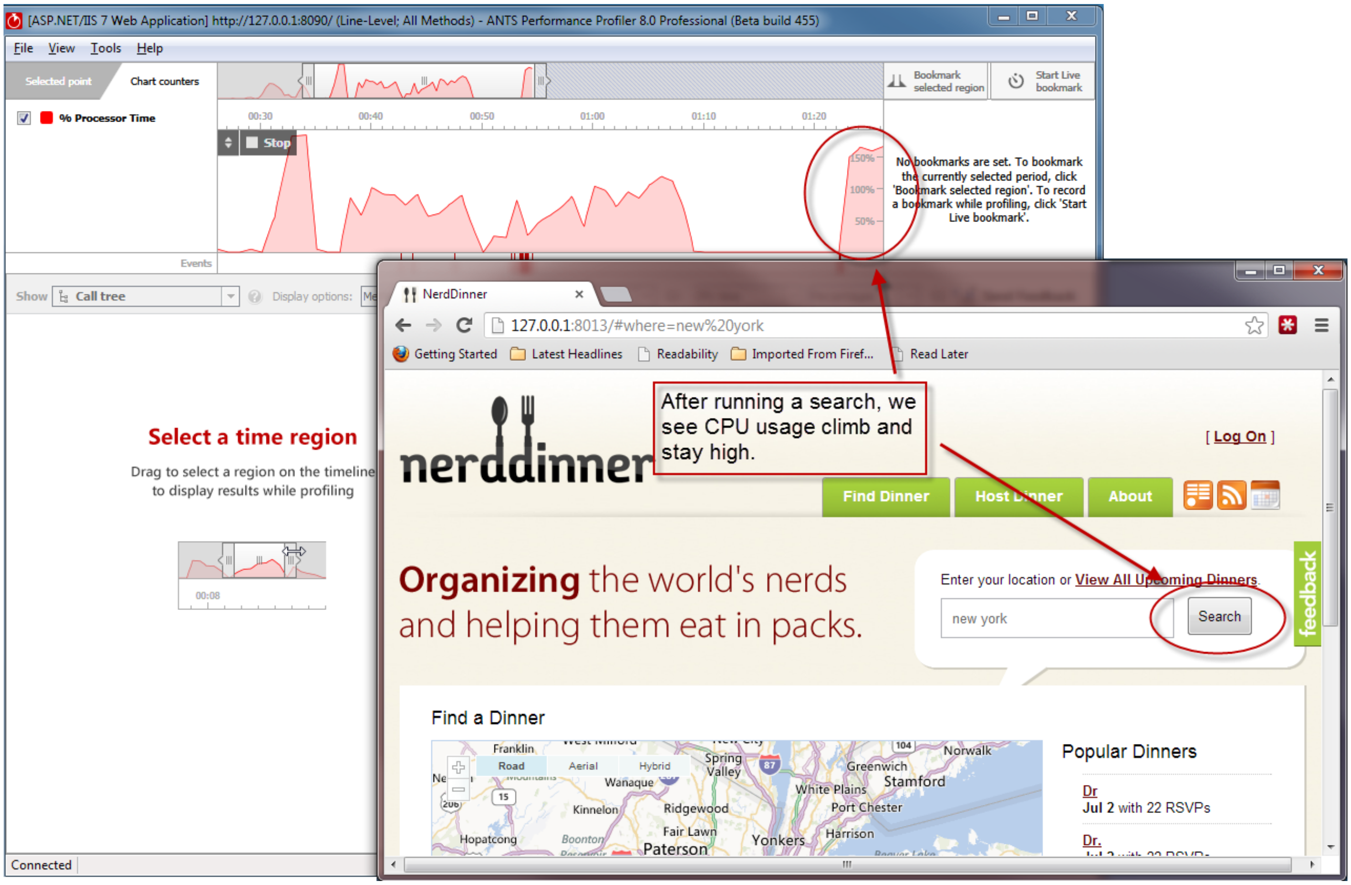
Organizing the world's nerds and helping them eat in packs.

Enter your location or View All Upcoming Dinners

Using NerdDinner

In this example, the NerdDinner site includes several pages that rely heavily on database queries, as well as some static HTML. Imagine users have reported that the site's location search feature is slow: we start the investigation by exercising that feature, entering a place name in the search box and hitting **Search**. The search is designed to return a list of events near a certain location.

While we use the site, ANTS Performance Profiler's timeline shows the CPU usage the application has caused:



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

Select a time region

Drag to select a region on the timeline to display results while profiling

After running a search, we see CPU usage climb and stay high.

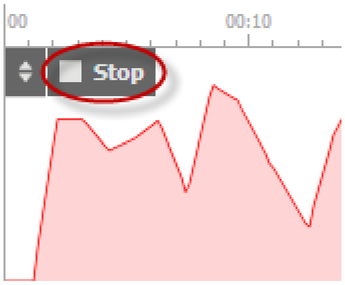
Organizing the world's nerds and helping them eat in packs.

Enter your location or View All Upcoming Dinners

There's a brief, expected CPU peak when the site launches. CPU then returns to near zero until we start our search, when it begins to grow to nearly 200%, staying high for long after the first results were returned. This clearly indicates a performance bottleneck in the search feature.

For more information on analyzing CPU usage, see The timeline.

Now we see where the bottleneck may lie, we can explore the results in detail: In the ANTS Performance Profiler window, click Stop to stop profiling.

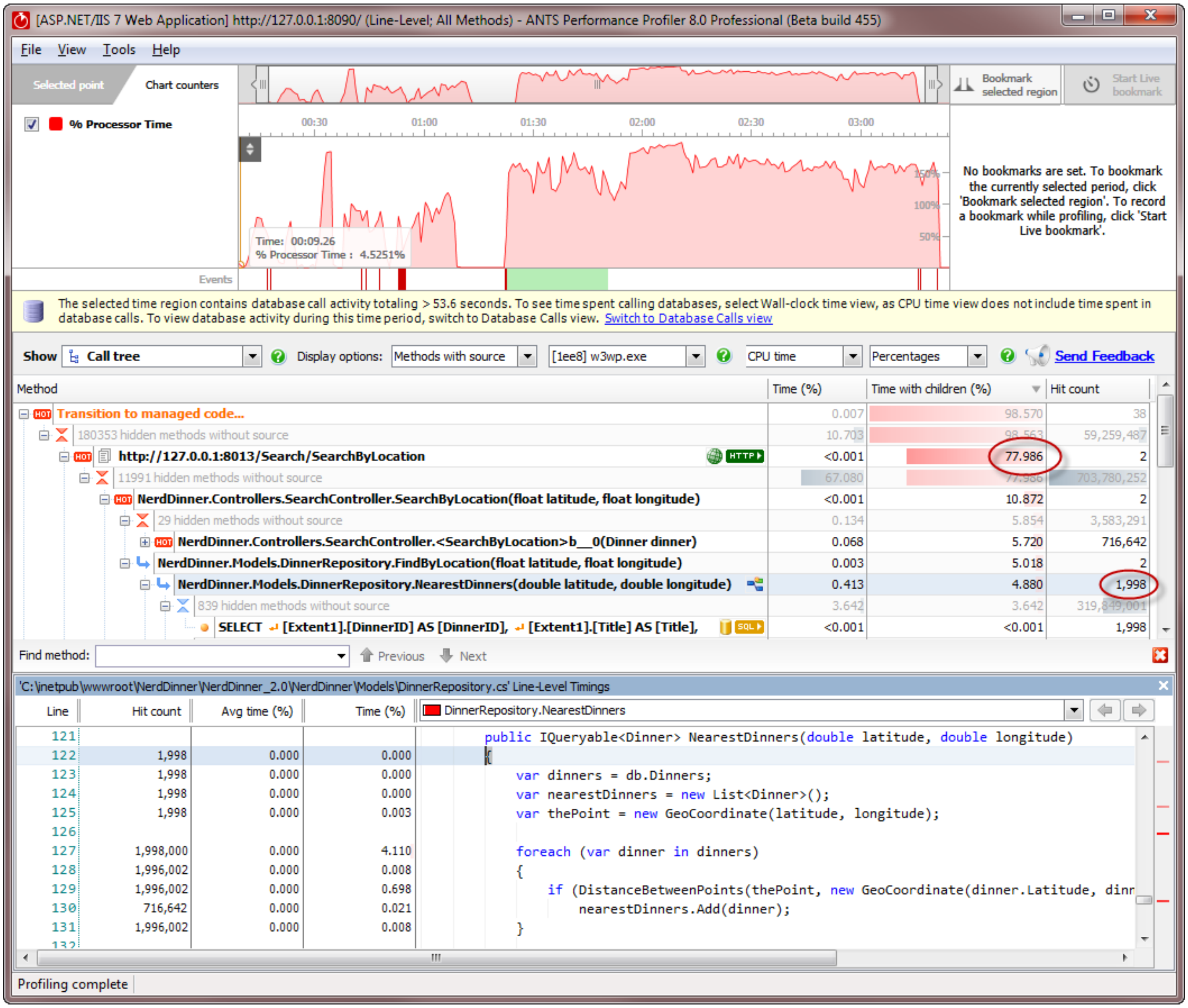


The website closes.

Analyzing the profiler's results

After a few moments, full results are shown.

At the top of the call tree, ANTS Performance Profiler shows the 'hottest' stack trace; that is, the code that contributes the greatest proportion of CPU time. This is usually a good place to start looking for opportunities to optimize the code.


[ASP.NET/IIS 7 Web Application] http://127.0.0.1:8090/ (Line-Level; All Methods) - ANTS Performance Profiler 8.0 Professional (Beta build 455)
File
View
Tools
Help
Selected point
Chart counters
% Processor Time
00:30
01:00
01:30
02:00
02:30
03:00
150%
100%
50%
Time: 00:09.26
% Processor Time : 4.5251%
Events
Bookmark selected region
Start Live bookmark

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

The selected time region contains database call activity totaling > 53.6 seconds. To see time spent calling databases, select Wall-clock time view, as CPU time view does not include time spent in database calls. To view database activity during this time period, switch to Database Calls view. Switch to Database Calls view

Show: Call tree
Display options: Methods with source
[1ee8] w3wp.exe
CPU time
Percentages
Send Feedback

Method	Time (%)	Time with children (%)	Hit count
Transition to managed code...	0.007	98.570	38
180353 hidden methods without source	10.703	98.563	59,259,487
http://127.0.0.1:8013/Search/SearchByLocation	<0.001	77.986	2
11991 hidden methods without source	67.080	77.986	703,780,252
NerdDinner.Controllers.SearchController.SearchByLocation(float latitude, float longitude)	<0.001	10.872	2
29 hidden methods without source	0.134	5.854	3,583,291
NerdDinner.Controllers.SearchController.<SearchByLocation>b__0(Dinner dinner)	0.068	5.720	716,642
NerdDinner.Models.DinnerRepository.FindByLocation(float latitude, float longitude)	0.003	5.018	2
NerdDinner.Models.DinnerRepository.NearestDinners(double latitude, double longitude)	0.413	4.880	1,998
839 hidden methods without source	3.642	3.642	319,849,001
SELECT [Extent1].[DinnerID] AS [DinnerID], [Extent1].[Title] AS [Title],	<0.001	<0.001	1,998

Find method: Previous Next

'C:\inetpub\wwwroot\NerdDinner\NerdDinner_2.0\NerdDinner\Models\DinnerRepository.cs' Line-Level Timings

DinnerRepository.NearestDinners

Line	Hit count	Avg time (%)	Time (%)
121			
122	1,998	0.000	0.000
123	1,998	0.000	0.000
124	1,998	0.000	0.000
125	1,998	0.000	0.003
126			
127	1,998,000	0.000	4.110
128	1,996,002	0.000	0.008
129	1,996,002	0.000	0.698
130	716,642	0.000	0.021
131	1,996,002	0.000	0.008
132			

```
public IQueryable<Dinner> NearestDinners(double latitude, double longitude)
{
    var dinners = db.Dinners;
    var nearestDinners = new List<Dinner>();
    var thePoint = new GeoCoordinate(latitude, longitude);

    foreach (var dinner in dinners)
    {
        if (DistanceBetweenPoints(thePoint, new GeoCoordinate(dinner.Latitude, dinr
            nearestDinners.Add(dinner);
    }
```

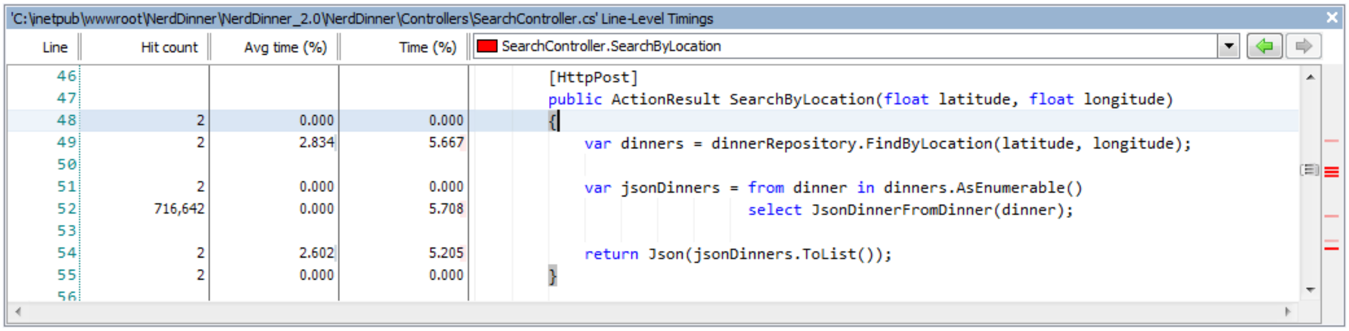
Profiling complete

At the top of each call tree stack, we see the HTTP request that triggered the calling of the .NET methods. As we expected, the hottest stack trace during the whole profiling session descends from the request *http://127.0.0.1:8013/Search/SearchByLocation*, child methods of which account for about 78% of the total time spent in the profiling session. (Notice that the port used for profiling, 8013, isn't the same as the original port, 8010).

Looking down the hot stack, we can see that this request called a .NET method, `NerdDinner.Models.DinnerRepository.NearestDinners (double latitude, double longitude)`, that was hit 1998 times - as was the SQL `SELECT` query it ultimately runs.

For more information on hot stacks and HTTP nodes, see The call tree.

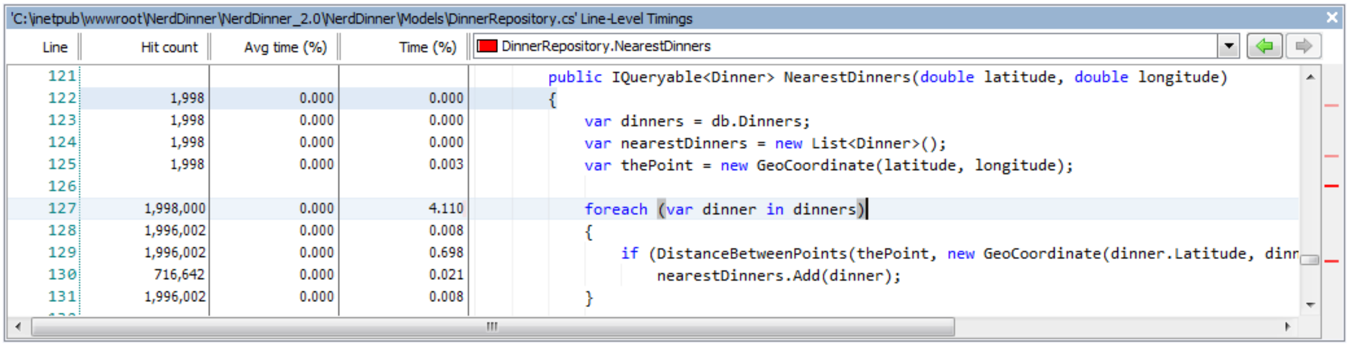
If we select the method's parent, `NerdDinner.Controllers.SearchController.SearchByLocation(float latitude, float longitude)`, we can view its source code. Because we used a profiling mode with line-level timings, we can also see where inside the method the greatest time was spent. This shows us that the method retrieves the full list of all recorded events from the database:



```
'C:\inetpub\wwwroot\NerdDinner\NerdDinner_2.0\NerdDinner\Controllers\SearchController.cs' Line-Level Timings

Line   Hit count   Avg time (%)   Time (%)   SearchController.SearchByLocation
  46                                                 [HttpPost]
  47                                                 public ActionResult SearchByLocation(float latitude, float longitude)
  48           2          0.000      0.000           {
  49           2          2.834      5.667               var dinners = dinnerRepository.FindByLocation(latitude, longitude);
  50
  51           2          0.000      0.000               var jsonDinners = from dinner in dinners.AsEnumerable()
  52     716,642          0.000      5.708                                 select JsonDinnerFromDinner(dinner);
  53
  54           2          2.602      5.205               return Json(jsonDinners.ToList());
  55           2          0.000      0.000           }
  56
```

The method `NerdDinner.Models.DinnerRepository.FindByLocation(float latitude, float longitude)` then tries to process the result set in the web page, to filter by location:



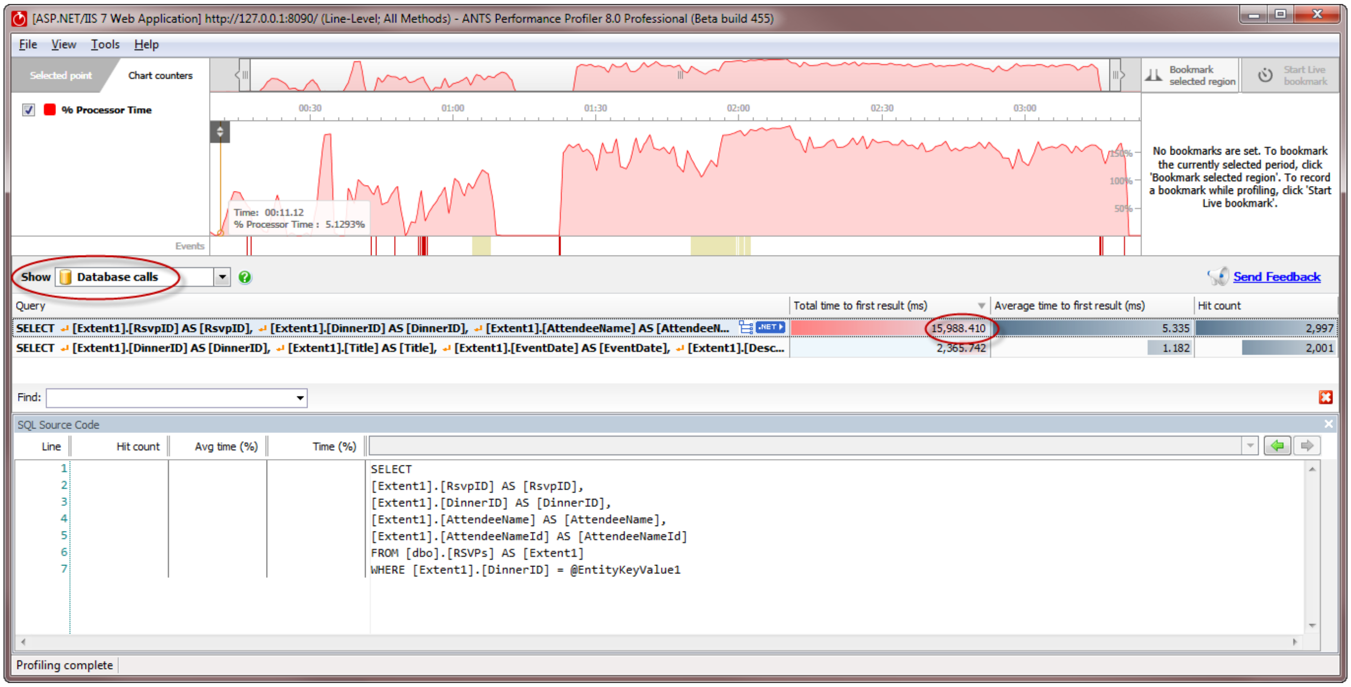
```
'C:\inetpub\wwwroot\NerdDinner\NerdDinner_2.0\NerdDinner\Models\DinnerRepository.cs' Line-Level Timings

Line   Hit count   Avg time (%)   Time (%)   DinnerRepository.NearestDinners
 121                                                 public IQueryable<Dinner> NearestDinners(double latitude, double longitude)
 122       1,998          0.000      0.000           {
 123       1,998          0.000      0.000               var dinners = db.Dinners;
 124       1,998          0.000      0.000               var nearestDinners = new List<Dinner>();
 125       1,998          0.000      0.003               var thePoint = new GeoCoordinate(latitude, longitude);
 126
 127   1,998,000          0.000      4.110               foreach (var dinner in dinners)
 128   1,996,002          0.000      0.008               {
 129   1,996,002          0.000      0.698                   if (DistanceBetweenPoints(thePoint, new GeoCoordinate(dinner.Latitude, dinr
 130     716,642          0.000      0.021                       nearestDinners.Add(dinner);
 131   1,996,002          0.000      0.008               }
```

These methods are good candidates for optimization. The same results could be achieved via AJAX calls, and by returning from the database only events that meet specified latitude and longitude criteria.

For more on the source code view, see Source code.

Switching to Database Calls view, we can see that the query to return the full results set was run thousands of times, summing to nearly 16 seconds just to return the first result for all the instances of the query:


[ASP.NET/IIS 7 Web Application] http://127.0.0.1:8090/ (Line-Level; All Methods) - ANTS Performance Profiler 8.0 Professional (Beta build 455)
File View Tools Help
Selected point
Chart counters
% Processor Time
00:30
01:00
01:30
02:00
02:30
03:00
Time: 00:11.12
% Processor Time : 5.1293%
150%
100%
50%
Events
Bookmark selected region
Start Live bookmark
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
Database calls
Send Feedback
Find:
SQL Source Code
Line
Hit count
Avg time (%)
Time (%)
Profiling complete

Query	Total time to first result (ms)	Average time to first result (ms)	Hit count
SELECT [Extent1].[RsvpID] AS [RsvpID], [Extent1].[DinnerID] AS [DinnerID], [Extent1].[AttendeeName] AS [AttendeeN...	15,988.410	5.335	2,997
SELECT [Extent1].[DinnerID] AS [DinnerID], [Extent1].[Title] AS [Title], [Extent1].[EventDate] AS [EventDate], [Extent1].[Desc...	2,365.742	1.182	2,001

```
SELECT
[Extent1].[RsvpID] AS [RsvpID],
[Extent1].[DinnerID] AS [DinnerID],
[Extent1].[AttendeeName] AS [AttendeeName],
[Extent1].[AttendeeNameId] AS [AttendeeNameId]
FROM [dbo].[RSVPs] AS [Extent1]
WHERE [Extent1].[DinnerID] = @EntityKeyValue1
```

Again, it's clear that this very broad request is being run repeatedly, contributing a large total running time. It would be more efficient to run a more precise request fewer times.

For more on SQL query timings, see The database calls view.

After profiling, we now have a clear idea of which HTTP requests are associated with slow performance, and which of our .NET methods contain the source of those slowdowns. We know which methods to rewrite to remove bottlenecks, and the steps we'll need to reproduce in the application to check that the problem has gone.