How can I make my applications scale well?

Asked 16 years, 3 months ago Modified 13 years, 4 months ago
Viewed 1k times

8

In general, what kinds of design decisions help an application scale well?

(Note: Having just learned about Big O Notation, I'm looking to gather more principles of programming here. I've attempted to explain Big O Notation by answering my own question below, but I want the community to improve both this question and the answers.)

Responses so far
1) Define scaling. Do you need to scale for lots of users, traffic, objects in a virtual environment?
2) Look at your algorithms. Will the amount of work they do scale linearly with the actual amount of work - i.e. number of items to loop through, number of users, etc?
3) Look at your hardware. Is your application designed such that you can run it on multiple machines if one can't keep up?

Secondary thoughts
1) Don't optimize too much too soon - test first. Maybe bottlenecks will happen in unforseen places.
2) Maybe the need to scale will not outpace Moore's Law,

and maybe upgrading hardware will be cheaper than refactoring.

algorithm language-agnostic scalability

Share

Improve this question

Follow

edited Nov 4, 2008 at 19:11

community wiki
4 revs
Nathan Long

7 Answers

Sorted by: Highest score (default)

11

The only thing I would say is write your application so that it can be deployed on a cluster from the very start. Anything above that is a premature optimisation. Your first job should be getting enough users to have a scaling problem.

Build the code as simple as you can first, then profile the system second and optimise only when there is an obvious performance problem.

Often the figures from profiling your code are counter-intuitive; the bottle-necks tend to reside in modules you didn't think would be slow. Data is king when it comes to optimisation. If you optimise the parts you think will be slow, you will often optimise the wrong things.

Share Improve this answer Follow

edited Sep 3, 2008 at 9:48

answered Sep 3, 2008 at 9:41

Simon Johnson
7,902 ● 5 ● 38 ● 50

6

Ok, so you've hit on a key point in using the "big O notation". That's one dimension that can certainly bite you in the rear if you're not paying attention. There are also other dimensions at play that some folks don't see through the "big O" glasses (but if you look closer they really are).

A simple example of that dimension is a database join. There are "best practices" in constructing, say, a left inner join which will help to make the sql execute more efficiently. If you break down the relational calculus or even look at an explain plan (Oracle) you can easily see which indexes are being used in which order and if any table scans or nested operations are occurring.

The concept of profiling is also key. You have to be instrumented thoroughly and at the right granularity across all the moving parts of the architecture in order to identify and fix any inefficiencies. Say for example you're building a 3-tier, multi-threaded, MVC2 web-based application with liberal use of AJAX and client side processing along with an OR Mapper between your app and the DB. A simplistic linear single request/response flow looks like:

```
browser -> web server -> app server -> DB -> app
server -> XSLT -> web server -> browser JS engine
execution & rendering
```

You should have some method for measuring performance (response times, throughput measured in "stuff per unit time", etc.) in each of those distinct areas, not only at the box and OS level (CPU, memory, disk i/o, etc.), but specific to each tier's service. So on the web server you'll need to know all the counters for the web server your're using. In the app tier, you'll need that plus visibility into whatever virtual machine you're using (jvm, clr, whatever). Most OR mappers manifest inside the virtual machine, so make sure you're paying attention to all the specifics if they're visible to you at that layer. Inside the DB, you'll need to know *everything* that's being executed and all the specific tuning parameters for your flavor of DB. If you have big bucks, BMC Patrol is a pretty good bet for most of it (with appropriate knowledge modules (KMs)). At the cheap end, you can certainly roll your own but your mileage will vary based on your depth of expertise.

Presuming everything is synchronous (no queue-based things going on that you need to wait for), there are tons of opportunities for performance and/or scalability issues.

But since your post is about scalability, let's ignore the browser except for any remote XHR calls that will invoke another request/response from the web server.

So given this problem domain, what decisions could you make to help with scalability?

1. Connection handling. This is also bound to session management and authentication. That has to be as clean and lightweight as possible without compromising security. The metric is maximum connections per unit time.
2. Session failover at each tier. Necessary or not? We assume that each tier will be a cluster of boxes horizontally under some load balancing mechanism. Load balancing is typically very lightweight, but some implementations of session failover can be heavier than desired. Also whether you're running with sticky sessions can impact your options deeper in the architecture. You also have to decide whether to tie a web server to a specific app server or not. In the .NET remoting world, it's probably easier to tether them together. If you use the Microsoft stack, it may be more scalable to do 2-tier (skip the remoting), but you have to make a substantial security tradeoff. On the java side, I've always seen it at least 3-tier. No reason to do it otherwise.
3. Object hierarchy. Inside the app, you need the cleanest possible, lightest weight object structure possible. Only bring the data you need when you

need it. Viciously excise any unnecessary or superfluous getting of data.

4. OR mapper inefficiencies. There is an impedance mismatch between object design and relational design. The many-to-many construct in an RDBMS is in direct conflict with object hierarchies (person.address vs. location.resident). The more complex your data structures, the less efficient your OR mapper will be. At some point you may have to cut bait in a one-off situation and do a more...uh...primitive data access approach (Stored Procedure + Data Access Layer) in order to squeeze more performance or scalability out of a particularly ugly module. Understand the cost involved and make it a conscious decision.
5. XSL transforms. XML is a wonderful, normalized mechanism for data transport, but man can it be a huge performance dog! Depending on how much data you're carrying around with you and which parser you choose and how complex your structure is, you could easily paint yourself into a very dark corner with XSLT. Yes, academically it's a brilliantly clean way of doing a presentation layer, but in the real world there can be catastrophic performance issues if you don't pay particular attention to this. I've seen a system consume over 30% of transaction time just in XSLT. Not pretty if you're trying to ramp up 4x the user base without buying additional boxes.
6. Can you buy your way out of a scalability jam? Absolutely. I've watched it happen more times than

I'd like to admit. Moore's Law (as you already mentioned) is still valid today. Have some extra cash handy just in case.

7. Caching is a great tool to reduce the strain on the engine (increasing speed and throughput is a handy side-effect). It comes at a cost though in terms of memory footprint and complexity in invalidating the cache when it's stale. My decision would be to start completely clean and slowly add caching only where you decide it's useful to you. Too many times the complexities are underestimated and what started out as a way to fix performance problems turns out to cause functional problems. Also, back to the data usage comment. If you're creating gigabytes worth of objects every minute, it doesn't matter if you cache or not. You'll quickly max out your memory footprint and garbage collection will ruin your day. So I guess the takeaway is to make sure you understand exactly what's going on inside your virtual machine (object creation, destruction, GCs, etc.) so that you can make the best possible decisions.

Sorry for the verbosity. Just got rolling and forgot to look up. Hope some of this touches on the spirit of your inquiry and isn't too rudimentary a conversation.

Share Improve this answer Follow

answered Sep 16, 2008 at 21:21

Ed Lucas

654 ● 7 ● 8

4

Well there's this blog called [High Scalibility]() that contains a lot of information on this topic. Some useful stuff.

Share Improve this answer Follow

answered Sep 3, 2008 at 9:38

Malik Daud Ahmad Khokhar
13.7k ● 24 ● 80 ● 81

3

Often the most effective way to do this is by a well thought through design where scaling is a part of it.

Decide what scaling actually means for your project. Is infinite amount of users, is it being able to handle a slashdotting on a website is it development-cycles?

Use this to focus your development efforts

Share Improve this answer Follow

answered Sep 3, 2008 at 9:38

svrist
7,110 ● 7 ● 46 ● 67

2

Jeff and Joel discuss scaling in the [Stack Overflow Podcast #19]().

Share Improve this answer Follow

edited Jan 18, 2021 at 12:38

Community Bot
1 ● 1

answered Sep 3, 2008 at 9:33

GateKiller

75.8k ● 75 ● 175 ● 204

1

FWIW, most systems will scale most effectively by ignoring this until it's a problem- Moore's law is still holding, and unless your traffic is growing faster than Moore's law does, it's usually cheaper to just buy a bigger box (at $2 or $3K a pop) than to pay developers.

That said, the most important place to focus is your data tier; that is the hardest part of your application to scale out, as it usually needs to be authoritative, and clustered commercial databases are very expensive- the open source variations are usually very tricky to get right.

If you think there is a high likelihood that your application will need to scale, it may be intelligent to look into systems like memcached or map reduce relatively early in your development.

Share Improve this answer Follow

answered Sep 3, 2008 at 9:57

Tim Howland

7,970 ● 4 ● 29 ● 46

Sorry, I strongly disagree with this. Often, by the time scalability becomes a problem, the only real cure is to rewrite the system almost from scratch, because the needed changes are often pervasive and deep. – RickNZ Dec 29, 2009 at 8:34

"Premature Optimization is the root of all evil". – Tim Howland
Dec 29, 2009 at 18:25

1

One good idea is to determine how much work each additional task creates. This can depend on how the algorithm is structured.

For example, imagine you have some virtual cars in a city. At any moment, you want each car to have a map showing where all the cars are.

One way to approach this would be:

```
for each car {
   determine my position;
   for each car {
     add my position to this car's map;
   }
}
```

This seems straightforward: look at the first car's position, add it to the map of every other car. Then look at the second car's position, add it to the map of every other car. Etc.

But there is a scalability problem. When there are 2 cars, this strategy takes 4 "add my position" steps; when there are 3 cars, it takes 9 steps. **For each "position update," you have to cycle through the whole list of cars - and every car needs its position updated.**

Ignoring how many other things must be done to each car (for example, it may take a fixed number of steps to calculate the position of an individual car), **for N cars, it takes N^2 "visits to cars" to run this algorithm**. This is no problem when you've got 5 cars and 25 steps. But as you add cars, you will see the system bog down. 100 cars will take 10,000 steps, and 101 cars will take 10,201 steps!

A better approach would be to undo the nesting of the for loops.

```
for each car {
  add my position to a list;
}
for each car {
  give me an updated copy of the master list;
}
```

With this strategy, the number of steps is a multiple of N, not of N^2. **So 100 cars will take 100 times the work of 1 car - NOT 10,000 times the work**.

This concept is sometimes expressed in "big O notation" - the number of steps needed are "big O of N" or "big O of N^2."

Note that this concept is only concerned with scalability - not optimizing the number of steps for each car. Here we don't care if it takes 5 steps or 50 steps per car - the main thing is that N cars take (X * N) steps, not (X * N^2).

Follow

edited Aug 3, 2011 at 13:05

answered Sep 3, 2008 at 9:56

Nathan Long

126k ● 103 ● 350 ● 463