

Art of Scalability

How we can design High Scalable Web Application

Haytham ElFadeel email: [HFadeel at HFadeel.com](mailto:HFadeel@HFadeel.com)

Art of scalability

Scalability principles:

In telecommunications and software engineering, scalability is a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged. For example, it can refer to the capability of a system to increase total throughput under an increased load when resources (typically hardware) are added. An analogous meaning is implied when the word is used in a commercial context, where scalability of a company implies that the underlying business model offers the potential for economic growth within the company. [Wikipedia]

If it's hard to understand the Wikipedia Scalability definition, so read this: At the simplest level, Scalability is about doing more of something. Scaling a web application is about allowing more people to use your application is same time. Scaling it's the ability to handle larger number of concurrent users.

Today we have two choices when it comes to scaling:

- Vertical Scalability: Vertical Scalability refers to adding resource within the same logical unit to increase the capacity. For example: Add more CPUs to your application server, or expanding the storage or the memory. The Vertical Scalability focus on the hardware only.
- Horizontal Scalability: Horizontal Scalability refer to add multiple logical units of resources and make them together work as a single unit. You can think about it like: Clustering, Distributed, and Load-Balancing. The Horizontal Scalability focus on the software and the hardware.

Now the big question: How we can architect and design our software to be Scalable? The architects strive to achieve linear scalability, which refer to the ability to maintain a consistent throughput rate proportionally as resources are added to the system. However, adding resources incurs additional overhead, making it difficult to achieve. [Royans K Tharakan](#) refers to this as a "scalability factor", and uses it to enumerate types of scalability:

- If the scalability factor stays constant as you scale. This is called linear scalability.
- But chances are that some components may not scale as well as others. A scalability factor below 1.0 is called sub-linear scalability.
- Though rare, it's possible to get better performance (scalability factor) just by adding more components (I/O across multiple disk spindles in a RAID gets better with more spindles). This is called supra-linear scalability.
- If the application is not designed for scalability, it's possible that things can actually get worse as it scales. This is called negative scalability.

As with a lot of things in software development, there is no one size fits all prescriptive approach that will solve your scalability problems. If you need scalability, urgently, going to vertical scaling is

probably will to be the easiest, but be sure that Vertical scaling, gets more and more expensive as you grow, and While infinite horizontal linear scalability is difficult to achieve, infinite vertical scalability is impossible.

On the other hand Horizontal scalability doesn't require you to buy more and more expensive hardware. It's meant to be scaled using commodity storage and server solutions. But Horizontal scalability isn't cheap either. The application has to be built ground up to run on multiple servers as a single application.

Scalability guidelines:

1. Decrease processing time:

In any optimization book, or course you will find that code optimization is about decrease processing time using better algorithms, code, design, or better strategy. Also decrease - request - processing is one of main Scalability guidelines. To increase the amount of work that an application does is to decrease the time taken for individual work units to complete. For example, decreasing the amount of time required to process a user request means that you are able to handle more user requests in the same amount of time. But how, here are some examples:

- **Caching:** if the data and the code can't be collocated, cache the data to reduce the overhead of fetching it over and over again.
- **Parallelization:** decrease the time taken to complete a unit of work by decomposing the problem and parallelizing the individual steps.
- **Remoting:** reduce the amount of time spent accessing remote services by, for example, making the interfaces more coarse-grained. It's also worth remembering that remote vs local is an explicit design decision not a switch and to consider the first law of distributed computing - do not distribute your objects.
- **Collocation:** reduce any overheads associated with fetching data required for a piece of work, by collocating the data and the code.
- **Pooling:** reduce the overhead associated with using expensive resources by pooling them.

We always try to introduce abstractions and layers that can make the previous techniques easy when are required. But between the highest abstractions and lowest abstraction usually the software developers fall. Yes, these concepts are great tools for decoupling software components, but they have a tendency to increase complexity and impact performance, particularly if you're converting between data representations at each layer. Therefore, the other way in which processing time can be minimized is to ensure that the abstractions aren't too abstract and that there's not too much layering. In addition, it's worth understanding the cost of runtime services that we take for granted

because, unless they have a specific service level agreement, it's possible that these could end up being the bottlenecks in our applications.

2- Partition:

Decreasing the processing time associated with other related concepts, such as Code Optimization as I said before, Partition, Asynchronous.

If you partition your work by Functional decomposition/Partition by function (When related pieces of functionality belong together, while unrelated pieces of functionality belong apart) Further, the more decoupled that unrelated functionality can be, the more flexibility you will have to scale them independently of one another. For example single database server sitting behind many web servers. When that starts to become the bottleneck, you have to change your approach and one way to do this is to adopt a partitioning strategy. Put simply, this involves breaking up that single piece of the architecture into smaller more manageable chunks. Partitioning that single element into smaller chunks allows you to scale them out and this is exactly the technique that large sites such as Amazon, Facebook use to ensure that their architectures scale. Partitioning is a good solution.

3- Asynchronously:

After you partition your application, the next key element to scaling is the aggressive use of asynchrony. If component X calls component Z synchronously, X and Z are tightly coupled, and that coupled system has a single scalability characteristic — to scale X, you must also scale Z. Equally problematic is its effect on availability. Going back to Logic 101, if X implies Z, then not-Z implies not-X. In other words, if X is down then Z is down. By contrast, if Z and X integrate asynchronously, whether through a queue, multicast messaging, a batch process, or some other means, each can be scaled independently of the other. Moreover, Z and X now have independent availability characteristics - A can continue to move forward even if B is down or distressed.

This principle can be applied up and down an infrastructure. Techniques like SEDA (Staged Event-Driven Architecture) can be used for asynchrony inside an individual component while retaining an easy-to-understand programming model. Between components, the principle is the same — avoid synchronous coupling as much as possible. More often than not, the two components have no business talking directly to one another in any event. At every level, decomposing the processing into stages or a phase, and connecting them up asynchronously, is critical to scaling.

4- Scalability is about concurrency:

As Researcher in Parallelism, I can say Scalability is about concurrency and the concurrency is about the parallelism :). Using the parallelism to decrease the processing time, especially if your work have asynchronously flow, is great. But remember the parallelism is not that easy. But there are some simple advices that can help to decrease the processing time using parallelism:

- If you do need to hold locks (e.g. local objects, database objects, etc), try to hold them for as little time as possible.
- Try to minimize contention of shared resources and try to take any contention off of the critical processing path (e.g. by scheduling work asynchronously).
- Any design for concurrency needs to be done up-front, so that it's well understood which resources can be shared safely and where potential scalability bottlenecks will be.

5- Requirements must be known

To build successful software system, you need to know what your goals are what you are aiming for. While the functional requirements are often well-known, it's the non-functional requirements that are usually absent or ignored.

If you do genuinely need to build a piece of software that is highly scalable, then you need to understand the next types of things:

- Target average and peak performance (Response time, latency, etc)
- Target average and peak load (Concurrent users, etc)
- Acceptable limits for performance and scalability

After you know and understand the requirements well, now you can start design and build the solution. The design, all performance and scalability decisions should be backed up by evidence, and this evidence should be gathered and reviewed from the start of the project and on a continuous basis thereafter. In other words; set measurable goals throughout the system, verify and measure the real performance and consider performance at all stages of the project.

Also you should consider the performance in every development step (Designing, Developing, and Testing). You can't simply say it's important to make the design the solution to be fixable and in the development we can optimize the code for the Scalability. If you do this you will pay at the end. You should think about the performance in every development step.

6- Avoid Distributed Transactions

We all know that distributed transactions have higher cost, actually this point make be back to the partition speak. You should know how to partition your application, and your data, because bad partition will make you need to perform distributed transaction.

7- Measure the right thing:

Back to [Performance anti-pattern post](#), you should measure the right things. For example you can't ask me 'How many user do you have in your Application?' to measure the scalability of my application. You should ask me the right question such as 'How many concurrent user do you have in your Application?'. Select and measure the right thing is very important, for more information please read 'Measuring and Comparing the Wrong Things' section into [Performance anti-pattern](#) post.

8- Understand the Semantic of the lower level libraries:

Abstraction is very good concept, but understanding the semantic of the lower level libraries is important to design and develop a scalable software system. If you have many application servers and integration between them, so you may ask yourself:

- Does your RPC infrastructure create TCP connection to all or some of your backend servers?
- What is the different between the APIs that exist into the lower level libraries?

9- Fact: Any Scalable System is a Distributed system:

As I said before: Partition your system into small pieces that can optimized or scale independently, this also mean: these small pieces can be distributed in different physical servers. So you can create a load balancing layer (layer that deliver the request to the most near and healthy server). In distributed system you need to take care of some points:

- **Fault Tolerance:**
There are important role in distributed systems, if you think that everything is reliable so you are wrong, you should take care of fault tolerance, and how you should handle any error are occur. For example: any sick server shouldn't accept new requests, and error in request processing should handle and isolated.
- **Monitor the servers healthy and load:**
Monitor the servers healthy and load is fundamental in any load balancing system, because this will help to deliver the request to the most near and not busy server, in other hand archive this data for offline processing and view will give you great ability to know what is the busy time, what about the servers healthy, etc. this information will help you to know when I should add more hardware or improve my software for more scale, also this information can tell you what is most slower piece in my system.

10- Different kind of data needs different kind of partition:

When I speak about partition, I was speaking about code and sub system partition, but data partition is also important. For example:

If you have photos search engine that provide thumbnails will the results. Partition the thumbnails data around many servers is so important, why?

The problem with thumbnails images is there are a lot of them and they are small around 10KB, and in every search request will make the browser send a lot of requests to view them. So partition the thumbnails images in many servers is important.

What I mean by the example, is partition the data is also important maybe more that software, and you should understand the semantic of the problem/Challenger that you have to be able to choice good solution.

References:

- Several videos about Scalability from Google video.
- O'Reilly - Building Scalable Web Sites.
- Wikipedia
- InfoQ.com
- My experience at [Kngine](#).