MATURE OPTIMIZATION HANDBOOK

CARLOS BUENO

carlos.bueno.org/optimization
Typeset in Calisto MT with XƎTEX.
Version 0014, Oct 2013

To Mom & Dad,
who showed us how to use the power tools
to take apart other power tools.

Table of Contents

Introduction

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time; **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth, *Structured Programming With go to Statements*

The trickiest part of speeding up a program is not doing it, but deciding whether it's worth doing at all. There are few clear principles, only rules of thumb.

Part of the problem is that optimization is hard to do well. It's frighteningly easy to devolve into superstitious ritual and rationalization. Then again, there can be big payoffs hidden in surprising places. That's why expert advice about performance tends to have a gnomic, self-contradictory flavor: "If you don't know what you are doing, don't do it! You'll know if you know what you are doing. And remember to design your programs for performance." The experts are acutely worried about encouraging more folly, yet can't quite bring themselves to ignore the possible gains.

Knuth's famous quote about premature optimization was never meant to be a stick to beat people over the head with. It's a witty remark he tossed off in the middle of a keen observation about leverage, which itself is embedded in a nuanced, evenhanded passage about, of all things, using *gotos* for fast and readable code. The final irony is that the whole paper was an earnest attempt to caution against taking Edsger Dijkstra's infamous remark about gotos too seriously. It's a wonder we risk saying anything at all about this stuff.

Structured Programming With go to Statements does make two valuable points about performance. Optimizing without measurement to guide you is foolish. So is trying to optimize everything. The biggest wins tend to be concentrated in a small portion of the code, "that critical 3%", which can be found via careful measurement.

While a proper measurement regime can tell you where optimization is likely to succeed, it says little about whether doing it is worthwhile. Knuth ultimately shoves that responsibility onto a hypothetical "good" and "wise" programmer, who is able to look past the witty remarks and dire warnings and decide on the merits. Great, but how?

I don't know either. Performance optimization is, or should be, a cost/benefit decision. It's made in the same way you decide just how much effort to put into other cross-cutting aspects of a system like security and testing. There is such a thing as too much testing, too much refactoring, too much of anything good. In my experience, it makes most sense on mature systems whose architectures have settled down.

The age of a piece of code is the single greatest predictor of how long it will live. Stastically speaking, you encounter a piece of code somewhere in the middle of its lifespan. If it's one month old, the odds are good it will be rewritten in another month. A five-year-old function is not "ready to be rewritten", it's just getting started on a long career. New code is almost by definition slow code, but it's also likely to be ripped out and replaced as a young program slouches towards beta-test. Unless your optimizations are going to stick around long enough to pay for the time you spend making them *plus* the opportunity cost of not doing something else, it's a net loss.

Optimization also makes sense when it's needed for a program to ship. Performance is a feature when your system has unusually limited resources to play with or when it's hard to change the software after the fact. This is common in games programming, and is making something of a comeback with the rise of mobile computing.

Even with all that, there are no guarantees. In the early 2000s I helped build a system for search advertising. We didn't have a lot of money so we were constantly tweaking the system for more throughput. The former CTO of one of our competitors, looking over our work, noted that we were handling ten times the traffic per server than he had. Unfortunately, we had spent so much time worrying about performance that we didn't pay enough attention to credit card fraud. Fraud and chargebacks got very bad very quickly, and soon after our company went bankrupt. On one hand, we had pulled off a remarkable engineering feat. On the other hand, we were fixing the wrong problem.

The dusty warning signs placed around performance work are there for a reason. That reason may sting a little, because it boils down to "you are probably not wise enough to use these tools correctly". If you are at peace with that, read on. There are many more pitfalls ahead.

1: Defining the Problem

"First, catch the rabbit."

a recipe for rabbit stew

Before you can optimize anything you need a way to measure what you are optimizing. Otherwise you are just shooting in the dark. Before you can measure you need a clear, explicit statement of the problem you are trying to solve. Otherwise, in a very real sense, you don't know what you are doing.

Problem definitions often can be taken off the shelf. They are a lot like recipes. Many people have been this way before and there is a large body of knowledge available to you. There's nothing magical or hard to understand; the key is to be explicit about the recipe you are following.

It's impossible to clearly define the problem of clearly defining the problem. But there is a way to judge the quality of a recipe. It must be specific enough to suggest a fix, and have an unambiguous way to tell whether the fix worked. A problem definition must be *falsifiable*. You have to risk being wrong to have a chance at gaining the truth. Let's start with a bad definition and make it better.

"WidgetFactoryServer is too slow."

What does "slow" mean here? It could be that it takes so long that a human notices (eg, more than 350 milliseconds) or that some consumer of WFS times out while waiting for a response, or perhaps it shows up on some measure of slow components. "Too slow" is a judgement about *walltime*, ie time passing according to the clock on the wall.

Two things generally contribute to walltime: computation on the local CPU and time spent waiting for data from storage or the network. You wrote WFS so you know that it doesn't read from disk or network. Another contributor in threaded code is waiting for locks, but WFS isn't threaded. So it's probably all in computation. This fits a familiar pattern. We can use a ready-made problem definition.

> "WidgetFactoryServer is transactional. It receives requests and emits responses. It is probably CPU-bound. So if we profile to discover which functions take up the most time, and optimize those, the total CPU time per transaction should decrease."

Good enough. It states a thesis about what resource is bottlenecked, suggests a method of finding the right places to optimize, and is clear about what result you expect to see. Happily, since we're already measuring CPU, there's no need for a separate measurement to test the result.

The actual optimization is almost anticlimactic. Alicia's profiling finds that the most expensive function call in WFS has a bit of clowniness in a tight loop, say it iterates a list of valid WidgetTypes over and over again, or recalculates a value that can be calculated once. Once she sees it, the fix is obvious and testable. The CPU time drops and there is much rejoicing.

Now the trouble starts

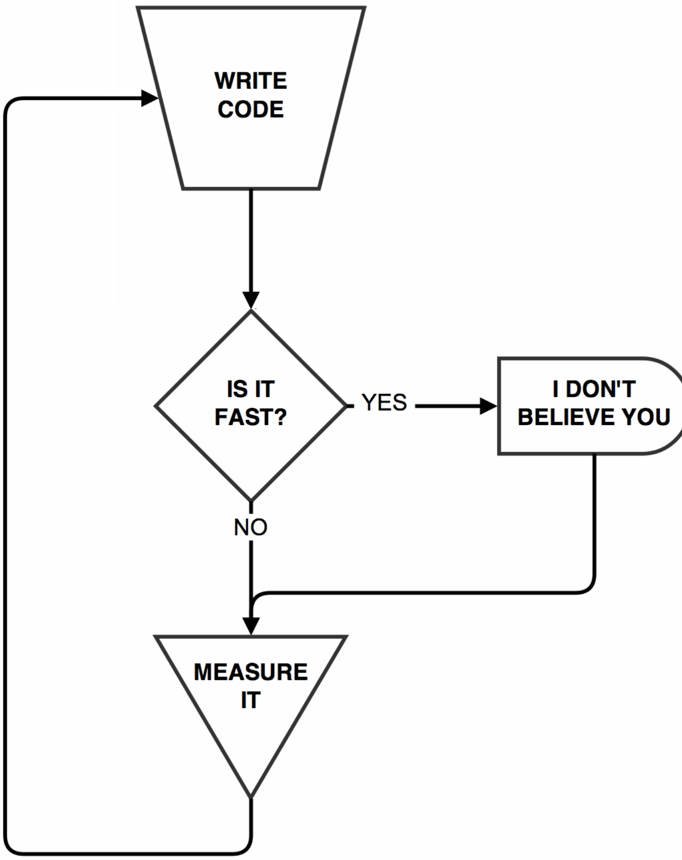
There is a rich & vibrant oral tradition about how to write fast programs, and almost all of it is horseshit. It's here in the afterglow of success that it takes root. Alicia finds that eliminating a piece of code that iterates WidgetTypes speeds up the program by 10%. She spends the next two weeks searching for this pattern and "optimizing" wherever she finds it. "Iterating WidgetTypes is slow!" she tells her fellow coders. That might be technically true, but does it matter? Are those other instances in the critical path? Do they show up in the measurements? Probably not. Yet another folk remedy is born. After all, it worked once.

All of this define-the-problem-then-measure stuff isn't like a set of training wheels you cast off once you've learned how to ride on your own. You really do have to approach every act of optimization as an experiment, starting over from scratch, making every assumption explicit and testable.

Computer systems are *astoundingly* complex, and it's silly to generalize too much from a given a performance bug. "We configured the number of threads to be twice the number of cores and saw this speedup" is good science, but it is *not* the same as "If you up the threads to 2X cores you will see a speedup". Wishing doesn't make it so.

> "I once generalized from a single data point, and I'll *never* do that again!"
>
> Achilles the Logician, *Lauren Ipsum*

The lessons learned from past optimizations aren't useless. They can hint very strongly at where to look. But we are never justified in trusting them blindly the way we trust, say, gravity. If someone gives you a performance tip, ask for the data. If they show you a benchmark, dig into the methodology. If they tell you "X is slow" or "Y is fast", take it under advisement. Then measure it yourself, every time.


WRITE CODE
IS IT FAST?
YES
I DON'T BELIEVE YOU
NO
MEASURE IT

2: Flying By Instruments

> “It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.”
>
> Donald Knuth, *Structured Programming With go to Statements*

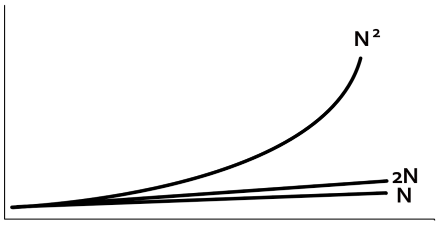
The scariest part of flying a plane is learning to trust your instruments. Humans are very bad at judging distance and orientation in three dimensions, even pilots. When your eyes and inner ear are telling you one thing and the ground is telling you another, the ground wins every time. You have to accept that there are very real human limitations that cannot be overcome by talent or practice.

In a similar way, performance work depends on “flying” by measurement. Humans are bad at predicting the performance of complex systems, even programmers. *Especially* the programmers. Our ability to create large & complex systems fools us into believing that we’re also entitled to understand them. I call it the Creator Bias, and it’s our number-one occupational disease. Very smart programmers try to optimize or debug or capacity-plan without good data, and promptly fall right out of the sky.

How a program *works* and how it *performs* are very different things. If you write a program or build a system, then of course you know how it works. You can explain its expected behavior, logical flow, and computational complexity. A large part of programming is playing computer in your head. It’s kind of our thing. By all means, use your brain and your training to guide you; it's good to make predictions. But not all predictions are good. Never forget that our human-scale understanding of what's supposed to happen is only a very rough approximation of what actually does happen, in the real world on real hardware over real users and data.

The Big O

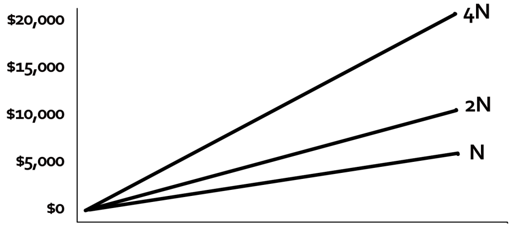
You may remember this chart from school. It shows the vast divide between the running times of different orders of computational complexity. O(N^2) so completely dominates O(N) that the coefficients hardly matter. This is a deep and important concept. Without complexity analysis we would waste a lot more CPU cycles than we already do.



While this chart says something absolutely true, it implies something else which is false. Looking at this graph it's easy to convince yourself that complexity analysis is all you need to know about performance, that the rest is just a rounding error.

In the real world, Big-O complexity is almost never the reason your program is slow. Look at the slope of the N-squared curve. In all probability, either N is so small that it doesn't really matter, or N is so large and its effects so obvious that any program with that kind of bug gets fixed quickly.

That means the stuff you have to contend with is the stuff your professor told you to ignore: coefficients, constant factors, and variability. Going from O(N^2) to O(N) is easy in the sense that you can do it on a whiteboard via pure analysis. Complexity bugs are fun and straightforward to teach, so that's what's taught. But how do you take an existing program from 2N to 1N? How do you know what the coefficient even is? (Just how high *is* that mountain, anyway? Will you clear it?)



Here, the Creator Bias kicks in again. Why not just analyze the program more deeply? I'm smart; coefficients don't sound that hard. The only problem with this approach is that you have to throw in everything: the language, the compiler that implements the language, the operating system and the hardware, all the data, and a lot more.

Imagine two computers with identical software, configuration, environmental conditions, and hardware specs. The only difference is that one has four 4GB memory chips and the other has one 16GB chip. Under many –but not all– workloads there will be a measurable difference in the throughput of these two systems. Even if you understand why that can happen, your guess is as good as mine (ie, useless) as to what the actual difference will be. It depends on every other aspect of the system.

The more detailed you make a model of a program, the more it becomes a slow, buggy simulation of that program. It doesn't matter how smart you are; it's a direct consequence of the Halting Problem. That's what Turing equivalence *means*. That's why being good at programming requires being good at playing computer. So you have to ask yourself a serious question: who's better than you at playing computer? Right. The computer.

In other words, learn to trust your instruments. If you want to know how a program behaves, your best bet is to run it and see what happens.

Even with measurements in hand, old habits are hard to shake. It's easy to fall in love with numbers that seem to agree with you. It's just as easy to grope for reasons to write off numbers that violate your expectations. Those are both bad, common biases. Don't just look for evidence to confirm your theory. Test for things your theory predicts *should never* happen. If the theory is correct, it should easily survive the evidential crossfire of positive and negative tests. If it's not you'll find out that much quicker. Being wrong *efficiently* is what science is all about.

Your job isn't to out-computer the computer. Your goal isn't to be right. Your goal is to discover what is. Think of it as optimizing for truth.

3: The Right Way to be Wrong

"It is better to do the right problem the wrong way than the wrong problem the right way."

Richard Hamming, *The Art of Doing Science & Engineering*

If you follow all the rules, measure carefully, etc, and it still doesn't work, it's possible that your problem definition isn't just falsifiable, it's *false*. You might be measuring the wrong thing, or optimizing the wrong layer of the stack, or misled about the root cause. This happened once to Facebook's HHVM team. It's an extreme example that doesn't come up very often, but milder versions happen every day.

HHVM is a virtual machine for the PHP programming language. Some of the time it runs bytecode that is dynamically generated ("just-in-time" aka JIT), but most of the time is spent running precompiled C++ code. The implementation wasn't completely settled, but there were legitimate reasons for making performance a feature.

They started off with a reasonable problem definition, similar to the one we used for WidgetFactoryServer. Just find the functions that consume the most CPU time and optimize them. But things didn't work out the way they expected.

> "HHVM today is about three times faster than it was a year ago. Then, as now, it spent about 20% of time in the JIT output, and about 80% in the C++ runtime.
>
> The great mystery for you to ponder, and I would hope for your book to explain, is that we got three times faster by focusing our optimization efforts on the code that was executing for

20% of the time, not 80% of the time. Go back and reread that.

When you internalize that this sort of thing really happens, in real programs that you're responsible for, and that we're not talking about pocket change, but a 3x difference in performance, it is scary.

Learning which functions the CPU cycles are being spent on can actively deceive you about what to optimize. The advice we give intermediate programmers about "premature optimization" and allowing profiling to drive your optimization efforts is, well, untrue. Or rather, it's a heuristic that helps compensate for even more dangerously wrong optimization impulses.

-- Keith Adams

So, what happened? How could they get huge wins out of *ignoring* the proverbial 80%? Because the minority code had indirect influence on the rest. Changes to the JIT code, nominally responsible for only 20% of CPU time, caused random, outsized performance effects throughout the system.

To understand why, remember that a computer really does only two things: read data and write data. Performance comes down to how much data the computer must move around, and where it goes. Throughput and latency *always* have the last laugh. This includes CPU instructions, the bits and bytes of the program, which we normally don't think about.[1]

The kinds of computers in use today have four major levels of "where data goes", each one hundreds to thousands of times slower than the last as you move farther from the CPU.

[1]This mental model is less simplistic than it appears. All computers are ultimately equivalent to a Turing Machine, and what does a Turing Machine do? It moves symbols around on a tape.

- *Registers & CPU cache*: 1 nanosecond
- *RAM*: 10^2 nanoseconds
- *Local drives*: 10^5 to 10^7 nanoseconds
- *Network*: 10^6 to 10^9 nanoseconds

Memory controllers try mightily to keep the first level populated with the data the CPU needs because every cache miss means your program spends 100+ cycles in the penalty box. Even with a 99% hit rate, most of your CPU time will be spent waiting on RAM. The same thing happens in the huge latency gap between RAM and local drives. The kernel's virtual memory system tries to swap hot data into RAM to avoid the speed hit of talking to disk. Distributed systems try to access data locally instead of going over the network, and so on.

HHVM was essentially "hitting swap" on the CPU. Actual machine code is data too, and has to be on the CPU in order to execute. Some JIT code would copy over and execute, pushing other stuff out of the CPU caches. Then it would pass control over to some function in the C++ runtime, which would not be in the cache, causing everything to halt as the code was pulled back out of RAM. Figuring this out was not easy, to say the least.

> Cache effects have a reputation for being scarily hard to reason about, in part because caches of all kinds are a venue for spooky action-at-distance. The cache is a stateful, shared resource that connects non-local pieces of your program; code path A may only be fast today because the cache line it operates on stays in cache across lots of invocations.

Making a change in unrelated code that makes it touch two lines instead of one can suddenly cause A to take a miss every time it runs. If A is hot, that "innocent" change becomes a performance catastrophe.

More rarely the opposite happens, which is even more frustrating, because of how clearly it demonstrates that you don't understand what determines performance in your program. Early on in development Jordan DeLong made an individual checkin, only affecting the JIT, that was a 14% (?!!) performance win overall.

-- Keith Adams

The surface problem was CPU time. But most of the time wasn't actually going to computation (ie, moving data around inside the CPU) it was spent fetching data from RAM into the CPU. Worse was that normal profiling, even advanced stuff like VTune and the Linux "perf" kernel module, weren't very useful. They will happily tell you what functions suffer the most cache misses. But they don't tell you *why* the data wasn't there.

The team had to come up with a different problem definition, which went something like this:

> CPU time in HHVM is dominated by CPU cache misses. If we somehow log *both* the function which suffered a cache miss and the function which *replaced* the data that was missed, we may find that a small number of functions cause most of the evictions. If we optimize those to use less cache space, we expect to see a reduction in both cache misses and CPU time spent across the board.

That's a mouthful, but it worked. To prove this theory they ended up gathering very expensive and detailed logs, for example, every datum accessed or CPU instruction executed. They then fed that through a cache simulation program, and modified the simulator to record the causal path from evictor to evictee, so they knew what functions to blame. Sort the list to find the worst offenders and optimize them. Not easy, but a lot more straightforward.

Everyone wants a cookbook to follow, and most of the time it's there. But how do new recipes get written? It happens when everything else fails, and you have to invent new ways to deal with, or even describe, the problem in front of you.

4: Continuous Systems

> "Time can't be measured in days the way money is measured in pesos and centavos, because all pesos are equal, while every day, perhaps every hour, is different."
>
> Jorge Luis Borges, *Juan Muraña*

In the age of the personal computer, software was made the way toasters (and pesos and centavos) are: lots of design up front, stamped out by the million, then sent off into the world to fend on their own. It was nearly impossible to get representative performance data from the field. On the other hand, any copy was as good as any other. If you made your copy run faster it would probably run faster everywhere. That is, when and if everybody upgraded to the newest version.

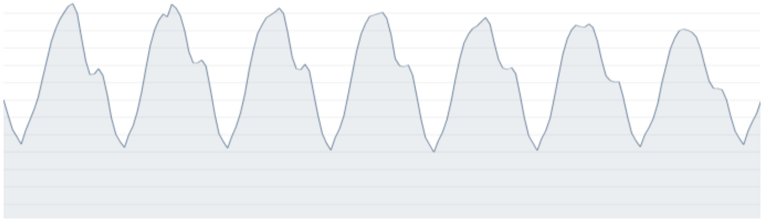
The other kind of software became dominant again 10 or 15 years ago. An internet app is more like a power plant than a design for a toaster. It's a unique, complicated artifact woven into a global network of other unique, complicated artifacts, half of them human beings.

Being able to give everyone the same version of your software at the same time was *the* killer feature of the internet. It's why we threw out decades of software and infrastructure and rewrote it inside the browser. But this new (old) model has other important consequences. To understand them, it's possible we should rely less on computer science and more on operations research and process control.

When a major component of your system is the entire world and the people in it, you can't really put a copy on the bench and test it out. On the other hand, getting live data and iterating on what you learn has never been easier. Instead of "versions" rolled out every year like cars, you constantly push out smaller changes and measure their effects.[2]

Spot checks and benchmarks aren't enough in this model. If something is important enough to worry about, it's important enough to measure all the time. You want a continuous, layered measurement regime. This makes it easy to run performance experiments in the same way that automated testing makes it easier to catch errors. It also removes a dependency on clairvoyance. If you happen to neglect to measure something you should have, it's not a disaster. You can start measuring it tomorrow. Flexibility in the instrumentation and analysis is key.

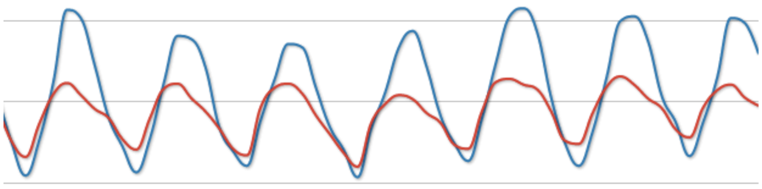
Most interestingly, a networked application follows cycles. Think for a moment about what it means to run a power plant. During the wee hours of the morning draw is low. Then people wake up. Alarm clocks and radios turn on (all at the same minute), then the electric ranges, office buildings, and factories. Elevators start running. People move from one part of the city to another and start using electricity there. School bells ring. Thermostats are tuned to the weather. If you were to chart a week of power draw, it would look something like this:



[2]There is also the hybrid model of mobile applications talking to servers, building both the power plant and the toaster that plugs into it. But this time round we're forcing automatic upgrades on the toasters.

The peak day is Monday, with everyone going back to work and school. The peak hour is noon, when power demand is more than twice the trough hour at 2am. There's another mini-peak around 7pm as people come home for dinner. It's easy to see that metrics averaged over an entire day or week are almost meaningless. Like dams, power plants are built for the peak not the average, and they have to deal gracefully with variation in demand. The same is true of network applications.

These diurnal and weekly cycles have important effects on systems, software, and the measurements you make. Here's a graph of the average per-transaction CPU time and CPU instructions for an internet application over a week. As the system heats up the CPU time spikes upward, partly due to increased computer utilization. The instructions metric is less susceptible to this but it still follows the cycles.



The mix of things people do is cyclical too. They check email in the morning; at night they play games and videos. Or think of a banking application. Users probably check their balance at all hours. Check deposits should cluster around Fridays, mid-month, and end of the month. Bank transfers probably spike a day or two later, as users pay their rent and bills.

Dealing with these cycles, making sure they don't interfere with your analysis, requires some care. When comparing two points in time, it's best to compare the same hour and day of the week. Running a quick experiment at midnight might not give you valid data. Running experiments side-by-side through at least one peak is more informative. This is where using A/B testing for performance optimizations really helps.

There are some neat tricks that take advantage of cycles. For example, the quickest way to reduce capacity demand might be to *turn off* expensive features for one hour per week. At very large scale this is less silly than it sounds. Another might be to move computation to off-peak time. Do you really need to generate those emails now, or could they be batched up and run in the afternoon? Precalculating or precaching data sounds expensive, but computer time in the trough is nearly free.

Some companies who use cloud providers keep a core set of machines on long-term lease, and spin up extra instances as needed. Renting by the hour is more expensive, and can leave you vulnerable to a resource crunch, but it's a decent calculated risk. If, of course, you actually do the calculations.

5: Instrumentation

"You know my method. It is founded upon the observation of trifles."

Sherlock Holmes, *The Bascombe Valley Mystery*

How do you profile? What do you measure? A problem definition hints at the bottlenecked resource you want to conserve, but instrumentation is hard to get right. There is lots of room to fool yourself. In this chapter we will focus on the recording of performance data. Where it goes and what you do with it will be discussed later on.

Let's begin with some jargon. A *measurement* is a number obtained during some kind of profiling event. It could be the number of CPU cycles spent, or time spent waiting for data fetching, the number of bytes sent or received, etc. It is almost always an integer, because we're dealing with discrete systems. If you come across a decimal it's because of the units (34.19 $msec = 34{,}190\ \mu sec$), or because it's not a single measurement but an average, like the CPU temperature or requests per second.

The *metadata* are attributes of the system or event that might influence the measurements. They include environmental facts, for example, the timestamp, the machine name, compiler version, the kind of transaction being done. They also include things learned during the event which are not numerical measurements, like error codes. The metadata is what you use to separate a goopy mass of numbers into coherent groups.[3]

[3] In machine learning and pattern recognition circles these are called *features*. In dimensional analysis, they are called *dimensions*. In the physical sciences you might hear them called *factors* or *nominal–*, *categorical–*, or *attribute variables*. All more or less the same thing.

A *sample* is a collection of measurements and metadata, an observation of a single event. It is a statement about that event, usually a transaction performed by the system. Technically the term "sample" can also refer to a set of observations taken from a larger set, eg "a 5% sample of the population". To avoid confusion, in this book "a sample" will always mean a single observation.

A *metric* is a statement about a set of samples. It's typically an aggregate number derived from some measurements, and a description of the subset of samples they come from. "The median walltime of WidgetFactoryServer, between 10am and 11am on 18 April 2013, in the West-coast datacenter, was 212 msec." Metrics can also be simple counts and proportions, eg "1.22% of hits returned an error code."

Computers are like onions

A good measurement regime is built up in layers. Avoid the temptation to measure the first thing that comes into your head, because it won't be random. It'll be the thing you feel you understand best, but our feelings are probably wrong. We optimize what we measure. Some things are easier to measure than others, so those tend to be optimized too much.[4]

Forget for a moment that you built the thing and start from first principles. Pretend you are examining an alien artifact that just fell out of the sky. You want to discover how it performs. Not how it *works*, how it performs. What does it consume? What does it output? Where does that happen? A good place to start is with basic "temperature and pressure" logs of the system. Then you add more detailed logging to help zero in

[4]On the other hand this is supposed to be a short book, so we will ignore the client side of performance work and focus on the server side.

on the hot spots. It's a process of recursively subdividing measurements to find concentrations of precious resources being consumed.

Your instrumentation should cover the important use cases *in production*. Make all the measurements you want in the lab, but nothing substitutes continuous real-world data. Think about it this way: optimizing based on measurements you take in a lab environment is itself a falsifiable theory, ie, that lab conditions are sufficiently similar to production. The only way to test that theory is to collect measurements in production too.

Let's say it's some kind of server application. There's a web server tier, a datastore, and maybe a caching layer. The web layer is almost always where the action is; if it's not the major bottleneck it still has lots of leverage. A baseline performance log of every hit to the web server doesn't have to be fancy, just counters of the resources consumed during each transaction. Here's an example, modeled on a real-world system:

```
Metadata
  timestamp          script_path
  server_name        status_code
  datacenter         ab_tests
  build_number       user_id
```

All of these should be cheaply available to the application layer. A common mistake is to collect too little metadata. Any input that affects the logic that executes, or might help you separate one pile of samples from another, is worth writing down. Logging the status code lets you count errors and keep inexpensive HTTP redirects from skewing the averages. If you log whatever A/B test buckets were in effect for a hit, you can use your A/B framework to experiment with different optimizations. If you log the application's build number you can connect changes in performance to changes in features.

The script_path is unfortunately a muddled concept. These days there is little connection between URI requested by the client and the "script" that handles it. I've yet to come up with a good name for this field, but it should contain a unique identifier for the major bit of code that ran during an event. If you are logging a hit to the home page, then "/" or "home" will do. If it was a call to an API you publish, then script_path should be the API method. If you have some kind of Controller architecture, use the name of the controller, and so on. The cardinality of this field should be low; a few thousand at most. If some URI parameter significantly changes the behavior, include it, but not all parameters.

```
Measurements
  walltime        db_count        cache_count
  cpu_time        db_bytes_in     cache_bytes_in
  memory_used     db_walltime     cache_walltime
  bytes_out
```

This is a decent set of measurements. They describe the major "consumables" of a computer (wall-clock time, CPU time, memory, network). They also split walltime into rough chunks, ie time spent waiting for the database and cache. They count how many data-fetching operations are done, and the ratio between the bytes fetched and sent to the client. Let's give this measurement system a name: Detailed Event & Resource Plotting, or DERP for short.

This minimal regime, applied to a system that's never had one before, would probably reveal a half-dozen things you can fix before lunch. And they will all be lurking in some place surprising.

Adding more layers

The next layer of logging is determined by what DERP tells you about your system. If it turns out that most of the time goes to CPU, then you can use a CPU profiler to find the expensive functions. If instead the system spends a lot of time waiting on the database, then you want to build a query logger.

Say that the time is going to database queries. The db_count is often high, as is db_walltime. Naturally you want to know which queries are slow. The expedient answer is to instrument the database. Most of them have a "slow query" log of some kind or another.

That approach has problems. It may be a lot of little queries causing the slowdown, not a few big ones. It can be hard to connect a query logged on the database to the code on the web tier that generated it. The same query might be triggered from multiple places. You'll end up stuffing more and more metadata into query comments then parsing them out later. And then there is the high cardinality of the queries themselves. We can do better.

In addition to counting database queries and the total time spent on the client (web server) side, log individual queries there too. The cardinality can be reduced by logging a normalized pattern. For example, this is the raw text the database might see:

```
SELECT photo_id, title, caption, ...
FROM photos
WHERE timestamp > 1365880855
  AND user_id = 4
  AND title LIKE '%kayak%'
  AND album_id IN (1234, 4567, 8901, 2345, ...)
  AND deleted = 0
ORDER BY timestamp
LIMIT 50 OFFSET 51
```

We can replace most of the literal strings and numbers with placeholders, since there's probably little performance difference between looking up user 4 and user 10, or "kayak" instead of "sunset". The explosion of IN clause values can be reduced to a count. Instead of a series of placeholders like `([N], [N], [N], [N], [N])` you log the nearest power-of-two, `[N8]`. You can take this quite far to reduce the number of unique queries: collapsing and sorting repeated clauses, unwinding subqueries, etc. In the extreme case you end up with a little language parser. It's fun, if you enjoy that sort of thing.

```
SELECT photo_id, title, caption, ...
FROM photos
WHERE timestamp > [N]
  AND user_id = [N]
  AND title LIKE [S]
  AND album_id IN [N8]
  AND deleted = [N]
ORDER BY timestamp
LIMIT 50 OFFSET 51
```

A sample in the DERP-DB log is structured very similarly to a DERP sample, except that it's a statement about an individual query, not the web hit as a whole:

```
DERP-DB
Metadata             Measurements
  timestamp            db_walltime
  server_name          db_bytes_in
  datacenter           db_query_bytes
  build_number         db_rows_examined
  script_path          db_rows_returned
  status_code
  ab_tests
  user_id
  db_query_pattern
  db_server_name
  db_error_code
```

Why have two logs that substantially overlap? Why not have a "request_id", log the common metadata only once, and join it up when you need it? Or put everything together in one log? Separate logging has a lot of nice properties, admittedly at the expense of disk space. You can easily imagine these logs as flat, denormalized database tables. Very fast and easy to query, which is important for exploration and charting. You also might want to have different sample rates or data retention for DERP-DB versus DERP.

Cache on the Barrelhead

Another layer of logging that seems to make sense here is on the caching system. Presumably it's there to store the results of expensive computations on the database or web tier for later reuse. How well does it do that? Its effectiveness depends on two metrics. The "hit rate" is how often the precomputed data is there when it's asked for. The "efficiency" describes how often a piece of data in the cache actually gets used. There are also ancillary things like the ratio between key and value length.

Just as with DERP-DB, a DERP-CACHE sample should describe a single data fetch. The important metadata are what operation was done (a GET, DELETE, WRITE, etc) the normalized key, the server being queried, and whether the fetch was successful. On the assumption that most database reads are cache misses, it also makes sense to add another field to DERP-DB, recording the normalized key which missed.

```
DERP-CACHE
Metadata                Measurements
  timestamp               cache_walltime
  server_name             cache_bytes_in
  datacenter              cache_key_bytes
  build_number
  script_path
  status_code
  ab_tests
  user_id
  cache_operation
  cache_key_pattern
  cache_server_name
  cache_hit
```

So now you know which query patterns take the most time, what script_paths they come from, and the efficiency of your caching system. Optimizing now comes down to whether the system is doing unnecessary queries, or whether their results can be cached, or perhaps the query itself (or the indexes, or the database) needs tuning. Only the data can tell you where to look next.

6: Storing Your Data

> "Time dissipates to shining ether the solid angularity of facts."
>
> Ralph Waldo Emerson, *Essays, 1841*

Our friends in the physical sciences have it worse. A thermometer can't actually measure the temperature at a given point in time. Heat takes time to transfer so a thermometer can only give the average temperature over some period. The more sensitive the measurements desired, the more the equipment costs. And there is always a lag.

You and I can make as many discrete (and discreet) measurements as we want. We can even *change* what measurements are done on the fly based on complex logic. And, of course, we know how to handle large amounts of data. So why do we throw the stuff away? Look at the documentation for a popular measurement system like RRDtool:

> You may log data at a 1 minute interval, but you might also be interested to know the development of the data over the last year. You could do this by simply storing the data in 1 minute intervals for the whole year. While this would take considerable disk space it would also take a lot of time to analyze the data when you wanted to create a graph covering the whole year. RRDtool offers a solution to this problem through its data consolidation feature.
>
> Using different consolidation functions (CF) allows you to store exactly the type of information that actually interests you: the maximum one minute traffic on the LAN, the minimum temperature of your wine cellar, the total minutes of down time, etc.
>
> -- oss.oetiker.ch/rrdtool/doc/rrdtool.en.html

This approach is *dead wrong*, and not because of its breezy claims about performance. Why do we force ourselves to guess every interesting metric in advance? To save disk space? Computing aggregates then throwing away your raw data is premature optimization.

In exchange for that saved space, you have created a hidden dependency on clairvoyance. That only works if all you will ever need to know is the "maximum one minute traffic". If later on you want to know what the average traffic was, or the 95th percentile, or grouped by protocol, or excluding some hosts, or anything else, you can't.

Making sense of a pile of data is a multidimensional search problem. We're programmers. We know how to handle search problems. The key is being able to freely jump around those dimensions. It's not possible to predict ahead of time the aggregate metrics, the search paths, you'll want to use later on. Storing all the raw data can be expensive, but on the other hand the combinatorial explosion of all possible metrics that can be derived from it is much larger. On the third hand, the usefulness of raw data drops off sharply as it ages.

There is a way to redefine the problem. What's happening is a clash between two use-cases. New data needs to be in a "high-energy" state, very fluid and explorable. Old data can be lower-energy, pre-aggregated, and needs to be stable over long periods of time.

So, store raw performance data in a very fast database, but only for the last few weeks or months. You can explore that raw data to discover the metrics you care about. *Then* you render it down to those road-tested metrics and shove them into RRD or whatever else you want for long-term storage. A buffer of recent raw data to help you diagnose novel problems, and stable historical records so you can track how well you're doing.

This brings us next to the storage engine of a system developed at Facebook called Scuba.[5] When Lior Abraham and David Reiss first pitched the idea, I thought it was nonsense. Store raw samples in *RAM*? No indexes? Every query is a full scan? Ridiculous. We were having a hard enough time keeping our performance data small enough to fit on *disk*. What were they thinking?

They were thinking about the future. The inflexibility of the tools we had at the time, and especially their pernicious dependency on clairvoyance, was getting in the way of us doing our job. We had only a few dimensions pre-aggregated and if the problem of the day lay somewhere else, we were stuck. Adding new dimensions was a pain in the neck, and didn't help with the problem you were trying to fix right now. The raw data was either gone or very slow to query.

Storing raw samples, all the metadata and measurements, in a flat table in RAM, makes every possible query equally cheap. It shortens the feedback loop from idea to results down to seconds, and allows truly interactive exploration. The short shelf-life of raw data becomes an advantage, because it limits the amount of expensive storage you need.

Here is the beginning of the README file for a measurement system developed by Square, Inc.

> Cube is a system for collecting timestamped events and deriving metrics. By collecting events rather than metrics, Cube lets you compute aggregate statistics post hoc.

[5] fb.me/scuba www.facebook.com/publications/148418812023978

-- github.com/square/cube

How to build one

The shortest way to describe this kind of storage system is as a search engine that can perform statistical functions. It needs to efficiently store large numbers of "documents" with flexible schemas, probably distributed over many servers. It must be able to retrieve all matching documents quickly, then reduce the measurements down to counts, sums, averages, standard deviations, percentiles, and so on. Grouping along the time dimension (eg, average walltime for the last few hours in 5-minute buckets) is also a must.

The systems of this kind I've seen tend to be hybrids: actual search engines taught to do statistics, statistical engines scaled up, or document stores with a layer of aggregation on top. I don't think a name for it has been generally agreed on, but it will probably be something like "analysis engine".

I hesitate to recommend building your own for serious production work; there are plenty of databases and charting libraries in the world already. But understanding their implementation by building a toy version will help illustrate the benefits of a system designed around raw samples.

The hardest part is scaling over multiple machines, especially the statistical functions. Leaving that aside you can hack a prototype without too much fuss using a standard relational database. The statistical functions are often lacking but can be faked with sufficiently clever SQL and post-processing. We'll also disregard the flexible schema requirement because we're going to avoid most of the normal headaches of managing relational database schemas. There will be only one index, on the time dimension; we will rarely delete columns and never do joins.

```
sqlite> .tables
derp     derp_cache     derp_db

sqlite> .schema derp
CREATE TABLE derp (
  timestamp       int,
  script_path     string,
  server_name     string,
  datacenter      string,
  ab_tests        string,
  walltime       int,
  cpu_time        int,
  ...
```

Each of the three logs in our example measurement regime can be represented as a flat, denormalized database table. We can query the tables to see walltime metrics for various kinds of transactions served by our system over, say, the last hour. If the definition of "metric" in the previous chapter on instrumentation sounded a lot like a database query, this is why.

```
sqlite> select script_path, round(avg(walltime)/1000)
  from derp where timestamp >= 1366800000
  and timestamp < 1366800000+3600
  group by script_path;

/account  | 1532.0
/health   | 2.0
/home     | 1554.0
```

```
/pay      | 653.0
/share    | 229.0
/signup   | 109.0
/watch    | 459.0
```

When a user is logged in, the application does a lot more work and fetching of data, for example their preferences and viewing history. The state of the user is then a significant influence on the resources consumed. Averaging in all the light hits from anonymous users can mask problems.

```
sqlite> select user_state, sum(sample_rate),
  round(avg(walltime)/1000), round(avg(db_count))
  from derp where timestamp ...
  and script_path = '/watch'
  group by user_state;

logged-in |  2540 | 1502.0 | 2.3
anon      | 11870 |  619.0 | 0.9
```

Including lots of environmental data in the DERP log lets you look at the performance of larger components of the system. Are the webservers equally loaded? How about the various datacenters they live in? If the dataset is sampled, the sum of the sample_rate field gives you a estimate of the total number of transactions performed.

```
sqlite> select server_name, sum(sample_rate),
  round(avg(walltime)/1000)
  from derp where timestamp ...
  group by server_name;
```

```
web1 | 19520 | 1016.0
web2 | 19820 | 1094.0
...
```

```
sqlite> select datacenter, sum(sample_rate),
  round(avg(walltime)/1000)
  from derp where timestamp ...
  group by datacenter;

east   | 342250 | 1054.0
west   | 330310 | 1049.0
```

By grouping the time dimension into 300-second buckets, you can see that the `/health` hit, which is used internally to make sure that a web server is still functioning and able to respond, has pretty stable performance, as expected.

```
sqlite> select
    group_concat(round(avg(walltime)/1000, 1), ', ')
  from derp where timestamp ...
  and script_path = '/health'
  group by timestamp/(60*5);

1.5, 2.1, 2.2, 1.3, 1.3, 1.3, 1.4, 1.1, ...
```

Isn't that nice? All of these metrics and thousands more can be generated from the same raw table. The simplicity of the queries helps too. You could easily imagine these numbers plotted on a time series chart. Later on that's just what we'll do.

A brief aside about data modeling

The long-running doctrinal disagreement between relational entity modeling (third normal form, OLTP, etc) and dimensional modeling (star schemas, data warehousing, OLAP, and so forth) is largely about the number of table joins needed to accomplish the task of analysis. In other words, it's an argument over performance.

Well, we know what to do about arguments over performance. "Define the problem and measure" applies just as much to the systems you build for measurement. The flat table structure described above is at the extreme of trading space for time: there are zero joins because all the metadata are attached to the sample. It's a starting point, but one that can take you a long, long, long way.

As your measurement system scales up you will likely not want a totally flat design that attaches obscure metadata like the the server's chipset, model number, rack id, etc to every sample. Instead you might add a small lookup table to the database and join on it, or implement some kind of dictionary compression. The point is that complicating your schema is an *optimization* to be done when the need arises, not because of data model orthodoxy. Don't overthink it.

7: Check Your Yardsticks

"Prudent physicists –those who want to avoid false leads and dead ends– operate according to a long-standing principle: Never start a lengthy calculation until you know the range of values within which the answer is likey to fall (and, equally important, the range within the answer is *unlikely* to fall)."

Hans Christian van Baeyer, *The Fermi Solution*

The people in the hardware store must have thought my father was crazy. Before buying a yardstick, he compared each one to a yardstick he brought with him. It sounds less crazy when you learn he's found differences of over $\frac{1}{4}$ inch per yard, something like 1%. That's a big deal when you are building. It's even worse when you aren't aware there's a problem at all.

The yardstick's reputation is so strong that we use it as a metaphor for other standards, eg the "yardstick of civilization". The entire point of their existence is that they should all be the same size. How could anyone manage to get that wrong? Well, manufacturing tolerances drift. Yardsticks are made from other yardsticks, which are made from others, and so on back in time. Errors propagate. Measurement is trickier than it appears.

The root cause is that no one bothered to check. Measurement software is just as likely to have bugs as anything else we write, and we should take more care than usual. A bug in user-facing code results in a bad experience. A bug in measurement code results in bad *decisions*.

There are simple checks you can do: negative numbers are almost always wrong. One kind of mistake, which would be funnier if it didn't happen so often, is to blindly record a CPU or walltime measurement of 1.3 billion seconds.[6] Other checks are possible between measurements. CPU time should never be greater than walltime in a single-threaded program, and the sum of component times should never be greater than the overall measure.

A harder bug to catch is missing points of instrumentation. This is where a well-factored codebase, with only a few callsites to maintain, really helps. You can also use the layers of your measurement regime to cross-check each other. Let's say DERP logs 1:1 and DERP-DB at 1:100. The weight-adjusted sums of DERP's db_count field and DERP-DB samples should be in close agreement. Ditto for errors and the weighted sums of db_wall_time or db_bytes_in. If the numbers don't add up you may have missed something.

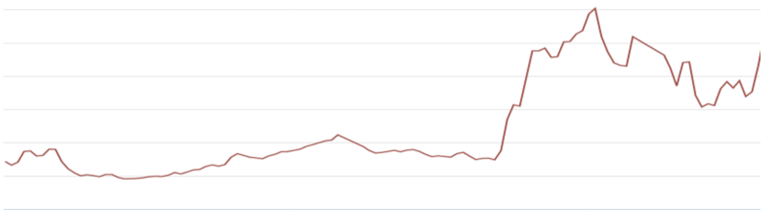
Even better is to log something in two different ways, to bring your own yardstick to the hardware store. To a programmer, having two systems that measure the same thing is duplicated work. To an experimentalist that's just independent confirmation. The more ways you can show something to be true the more likely it *is* true. The second log doesn't have to be fancy. Your database probably has enough statistics built in that you can monitor periodically and compare with your logs.

Or it can be very fancy indeed. For a long time, Jordan Alperin hadn't been satisfied with the error logs that were coming in from certain kinds of client software. A large portion had no information to go on, no line numbers or even error messages. Some had messages apparently localized to the user's language: *"x is niet gedefinieerd"*.

[6] Or perhaps 1.4 billion, depending on when you read this. Duration is measured by recording a Unix timestamp before and after some event, and subtracting start from end. If you're not careful, you can get 0 in either of those values...

Jordan decided to look at where the rubber met the road, a built-in event called window.onerror. It's supposed to pass in three arguments: an error message, the URL of the code, and a line number. The internet being what it is, he suspected that some people were not following the spec. So he rewrote the error handler to accept any number of arguments and log them blindly. It turns out that some clients pass in a single error object, others only two arguments, still others *four*. And, of course, he found flaws in the error-handling code which generated more errors to muddy the waters further.

Running both logging schemes side-by-side established a new lower bound on the number of errors. Only then could Jordan start the task of figuring out what those errors were, including a whole class previously unknown.



On top of the exciting opportunities for writing bugs yourself, the measurements you rely on may not be what they seem. Take CPU time. The idea is simple enough: on mutitasking computers a program can be scheduled in and out multiple times per second, even moved from one core to another. CPU time is the number of milliseconds during which your program was executing instructions on a chip.

But are all milliseconds equivalent? CPUs aren't simple clockwork devices anymore, if they ever were. As we saw with HHVM, some (most!) of the time can be spent waiting for data in RAM, not actually doing work. It's not just your program's cache-friendliness at play, either. A server at 90% utilization behaves very differently from an idle machine with plenty of memory bandwidth. And what happens when you have several generations of servers in your fleet, some 2GHz and some 3GHz? You can't average their measurements together if you want a long-term metric that makes sense. And just in case you were thinking of scaling CPU time by clock speed, modern chips can *adjust their clock speeds* based on all kinds of environmental data.

One way out of that particular hole is to take a second measure of computation alongside CPU time: retired CPU instructions. In general, instruction counts are less dependent on the factors that influence time. This measurement is harder to get; you have to rummage around inside the kernel and special chip vendor features. And not all instructions are equivalent. But for CPU optimization at the application level, it can be worth the trouble to collect.

8: Ontology

"One of the miseries of life is that everybody names things a little bit wrong."

Richard Feynman, *Computers From The Inside Out*

If you have the luxury of building a measurement regime from scratch, make sure you get the names and units right.

A long time ago, back at the beginning of the world, I was helping to build a website to sell computer products. The data was dumped nightly from a COBOL-based mainframe database. The prices made no sense: a cheap mouse listed at 9990. "Oh, that," said the crusty senior engineer. "We store all money data in *mills*. One-thousandth of a dollar. That way we avoid floating-point errors and decimal errors when we do discounts and commissions. It's a real thing; look it up."

He was right. The mill is a legal unit of currency. Even if it wasn't this would be a good idea. As long as everyone uses the same units, and those units have greater precision than you need, whole classes of errors can be avoided. Sometimes the old folks know what they are talking about.

A measurement regime for performance deals in a few major currencies: time, instructions, and bytes. The handiest unit of time is probably the millisecond (*msec*), but sometimes you need more precision. It's not as though storing a measurement as 1 billion microseconds takes more space than 1 million milliseconds. They are all fixed-length integers in the end.

You do have to think about aggregation and overflow. Fortunately an unsigned 64-bit integer can represent over half a million years' worth of time in microseconds (*μsec*). I wouldn't want to go on record as saying 64 bits is enough for anyone, but it should serve for nearly any system you contemplate.

Designing your units to the data types, eg reducing precision to make all expected values fit into a 32-bit integer, is a recipe for regret. Define the precision you want, be generous with yourself, implement it, and then make it more efficient. You can fit 71 minutes' worth of microseconds into an unsigned int32, so it's possible you can get away with storing raw measurements as uint32 and casting to uint64 during aggregation. But keep that optimization in your back pocket for when you really need the space.

Counting raw CPU instructions is probably overkill. One microsecond is enough time to execute a few thousand instructions. We don't want to overflow, and given that a microsecond is precise enough for most purposes, our base unit could be the kiloinstruction, or *kinst*.

Bytes should be stored as bytes. The kilobyte is too coarse for many measurements, and no one remembers to divide or multiply them correctly anyway (1 KB = 1,024 B). There is a slight danger of overflow, however, and some difference in opinion about the proper unit. Network traffic is traditionally measured in bits, 1/8th of a byte. 125 gigabytes per second, measured in bits, would overflow a uint64 counter in seven months. Five years ago a terabit of traffic was nearly unthinkable. Today it's within the reach of many systems. Five years from now you'll doubtless get a terabit for free with your breakfast cereal.

Remember that we're talking about storage, not display. It can be tiresome to deal with large numbers of significant digits. The user interfaces on top of the data should help humans cope by rounding values to megabytes and CPU-years, or otherwise indicating orders of magnitude. When working with large raw numbers I configure my terminal to display millions & billions & trillions in different colors.

Attack of the Jargon

Naming things has been half-jokingly called the second-hardest problem in computer science. Anyone can name the things they build anything they want, and they do. That's the problem. The computer doesn't care about names. They're for the benefit of humans so there are no technical arguments to fall back on. Excess jargon is the sawdust of new technology, and the mental friction it imposes is scandalous. Whoever figures out how to sweep it up does a service to mankind.

Take the word we've been using for intervals of real time, "walltime". Perhaps it's more properly called "duration". Time spent on the CPU could be called "cpu_duration"; time spent waiting for the database "db_duration" and so on. And why not be explicit about the units, eg "duration_cpu_usec"? If you have a strong preference either way, I humbly suggest that it's a matter of taste and not universal truth. Walltime sounds more natural to me because that was the jargon I was first exposed to. But who actually has clocks on their walls any more? The term is as dated as "dialing" a phone number.

For that matter, take instructions. Now that we've decided to round to the nearest thousand, is the name "instructions" misleading? Is "kilo_instructions" too cumbersome to type? Is "kinst" too obscure to remember?

This all might sound mincing and pedantic, but a) you have to pick *something* and b) you'll have to deal with your choices for a long time. So will the people who come after you. You can't clean up the world but you can mind your own patch.

Even an ugly scheme, if it's consistently ugly, is better than having to remember that walltime here is duration over there and response_time_usec somewhere else. Whatever ontology you build, *write it down* somewhere it will be noticed. Explain what the words mean, the units they describe, and be firm about consistency.

9: Visualization

> "A four-year-old *child* could understand this report! (Run out and find me a four-year-old child. I can't make head or tail out of it.)"
>
> Groucho Marx, *Duck Soup*

As useful as it is to keep all those correctly-named, double-checked, raw performance numbers, no one actually wants to look at them. To understand the data, not just user data but any data, we need to summarize measurements over a large range of possible values.

You keep using that word

So far we've been using mean average in all of our metrics. Everyone understands how avg() is calculated: take the sum of the numbers and a count of how many numbers there are, then divide one into the other. But averages lie to us. The moment Bill Gates steps onto a city bus, *on average* everyone is a billionaire.

Let's look at a simple metric, walltime per hit. The average walltime of hits in our DERP dataset, minus the cheap healthchecks, is 439 milliseconds. Does that sound slow? Is it slow? What part is slow? Hard to know.

```
sqlite> select round(avg(walltime)/1000), count(1)
  from derp where script_path != '/health';

439.0 | 3722
```

Let's split the range of values into ten bins of 100 msec each, and measure the percent of samples that fall in each one.

```
select round(walltime/1000/100) * (100) as bin,
    round(count(1)/3722.0, 2) as percent
  from derp
  where walltime/1000 < 1000
    and script_path != '/health'
  group by bin;

0.0   | 0.41
100.0 | 0.17
200.0 | 0.1
300.0 | 0.06
400.0 | 0.04
500.0 | 0.02
600.0 | 0.02
700.0 | 0.02
800.0 | 0.02
900.0 | 0.02
```

This very rough histogram tells us that 40% of samples have a walltime between 0 and 100 msec. It's hard to see from this list of numbers, but the sum of all ten bins is only about 88%, so there are plenty outside our upper limit of 1 second.

This query doesn't give you *percentiles*, which is something slightly different. The median, aka 50th percentile or p50, is the number of milliseconds that is more than half of the samples and less than the other half. Since the first two of ten bins contain 58% of the samples, we can guess that the median value is somewhere near the beginning of the second bin, ie between 100 and 200 msec.

Knowing the median (and any other percentile) more accurately is easy. Just divide the histogram into many more bins, say in increments of 15 msec. Sum from the top until the total of bin percents gets close to 0.5 (or 0.25 or 0.95, etc). It's still an approximation, but the margin is smaller. In this case, the median is somewhere between 120 and 135 msec, and the p75 is close to 405 msec.

```
select round(walltime/1000/15) * (15) as bin,
    round(count(1)/3722.0, 2) as percent
  from derp
  where walltime/1000 < 1000
    and script_path != '/health'
  group by bin;

0.0   | 0.0
15.0  | 0.02
30.0  | 0.1
45.0  | 0.11
60.0  | 0.08
75.0  | 0.06
90.0  | 0.05
105.0 | 0.03
120.0 | 0.03
135.0 | 0.03
150.0 | 0.02
165.0 | 0.02
...
```

So, hiding behind that mean old average number is a much more complex story. Fully 75% of samples have a response time lower than the average. Most of them are grouped around 40 to 60 msec, ten times faster than you would be led to believe by the headline number. Even in plain SQL there are silly tricks to make this distribution more clear.

```
select (walltime/1000/20) * 20 as bin,
    replace(substr(quote(zeroblob(
    (round(count(1)/11) + 1) / 2)), 3,
    round(count(1)/11)), '0', '-')
from derp
where walltime/1000 < 400
  and script_path != '/health'
group by bin;

  0|--
 20|----------------------------
 40|------------------------------------------------
 60|-----------------------------------
 80|------------------------
100|----------------
120|--------------
140|----------
160|-------
180|-------
200|--------
220|-------
240|------
260|----
280|------
300|---
320|----
340|----
360|-----
380|--
```

I warned you it was silly. With sufficient stubbornness it is possible to wring insight out of nothing but a SQL prompt, but in this day and age no one should have to.

Minimal pictures, maximal flexibility

This might appear counterintuitive, but the fewer types of visualizations you add to your tool, the better off you'll be. Remember that the goal is not pretty pictures, it's insight. A visualization tool should first focus on the fluidity of exploration, which in turn depends on the composability of its features. A dataset with ten metadata columns can be thought of as a ten-dimensional space, with one or more measurements residing at the points. Helping the user navigate and collapse that space is the primary task.

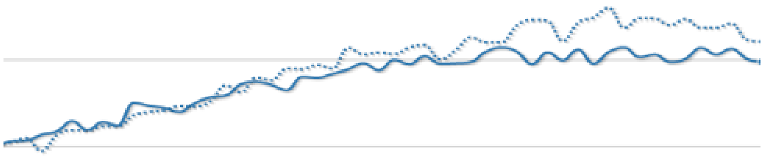
The main activities are to describe trends, show the distribution of a set of samples, to compare sets of samples, and to find correlations between dimensions. As a general principle, if you ever catch yourself doing mental arithmetic, or pointing to two spots on a graph, or (worse) two separate graphs, that means your tool is missing a feature.

Start with a raw table view. All metadata and measurement columns are visible, and each row is a raw sample. The first feature should be the ability to hide the columns that are unimportant to the question being asked. The second feature is to group rows together by the values in the metadata columns. The third is to compute aggregate metrics (counts, averages, percentiles) over the grouped measurements. The fourth is to overlay two tables on top of each other and calculate the differences in each cell. This allows you to compare the performance of two datacenters, or versions of the software, or anything else.

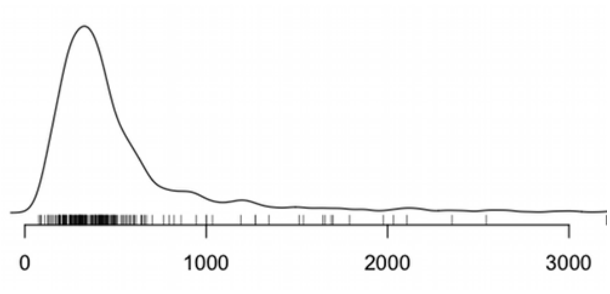
So far what we have isn't much more powerful than a spreadsheet (though spreadsheets are more powerful than they look). One special dimension of the data is time. Controlling the interval of time being queried should require a minimum of clicks, keystrokes, or thought. Humans are pretty bad at understanding raw time values but we are pretty good at spatial reasoning. I've found it's easiest to deal with relative times like "one hour ago" and "last week". Requiring too much precision from the user serves little purpose.

For a given interval of time, say one hour ago to now, calculate the median cpu_time in one-minute buckets. You could display this data as a series of rows, one for each time bucket, but that would be painful for the user to grok. It's better to add a second type of visualization, the time series graph. There have already been many examples of that graph in this book because we're often concerned with showing changes over time.

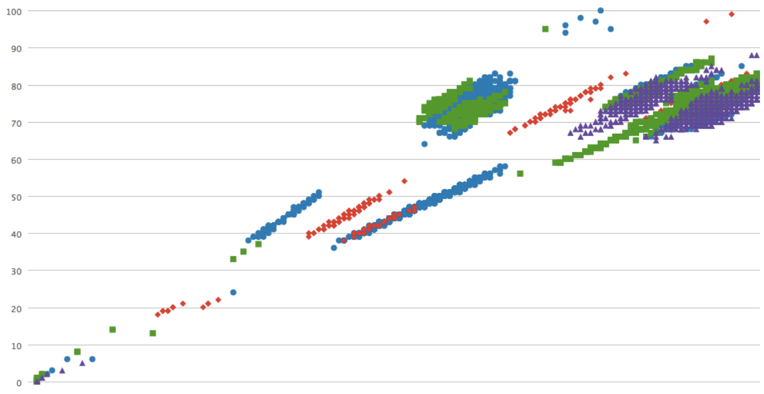
When adding a new type of visualization, it pays to add the same composable features as the rest: hiding columns, grouping samples, calculating metrics, and showing differences. For the time series graph, that last feature can be implemented by drawing a second dotted line, like so:



The third important visualization is the histogram or density graph. Its purpose is to show the full distribution of values in a set of samples, just like the silly SQL graph above. Comparing histograms is tricky to get right, but it's worth the time to experiment to get to something you like.



The fourth great chart is the scatter plot. It's rare for metadata and measurements to be completely independent, and the scatter is a quick way to see whether there is a relationship between them. Here's a chart showing the relationship between CPU utilization and request throughput over a large number of servers. The relationship is nearly linear, as you'd expect from a system that's CPU-bound. The angle of the different "streaks" are interesting, probably evidence of different chip speeds, memory arrangements, or workloads.



Chains of reasoning

Another general principle is to design for *chains of reasoning*, not just individual views of the data. In the course of finding the answer to a question, it's very likely the user will need to run multiple queries, perhaps narrowing down a previous query, or rotating along another dimension or metric, or even backtracking several steps.

The system should be built around the idea of handling the combinatorial explosion of possible metrics and views on the data. Only a few of those views are important, and the raw data ages out relatively quickly. So every query that a human actually looks at is probably special and should be saved: not just the description of the query, but the actual data. If you do it right, this would cost at most a few tens of kilobytes per. Give each saved view a short, unique URL they can share around. Collections of those views are a record of a chain of reasoning.

That's a fancy way of saying "permalink", and it's nothing novel. The ideas in this chapter are only examples. To get the most out of your hard-won data, apply modern user-experience design to the task at hand. If the effort is worth it for your users' experience, it's worth it for yours.

10: Monitoring & Diagnosis

> "Work in the real world involves detecting when things have gone awry; discriminating between data and artefact; discarding red herrings; knowing when to abandon approaches that will ultimately become unsuccessful; and reacting smoothly to escalating consequences."
>
> Richard Cook, *Gaps in the Continuity of Care and Progress on Patient Safety*

The point of measuring all this stuff in production is to know what's going on, and to be able to find the causes of meaningful changes in the metrics. That sounds a lot like operations, doesn't it? With networked applications the line between operations and performance gets blurry.

Monitoring

Whether you call them greenboards, or gauges, or blinkenlights, or heads-up displays, a dashboard is the main fusebox of your system. The metrics on the board should reflect your understanding of how the system behaves and misbehaves.

The dashboard is a really good place to apply that trick of stating theories in terms of what they predict *won't* happen. Finish the following sentence by filling in the blanks:

> While the system is operating normally, the ________ graph should *never* ________.

There will be several good answers, and they will tend to be fundamental things like network traffic, CPU utilization, and transaction volume. Those answers, expressed as graphs on your dashboard, should function like electrical fuses: simple, robust, and not prone to false positives or negatives. A fuse popping conveys only one bit of information, but it's an important bit, and almost always a cause for action.

It's a common mistake to overload your dashboard with too much information. The fanciest hospital monitor can describe something as complex and important as a living being with less than ten metrics. *While the human is operating normally, the heart rate graph should never change drastically, go below 60, or above 100.* You can do the same with a computer system. A dashboard is for monitoring, not diagnosis. It's only job is to tell you that something is wrong, and give a rough clue about where to look next.

That's not to say you can't have lots of graphs. It's fine to have lots of graphs, say memory pressure and CPU, one for every server. What you don't want is too many metrics, too many *kinds* of graphs, confusing the separate duties of monitoring and diagnosis.

Diagnosis

One day you wake up to a popped fuse. The response times on the web tier have gone up by 10%. First you want to find where the increase is concentrated. The assumption is that all things are equal except for one anomaly. It's a decent heuristic as long as you keep in mind that it is a heuristic, and be prepared to change your mind. The more complex and volatile a system, the more often a single surface problem has multiple causes.

Rich metadata and a flexible system for visualization and analysis are your friends. Go back to see when the increase happened. Was is a sharp jump, or something gradual? Note that interval of time with a little bit on either side, so you can refer to it later. Then compare a large set of samples from before and after to see the magnitude of the change. Every hour is different, but per-hit or per-user metrics should be good enough for now.

Comparing those two intervals of time, subdivide the set of samples by some dimensions, say by datacenter, or product or script_path. If you're lucky, the increase will not be across the board but mostly in one of those buckets. A latency bump localized to a datacenter would suggest a systems problem like an overloaded database machine. Product or script_path changes suggest some code or configuration change. Check the log of changes (you have a log of changes, right?) and see whether any of them correspond in time to the performance regression.

If you're not lucky and the rise happens everywhere, that means you're probably looking at the wrong layer of the stack. Till now you've been using gross walltime. Which component of walltime is acting up? In the example system we've been using throughout this book, there are three components: CPU time, database time, and cache fetching time. Graph all three measurements from the DERP dataset and see which one is at fault. From there you would jump to the measurements for that system to narrow down the problem further.

This process of recursively subdividing a pile of measurements, guided by both what you see and your mental model of the system it's describing, is almost but not quite mechanical. It's also not concerned with causes or making up stories about what you think is wrong. Or, if you must think about root causes, flip it around: think about what *can't* be true if the system is misbehaving in the way you think it is. Then try to prove or disprove it using the data.

"I don't believe in the Performance Fairy."

Jeff Rothschild

Notice that I wrote "meaningful changes" and not regressions. Let's say a new version of the application is rolled out, and the average CPU time per hit *improves* by 10%. That's great, right? High-fives all round and break for lunch? Not if you weren't expecting it. There are lots of possible reasons for a sudden performance improvement, and very few of them are good.

- Crashing is cheap. Some transactions might be fataling early. Check the error rates.
- Perhaps there are no errors logged, but no data returned either. Check the overall volume of "network egress", bytes flowing out of the system. Check the average bytes sent out per transaction. Check the distribution.
- Another culprit could be a flood of new, very cheap hits that skew the averages. Internal health checks are a favorite way to cause measurement bugs.
- Did you buy some new, faster servers?
- Did you turn off some old, slower servers?
- Did some servers just die?

These are examples, not hard & fast rules. We're not looking for "the" way to diagnose. It's more like the game of Twenty Questions. Two people will start from different points and go by different routes, but end up at the same conclusion. Uniformity of the process doesn't matter, only convergence. The important thing is to have a flexible, layered measurement regime that allows you to check on your hypotheses, and, like Twenty Questions, quickly eliminate large swathes of the field of possibilities until you find your bogey.

Think hierarchies

Over time you build up knowledge about failure modes. Any time an incident is solved using a non-standard metric, or view, or filter, or what-have-you it should added to a diagnosis tool, which is essentially a lookup table for common paths taken during diagnosis. Here are some metrics that are probably relevant to any large networked application:

- Error rates
- Latency (average, median, low and high percentiles)
- CPU Time / Instructions
- Network bytes in & out
- Requests per second
- Active users
- Active servers
- Server utilization (CPU, RAM, I/O)
- Database queries
- Cache hit / miss rates

Never mind that you've already filled up a giant display and I've surely missed something important. These metrics should be further broken down by country, datacenter, WWW versus API versus mobile, and a half-dozen other dimensions.

Sketch them all out on a big piece of paper or whiteboard, then start to group them into hierarchies. Animal, vegetable, or mineral? What metrics lead to other metrics during diagnosis? How should they be linked? If you could "zoom in" on the data like an interactive map, which small things should become larger? Which metrics assure the accuracy of others?

> When the _______ is operating abnormally, the _______ graph can eliminate _______ as a possible cause.

This "diagnosis tool" is something in between the sparse top-level dashboard and the free-for-all malleable dataset we've built in this book. The minimum possible form it could take is a collection of links to views on the data that have proven useful in the past. There's a good chance you have something like this already. Whether it's in your head, your browser's bookmarks, or your operations runbook, take the time to curate and share it with everyone on your team. It embodies most everything you know about the real behavior of your system.

> "Correlation doesn't imply causation, but it does waggle its eyebrows suggestively and gesture furtively while mouthing 'look over there'."
>
> Randall Munroe, *xkcd.com/552*

11: Wholesale Optimization

"Civilization advances by extending the number of important operations which we can perform without thinking about them."

Alfred North Whitehead, *Symbolism: Its Meaning And Effect*

The most powerful tool of the computer programmer is the computer itself, and we should take every chance to use it. The job of watching the graphs is important but time-consuming and errorful. The first instinct of any programmer who has had that duty is automation. The second instinct is usually to solve it with thresholds, eg 1,400 msec for walltime, and trigger an alarm or email when a threshold is crossed.

For every difficult problem there is a solution which is simple, obvious, and wrong. Red lines, pressure gauges, and flashing lights are familiar movie tropes that create an air of urgency and importance. But, considering everything else Hollywood writers get wrong about computers, we should think very carefully about taking design cues from them.

Alert readers will notice three problems with static thresholds. First, a dependency on clairvoyance snuck in the back door: how do you decide what number to use? Second, there's only one number! The performance characteristics of internet applications vary hour-by-hour. The third problem is more subtle: you probably want to know when the metric you're watching falls *below* the expected range, too.

Static thresholds *do* make sense when making promises about your system to other people. Service Level Agreements (SLAs) often include specific response time promises, eg "The API will always respond in less than 500 msec". But that's business, not science. Declaring an acceptable limit doesn't help you learn what actually happens. You can't have anomaly detection without first discovering what defines an anomaly.

When you come across fancy names like "anomaly detection" and "fault isolation", it's a good bet that the literature is full of fancy ways to implement it. A quick search for that term reveals that you can choose among replicator neural networks, support vector machines, k-nearest neighbor, and many more.

The simplest way to get the job done is to choose a handful of metadata and measurements that tend to have the most impact on performance. Time is the first and most obvious one. Then comes the transaction type (eg script_path), the machine or group of machines (host, datacenter), the version of the software (build_number) and so on. For each combination of those dimensions, create a few metrics that characterize the performance envelope: say, the 25th, 75th, and 99th percentiles of walltime and cpu_time. You want the same kind of information that is on your dashboard, just more of it.

Given those metrics you can determine what values are "normal" for various times of the day by looking at historical data. If the 99th percentile walltime for /foo.php between 2:20PM and 2:40PM for the last few Wednesdays was 455 msec, then throw an alarm if this Wednesday at 2:25pm the metric deviates too much from what the data predicts, on a percentage basis or something fancier like the root mean square error.

How much is too much? You can run experiments to discover how many alarms would have been thrown for a given range. Hold back one day (or week) of historical data from the learning set and run your detection code over it. How much noise you're willing to put up with is up to you.

Garbage in, garbage out

Before you get too excited, I should point out that none of this matters if you don't already have a good mental model of your system's behavior, informed by the data. Take it slowly. Automating a bad manual process will only create a bad automatic process.

It's a common mistake to jump right to the reporting stage and skip over the hard work of defining the problem and discovering the sources of meaningful insight peculiar to the system you are responsible for. Time of day and day of week are *huge* factors in performance. We can guess at what the others will be, but the only way to know for sure is to obsess over the data.

Done right, you'll incrementally replace graph-watching, an important part of your job, with a small shell script. Done wrong, you'll create an autopilot that will happily crash into a mountain. It might happen anyway as conditions change and new factors pop up. Everybody forgets about DNS, kernel versions, and internal network bandwidth until they cause trouble.

However you decide to implement anomaly detection, keep it simple. It doesn't have to be smarter than a human. It only has to be more complete and easily extended.

Don't stop there

What happens after a graph goes wild and an alarm is thrown? It's very often the case that multiple alarms will have one cause, and vice-versa. For example, a sharp increase in cpu_time will probably also cause an increase in walltime. Ditto for a database problem which increases the time spent waiting for the databases to respond to queries. One way to coalesce redundant alarms is by noting the overlaps between the sets of anomalous samples, especially in time.

Another way to make alarms smarter is to burrow deeper into the data. Once an alarm condition is met, say overall walltime, your monitoring system could quickly try many combinations of dimensions in that set of samples to see whether the anomaly can be localized further, say by datacenter or user type. This is similar to the "almost mechanical" recursive subdividing of the dataspace discussed earlier on in the book.

And there goes most of the tedious work: discovering that there is a problem in the first place, and narrowing it down to likely spots.

The hidden benefit of a learning system like this is what it will tell you about the measurement system itself. You will almost certainly find data quality problems, measurement bugs, and math errors. Actually *using* all of the data, not just manually spot-checking pretty graphs, is like running a test suite on your measurement code. And once all those bugs are fixed, you'll have no choice but to accept the fact that the system you're measuring has a lot more variance than you expected, even when it's running "normally".

> "When you successfully use a computer you usually do an equivalent job, not the same old one... the presence of the computer, in the long run, changed the nature of many of the experiments we did."
>
> Richard Hamming, *The Art of Doing Science & Engineering*

12: Feedback Loops

> "A hallucination is a fact, not an error; what is erroneous is a judgment based upon it."
>
> Bertrand Russell, *On The Nature of Acquaintance*

Sooner or later, someone on your team is going to try to feed a system's performance data back into itself, and teach it to react. This is simultaneously a wonderful and terrible idea, fully into the realm of control theory.

The general technique has been around for ages. The float in a toilet tank, which raises with water level and controls the flow of water in, works on the same principle. In the graphics world, closed loop calibration is used to make sure what you see on a monitor is what you get on the printer. Every serious measurement device has calibration designed into it.

There are good places and bad places for this sort of advanced nerdery. Aside from general correctness, feedback loops for distributed computer systems come with three hard problems you have to solve: reaction time, staircase mode, and oscillation.

Reaction time means how quickly the entire system can react to changes in the feedback it's collecting. Gathering 5 minutes' worth of data will obviously limit your loop's reaction time to five minutes, unless you calculate some kind of moving average. There is also propagation delay: we like to pretend that state changes happen in an instant. But the more computers you have, the less and less that's true.

Staircase mode is a quality you want to design for. Elevators don't have great failure modes. Escalators, on the other hand, never really break. They just become staircases. When your feedback loop fails, and it will, think about whether it becomes a staircase or a death trap.

Oscillation happens when one or more components overcompensate. Think of those awkward moments when you and someone walking in the opposite direction get in each other's way. You step to one side but a fraction of a second later they do too. So you step the other way, but before you can react they follow, etc.

Adaptive sampling

Suppose that at peak hour you get so much traffic that your measurement system can't keep up. So you sample down the stream of data, say 1:10. But given that sampling rate, at the trough hour there isn't enough data to make meaningful decisions from. It would be nice to keep the volume of samples constant and vary the sample rates instead. At peak it goes to 1:10; at trough perhaps 1:3.

Implementing this doesn't sound too hard. You maintain a moving average of the sample volume and every few minutes adjust the rate to keep the volume steady. Oh! And build separate sample rates and counters for different datacenters. And also script_paths, so you can ditch uninformative samples in favor of oversampling the rare hits.

On a regular basis you will have to update the sampling configuration across all the servers in your fleet. The delay of collecting the volumes and distributing new rates may be on the order of minutes. If the loop breaks at midnight in the trough, that high sample volume may overwhelm you at the peak. You can mitigate these problems but only through more complication.

In my opinion, a feedback loop for sampling is overkill. Your diurnal cycle of traffic volume is probably well-known and predictable. The ratio of traffic among your datacenters and types of transactions is usually predictable too. You could just look at the last few week's of data and build a curve of "rate multipliers" for every 30-minute chunk of the day. You only need to recalculate that curve every few weeks, at most.

Live load testing

Let's say you want to optimize a system for throughput, eg, how many transactions it can handle through per second, within some bound of acceptable response time. In the production system you may only reach that red line once per day, if ever. You could make up a model in the lab to simulate load by replaying traffic, but that has its own set of problems.

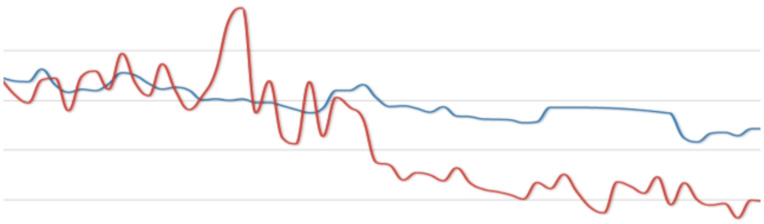
One way is to constantly push a small number of production machines to the red line, with real traffic, and keep them there. The efficiency of your system can be measured by the transactions per second those loaded machines perform. These loaded production boxes can be available all the time for iterative tests, especially configuration tuning.

First you define the red line for your system, the point past which performance is unacceptable. Let's say you think a 500 ms median response time should be the limit.

Now you build a feedback loop between the server you're going to overload and the load balancer that gives it traffic. The load balancer measures the response time over some period. As long as the 30-second median is under 500 ms it will incrementally send the server a larger and larger proportion of the traffic. When the response time goes over, the load balancer backs off a little to help it recover. This works well primarily because the feedback loop is simple and short.

This brings us next to a measurement system developed at Facebook called "Dyno".[7] It helps benchmark the efficiency of the web server tier by holding every factor constant –the code being run, the traffic mix, compiler, hardware, configuration, etc– so that you can change one thing at a time to see how it affects peak performance, as measured by requests per second (RPS).

Let's say there is a new release of the application to test. Push it to *half* of the load test pool, so you are running two versions of the app concurrently. Then you can compare the RPS of the old and new versions side-by-side to see if there is an improvement or regression.



Global traffic routing

You know what could use a giant, complicated feedback loop? Yes. The motherlovin' internet.

When you have a lot of datacenters and users all over the world, an important problem is how to route the right user to the right datacenter. There's a lot of trickery in defining "right".

[7] fb.me/dyno-paper

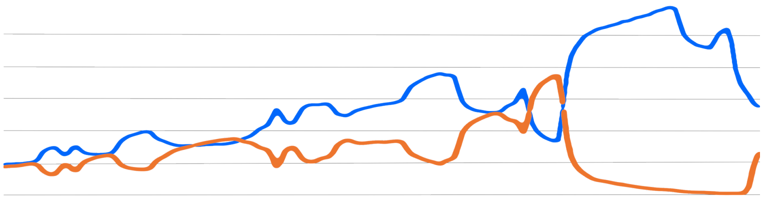
One layer of the "Cartographer" system measures the network latency between users (grouped by DNS resolver) and various datacenters around the planet. No geographic information is used; only packet round-trip times (RTT). Assuming that one datacenter will be faster for most of the users behind a given resolver, you can use your DNS system to send the users to that one. Global internet routing tends to change slowly (though specific routes change quite fast) so the feedback loop can be hours or even days in length.

But life is never that simple. Humans are not evenly distributed around the world, and datacenters have only so much capacity. If you were to direct traffic solely on RTT, odds are good it will be unbalanced, overwhelming whichever datacenter happens to be closest to the most users. So there is a second constraint, the available capacity of each building full of servers to send traffic to.

The feedback loop, then, has to be able to spread load to satisfy both constraints. This is done by dynamically changing the answers your DNS servers give out in response to domain name lookups. If resolver 1234 is closest to datacenter A, but A is at capacity, the DNS server would respond with the IP address of datacenter B, a bit farther away but with RPS to spare. That means the DNS server (more precisely, the system that generates the DNS maps) needs accurate, recent feedback about datacenter health.

So far so good. Let's code that up. How hard could it be?

> “ This is the best one I can find. I sadly don't have the habit of snapshotting crazy graphs when I see them.
>
> The orange and blue lines are two clusters running a really early version of Cartographer. The feedback loop was not well timed; you can see it drastically over/under correcting. This was Cartographer failing to account for DNS propagation delay and data-point delay in our measurement system. The combination caused brutal oscillation.
>
> Orange would shed load and the traffic would dump into blue. Then the inverse would happen a few minutes later when Cartographer finally got data informing it just how screwed up things were.
>
> -- Alex Laslavic



When it was first turned on, Cartographer started sloshing more and more traffic around the internet until Alex had to turn it off. There were two bugs in the system working together.

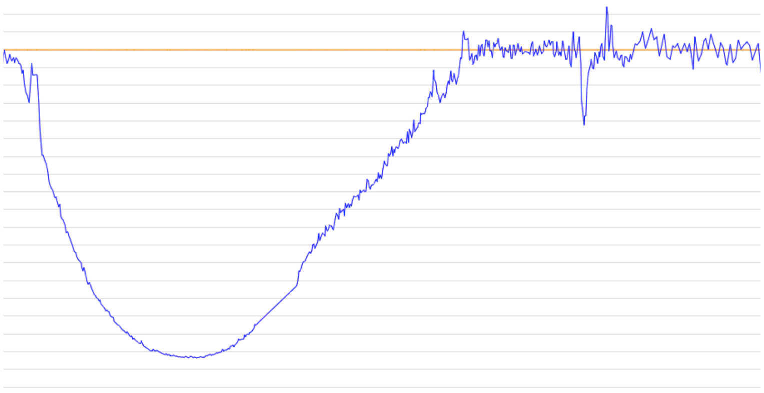
The system which measured capacity utilization was too slow. It took minutes to gather enough data, and dumped it out every 5 minutes. Cartographer would suddenly see a large imbalance and issue a change in the map to correct it. Always too late.

Compounding the problem, and the reason the oscillations increased in magnitude with each swing, was that this early version of the software assumed that its orders would be carried out immediately. Generating the new map and pushing it to the DNS servers was actually pretty fast; they would start answering DNS lookups according to what Cartographer told them to as soon as the file copied over. But DNS *resolvers*, owned not by Facebook but by ISPs around the world, cache the results of old lookups for quite a while. There are many layers of caching all the way down to the user's computer. It's built into the design of the domain name system.

Fixing this feedback loop meant fixing both the response time and Cartographer's assumptions about how the internet works. Straightforward engineering got the measurement system's latency down to less than 30 seconds so Cartographer could work with fresher data.

Then Alex had to measure and model DNS propagation delay, ie, how quickly a change is reflected by the global population of DNS resolvers. He started with a domain that pointed all requests to datacenter A. Then he changed it to point to datacenter B. Then he observed how long it took for the traffic to move over.

It turns out that DNS propagation is reasonably similar to exponential radioactive decay. Cached DNS queries have a measurable half-life. If it takes N minutes for half of the traffic to switch over, at 2N minutes only one-quarter remains, at 3N one eighth, and so on. Incorporating that model into Cartographer allowed it to predict ahead of time how the whole system should look, and refrain from overcorrecting. Here is the chart from a cluster trying to hit a specific volume of traffic.



13: Starting Over

> “Those are my principles. If you don’t like them, I have others.”
>
> Groucho Marx *(attributed)*

Throughout this book we’ve relied on a particular point of view about the nature of performance optimization. It’s time to haul it into the light.

> The goal is to reliably make a system more efficient in time, compute, memory, etc. Only a fraction of the code matters for this, hence the term “bottleneck”. A continuous cycle of measurement to find high-leverage spots and isolated fixes for them works best.

As a problem definition it’s not very good. The test at the end doesn’t really address the assertions, and isn’t obviously falsifiable. There’s a lot of slipperiness packed into the words “reliably” and “best”.

The reductionism on display here should be a red flag to anyone with a sense of history. Breakthroughs happen. You never know when one might be just beyond the horizon. Tweaking sorting algorithms for better performance used to be serious & vital work, something of a competitive sport among programmers. But once Tony Hoare invented quicksort it was all over but the shouting. [8]

[8] Even better results have appeared over the last 55 years, but almost all of them are elaborations or hybrids of the basic idea.

Breakthroughs happen unpredictably. Even if you're only talking about ideas that are not new but new to you, you can't count on finding them when you need to. But we *can* talk about times starting over turned out to be a good idea. Maybe it's possible to recognize signs that a redesign is needed.

Incidental vs. Fundamental

There seem to be two rough categories of bottleneck. Incidental bottlenecks tend to be isolated and fixable without much fuss. Even if they require a lot of clever work to implement, they don't threaten to upend the basic premises of your design, and their magnitudes can be large. This is the happy case. I wish you a long career of easy wins from this end of the spectrum.

Fundamental bottlenecks are harder to fix, and harder to *see*, because they are caused by some fact of nature or an assumption the system is built around. An infinitely-fast network application is still subject to the speed of light. A pizza shop's fundamental bottleneck is the size of its oven. Rockets are limited by the need to lift the weight of their own fuel.

Once a fundamental bottleneck is identified, you have to decide whether it's possible to remove it without collapsing the rest, and whether that's worth doing. You should be able to take an educated guess at the magnitude of the potential win. Systems analysis comes back onto the stage, plus a healthy slug of creativity & judgement.

Software has a present value and future value, however difficult they are to quantify. Rewrites can't be decided solely on technical merits. In theory you could take any existing piece of software, clone it in assembler and end up with a faster program... written in assembler.

NoSQL in SQL

Often the bottleneck is a product of old cruft, ideas the system was shaped to handle but that are no longer necessary. Engineering practice changes surprisingly slowly compared to facts on the ground, but you should be careful to throw out only the bits that are obsolete.

FriendFeed, a link-sharing service, had a database problem. They'd done all the usual things to handle increasing data size and traffic: sharding across many machines, caching when it made sense. It turned out that scaling the software and data model they had wasn't a big deal. The big deal was that their model was hard to change.

> In particular, making schema changes or adding indexes to a database with more than 10-20 million rows completely locks the database for hours at a time. Removing old indexes takes just as much time, and not removing them hurts performance... There are complex operational procedures you can do to circumvent these problems... so error prone and heavyweight, they implicitly discouraged adding features that would require schema/index changes...
>
> MySQL works. It doesn't corrupt data. Replication works. We understand its limitations already. We like MySQL for storage, just not RDBMS usage patterns.
>
> -- Bret Taylor
> backchannel.org/blog/friendfeed-schemaless-mysql

Their business was built around a particular database, and that database had become a bottleneck for performance tuning and adding new features. It was so hard to change the table schemas that they often just didn't do it. There were plenty of other things they liked about MySQL and at the time (2008) there weren't many solid alternatives. So instead of betting the company on a completely new kind of oven, they decided to ditch the features they didn't need through creative abuse of the system.

FriendFeed's initial data model was probably what you'd expect in production relational databases, somewhere in between 2nd and 3rd Normal Form.

```
CREATE TABLE items (
    id string,
    user_id string,
    feed_id string,
    title string,
    link string,
    published int,
    updated int,
    ...

CREATE TABLE users (
   id string,
   name string,
   created int,
   ...
```

They had already sharded these tables over many machines, so an item with id 1234 would live on database A while the owner of that data would live on database Q. That meant joins and subqueries were already off the table, so to speak. Their application was used to running a "query" as set of sharded queries it would assemble later.

Much of the point of row-oriented table schemas is that the data can be stored in a compact form. This in turn makes complex manipulations at the database layer more efficient. The downside is that they can only be changed by dumping the data and loading it all back in. It's another case of trading clairvoyance for speed, and it often makes sense. But if you're not using those features of the database, then there's no need to chain yourself to their limitations.

After lots of testing and discussion, FriendFeed moved to a "bag of words" model that stored their data in an opaque BLOB, and used auxiliary index tables for the dimensions they wanted to index on. All MySQL knew about was an entity's id and timestamp.[9] The BLOB contained binary data (compressed Python objects) that only the application needed to know about.

```
CREATE TABLE entities (
    id BINARY(16) NOT NULL,
    updated TIMESTAMP NOT NULL,
    body MEDIUMBLOB,
    UNIQUE KEY (id),
    KEY (updated)
) ENGINE=InnoDB;
```

At this point they had a key / value store that's efficient for retrieving objects if you know the id, and perhaps for retrieving them by a time range. They could change the "schema" simply by writing new fields into the opaque BLOB field. As long as the application understands the format, the database doesn't care. To make this more useful, say for finding all entries by a given user, they added an index table and populated it.

[9]There was also an autoincrement primary key field to force the storage engine to write entries in the order they were inserted.

```
CREATE TABLE index_user_id (
    user_id BINARY(16) NOT NULL,
    entity_id BINARY(16) NOT NULL UNIQUE,
    PRIMARY KEY (user_id, entity_id)
) ENGINE=InnoDB;
```

Since creating or dropping a table doesn't affect other tables this is quick and atomic. They had a background process read from the entities table and populate the index table. Once that was done, getting all recent entries from a user could be done in two fast queries:

```
SELECT entity_id
FROM index_user
WHERE user_id = [S]

SELECT body
FROM entities
WHERE updated >= [N]
  AND entity_id IN (...)
```

To build the "friend feed" of recent links shared by your friends, they could intersect the results of three queries:

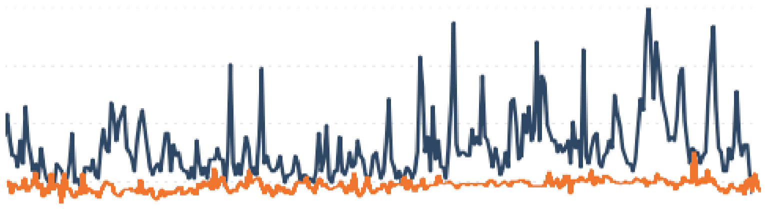
```
SELECT user_id
FROM index_friend
WHERE friend_id = [S]

SELECT entity_id
FROM index_user
WHERE user_id IN (...)
```

```
SELECT body
FROM entities
WHERE updated >= [N]
  AND entity_id IN (...)
```

Complex queries with compound indexes become a series of fast unique-key lookups. The act of adding or deleting indexes was decoupled from serving production traffic. There are many elaborations you could add to a design like this. Most queries have a time component. Adding a timestamp to all of the index tables should shrink the number of entity_ids the application needs to intersect. The important thing is they could now try it and see, by creating a new index table alongside the old one, without first suffering through a week-long schema migration.

The hidden coup of FriendFeed's storage rewrite was increased predictability of the system's performance. Because they were able to use exactly the right unique-key indexes, in the right order, the range of response times got narrower. Before they had been stuck trying to handle all query combinations with a few hard-to-change compound indexes. That meant that many queries were suboptimial, causing large swings in performance.



Shrinking the variance of your system's response time, even if it makes the average *slower*, is a huge win that spreads good effects all over the place. It makes load balancing that much easier because units of work are more interchangable. It reduces the possibility of server "spikes", temporary imbalances, filled queues, etc etc and so forth. If you could magically optimize a single metric in any system, it should be the standard deviation.

Where to Look Next

> "As I've said, much of this advice –in particular, the advice to write a good clean program first and optimize it later– is well worn... For everyone who finds nothing new in this column, there exists another challenge: how to make it so there is no need to rewrite it in 10 years."
>
> Martin Fowler (10 years ago), *Yet Another Optimization Article*

Don't stop now! There are many books on computer performance, but there's also a rich & relevant literature outside our field. For example, *Mature Optimization* does not include very much material about statistics or sampling, because I felt I could not treat the subject as it deserves.

Structured Programming With go to Statements: This paper is the source of the phrase "premature optimization". The title sounds dry today, but it was meant to be provocative, banging together two incompatible ideas, something like "Monadic Programming In Assembler". The holy war over gotos is long dead, but the social causes of the silliness Knuth was trying to counter are very much alive.

Principles of Experiment Design & Measurement: This book is a little gem, an introduction to experimental discipline in the physical sciences. Goranka Bjedov lent me a copy by chance, and precipitated the writing of this book.

How Complex Systems Fail: Richard Cook is one of the most interesting and influential writers about operations research. This is his most famous essay about the nature of failure and

system degradation, which was reproduced in John Allspaw's book *Web Operations*.
`http://www.ctlab.org/documents/Ch+07.pdf`

The Goal: I've always felt that logistics is the closest cousin to computer engineering. They have all the same problems, just with atoms instead of bits. The best introduction is probably Eliyahu Goldratt's *The Goal*. It's written as a novel: the story of factory manager Alex Rogo and how he turns his widget factory around.

Now You See It: Stephen Few writes big beautiful coffee-table books about visualizing data in meaningful ways.

Programmers Need To Learn Stats Or I Will Kill Them All: Zed Shaw's modest thoughts on the utility of measurement and statistics for applied computer science.
`zedshaw.com/essays/programmer_stats.html`

Think Stats: Probability and Statistics for Programmers: This book is intended as an introductory text, and follows a single case study drawing on a large set of demographic data.
`thinkstats.com`

Handbook of Biological Statistics: This is a quite good survey of statistical tools and thinking which came out of the biology department of the University of Delaware.
`udel.edu/m cdonald/statintro.html`

Statistical Formulas For Programmers: Evan Miller has posted many helpful ideas and explanations on his blog, and also wrote *Wizard*, a simple application for exploring and visualizing data.
`evanmiller.org/statistical-formulas-for-programmers.html`

Characterizing people as non-linear, first-order components in software development: In 1999 Alistair Cockburn published one of those rare papers that says everything it needs to say in

the title. Software is not merely made by people; it is made *of* people, and so understanding people is as important as data structures and algorithms.
`a.cockburn.us/1715`

The Art of Doing Science & Engineering: Nearly every time I have had a good idea about what happens at the intersection of humans and computers, it turns out that Richard Hamming already said it, more clearly, before I was born. This book about "learning to learn" was developed from a course he taught for many years at the US Naval Postgraduate School.

Thank You

Too many people to list helped with this book. Special thanks to Bret Taylor, Keith Adams, Alex Laslavic, and Jordan Alperin for permission to quote them in detail. Goranka Bjedov is to blame for the idea of writing a short book on measurement. Adam Hupp, Mark Callaghan, Okay Zed, Shirley Sun, Janet Wiener, Jason Taylor, & John Allen all gave valuable advice and corrections. Boris Dimitrov deserves extra praise for gently leading me realize that I know next to nothing about statistics. This project could not have happened at all without the help of Alex Hollander, Tom Croucher, and Alma Chao.

And thank you for reading. I hope it was interesting.

About the Author

After 18 years in industry, Carlos Bueno has forgotten more about coding and design than someone with a better memory. Most recently he was a member of Facebook's Performance team, saving the company bags of cash through careful measurement and mature optimization. Carlos is also the author of *Lauren Ipsum*, an award-winning children's novel about computer science.

carlos@bueno.org
carlos.bueno.org/optimization