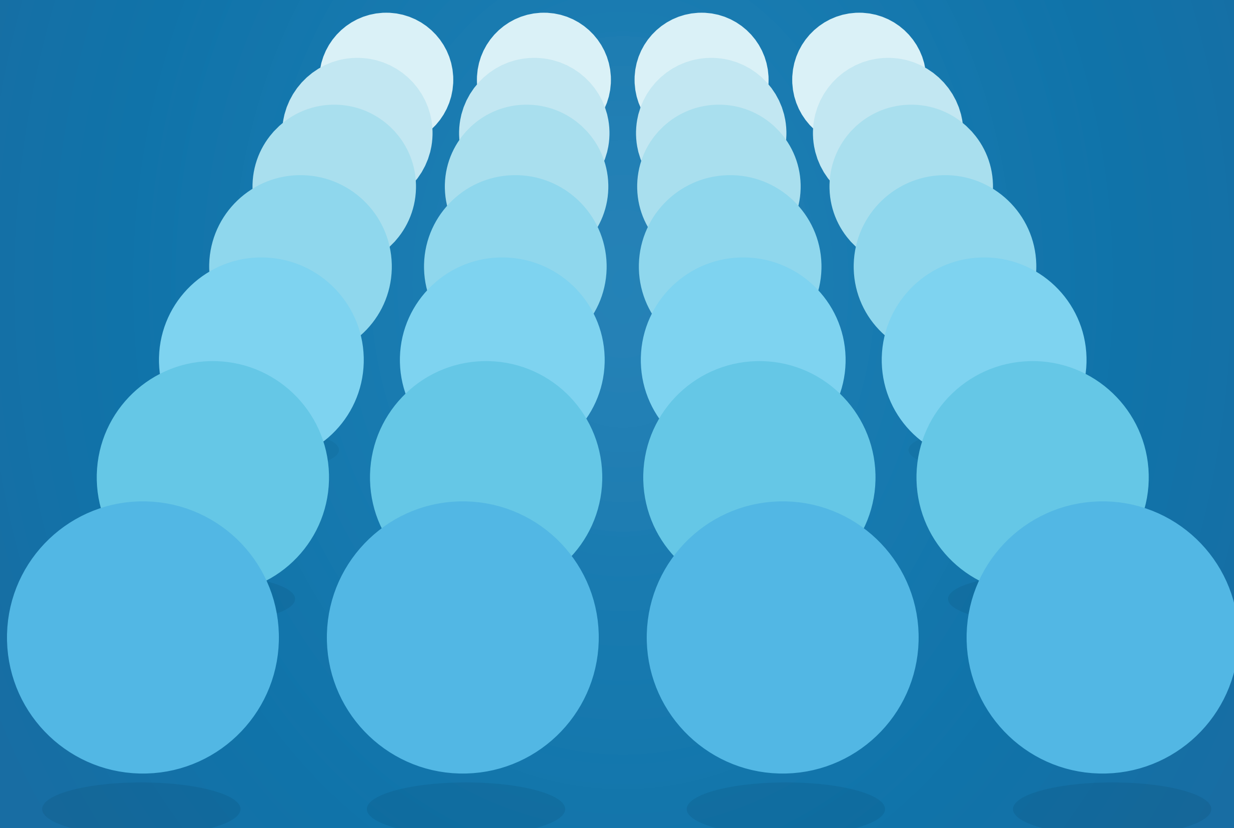
Everything You Need To Know About

Scalability



October 30, 2015

VividCortex

Meet the Author

Baron Schwartz

Baron is a performance and scalability expert who participates in various database, opensource, and distributed systems communities. He has helped build and scale many large, high-traffic services for Fortune 1000 clients. He has written several books, including O'Reilly's best-selling High Performance MySQL. Baron has a CS degree from the University of Virginia.

Table of Contents

Introduction

Making systems big, fast, and efficient is one of the most interesting and satisfying things I've done. It's a great feeling when you fix a bottleneck, and get dramatically improved performance at scale. You suddenly realize how wasteful your systems were before the improvement.

I've participated in lots of projects that have produced those kinds of outcomes. It's no coincidence that the best results came from the projects where the most disciplined analysis was performed up front. Like performance optimization, scalability optimization can be a real mystery unless you have an accurate model of how the world works, good measurements of how your systems are performing, and the ability to identify the problem and its likely fix with high certainty.

What's even better than fixing problems with scalability is the ability to design systems (and organizations) that scale in the first place. This is worth its weight in gold.

Scalability is quite a scientific topic, but it doesn't need to be mysterious. It's true that queueing theory is intimately involved, and queueing is complicated and unintuitive. But what's really cool is that scalability, correctly understood, is quite straightforward despite the complexity of what's going on behind the scenes.

I wrote this book to help you understand the simple, but profoundly powerful, truths about scalability. I also wanted to help you understand the connections between scalability and other disciplines, such as performance optimization or the study of queueing. My hope is that this book is as transformational and rewarding for you as the process of learning these concepts has been for me.

What is Scalability?

Scalability is ambiguous for many people—a vague term often tossed about in conference presentations, for example. It's often used in ways confusingly similar to performance, efficiency, capacity, availability, and many other terms related to making things big and fast.

Wikipedia's definition of scalability, borrowed from a 2000 paper by André B. Bondi, is "the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth." This isn't wrong, but it's still a bit informal, and this book needs a more formal definition.

Dr. Neil J. Gunther provides one such definition: scalability is a *function*. I read his books and heard him speak, but it was still a year or so before I understood. *Scalability can be defined as a mathematical function*, a relationship between independent and dependent variables (input and output). This is the type of formal definition we need to model and analyze scalability.

The most important part of understanding such a scalability model is choosing the correct variables to describe the way systems really operate. Bondi's definition provides a good clue: *work* is the driving factor of scalability. Useful ways to think about work include, to mention a few,

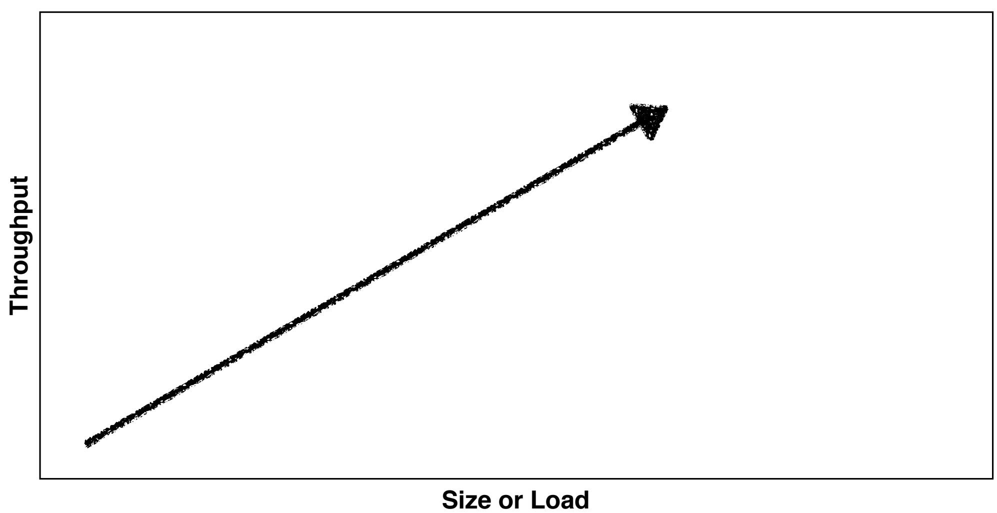
- Units of work (requests).
- The rate of requests over time (arrival rate).
- The number of units of work in a system at a time (concurrency).
- The number of customers or users sending requests.

Each of these can play sensible roles in the scalability function, depending on how you view it. For example, it's quite common to configure the number of threads a benchmark runs to send requests to a database. The benchmark usually sends requests as fast as possible, assuming zero think time, so the arrival rate is related to, but not strictly controlled by,[1] the benchmark configuration.

You could say that the amount of work requested is the input to the benchmark's scalability function, and the completion rate is the output.

In another scenario, you might vary the number of CPUs for the system under test (SUT) while holding constant the load per CPU, or if it's a clustered database, vary the cluster size and hold constant the load per node. In this case, the independent variable is the system size and the dependent variable is the completion rate.

In most cases I've analyzed, either size or load are the sensible independent variables for the scalability function. So for the purposes of this book, we'll consider scalability to be *a function of size or load*. The dependent variable will be the rate at which the system can complete work, or *throughput*. The hope is that the system should complete more work as size or load grow, so it should be an increasing function.



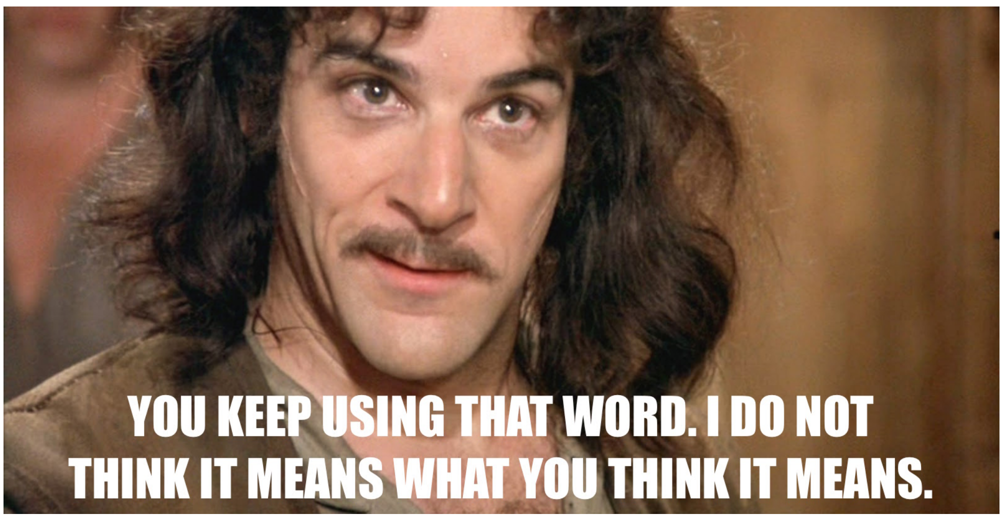
For those who are like me and need extra emphasis, I'll repeat that this is

1 Because it is determined by how quickly the database finishes each request.

a mathematical function, with size or load on the X axis, and throughput on the Y axis. We'll make this more precise later.

Linear Scalability: The Holy Grail

In my experience, never was a marketechture slide deck created that mentions scalability without also including the word "linear." But as you might expect, "linear scalability" is not what people would have you believe.



Hand-waving claims of linear scaling usually coincide with vague definitions of scalability, and people who know a lot about scalability rarely say the word "linear." Here are a few of the misdefinitions of linear scalability I've heard:

- A web architect at a conference said, "I designed our system to be shared-nothing so it would be linearly scalable." He meant there was no single resource or system imposing a hard upper limit on how many servers could be added to the system. But he didn't really know whether his system actually scaled linearly.
- A technical evangelist giving a presentation about a clustered

database said, “adding a node to the cluster adds a predictable amount of capacity.” Predictable isn’t the same as linear.

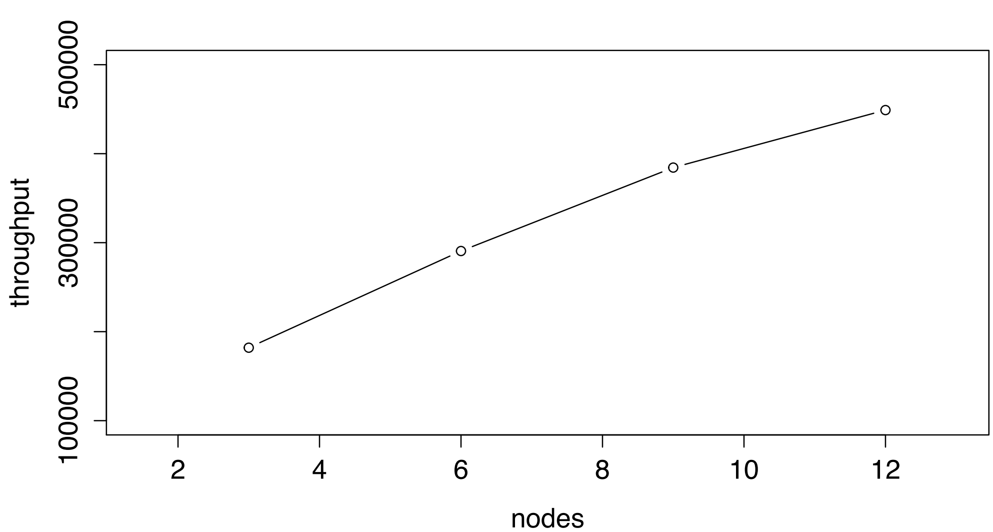
- A sales presentation for another clustered database said the database “scales linearly, with a linearity factor of 97%,” meaning that each additional node increases the system’s capacity by 0.97 times the amount the previous node added. That’s a curve, not a line. (Later you’ll learn how to instantly determine the asymptotic upper bound on such a system’s total capacity.)

This may seem like a pointless rant, but it’s actually important if you want to be able to design and improve highly scalable systems.

Spotting bogus linearity claims is fun. Here are some ways “benchmarketing” makes systems appear linear:

- Show graphs without numbers, so readers can’t do the math.
- Show graphs with nonlinear axes.
- Begin the axes, especially the Y axis, at a nonzero value.

Here is a real example, redacted to protect the not-so-innocent, that employs some of these tricks.



Looks pretty linear, doesn't it? Yet if you do the math, it's nowhere near linear. It's an optical illusion, because the X axis begins around 1.45 instead of zero, and the Y axis starts at 100000, so you can't tell that the chart isn't going to intersect the origin if you extend it downwards.

The real test of linearity is whether the transactions per second per node remains constant as the node count increases. The chart's original source mentioned that throughput increased from "182k transactions per second for 3 nodes to 449k for 12 nodes." The math is easy: the system achieves 60700 transactions per second per node at 3 nodes, but only 37400 at 12 nodes, which represents a *39% drop in throughput* versus linear scalability. If it actually scaled linearly, it would achieve 728k transactions per second at 12 nodes.

Linear means linear, folks! And seemingly small amounts of nonlinearity really matter, as you'll see later, because small sublinear effects grow very quickly at larger scale.[1]

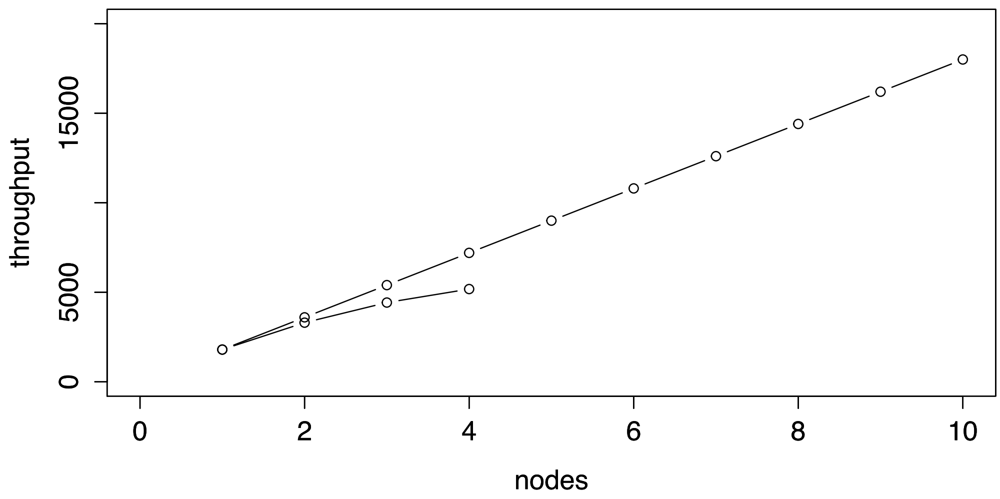
Why Systems Scale Sublinearly

Linear scalability is the ideal, yet despite the claims, systems that actually scale linearly are rare. It's very useful to understand the reasons for this, because a correct understanding of scalability, and the reasons and sources of sublinear scaling, is the key to building more scalable systems. That's why it's really important to be a linearity skeptic. It's not just being pedantic.

The best way to think about linearity is as a ratio of the system's performance at a size of 1. Neil Gunther calls this the *efficiency*. If a system produces 1800 transactions per second with 1 node, then ideally 4 nodes produce 7200 transactions per second. That would be 100% efficient. If the system loses a bit of efficiency with each node and 4

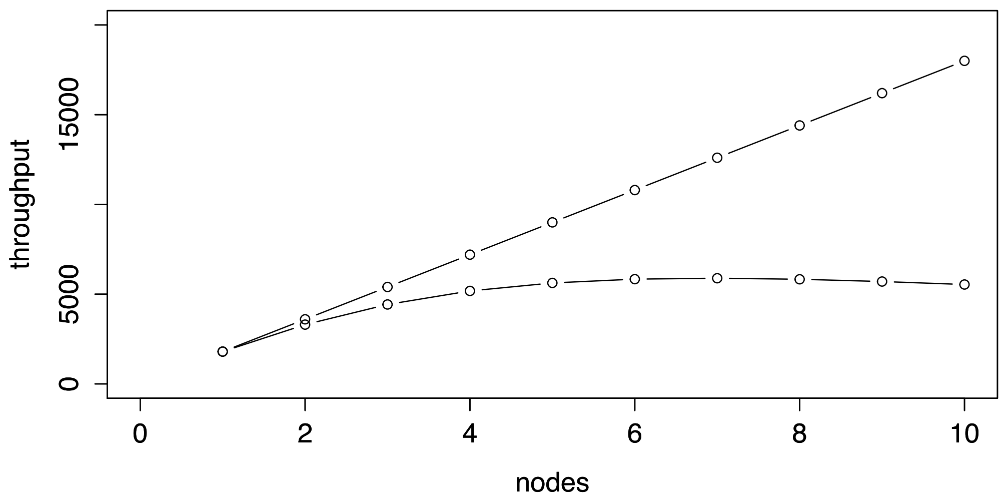
1 In fact, they grow—wait for it—nonlinearly!

nodes produce, say, 5180 TPS, then the 4-node system is only 72% efficient:



If you do this math, you'll often be surprised at how large the efficiency loss is. Graphs can be deceptive, but the numbers are quite clear.[1]

In the real world there's almost always some efficiency loss, and if you can figure out why, you may be able to fix it. In fact, you've probably noticed that real systems tend not only to fall behind linear scalability a bit, but actually exhibit *retrograde* scalability at some point:



[1] Drawing a linear scaling line on the graph helps, too. Without that line, the eye tends to see the graph as more linear than it really is, and the efficiency loss becomes less obvious.

This is quite common in the real world—you scale things up and at some point your system starts going backwards and *losing* performance, instead of just gaining more and more slowly. In the MySQL 5.0 days, for example, it was common to see people upgrading from 4-core servers to 8-core servers and losing performance.

Why does this happen? Why don't systems scale linearly, and why do they sometimes show retrograde scalability?

According to Neil Gunther, there are two reasons: **contention** and **crosstalk**. Contention degrades scalability because parts of the work can't be parallelized and queue up, so speedup is limited. Crosstalk introduces a coherency penalty as workers (threads, CPUs, etc) communicate to share and synchronize mutable state. We'll explore these effects in the next section.

The Universal Scalability Law

Neil Gunther's Universal Scalability Law (USL) provides a formal definition of scalability,[1] and a conceptual framework for understanding, evaluating, comparing, and improving scalability. It does this by modeling the effects of linear speedup, contention delay, and coherency delay due to crosstalk.

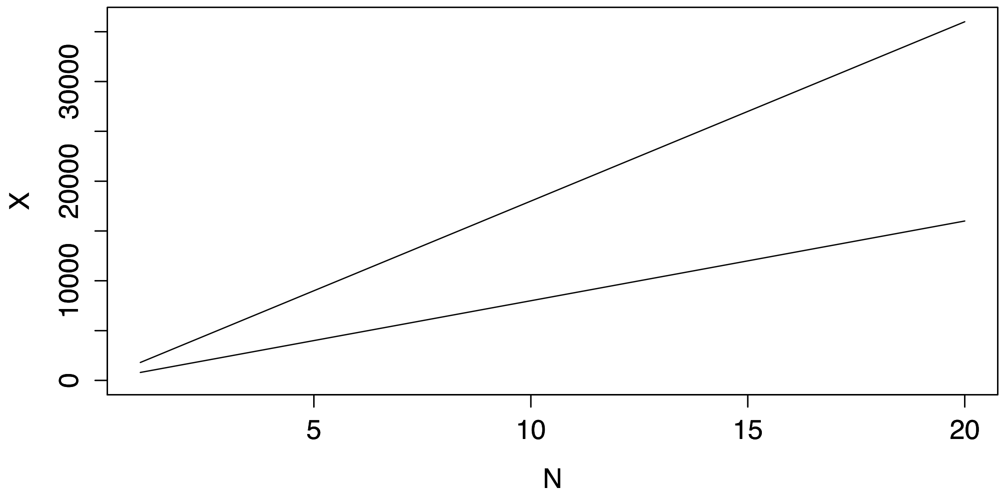
Let's see how this works, piece by piece. An ideal system of size 1 achieves some amount λ of throughput X, in completed requests per second. Because the system is ideal, the throughput doubles at size N=2, and so on. This is perfect linear scaling:

$$X(N) = \frac{\lambda N}{1}$$

The λ parameter defines the slope of the line. I call it the *coefficient of performance*. It's how fast the system performs in the special case when

1 Neil Gunther originally called a slightly different form of the USL "superserial," and you may encounter this terminology, especially in older books and papers.

there's no contention or crosstalk penalty. Here are two ideal systems, with λ of 1800 and 800, respectively.



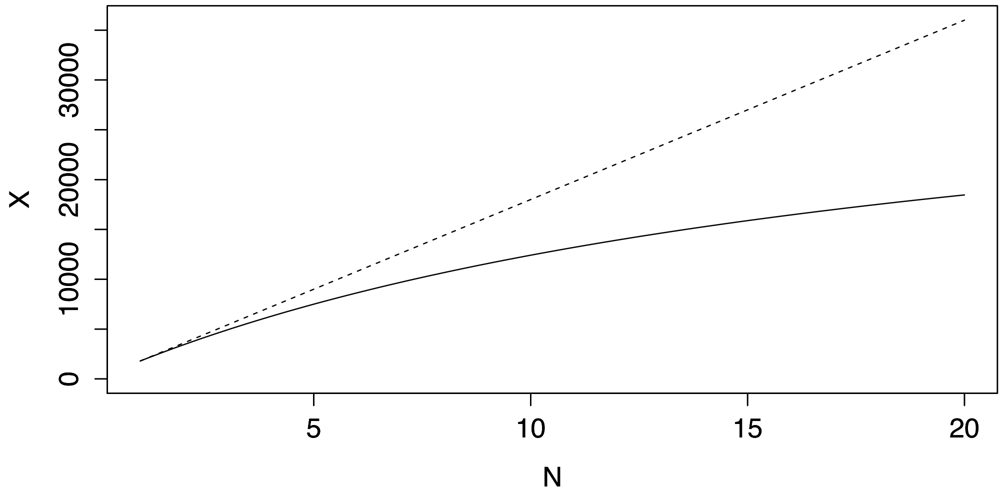
Note that every linearly scalable system is just as scalable as any other, regardless of the slope of the line. They have different performance but identical scalability characteristics: speedup is unlimited.

Contention appears in most systems[1] at some point, for example as a final stage of assembling the multiple outputs generated in parallel into a single final result. As parallelization increases, contention becomes the limiting factor. This is codified in Amdahl's Law, which states that the maximum speedup possible is the reciprocal of the serial fraction. We'll add a term to the denominator expressing the serial fraction of the work, multiplied by σ, the coefficient of contention, and now we have Amdahl's Law:

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1)}$$

A system with contention will asymptotically approach a ceiling on speedup. If σ is .05, for example, the speedup approaches 20. Let's see that graphically:

1 Including teams of people. This is a joke but it's also true in the queueing sense.



Remember the system I mentioned earlier, which a salesperson claimed to have 97% scalability with each additional node? That's 3% loss of scalability per node, so this system will never achieve a speedup factor of more than 33, no matter how many nodes it has.

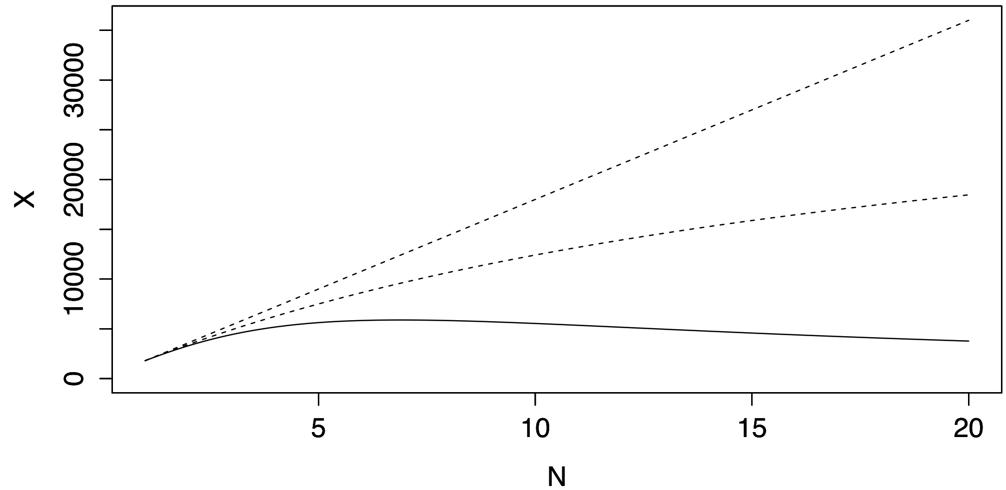
The last bit is the crosstalk penalty, also called the consistency or coherency penalty.[1] Crosstalk potentially happens between each pair of workers in the system (threads, CPUs, servers, etc). You probably remember that the number of edges in a fully connected graph is $n(n-1)$. We represent the amount of crosstalk with another term, multiplied by κ, the coefficient of crosstalk. This is the USL:

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)}$$

The crosstalk penalty grows fast. Because it's quadratic,[2] eventually it grows faster than the linear speedup of the ideal system we started with, no matter how small κ is. That's what makes retrograde scalability happen, as you can see in the following chart:

1 I call it crosstalk because in my opinion it's the best description of the pairwise communication that must occur to make distributed data or other shared resources consistent or coherent.

2 The cost of $n(n-1)$ is $\mathcal{O}(n^2)$. If you're not familiar with it, this blog post introduces Big-O notation.



That's the Universal Scalability Law in all its glory. This plot has the same parameters as the ones I showed before, where a system of size 4 produced only 72% of its ideal output. That system has 5% contention and 2% crosstalk, and now that I've plotted it out to size 20 you can see it's embarrassingly inefficient. In fact, we should have given up trying to scale this system after size 6 or so.

This shows visually how much harm a "small amount" of nonlinearity can do in the long run. Even very small amounts of these damaging coefficients will create this effect sooner or later (mostly sooner). This is why it's rare to find clustered systems that scale well beyond a couple dozen nodes or so. If you'd like to experiment with this interactively, I've made a graph of it at Desmos.

The USL's Relationship to Queueing Theory

The USL is closely related to queueing theory. Neil Gunther proved that it's equivalent to synchronous repairman queueing. If you're not familiar with queueing theory, I wrote an approachable introduction called Everything You Need To Know About Queueing Theory.

The two causes of sublinearity have important relationships to queueing theory. Contention, the first term we added to the denominator to obtain Amdahl's Law, expresses the penalty from queueing delay that occurs when there is competition for shared resources—the servers (in the queueing theory sense) that process work from queues.

The queue length is *nonlinear* with respect to utilization and therefore to offered load. Queueing theory is confusing and counterintuitive! As the queues lengthen, the queueing delay lengthens in direct proportion.

As you probably know, queueing theory treats service time—the amount of time it takes to complete a job after it leaves the queue and enters service—as independent of utilization or queueing. The job takes as long as needed to execute all the instructions, whether the server is busy or idle. The customer's total wait time at a busy server is longer only because of queueing delay.

Coherency penalty, which comes from crosstalk, actually expresses an *increase in service time* that is not due to queueing delay. As the system has to do more crosstalk to synchronize mutable shared state, the jobs take longer and longer. This is *not* due to queueing—the job is already out of the queue and in service.

These effects are clearly visible in the USL when it enters the region of retrograde scalability. An increase in service time is the only thing that can explain retrograde scalability. If the service time remained constant, throughput would be a flat line. Queue length and queue wait time cannot explain retrograde scalability; queueing can only cap throughput, not decrease it.

Measuring Scalability

To recap, at this point we've figured out the right dimensions for a formal model of scalability that seems to behave as we know real systems behave, and examined Neil Gunther's USL, which fits that framework well and gives us an equation for scalability. (Are you excited yet?)

Now what do we do with it?

Great question! It turns out we can do a lot of extremely useful things with the USL. Unlike a lot of models of system behavior, this one is actually practical to apply in the real world. That's the real genius of it, in fact. Not only is the equation uncomplicated, but the variables it describes are easy to get most of the time.

I use the USL mostly for modeling system scalability, by working backwards from observed system behavior and estimating the likely coefficients. To accomplish this, you need a set of measurements of the system's load or size (usually concurrency or node count) and the corresponding throughput. Then you *fit* the USL to this dataset, using nonlinear least squares regression. This is a statistical technique that finds the optimal coefficient values in order to calculate a best-fit line through the measurements. The result is values for λ, σ, and κ.

If you're reading about the USL in Neil Gunther's books, he takes a different approach. First, he doesn't use regression to determine λ, he assumes that you can measure it in a controlled way at $N = 1$. (I've often found that's not true for me.) Secondly, there are a couple of different forms of the USL—one for hardware scaling and one for software scaling—which are the same equation, but with different parameters. For simplicity I'm treating them as interchangeable. I will write a bit more about this later.

Examples of systems I've analyzed with the USL include:

- Black-box analysis of networked software simply by observing and correlating packet arrivals and departures, looking at the IP addresses, port numbers, and timestamps. From this I computed the concurrency by averaging the amount of time the system was busy servicing requests over periods of time. The throughput was straightforward to get by counting packet departures.
- MySQL database servers. Some of its `SHOW STATUS` counters are essentially equivalent to throughput and concurrency.
- Linux block devices (disks) by looking at `/proc/diskstats`, from which you can get both instantaneous and average concurrency over time deltas, as well as throughput (number of I/Os completed).
- Lots and lots—and lots—of benchmark results.

I've built a variety of tools to help clean, resample, and analyze the data before arriving at satisfactory results. Most of them were commandline, though these days I use R more than anything else. This is an important topic: you will get dirty data, and that will make your results less useful. You need to visualize both in scatterplot form as well as in time-series form and ensure you're working with a relatively consistent set of data. You can remove individual points or trim the time range you use, and you may need to experiment with averaging the data over time to get good results.

As for the R code, I'll give a little bit of a quickstart to show the soup-to-nuts approach. You'll save the data into a delimited file, with column headers `size` and `tput`. Then you'll load this into a variable in R and regress it against the USL.

Here's a complete sample, based on a benchmark that Vadim Tkachenko ran at Percona:

```
size tput
1 955.16
2 1878.91
3 2688.01
4 3548.68
5 4315.54
6 5130.43
7 5931.37
8 6531.08
9 7219.8
10 7867.61
11 8278.71
12 8646.7
13 9047.84
14 9426.55
15 9645.37
16 9897.24
17 10097.6
18 10240.5
19 10532.39
20 10798.52
21 11151.43
22 11518.63
23 11806
24 12089.37
25 12075.41
26 12177.29
27 12211.41
28 12158.93
29 12155.27
30 12118.04
31 12140.4
32 12074.39
```

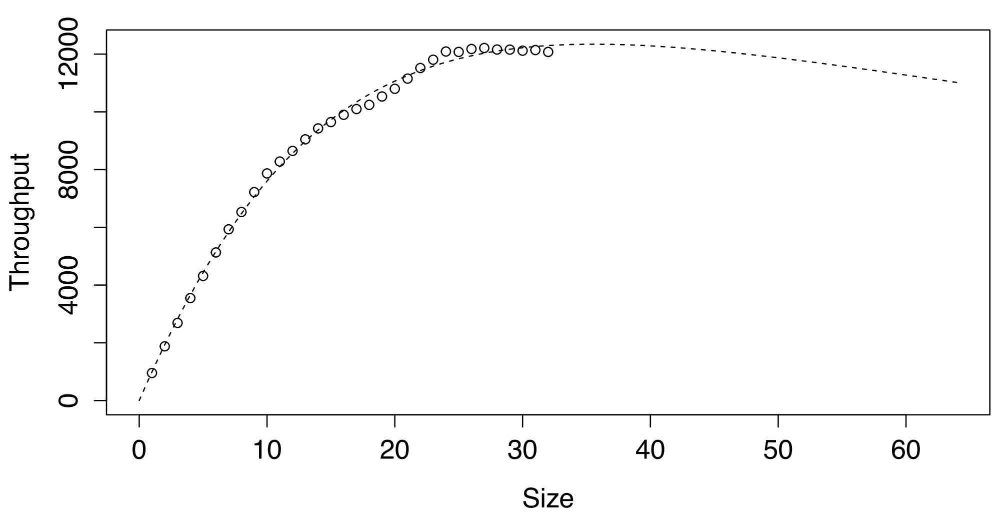
Save that data into a file, say, `benchmark.txt`. Then load it and run the following commands:

```
benchmark <- read.csv("/path/to/benchmark.txt", sep="")
usl <- nls(tput ~ lambda*size/(1 + sigma * (size-1) + kappa * size *
   (size-1)), benchmark, start=c(sigma=0.1, kappa=0.01, lambda=1000))
summary(usl)
sigma <- coef(usl)['sigma']
kappa <- coef(usl)['kappa']
lambda <- coef(usl)['lambda']
u=function(x){y=x*lambda/(1+sigma*(x-1)+kappa*x*(x-1))}
plot(u, 0, max(benchmark$size)*2, xlab="Size", ylab="Throughput", lty="dashed")
points(benchmark$size, benchmark$tput)
```

The results are as follows:

$$\begin{array}{ll} \lambda & 995.6486 \\ \sigma & 0.02671591 \\ \kappa & 0.0007690945 \end{array}$$

Note the extremely small value for κ which nonetheless degrades scalability before N becomes very large. Here's the resulting plot:



If you're an R user, that's probably all you need to get going. You really should do more diligence, such as checking the R^2 value of the fit. But instead of doing all this work manually (which you can certainly do if you want), I suggest trying the USL package from CRAN. It has many features built in, although it does have some limitations.

One final thing: if the κ coefficient has a nonzero value, the function has a maximum. You can find the size of the system at that maximum as follows:

$$N_{max} = \left\lfloor \sqrt{\frac{1-\sigma}{\kappa}} \right\rfloor$$

Of course, to find the maximum predicted throughput, you just plug N_{max} into the USL equation itself. Doing so with the coefficients in this example predicts the system's throughput will increase until $N = 35$, which in this case means 35 threads, and the peak throughput will be 12341 queries per second. It also found λ, the throughput at $N = 1$, to be 995 QPS, which is close to the actual value of 955.

It's always interesting to use the USL on a subset of the performance data, such as the first third or so, to see how well it predicts the higher N values. This can be quite educational.

Note that you should have at least half a dozen or so data points in order to get good results in most circumstances. In practice I usually try to capture at least a dozen for benchmarks, and more—often thousands—when analyzing systems that aren't in a controlled laboratory setting.

Relating Scalability and Performance

Throughput and latency are two common measures of performance. Most benchmarks measure overall system throughput, and claims of performance are almost always in throughput terms: "a million transactions per second," and so on. Benchmarks usually define performance as how much work the system can do.

On the other hand, users care mostly about the performance of individual requests. A user's opinion about your website's performance is based entirely on how quickly pages load and render. From this viewpoint, as Cary Millsap says, *performance is response time*.

Which view is right? Both. System performance is measured in throughput, and request performance is measured in latency.

We've seen that the USL can model and forecast how system size affects throughput. Can it also model latency? Yes, it can when the independent variable is concurrency, because of a relationship called Little's Law:

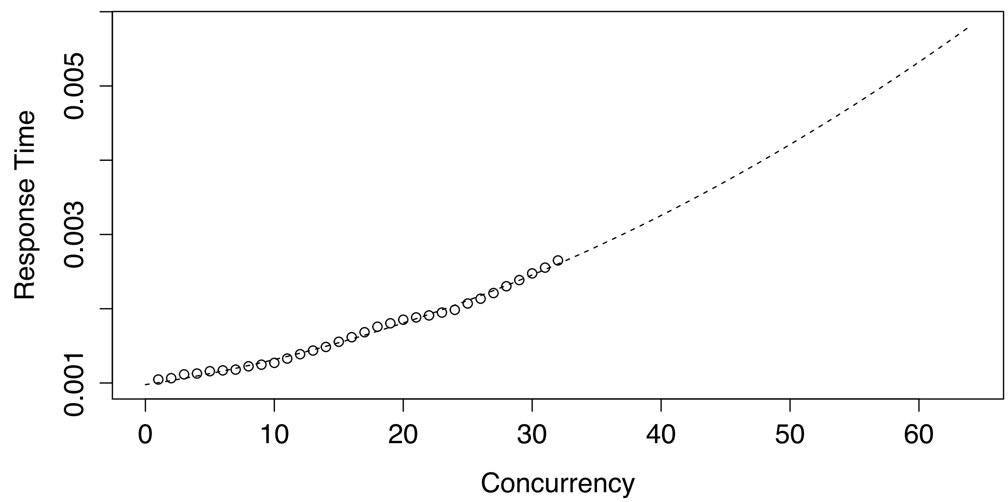
$$N = XR$$

Little's Law says that the mean number of requests resident in a system is equal to the throughput times the mean response time. This relationship is valid for stable systems, in which all requests eventually complete.

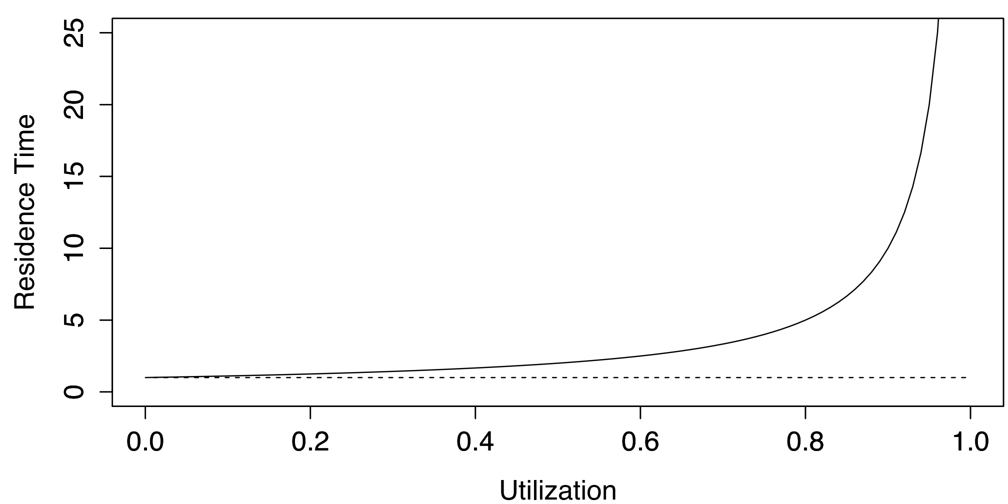
It's straightforward to use Little's Law together with the USL and solve for response time as a function of concurrency. The result is a quadratic function:

$$R(N) = \frac{1 + \sigma(N-1) + \kappa N(N-1)}{\lambda}$$

According to the model, response time is related to the square of concurrency. Here's the same data we saw previously, now inverted and used to predict response time.



Be careful not to confuse this chart with another famous "hockey stick" chart, that of response time versus utilization, which is familiar from queueing theory:

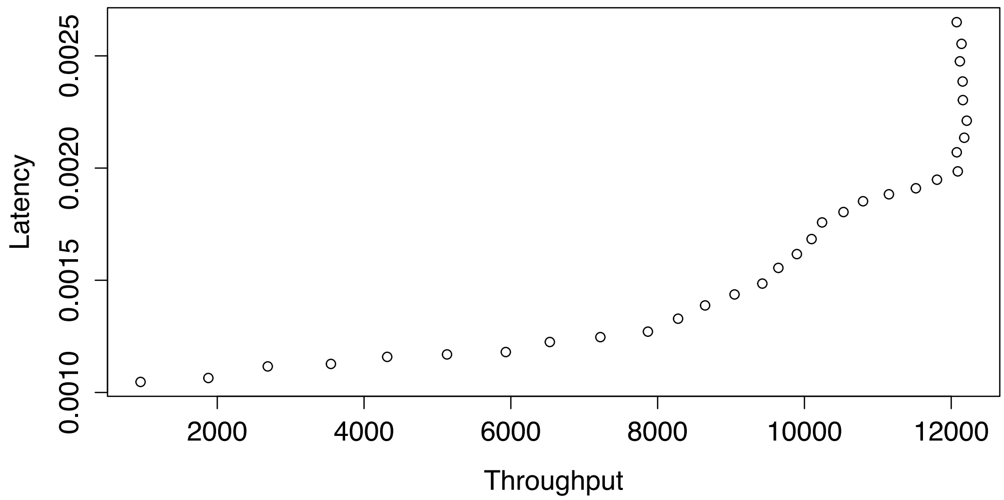


The difference is that one chart uses utilization as the independent variable, which ranges only from 0 to 1, whereas the other uses concurrency, which has no fixed upper limit.

The inverted USL predicts mean response time. If you wanted to, you could use queueing theory to assess this in a more nuanced way, such as the probability that any given request needed to wait more than a set amount of time.

XXXXX

It may appear to be true as long as you don't pass the point of maximum throughput and enter the territory of retrograde scalability, but it's easy to see such a plot is not actually a function of throughput after that happens:



You can see a hint in the chart that a single throughput value may produce multiple values for latency, just as a single value of throughput may be associated with multiple values of concurrency in the USL if there's a nonzero coherency penalty.

And the point is, that *does* happen. We all know that this is what happens in the real world. Anyone who's benchmarked, modeled, and analyzed system behavior has seen retrograde scaling.

If there's no retrograde scaling—that is, if κ = 0—then this model is actually valid. After eliminating the coherency term, substituting in Little's Law, and solving for latency as a function of throughput, the result is an exponential function:

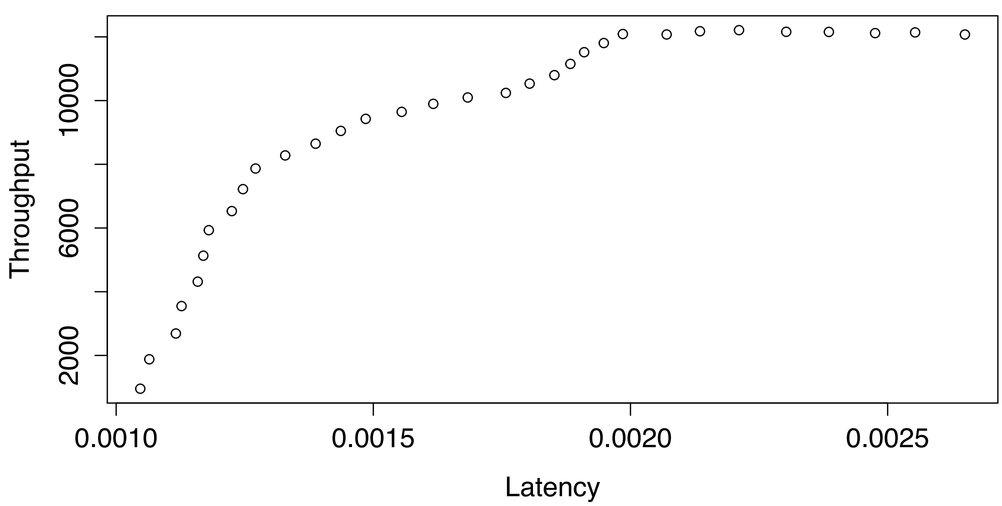
$$R(X) = \frac{\sigma - 1}{\sigma X - \lambda}$$

What this says is that systems that scale according to Amdahl's Law have exponentially increasing response time as throughput increases. Response time goes to infinity as throughput approaches λ/σ:

cisco-x-v-r-amdahl.pdf

The reason for this is that once you enter retrograde scalability, throughput is no longer under your control (e.g. you've saturated resources, service times are increasing quadratically, and you cannot simply pick an arrival rate for requests and get the SUT to perform them for you). In general, even in clean-room conditions, it is nearly impossible to predictably drive a desired throughput for a system, unless you maintain a very low utilization so requests don't queue. Throughput would only make sense as an independent variable, in the general case, if it were under your direct control. Therefore, latency as a function of throughput is not the way I would choose to plot the data.[1]

In fact, I think exactly the opposite is the correct relationship. Pick a latency, and you can find the corresponding throughput at which that latency will occur, just as I showed in the capacity planning thought exercise. Once again using the Percona benchmark data:



I'm leaving this as a reminder to myself to solve the USL equation for

1 No, not even if Neil Gunther does.

throughput as a function of latency someday, using Little's Law. If you do it first, please contact me and let me know.

XXX

$$X(R) = \frac{\sqrt{\sigma^2 + \kappa^2 + 2\kappa(2\lambda R + \sigma - 2)} - \kappa + \sigma}{2\kappa R}$$

Capacity Planning with the USL

"How much load can this system sustain?" is a common question in capacity planning. The practical purpose is usually something like the following:

- How soon will the system begin to perform badly as load increases?
- How many servers will I need for the expected holiday load?
- Is this system close to a point of failure?
- Are we overprovisioned? By how much?

Capacity planning is often a difficult problem because it's hard to tell what a system's true capacity is. The USL can help you estimate this.

Conventional ways to determine system capacity are often difficult, expensive, and don't give results you can really believe in. For example, you can set up load tests, but it takes a lot of work and time, and the results are suspect because the workload is always artificial in some way.

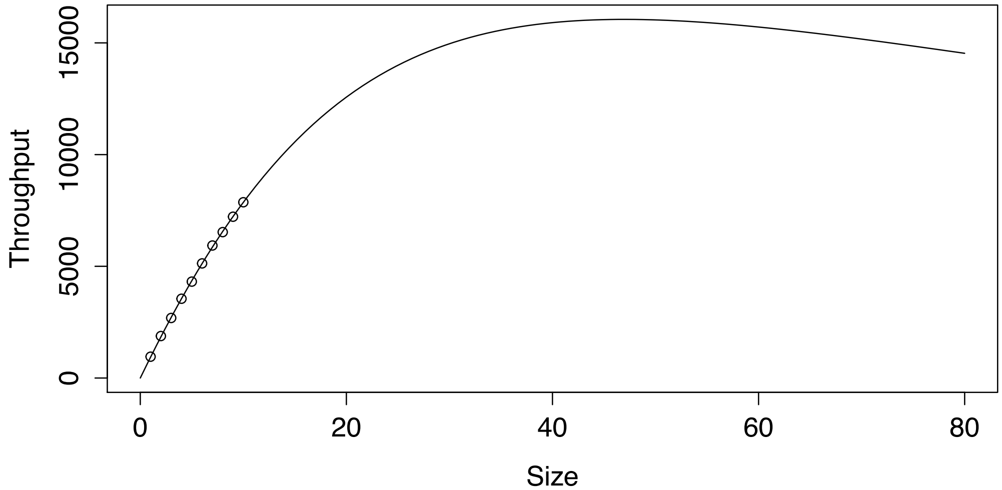
You can also run benchmarks, but most benchmarks are pretty useless for predicting a system's usable capacity. In addition to being an artificial workload, they push a system to its maximum throughput and beyond. As I mentioned, it's rare for benchmarks to be run by people who understand the importance of latency. But when I do see benchmarks that measure latency percentiles, the systems almost always perform very badly at their peak throughput.[1]

Another way I've tried to predict system capacity in the past is with queueing theory, using the Erlang C formula to predict response time at a given utilization. Unfortunately, this requires that you know service times, which are often impossible to obtain. You can measure total response time, but that includes waiting time in the queue, so it's not the same thing as the service time. The utilization is also often deceptive, because the real utilization of the resources you're trying to model can be difficult to measure correctly too. Most people I know consider the Erlang approach to be difficult to apply.

If load tests, benchmarks, and queueing theory are difficult to use, can the USL help? Yes, it can. Because the USL is a *model*, it can help you predict how a system will perform under load beyond what you can observe. The USL's point of maximum predicts the system's maximum throughput, so it's a way to assess a system's capacity. It can help you get a better idea of how close you are to the system's maximum capacity.

Here's an example. Imagine that we had measured the first 10 data points in the Percona benchmark, in a live production environment serving real users, not a lab. Here's the result of fitting the USL to the data:

1 This is a problem with the way benchmarks are usually designed, in my opinion. I'd really prefer for the benchmark system to be intelligent enough to back off and reduce pressure on the system under test if it violates a service level objective (SLO). Ideally, this is defined as a quantile, such as 99th percentile latency less than 10ms. A smart benchmark would throttle load, eventually finding a longterm stable arrival rate at which the SUT can consistently perform well.



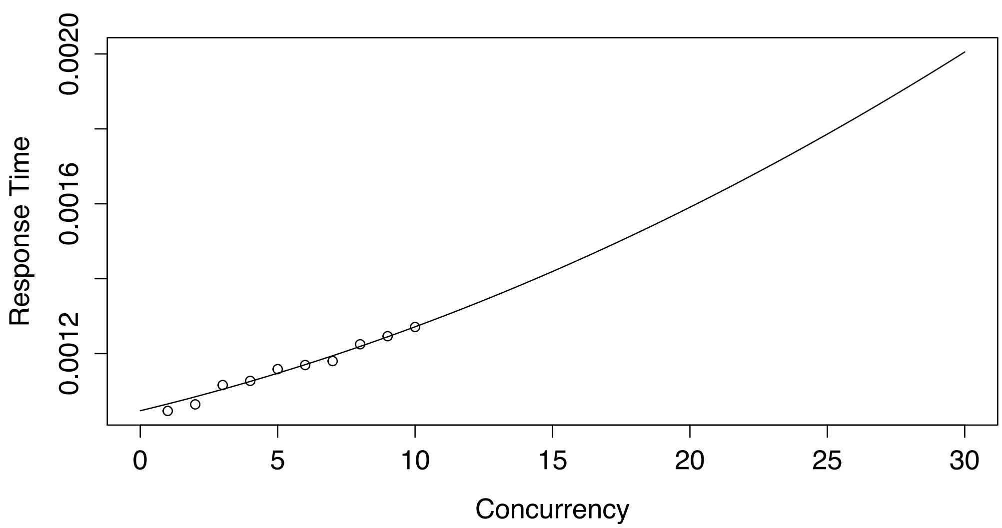
Using the formula $N_{max} = \sqrt{(1-\sigma)/\kappa}$, the USL predicts a max of 16,049 queries per second at a concurrency of 46 threads.

I have a rule of thumb for using the USL to project out into the unknown. I've seen so many systems that appear to be scaling beautifully, cleanly fitting the USL—just as this one does—and then they hit rough waters, that I don't trust anything farther out than twice the measured throughput or twice the measured concurrency, whichever comes first. And that's if I'm not seeing telltale signs of leveling off or retrograde throughput. If I see those signs, I lower my expectations accordingly. I will also include other information such as CPU utilization to guide my estimates, if I have it, but in the absence of more data this is a good way to keep expectations capped.

Back to the model: I have measurements only to $N = 10$, where observed throughput is 7,867, so I'm going to compute the predicted throughput at $N = 20$. The result is a forecasted throughput of 12,572. This is less than twice my maximum observed throughput, so I'll allow it. In my experience, it's an optimistic but not unrealistic guess that I won't get more than about 12,500 queries per second from this server. (As you may remember, this system topped out at 12,211 QPS.)

The outcome is that my system appears to be operating at about half of its maximum capacity. However, as discussed previously, maximum throughput isn't maximum *usable* capacity. Again, when the system is at its maximum throughput, response time is probably terrible, and will be extremely inconsistent. That's why it's more important to focus on the system's maximum throughput within the constraints of a service level objective.

As a first step towards this, I can use average latency to help understand the potential QoS end-users will get from this server. Again using the rearranged form of the USL to forecast response times, I obtain the following:



Using the estimated coefficients and the formula for response time, $R(N) = (1 + \sigma(N-1) + \kappa N(N-1))/\lambda$, I can predict a mean response time of 0.00159 seconds at 20 threads. Let's imagine that this is unacceptable; I need mean response times to be 15ms or less. Solving the response time equation as a function of latency lets me use it to compute the maximum usable concurrency. The resulting equation has two roots; I'm only interested in the positive one:

$$N(R) = \frac{\kappa - \sigma + \sqrt{\sigma^2 + \kappa^2 + 2\kappa(2\lambda R + \sigma - 2)}}{2\kappa}$$

Plugging in an R target of 0.0015 yields N = 17, so if I want to avoid violating my SLO I can't drive my server higher than approximately 11,450 QPS. All of which is to say I'm actually at about two-thirds of my usable capacity; I can grow traffic about 150% before I get into trouble, if I'm lucky.

This process is something like what I might use if I were encountering this server in the wild. It's not perfect; as Niels Bohr said, "It's hard to make predictions, especially about the future." Despite the uncertainty that remains, this approach is much better than staring at a chart and thinking, "I don't know, it looks like it's scaling linearly and CPU utilization is only 10%, so I guess we have a lot of headroom!"

The USL has a few nice properties that make it suitable for this type of capacity planning:

- It's a "black box" technique, which uses data that's usually easy to get.
- Gathering data and using regression to analyze it is also easy.
- The USL is a relatively simple model, so people like me can understand the math.
- The USL is highly intuitive in comparison to most other approaches.

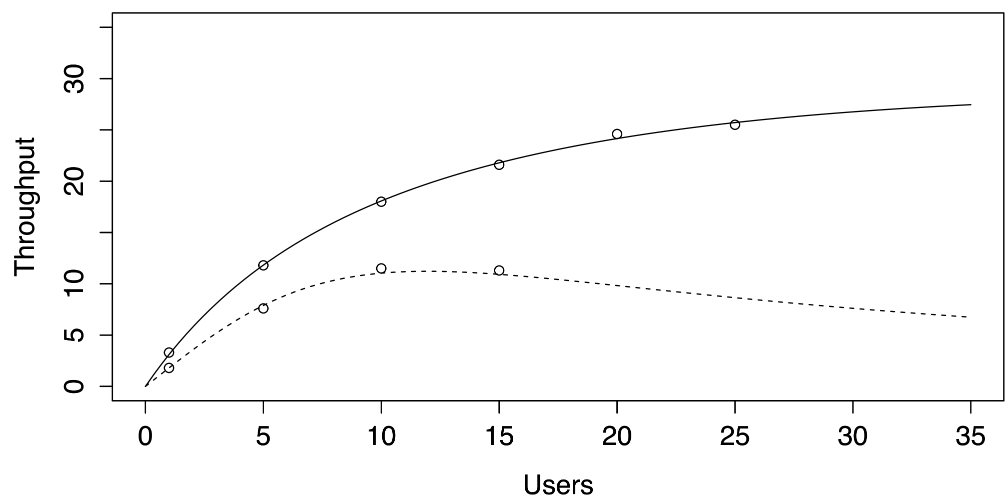
I would just repeat my caution that a lot of systems perform worse than the USL predicts they will, because their degradation in scalability at larger sizes is more severe than predicted. This is why I suggest viewing the USL's prediction as optimistic: "I won't count on being able to scale this system as high as the USL predicts I can."

You can also combine the USL with selected techniques from queueing theory, such as the Square Root Staffing Rule, to forecast how much capacity is needed and what quality of service it will provide. See the aforementioned queueing theory book for more on this topic.

Using the USL to Improve Scalability

One of the best uses of the USL is to explain *why* a system doesn't scale as well as it might. Armed with this knowledge, you can get clues about where to look for bottlenecks, so you might be able to alleviate them and improve the system's scalability. With practice, you'll also develop a mindset of scalability, building intuition about which design decisions can cause serious sublinearity.

An example will help illustrate. A few years ago PayPal published benchmark results of a Java application they rewrote in NodeJS. I analyzed their benchmark results and wrote about it on the VividCortex blog. Here are the plots and the key scalability parameters:



	σ	κ
Java	0.000011	0.006323
NodeJS	0.080319	0.000222

These systems scale very differently,[1] and for very different reasons. In a nutshell, the Java benchmark shows much higher crosstalk penalty,

[1] Both of them scale pretty poorly, in fact.

whereas the NodeJS benchmark exhibits more contention from queueing and serialization. Examining the architectures of the two systems reveals why: the Java app is multi-threaded and NodeJS is single-threaded with an event loop, and the PayPal blog post even mentions that they used “a single core for the NodeJS application compared to five cores in Java.”

This is a great real-life example of key scalability tenets:

- Make things as parallelizable as possible to avoid serialization and queueing.
- Avoid crosstalk and synchronization.

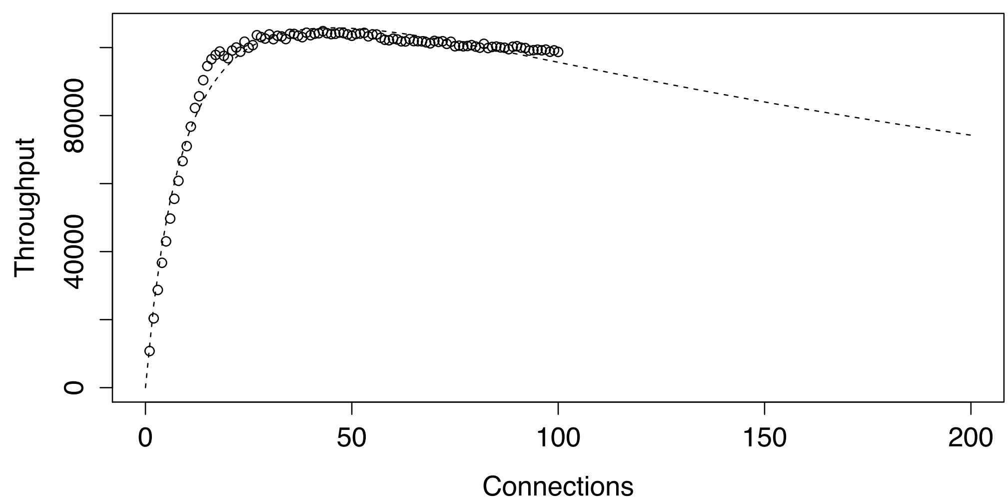
If you’re using the USL to model and analyze system scalability, another valuable practice is to approach the USL as a pessimistic scenario. Synchronous repairman queueing, the basis of the USL, is actually a worst-case in terms of the amount of queueing delay that occurs in a system. This is another way of saying that well-built systems theoretically ought to scale *at least* as well as the USL predicts. This should prompt you to ask the question, “why is this system degrading more than it should?” The answer to that is to look at whether the system degrades because of contention (queueing, serialization) or crosstalk (synchronization, communication, pairwise data interchange). If you can identify the likely cause, you might suspect that you need to look at mutex contention, for example.

Thinking Critically About The USL

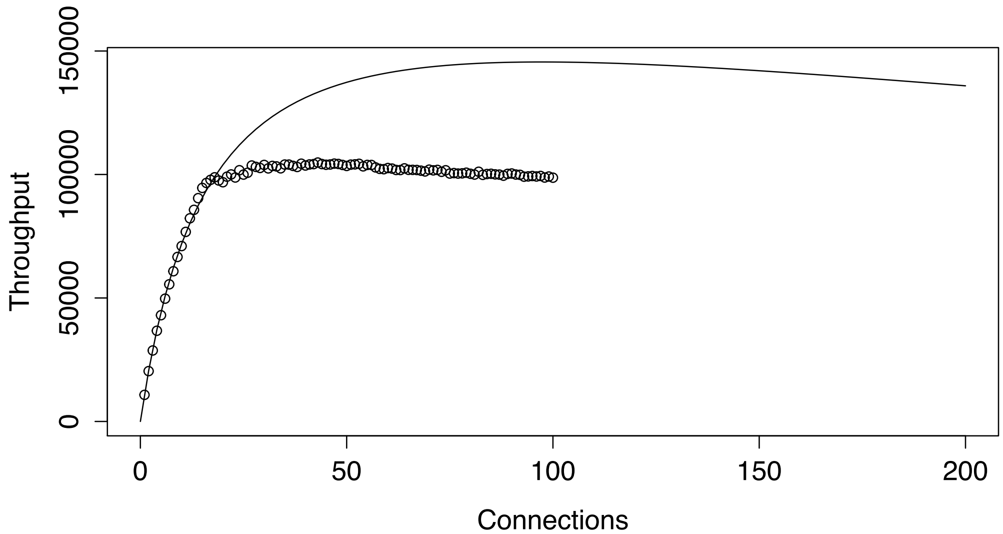
For many people, the USL is a huge shift in mindset. I know it was for me. But is it the be-all and end-all? Of course not.

Not only is the USL not the answer for every problem related to scalability or capacity planning, sometimes it doesn’t even work well for the problems it *can* solve. The examples I’ve shown thus far are remarkable in that they're extremely clean data. Real-world systems often have noisy data that doesn't "look like" the USL much at all. It may look more like something your cat threw up on the screen. Even when there is a nice-looking shape to the data, regression can produce unphysical results, or refuse to produce results.

And then there are the systems for which you have nice clean-room measurements, highly reproducible, but they just don't seem to fit the USL very well. I could show many examples. I've seen systems that appear to scale nicely, following the USL, but suddenly flatline instead of continuing a graceful curve, or abruptly degrade way faster than predicted—or the reverse, appearing to degrade more *slowly* than predicted once we exit Amdahl territory and enter the region where the higher-order terms prevail. Here's an example:



Maybe this is queue saturation or resource saturation, or maybe it's something else, but it is not something the USL has the capability to model or predict. If you'd tried to predict this behavior by fitting the USL to the first dozen or so data points, you'd get something like this:



The USL doesn't give you the tools to see those kinds of behavioral shifts coming. There's no way to tell it how many servers are servicing queues, for example. I've seen many benchmarks that fit the USL nicely with increasing numbers of threads until there is one thread per physical CPU core; then there's an abrupt change such as those in the diagram. I've seen others where a resource such as network bandwidth becomes a limiting factor.

If the USL is incomplete, what is it good for? The answer is lots.

First of all, a model is better than no model. In the absence of a model explaining the workings of system scalability, there isn't even a point of comparison to assess your expectations and results. There's no frame of reference to say, "I think this system should scale better than it does," or "This system's behavior makes no sense." Whether the USL is applicable to a given problem or not, it still provides a framework. Without it, why not just draw lines at will? You could get a set of French curves and follow your muse. No one could say you're wrong.

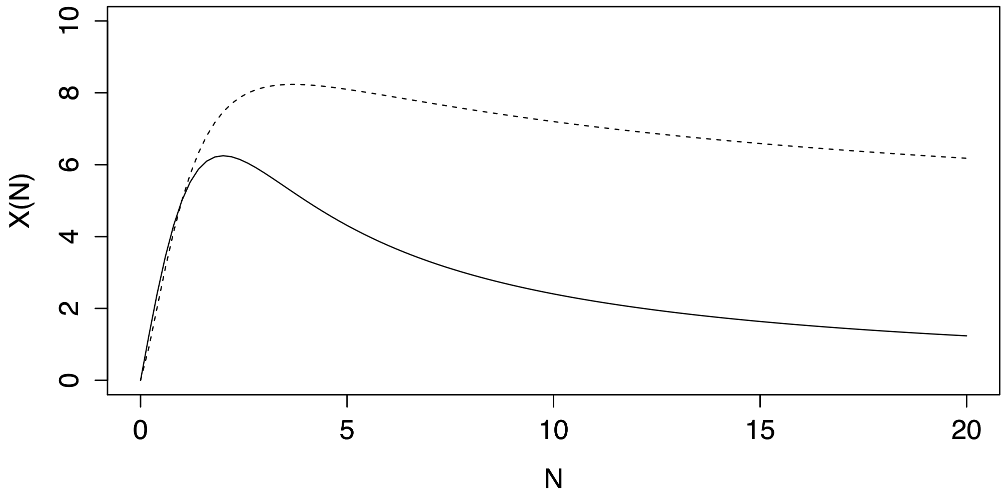
In an objective sense, the USL is both incomplete and wrong. If it were complete and correct, for example including knowledge about capped resources such as number of CPUs (which is a vitally important parameter for queueing theory problems) or network bandwidth, it would certainly describe more systems and scenarios than it does in my experience. As Richard Feynman said in a 1964 lecture at Cornell University, "If it disagrees with experiment, it's wrong. That simple statement is the key to science."

This is not to say the USL isn't useful. As George E. P. Box famously said, "all models are wrong, but some are useful." The USL is incredibly useful. But we must not put it onto a pedestal and worship it.

I'd also like to note that one could easily conjecture and analyze other USL-like models. For example, given that many computer algorithms, such as those that might perform pairwise interchange to cause the coherency penalty, can be shown by analysis to be $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$, perhaps we could imagine that the quadratic κ term doesn't really behave like its worst-case, and the term could be replaced by a logarithmic term, like so?

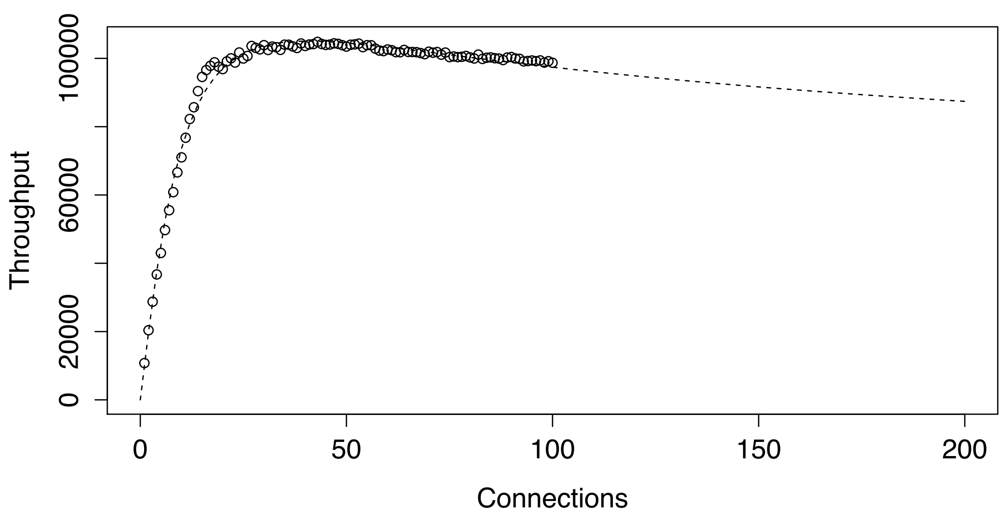
$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa \log(N)(N-1)}$$

To give some visual intution of how this differs from the accepted form of the USL, here they are together on a single plot. The standard USL is the solid line and the logarithmic variant is the dashed line.



You might think that you could choose parameters to the standard USL to make it follow the dashed line better, but it doesn't work. The functions are of different order and type, and won't behave the same.

Would this be a better model for the data I showed previously? Visually, it does appear to model the observed data somewhat more closely:

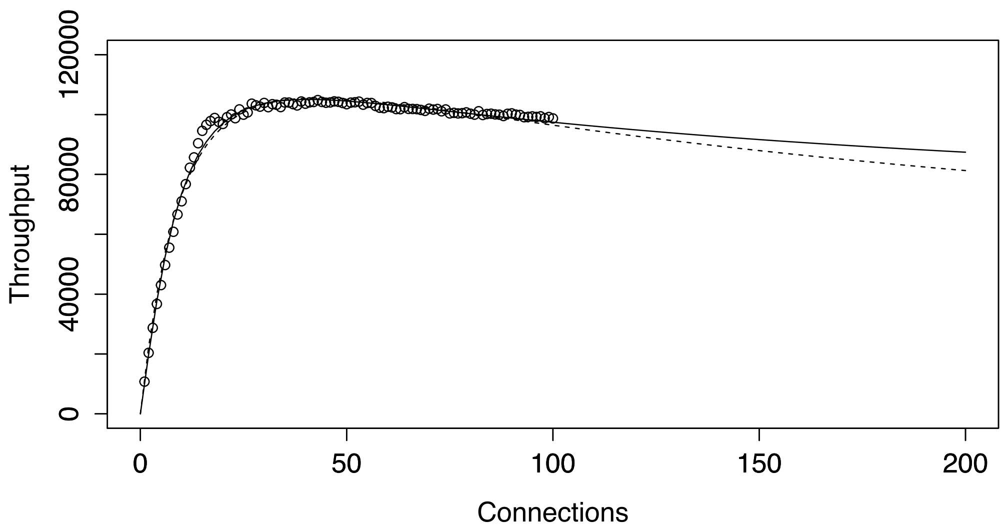


Jayanta Choudhury has suggested changes to the USL to model cases such as this, where resources apparently become saturated. His

Asymptotically Improved Super-Serial Law[1] is slightly more complex:

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \sigma\kappa N^{\beta}(N-1)}$$

The β parameter ranges from 0 to 1, inclusive. The following plot shows the logarithmic variant I proposed in a solid line, and the AISSL in a dashed line. As you can see, the logarithmic variation fits the data better.



Unfortunately it's a bit difficult to estimate parameters for the AISSL, making it harder to use than the USL.

As I'm sure you can imagine, you could play games like this all day long. Dreaming up an idea is much easier than proving that it's correct or showing how it might arise analytically from the underlying mechanisms as we understand them to operate in the system.

The more practical way to look at the examples I've given in this section is to stop trying to predict what happens after retrograde scalability or resource saturation kicks in. There are at least two good reasons for taking this pragmatic approach.

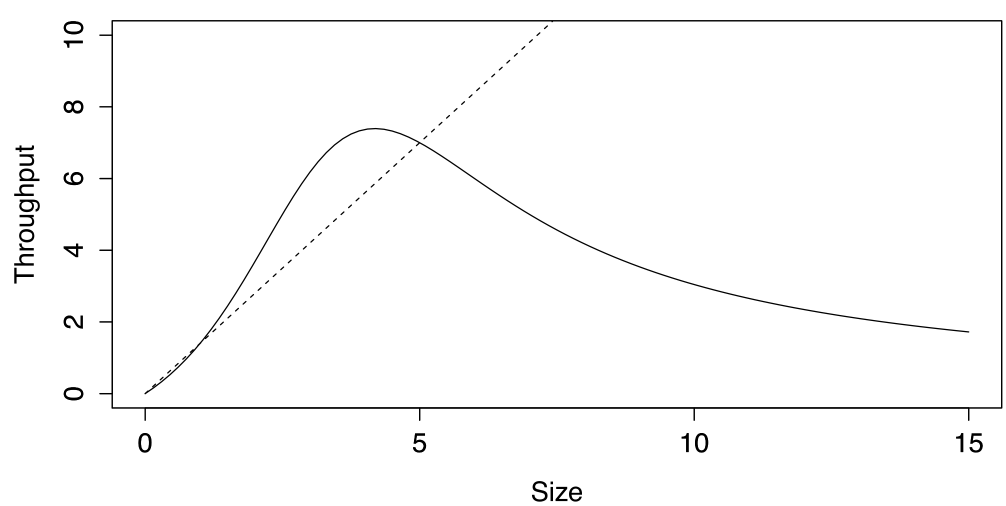
1 See *Parameter Estimation of Asymptotically Improved Super-serial Scalability Law* by Dr. Jayanta Choudhury, of TeamQuest Corporation.

1. **It's a different model.** We know the system's behavior is being influenced by factors that weren't present at a smaller size. It's not just different parameters, the entire model has changed, and it's doubtful that a single model exists that could explain those wildly different behaviors.

2. **It's pointless.** When we see retrograde scaling, we know the system has gone past the point where it's in trouble. Nothing good can come of pushing it further, so why should we try to model how bad it is? It's a lost cause with no practical purpose.

Superlinear Scaling

I've spent some time analyzing the possible causes of sublinear scaling. Is superlinear scaling possible? As I've worked with the USL over the years, I've found a number of cases where systems apparently do scale superlinearly. It manifests as a negative σ coefficient and a USL curve that has a more complex shape, rising above linear and then below again:



At first I dismissed this result as unphysical (how can there be less than zero contention?), but after repeatedly seeing this happen and having many conversations with Neil Gunther about it, I started to wonder. So did he, and eventually he was able to reproduce and explain the effect on a large-scale Hadoop TeraSort benchmark.

The TeraSort case is quite a specific one. In the more general case, I would explain superlinear scalability as a disproportionate scaling of some resource relative to the load placed upon it, creating an economy of scale. For example, adding more nodes to a clustered database system adds more memory; if the dataset size is not scaled proportionately, then more of the data fits into memory on each node, and access times improve relative to disk reads. Any resource that is more efficient when shared than when used singly may cause this effect.

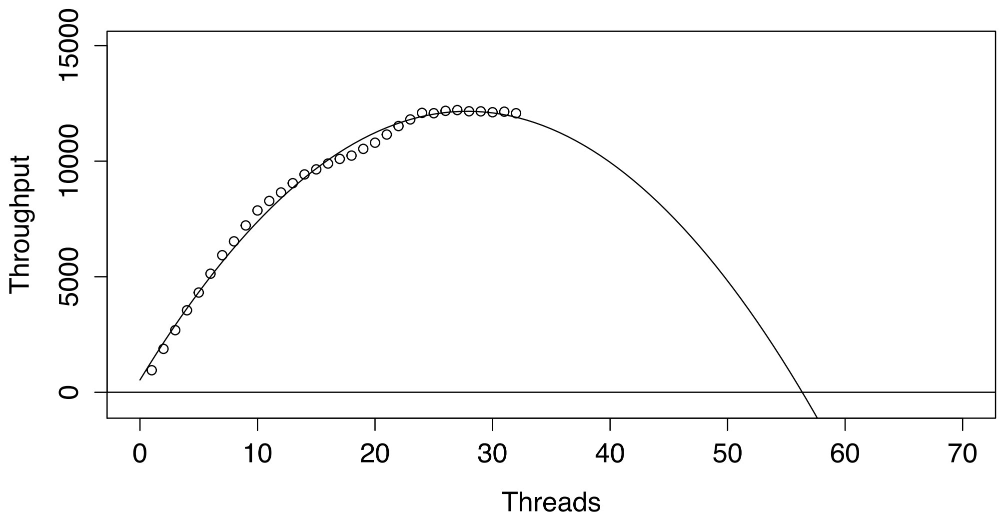
It's worth noting that this initial boost, depending upon its cause, may be countered by a correspondingly disproportionate "payback" later when performance falls quickly below linearity again.

Another special case to be aware of is that some clustered systems behave differently at sizes 1 and 2 than they do at 3 and above. For example, at size 1 there is no crosstalk or contention—it isn't a distributed system. At size 2 special-cases may be in play. At size 3 and above, usually generic algorithms and techniques suited for any size n are in use. Some clustered systems have to be benchmarked or measured at larger sizes in order to avoid skew from these effects.

Other Scalability Models

In addition to the USL and the variants I've already discussed, there are many other potential models of scalability you could consider. In my opinion some are good, some are useful, some are not. I also believe that there is much more work to be done on this topic.

Some alternative theories, however, are just garbage. Chief amongst them is "quadratic scalability." Observing that systems under increasing amounts of load will first increase in throughput, then level off and begin to decline, some people get the bright idea that they should "fit a curve to it." The curve is always a quadratic polynomial and the fitted curve ends up being a parabola opening downwards. Let's see how this looks on the benchmark data from Percona once again:



Look, Ma, it's a great fit! I don't even need to compute the R^2 value to know that. There's just one problem: it predicts that at some point we'll achieve negative throughput.

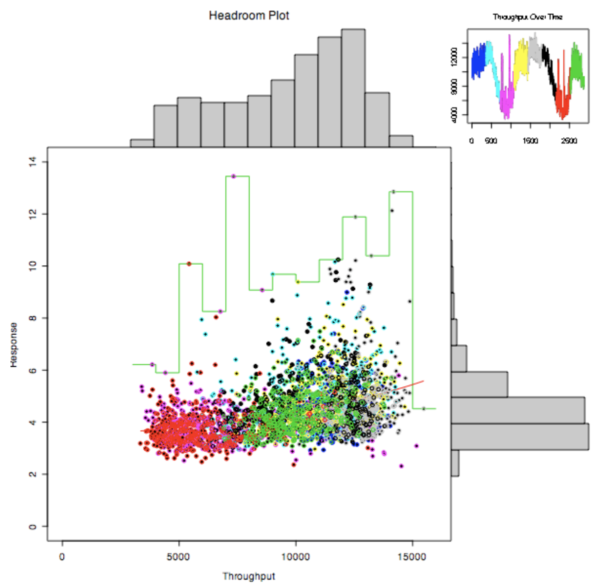
You should ignore this model because by definition it doesn't work. Rather than use it, I'd suggest that you get out that French curve set again and get in touch with your inner artist.

Another model that I've seen is latency as a function of throughput. Here are three examples:

- New Relic's scalability chart, which plots latency as a function of throughput and renders a smoothed line through the points. The line has no predetermined form and doesn't express any particular model.
- AppDynamics's scalability analysis feature does the same thing, but fits a parabola through the lines instead of a polynomial of arbitrarily high degree.

- Cockcroft headroom plots, by Adrian Cockcroft. These are also latency-versus-throughput charts, but add some histograms and other useful visual cues around the edges.

These three express essentially the same belief about the relationship between variables–that throughput is an independent variable and determines latency. The differences between them are relatively minor. Here's an example of a Cockcroft Headroom Plot:



in graphical form, but lacking formal analysis or a mathematical definition,

The only trouble is, the assumption that latency is determined by throughput is not correct in the general case contemplated by the USL.

Hardware and Software Scalability

I mentioned that Neil Gunther actually defines two forms of the USL, one for hardware scaling and one for software. They're essentially the same equation, with different Greek letters.

For most purposes it's not important to care about the distinction.

However, it is *very* important, in general, to have a firm grasp on the meaning of the X axis in the USL chart. I've been somewhat casual about it, essentially treating it as a generic metric of size by which we scale the system or workload, but I should be more precise.

In fact there are at least three important dimensions of the work a system performs, and how it scales, and these three interact with each other. A correct understanding of the concepts is important to get sensible results:

1. **Drivers.** The number of things producing work requests for the system. In a benchmark, for example, this is typically the configured concurrency of the benchmark—that is, the number of driver threads. It could also be the number of connections to the database, the number of users on a web application, and so on.
2. **Servers.** The number of servers as defined in queueing theory. It could be the number of CPUs in a server, the number of servers in a cluster, or the like.
3. **Data.** The size of the dataset. This will most typically be in the usual units—megabytes or gigabytes, number of rows—but will occasionally be the number of logical partitions in the dataset ("shards"). A VoltDB benchmark that I analyzed once needed to be couched in terms of partitions, because of the configured per-partition redundancy.

Now, what's really important to understand is that everything needs to be held constant relative to the unit of scale you're using, so you are changing only one variable at a time. If you're measuring scalability at different cluster sizes, for example, you need to grow the number of driver threads and the data size proportionately to the number of nodes in the cluster, so each node receives the same amount and rate of work to perform upon the same amount of data no matter the cluster size. (If you hold the dataset constant and increase the number of nodes, you'll get superlinear scalability.)

Asid from this caveat, the Universal Scalability Law really is universal and is a framework that can be applied to many different situations, as long as you can define the variables correctly.

Conclusions

Further Reading

* GCaP * Look at the Percona white paper all of the books in my picture - everything you've learned applies to organizations and their communications designs too.

About VividCortex

VividCortex is a SaaS database performance monitoring platform. The database is the heart of most applications, but it's also the part that's hardest to scale, manage, and optimize even as it's growing 50% year over year. VividCortex has developed a suite of unique technologies that significantly eases this pain for the entire IT department. Unlike traditional monitoring, we measure and analyze the system's work and resource consumption. This leads directly to better performance for IT as a whole, at reduced cost and effort.

Related Resources From VividCortex

Everything You Need To Know About Queueing Theory

This highly accessible introduction demystifies queueing theory without using pages full of equations, helping you build intuition about it.

Case Study: SendGrid

VividCortex has been instrumental in finding issues. It's the go-to solution for seeing what's happening in production systems.