A day in the life of a slow page at Stack Overflow

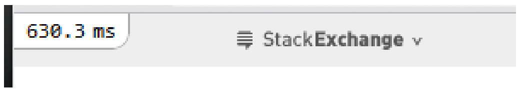
over 14 years ago

In this post I would like to walk through our internal process of tuning a particular page on Stack Overflow.

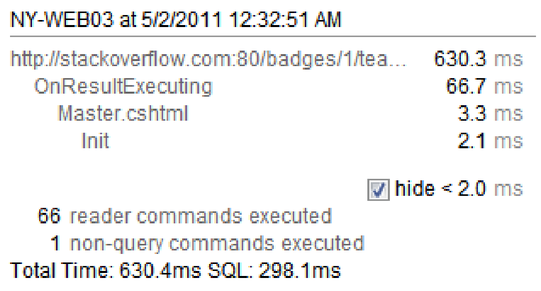
In this example I will be looking at our badge detail page. It is not the most important page on the site, however it happens to showcase quite a few issues.

Step 1, Do we even have a problem?

As a developer, when I browse through the sites I can see how long a page takes to render.



Furthermore, I can break down this time and figure out what is happening on the page:



```
NY-WEB03 at 5/2/2011 12:32:51 AM
http://stackoverflow.com:80/badges/1/tea...   630.3 ms
  OnResultExecuting                            66.7 ms
    Master.cshtml                               3.3 ms
      Init                                      2.1 ms

                                   hide < 2.0 ms
  66 reader commands executed
   1 non-query commands executed
Total Time: 630.4ms SQL: 298.1ms
```

To make our lives even easier, I can see a timeline view of all the queries that executed on the page (at the moment as an html comment):

```
T+3ms [1.26ms] SELECT [t0].[Id], [t0].[Class], [t0].[Name], [t0].[Descript
[t0].[AwardF

T+21ms [11.29ms] SELECT COUNT(*) AS [value] FROM [dbo].[Users2Badges] AS [
OUTER JOIN

T+274ms [49.84ms] SELECT [t7].[Id], [t7].[UserId], [t7].[BadgeId], [t7].[D
[PostTypeId], [

T+324ms [1.04ms] SELECT TOP (1) [t0].[Id], [t0].[PostTypeId], [t0].[Creati
[LastEditorUserId], [t

T+325ms [0.93ms] SELECT TOP (1) [t0].[Id], [t0].[PostTypeId], [t0].[Creati
[LastEditorUserId], [t

T+326ms [1.06ms] SELECT TOP (1) [t0].[Id], [t0].[PostTypeId], [t0].[Creati
[LastEditorUserId], [t

T+327ms [0.94ms] SELECT TOP (1) [t0].[Id], [t0].[PostTypeId], [t0].[Creati
[LastEditorUserId], [t

T+328ms [0.96ms] SELECT TOP (1) [t0].[Id], [t0].[PostTypeId], [t0].[Creati
```

Twitter updates

@samsaffron

Stack Overflow

recent questions

recent answers

Online content

Off the Rails

Measuring Ruby

Why I am excited about Ruby 2.1?

Docker Deploys on Ruby Rogues

Discourse on Ruby Rogues

About me

I am a co-founder of Discourse. I live in sunny Sydney. I love writing about performance, Ruby, Mini Profiler and technology related topics.

This is made possible using our mini profiler and a private implementation of DbConnection, DbCommand and DbDataReader. Our implementations of the raw database access constructs provides us with both tracking and extra protection (If we figure out a way of packaging this we will be happy to open source).

When I looked at this particular page I quickly noticed a few concerning facts.

1. It looks like we are running lots of database queries, our old trusty friend N+1.
2. I noticed a very concerning fact, more than half of the time is spent on the web server.
3. We are running some expensive queries, in particular a count that takes 11ms and a select that takes about 50ms.

At this point, I need to decide if it makes sense to spend any more effort here. There may be bigger and more important things I need to work on.

Since we store all of the haproxy logs (and retain them for a month), I am able to see how long it takes to render any page on average, as well as how many times this process runs per day.

```
select COUNT(*), AVG(Tr) from
dbo.Log_2011_05_01
where Uri like '/badges/%'


----------- -----------
26834       532
```

This particular family of pages is accessed 26k times a day; it takes us, on average, 532ms to render. Ouch … pretty painful. Nonetheless, I should probably look at other areas; this is not the most heavily accessed area in the site.

That said, Google takes into account the speed it takes to access your pages, if your page render time is high, it will affect your Google Ranking.

I wrote this mess. I know it can be done way faster.

Code review

The first thing I do is a code review,

```
var badges = from u2b in DB.Users2Badges
    join b in DB.Badges on u2b.BadgeId equals b.Id
    from post in DB.Posts.Where(p => p.Id == u2b.ReasonId && b
    from tag in DB.Tags.Where(t => t.Id == u2b.ReasonId && b.Ba
    where u2b.BadgeId == badge.Id
    orderby u2b.Date descending
    select new BadgesViewModel.BadgeInfo
    {
        Users2Badge = u2b,
        Post = post,
        Tag = tag,
        User = u2b.User,
        Badge = badge
    };
    var paged = badges.ToPagedList(page.Value, 60);
```

A typical LINQ-2-SQL multi join. ToPagedList performs a Count, and offsets the results to the particular page we are interested using Skip and Take.

Nothing horribly wrong here, in fact this probably considered a best practice.

However as soon as we start measuring I quickly notice that even though the SQL for this on local only takes 12 or so milliseconds, the total time it takes to execute the above code is much higher, profiling the above block shows that a 90ms execution time.

The LINQ-2-SQL abstraction is leaking 78ms of performance, OUCH. This is because it needs to generate a SQL statement from our fancy inline LINQ. It also needs to generate deserializers for our 5 objects. 78ms of overhead for 60 rows, to me seems unreasonable.

Not only did performance leak here, some ungodly SQL was generated ... twice:

[illegible]

Dapper to the rescue

The first thing I set to do is rewrite the same query in raw SQL using dapper. Multi mapping support means it is really easy to do such a conversion.

```
    var sql = @"select  ub.*, u.*, p.*, t.* from
    (
        select *, row_number() over (order by Date desc) Row
        from Users2Badges
        where BadgeId = @BadgeId
    ) ub
    join Badges b on b.Id = ub.BadgeId
    join Users u on u.Id = ub.UserId
    left join Posts p on p.Id = ub.ReasonId and b.BadgeReasonT
    left join Tags t on t.Id = ub.ReasonId and b.BadgeReasonTy
    where Row > ((@Page - 1) * @PerPage) and Row <= @Page * @P
    order by ub.Date desc
    ";

    var countSql = @"select count(*) from Users2Badges ub join

    var rows = Current.DB.Query<Users2Badge, User, Post, Tag,
                {
                    Users2Badge = ub,
                    User = u,
                    Post = p,
                    Tag = t,
                    Badge = badge
```

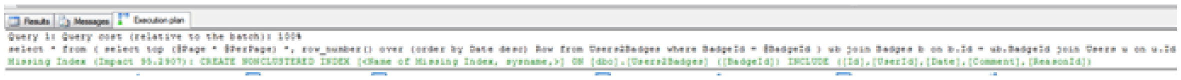
```
             }, new { Page = page, BadgeId = badge.Id, PerPa
    var count = Current.DB.Query<int>(countSql, new { Id = id.\
```

I measure 0.2ms of performance leak in dapper … sometimes. Sometimes it is too close to count. We are making progress. Personally, I also find this more readable and debuggable.

This conversion comes with a multitude of wonderful side effects. I now have SQL I can comfortably work with in SQL Management Studio. In SQL Management Studio, I am superman. I can mold the query into the shape I want it and I can profile it. Also the count query is way simpler than the select query.

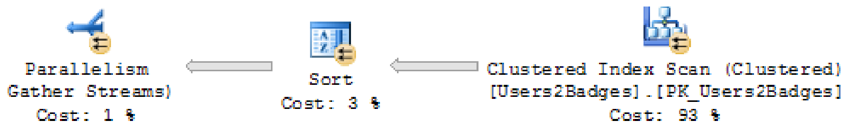
We are not done quite yet

Removing the ORM overhead is one thing, however we also had a few other problems. I noticed that in production this query was taking way longer. I like to keep my local DB in sync with production so I can experience similar pains locally. So this smells of a bad execution plan being cached. This usually happens cause indexes are missing. I quickly run the query with an execution plan and see:



```
Query 1: Query cost (relative to the batch): 100%
select * from ( select top (@Page * @PerPage) *, row_number() over (order by Date desc) Row from Users2Badges where BadgeId = @BadgeId ) ub join Badges b on b.Id = ub.BadgeId join Users u on u.Id
Missing Index (Impact 95.2907): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users2Badges] ([BadgeId]) INCLUDE ([Id],[UserId],[Date],[Comment],[ReasonId])
```

And one big FAT line:


Parallelism
Gather Streams)
Cost: 1 %
Sort
Cost: 3 %
Clustered Index Scan (Clustered)
[Users2Badges].[PK_Users2Badges]
Cost: 93 %

Looks like the entire clustered index on the Users2Badges table is being scanned just to pull out the first page. Not good. When you see stuff like that it also introduces erratic query performance; What will happen when the query needs to look at a different badge or different page? This query may be “optimal” given the circumstances, for page 1 of this badge. However, it may be far from optimal for a badge that only has 10 rows in the table. An index scan + bookmark lookup may do the trick there.

In production, depending on who hit what page first for which badge, you are stuck with horrible plans that lead to erratic behavior. The fix is NOT to put in OPTION (RECOMPILE) it is to correct the indexing.

So, SQL Server tells me it wants an index on Users2Badges(BadgeId) that includes (Id,UserId,Date,Comment,ReasonId). I will try anything once, so I give it a shot, and notice there is absolutely no performance impact. This index does not help.

Recommendation from the execution plan window are often wrong

We are trying to pull out badge records based on badge id ordered on date. So the correct index here is:

```
CREATE NONCLUSTERED INDEX Users2Badges_BadgeId_Date_Includes
ON [dbo].[Users2Badges] ([BadgeId], [Date])
INCLUDE ([Id],[UserId],[Comment],[ReasonId])
```

This new index cuts SQL time on local by a few milliseconds, but more importantly, it cuts down the time it takes to pull page 100 by a factor of 10. The same plan is now used regardless of Page or Badge from a cold start.

Correcting the N+1 issue.

Lastly, I notice 50-60ms spent on an N+1 query on my local copy, fixing this is easy. Looks like we are pulling parent posts for the posts associated with the badges, the profiler shows me that this work is deferred to the View on a per row basis. I quickly change the ViewModel to pull in the missing posts and add a left join to the query to pull them in. Easy enough.

Conclusion

After deploying these fixes in production, load time on the page went down from 630ms to 40ms, for pages deep in the list it is now over 100x faster.

Tuning SQL is key, the simple act of tuning it reduced the load time for the particular page by almost 50% in production. However, having a page take 300ms just because your ORM is inefficient is not excusable. The page is now running at the totally awesome speed of 20-40ms render time. In production. ORM inefficiency cost us a 10x slowdown.

Posted by: Sam Permalink | Comments (35)

Comments

Rob_Farley over 14 years ago

Yup â€“ tuning SQL is the key. It's one of those things that many developers don't realise, and why I'm very happy to be working the SQL space. I'm also very proud to think that a query of mine is in the StackOverflow codebase. :)

http://stackoverflow.com/questions/1176011/sql-to-determine-minimum-sequential-days-of-access/1176255#1176255

Rob

Paul_White over 14 years ago

Even with an optimal index, using ROW_NUMBER to find a page can be less than optimal. See why at:

http://www.sqlservercentral.com/articles/paging/69892/

http://www.sqlservercentral.com/articles/paging/70120/

Paul

Sam Saffron over 14 years ago

Yeah the new denali feature is going to be awesome: denali paging

with this particular page the biggest gain left would be to cache the counts... since that is the most expensive query left on the page, but still it would only shave off 4/5ms ... not really worth it imho, keep in mind the row here is actually fairly narrow

James over 14 years ago

Since your ORM tax is only going to get proportionately bigger as the site continues to grow, how long before you guys decide to just rip out Linq2Sql completely and replace it with Dapper?

Sam Saffron over 14 years ago

Well, we have been ripping it out of most of the select heavy areas. Not sure when or if L2S will be gone, our update paths still heavily depend on it.

Paul_White over 14 years ago

Hi Sam,

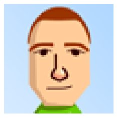
Yes, for the most part, I agree with you. The paging optimizations I linked to will not be essential in every situation, but it is something to bear in mind. They're almost never less efficient than doing it the ROW_NUMBER way. And who knows, the table might get wider in futureâ€¦?

The OFFSET clause in Denali is almost entirely syntactic sugar; there are no performance benefits over a ROW_NUMBER implementation. See http://sqlblogcasts.com/blogs/sqlandthelike/archive/2010/11/10/denali-paging-is-it-win-win.aspx for an example.

It's fair to say that OFFSET represents a bit of a missed opportunity. Maybe in SQL12 :)

Paul

Sam Saffron over 14 years ago



@Paul interesting link.

will keep all of this in mind next time around. Good point.

Hoping the team improve perf around OFFSET, its such a key feature for people who build web sites.

Martijn_Verburg over 14 years ago

Awesome post!! Really nice to see a problem dissected like this. And it's a valuable lesson to developers out there that you cannot simply rely on your ORM technology to automatically deliver you performance.

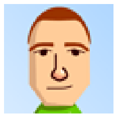
Please keep making more posts like these!

Cheers, Martijn (@karianna, @java7developer)

Meandmycode over 14 years ago

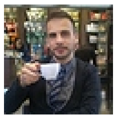
I find it interesting that you discover an n+1 query problem but the first thing you do is rip out the orm, it would've been nice to objectively approach this and say, actually once I added the index and fixed the query, query translation was still too slow.. the reason I say this is that it just seems like you want to use your object materialiser regardless, even if this isn't the case

Sam Saffron over 14 years ago



the N+1 impact was between 50 and 60ms ... compared to the rest this is not huge (in prd we were talking about 15% at the worst case), I ripped the ORM out first cause it was a huge offender, the biggest offender in my local copy. You can actually see the N+1 query in the screenshot, with the time it takes and the amount it offsets the total time ...

Jack_Nova over 14 years ago



Owesome!

Ryan_Lowdermilk over 14 years ago

Bravo! Nicely done!

Harry over 14 years ago

You say that the original performance problem was because LINQ2SQL â€œneeds to generate a SQL statement from our fancy inline LINQâ€.

I'd have thought the first thing to try in this case would be to use a compiled query (http://msdn.microsoft.com/enus/library/bb399335.aspx). It should make a significant difference if the SQL generation is the biggest performance offender. Is there a reason you didn't try this approach?

I haven't used LINQ2SQL much, but with LINQ2Entities have seen a factor of ten improvement when switching a complex query to a compiled query.

Of course, that only makes a difference to the ORM layer. It seems like adding the new index and resolving the N+1 issue would be required in any case.

Sam Saffron over 14 years ago

Harry,

A few reasons I avoided this, there is the delegate dance and clunky coding I would need to introduce for 2 queries, this also decreases extensibility since I would need new compiled queries for every filter I add.

I was also uncomfortable with the SQL it was generating, my handcoded solution is faster.

Finally, dapper overhead is negligible, when I did the perf testing for dapper I noticed the even compiled queries introduce a non-trivial overhead.

For completeness I just did a basic port of compiled query, L2S was running at 87ms, dapper at 11.9ms

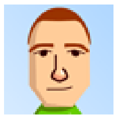
In this case I did not even handle the N+1 in L2S, eager loading in L2S is pretty tricky.

gist here

Cagdas over 14 years ago

By the way, is that an add-on displaying the time it took to render the page on the top left corner? If so, mind sharing its name? :)

Sam Saffron over 14 years ago



We are still trying to figure out a way packaging and distributing it.

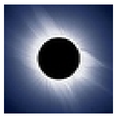
Kowsik over 14 years ago

Went through something similar with a location-aware iPhone app with SQL nastiness in the backend. Problem turned out to be running large geospatial queries on a column that wasn't indexed. SQL is tricky business.

Jim_Lawruk over 14 years ago

This is an awesome post! Thanks for taking the time to walk us through this, and thanks to Stack Overflow for taking the time to optimize for speed.

Javin over 14 years ago



Completely agree with Jim , excellent post and the walk through is really useful for anyone who is doing some page optimization, though it looks you guys got a great tool called mini profiler,how about sharing that tool ?

Javin

Joris_Witteman over 14 years ago

It takes you seven steps to decide to turn on indexing on a table and move to a JOIN type query?

Something tells me you have terrible overhead in your project.

Felipe over 14 years ago

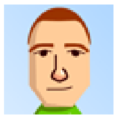
Sam,

At your select: select *, row_number() over (order by Date desc)

You can add `count() over()`, then you dont need another select to count, and the impact is very low, example:

```
WITH P AS(
SELECT
*,
ROW_NUMBER() OVER(ORDER BY cd_produto) as Linha,
COUNT(1) OVER() as Total
FROM produto)
SELECT * FROM P WHERE Linha BETWEEN 10 AND 20
```

Sam Saffron over 14 years ago



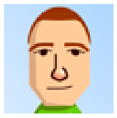
I had a look at that trick and it did not really help out, the separate count query can be much narrower so it is able to use other indexes. This trick seems to hurt perf for page 1 and early pages.

Kyro over 14 years ago

You mention that stackoverflow uses private implementations of DbConnection, DbCommand and DbDataReader, and that you trying to figure out a way to package and open source them.

I'm very eager to see how stackoverflow handles tracking at the Connect/Command/Reader level. Could you provide a general idea of how these are implemented?

Sam Saffron over 14 years ago



The general idea is that we have simple proxy objects that delegate to the actual classes that do the work and measure. This information is tracked in the base class and the interrogated at the end of the request.

Alexander_Nyquist over 14 years ago

Thanks for a very interesting and informative post. Always fun to read about real-world problems.

What way do you use to hook into the pipeline so that you can log all rendering events etc.?

Sam Saffron over 14 years ago

we proxy the connection object, we expect to open source this work in the next few weeks

Mohamad about 14 years ago

Sam, I'm just wondering, when you say page load time went from 630 to 40ms, is that the total page load time? The entire request? I mean, is there no overhead associated with the application/framework itself? That's insanely fast! I know this has a lot to do with hardware, but my pages never ever load in any where near that amount of speed, queries or no queries, on my home machine, which I imagine is a lot faster than my share hosting!

Stephen about 14 years ago

Hey Sam,

In the Dapper to the rescue code, are you even using the parameter PerPage which you set to 60, but you have hard code in the sql?

Thanks man for the most awesome Dapper.Net

Sam Saffron about 14 years ago

great catch ... thanks for that 🙂

Abelito almost 14 years ago

This is a great post, that was a really interesting read!

Enamrik over 13 years ago

If you had the chance to do StackOverflow again, would you use Dapper.NET exclusively and not use a full ORM?

Sam Saffron over 13 years ago

yes, if I were doing it all over I would just use dapper and a bunch of custom helpers like rainbow, sql builder etc.

Yannis_Rizos about 13 years ago

Hey Sam, your last comment spurred a ProgSE question â€“ Care to answer?

Date_Command_In_Unix about 13 years ago

Writing a good sql is the most important to get the query results quickly. The query needs to be tuned well. The tuning depends on lot things like

Operating system (unix/linux) processing power. DB configuration ETC

Andrei Rînea over 10 years ago

Yet more proof to avoid ORMs and use microORM’s with parameterized queries or SPROCs.

Leave a Comment