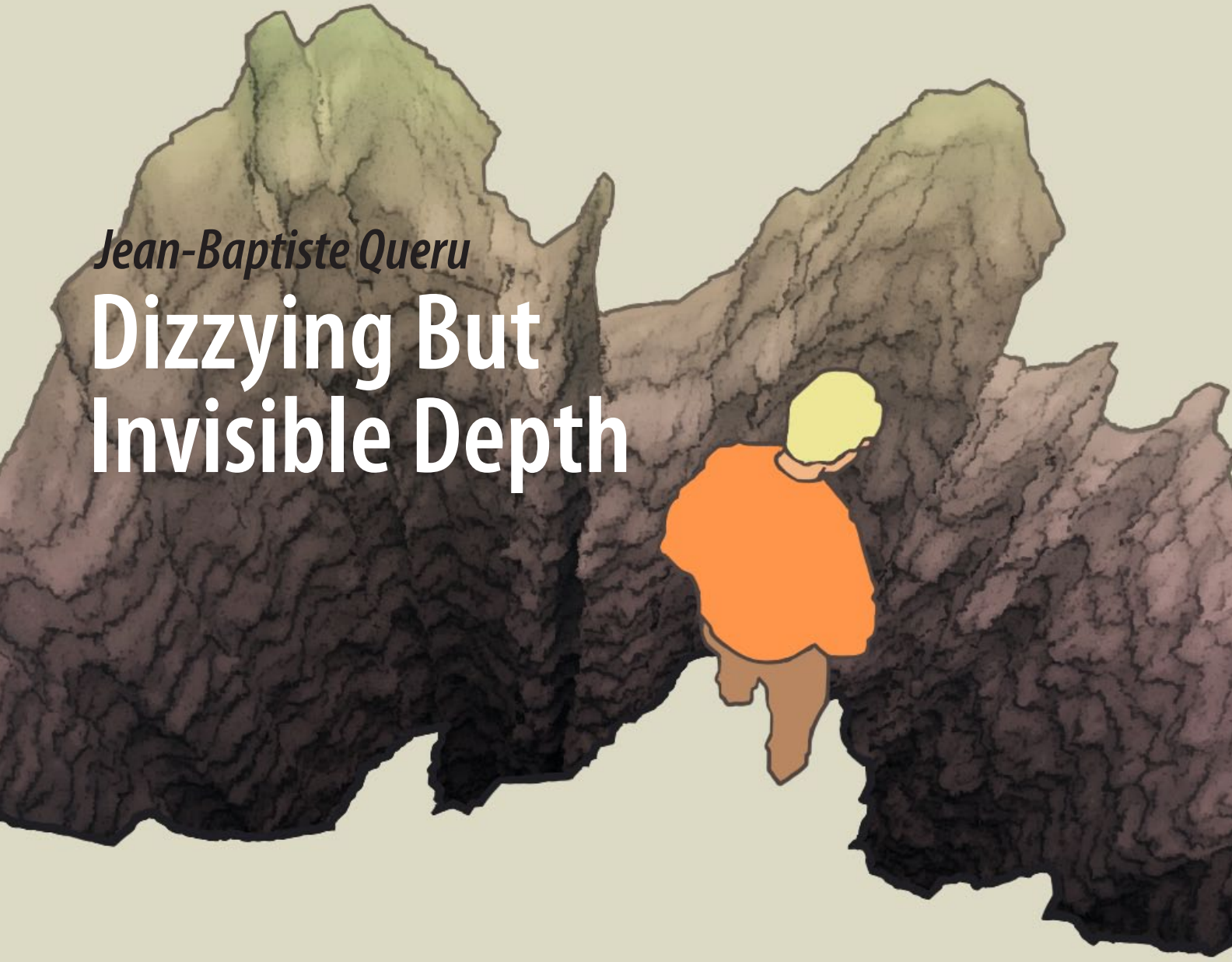


Jean-Baptiste Queru

Dizzying But Invisible Depth

HACKERMONTHLY

Issue 25 June 2012

Our users are saying...

"I enjoy the feeling of being recognized but mostly I enjoy knowing my friends are happy."

"It's cool realising that people thought enough about your contribution to give props."

When your people feel appreciation every day they are **happier**, more **engaged** and **more productive**.

Get started today at dueprops.com

Due Props
Better Recognize

hackernewsletter

Days go by quick. Then an entire week. What did you miss on Hacker News?

Before you see another sunrise subscribe to **Hacker Newsletter**, a weekly email of the best articles from Hacker News curated by hand. Afterwards you won't miss another great article.

Visit http://hackernewsletter.com/hm to subscribe!

Curator
Lim Cheng Soon

Contributors
Jean-Baptiste Queru
Carlos Bueno
Avichal Garg
Rob Fitzpatrick
Eric Karjaluoto
Rohin Dhar
Ed Weissman
Rob Ousbey
Aceex
Carl Lange
Salvatore Sanfilippo
Jeff Atwood
Yan Pritzker
Eric Naeseth
John D. Cook

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrators
Jaime G. Wong
Geona Demyun

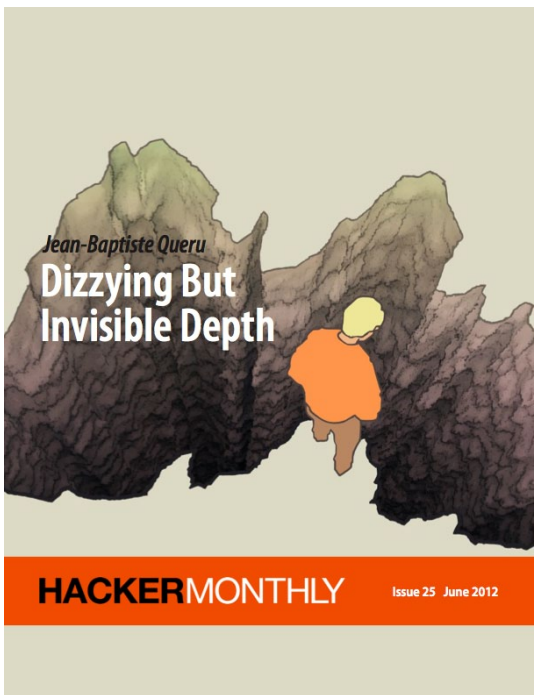
Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

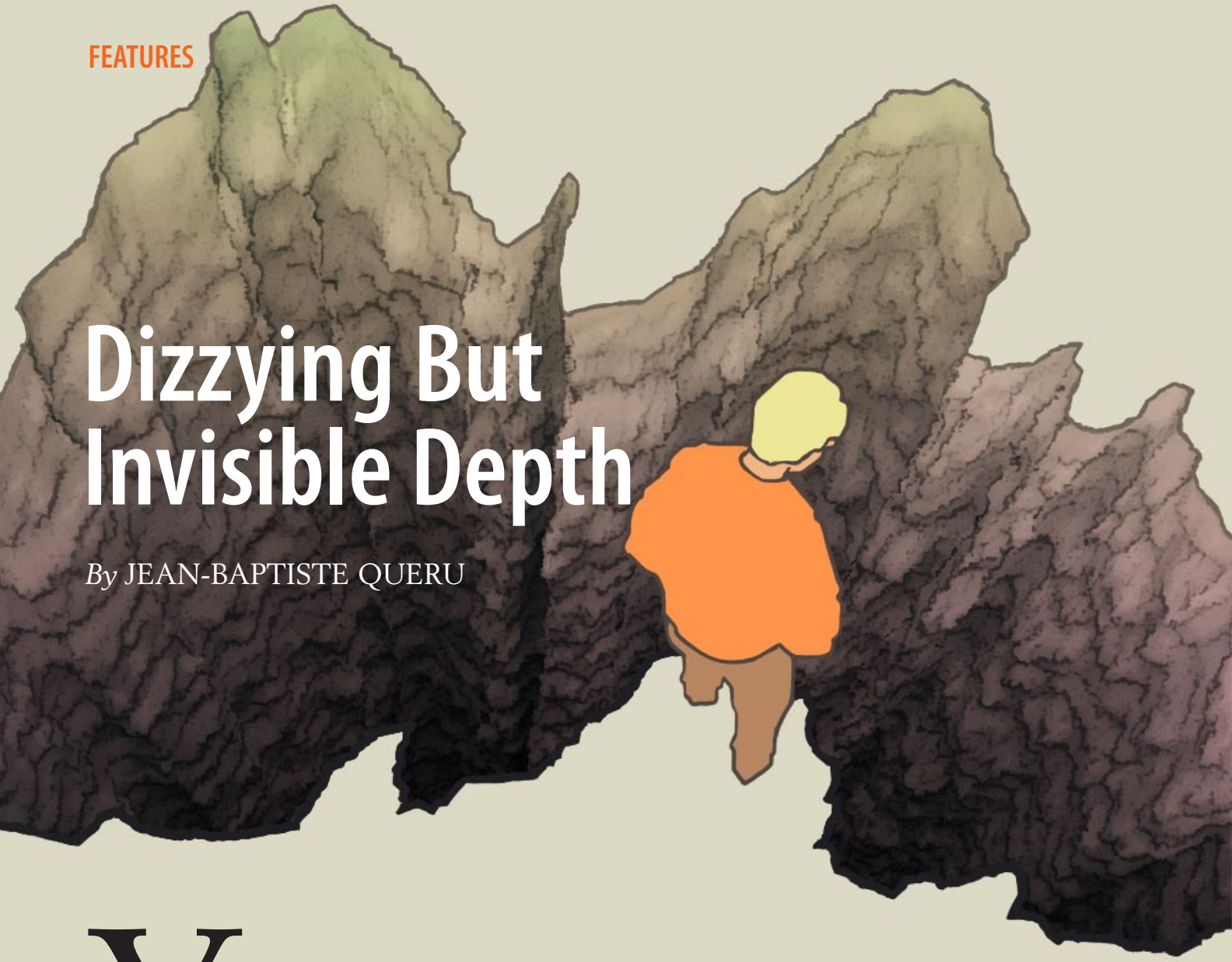
Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Illustration: Jaime G. Wong

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-25*



Dizzying But Invisible Depth

By JEAN-BAPTISTE QUERU

黃國富

You just went to the Google home page.

Simple, isn't it?

What just actually happened?

Well, when you know a bit of about how browsers work, it's not quite that simple. You've just put into play HTTP, HTML, CSS, ECMAscript, and more. Those are actually such incredibly complex technologies that they'll make any engineer dizzy if they think about them too much, and such that no single company can deal with that entire complexity.

Let's simplify.

You just connected your computer to www.google.com.

Simple, isn't it?

What just actually happened?

Well, when you know a bit about how networks work, it's not quite that simple. You've just put into play DNS, TCP, UDP, IP, Wifi, Ethernet, DOCSIS, OC, SONET, and more. Those are actually such incredibly complex technologies that they'll make any engineer dizzy if they think about them too much, and such that no single company can deal with that entire complexity.

Let's simplify.

You just typed www.google.com in the location bar of your browser.

Simple, isn't it?

What just actually happened?

Well, when you know a bit about how operating systems work, it's not quite that simple. You've just put into play a kernel, a USB host stack, an input dispatcher, an event handler, a font hinter, a sub-pixel rasterizer, a windowing system, a graphics driver, and more, all of those written in high-level languages that get processed by compilers, linkers, optimizers, interpreters, and more. Those are actually such incredibly complex technologies that they'll make any

> **“We can barely comprehend the complexity of a single chip in a computer keyboard, and yet there’s no simpler level.”**

engineer dizzy if they think about them too much, and such that no single company can deal with that entire complexity.

Let’s simplify.

You just pressed a key on your keyboard.

Simple, isn’t it?

What just actually happened?

Well, when you know a bit about how input peripherals work, it’s not quite that simple. You’ve just put into play a power regulator, a debouncer, an input multiplexer, a USB device stack, a USB hub stack, all of that implemented in a single chip. That chip is built around thinly sliced wafers of highly purified single-crystal silicon ingot, doped with minute quantities of other atoms that are blasted into the crystal structure, interconnected with multiple layers of aluminum or copper, that are deposited according to patterns of high-energy ultraviolet light that are focused to a precision of a fraction of a micron, connected to the outside world via thin gold wires, all inside a packaging made of a dimensionally and thermally stable resin. The doping patterns and the interconnects implement transistors, which are grouped together to create logic gates. In some parts of the chip, logic gates are combined to create arithmetic and bitwise functions, which are combined to create an ALU. In another part of the chip, logic gates are combined into bistable loops, which are lined up into rows, which are combined with selectors to create a register bank. In another part of the chip, logic gates are combined into bus controllers and instruction decoders and microcode to create an execution scheduler. In another part of the chip, they’re combined into address and data multiplexers and timing circuitry to create a memory controller. There’s even more. Those are actually such incredibly complex technologies that they’ll make any engineer dizzy if they think about them too much, and such that no single company can deal with that entire complexity.

Can we simplify further?

In fact, very scarily, no, we can’t. We can barely comprehend the complexity of a single chip in a computer keyboard, and yet there’s no simpler level. The next step takes us to the software that is used to design the chip’s logic, and that software itself has a level of complexity that requires going back to the top of the loop.

Today’s computers are so complex that they can only be designed and manufactured with slightly less complex computers. In turn the computers used for the design and manufacture are so complex that they themselves can only be designed and manufactured with slightly less complex computers. You’d have to go through many such loops to get back to a level that could possibly be re-built from scratch.

Once you start to understand how our modern devices work and how they’re created, it’s impossible to not be dizzy about the depth of everything that’s involved, and to not be in awe about the fact that they work at all, when Murphy’s law says that they simply shouldn’t possibly work.

For non-technologists, this is all a black box. That is a great success of technology: all those layers of complexity are entirely hidden, and people can use them without even knowing that they exist at all. That is the reason why many people can find computers so frustrating to use: there are so many things that can possibly go wrong that some of them inevitably will, but the complexity goes so deep that it’s impossible for most users to be able to do anything about any error.

That is also why it's so hard for technologists and non-technologists to communicate together: technologists know too much about too many layers, and non-technologists know too little about too few layers to be able to establish effective direct communication. The gap is so large that it's not even possible anymore to have a single person be an intermediate between those two groups. That's why we end up with those convoluted technical support call centers and their multiple tiers. Without such deep support structures, you end up with the frustrating situation that we see when end users have access to a bug database that is directly used by engineers: neither the end users nor the engineers get the information that they need to accomplish their goals.

That is why the mainstream press and the general population have talked so much about Steve Jobs' death and comparatively so little about Dennis Ritchie's Steve's influence was at a layer that most people could see, while Dennis' was much deeper. On the one hand, I can imagine where the computing world would be without the work that Jobs did and the people he inspired: probably a bit less shiny, a bit more beige, a bit more square. Deep inside, though, our devices would still work the same way and do the same things. On the other hand, I literally can't imagine where the computing world would be without the work that Ritchie did and the people he inspired. By the mid 80s, Ritchie's influence had taken over, and even back then very little remained of the pre-Ritchie world.

Last but not least, that is why our patent system is broken: technology has done such an amazing job at hiding its complexity that the people regulating and running the patent system are barely even aware of the complexity of what they're regulating and running. That's the ultimate bikeshedding: just as the proverbial town hall discussions about a nuclear power plant end up being about the paint color for the plant's bike shed, the patents on modern computing systems end up being about screen sizes and icon ordering. In both cases those are the only aspects that the people involved are capable of discussing, even though they are irrelevant to the actual function of the overall system in question. ■

Jean-Baptiste Queru is a software engineer working at Google on the Android Open-Source Project, specializing in large-scale source code management, with 14 years of experience in mobile technologies and operating systems.

Reprinted with permission of the original author. First appeared in *hn.my/dizzy*

Illustration by Jaime G. Wong.

How Bots Seized Control of My Pricing Strategy

By CARLOS BUENO

BEFORE I TALK about my own troubles, let me tell you about another book, *Computer Game Bot Turing Test*. It's one of over 100,000 "books" "written" by a Markov chain running over random Wikipedia articles, bundled up and sold online for a ridiculous price. The publisher, Betascript, is notorious for this kind of thing.

It gets better. There are whole species of other bots that infest the Amazon Marketplace, pretending to have used copies of books, fighting epic price wars no one ever sees. So with *Turing Test* we have a delightful futuristic absurdity: a computer program, pretending to be human, hawking a book about computers pretending to be human, while other computer programs pretend to have used copies of it. A book that was never actually written, much less printed and read.

The internet has everything.

Computer Game Bot Turing Test (10/3/2010)
by Lambert M. Surhone (Editor)

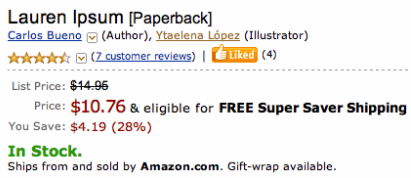
Format	BN.com	Used/New from
Paperback	**$47.00**	$46.99

This would just be an interesting anecdote, except that bot activity also seems to affect books that, you know, actually exist. Last year I published my children's book about computer science, Lauren Ipsum [hn.my/lauren]. I set a price of $14.95 for the paperback edition and sales have been pretty good. Then last week I noticed a marketplace bot offering to sell it for $55.63. "Silly bots", I thought to myself, "must be a bug." After all, it's print-on-demand, so where would you get a new copy to sell?

Then it occurred to me that all they have to do is buy a copy from Amazon, if anyone is ever foolish enough to buy from them, and reap a profit. Lazy evaluation, made flesh. Clever bots!

Then another bot piled on, and then one based in the UK. They started competing with each other on price. Pretty soon they were offering my book below the retail price, and trying to make up the difference on "shipping and handling." I was getting a bit worried.

The punchline is that Amazon itself is a bot that does price-matching. Soon after the marketplace bot's race to the bottom, it decided to put my book on sale! 28% off. I can't wait to find out what that does to my margin. (Update: nothing, it turns out. Amazon is eating the entire discount. This is a pleasant surprise.)



Lauren Ipsum [Paperback]
Carlos Bueno (Author), Ytaelena López (Illustrator)
(7 customer reviews) | Liked (4)
List Price: ~~$14.95~~
Price: $10.76 & eligible for **FREE Super Saver Shipping**
You Save: $4.19 (28%)
In Stock.
Ships from and sold by **Amazon.com**. Gift-wrap available.

My reaction to this algorithmic whipsawing has settled down to a kind of helpless bemusement. I mean, the plot of my book is about how understanding computers is the first step to taking control of your life in the 21st century. Now I don't know what to believe.

It's possible that the optimal price of Lauren Ipsum is, in fact, $10.76, and I should just relax and trust the tattooed hipster who wrote Amazon's pricing algorithm. After all, I no longer have a choice. The price is now determined by the complex interaction of several independent computer programs, most of which don't actually have a copy to sell.

But I can't help but think about that old gambler's proverb: "If you can't spot the sucker, it's you." ■

Carlos Bueno is an engineer at Facebook. He writes occasionally about programming, performance, internationalization, and why everyone should learn computer science.

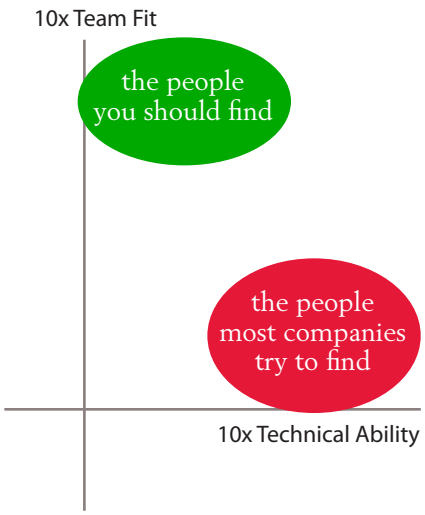
Reprinted with permission of the original author.
First appeared in *hn.my/bot* (bueno.org)

Focus on Building 10x Teams, Not on Hiring 10x Developers

By AVICHAL GARG

There are a lot of posts out there about identifying and hiring 10x engineers. And a lot of discussion about whether or not these people even exist. At Spool [spool.com], we've taken a very different approach. We focused on building a 10x team.

We believe that the effort spent trying to hire five 10x developers is better spent building one 10x team.



Startups optimize hiring for individual rather than team contributors

10x matters because of the Economics of Superstars

The "Economics of Superstars" [hn.my/superstar] observes that in some industries, marginally more talented people/groups generate exponentially more value.

The Economics of Superstars phenomenon requires a distribution channel to move a large volume of goods. For superstar athletes, television enables endorsements and merchandise sales. For software developers, the Internet enables scalable distribution of digital goods.

Finding a way to be 10x better than median can now generate exponentially more value for people who make digital goods.

In software, the superstar is the team, not the individual

In the Economics of Superstars, if an individual has tremendous control over the outcome (points scored in a basketball game), that individual is the beneficiary. So Kobe gets a big chunk of the value he generates for the team, stadium, and advertisers.

Software development, however, is more like rowing. It's a team sport that requires skill and synchronization. This applies at all scales. On a three-person boat, one person out of sync will stall your boat. As you get bigger, no single developer can impact your team's performance, so again synchronization is key.

Making your team as efficient as possible is what determines long-term success.

A bunch of 10x people != A 10x team

Most hiring processes assume that if you find a great developer and put them on a great team, the individual and team will do well. Good teams try to nail down "culture fit," but this is usually only based on whether the candidate gets along with the team.

Throwing together a bunch of great developers who get along does not make for a 10x team.

How to Think About Building a 10x Team

Building a 10x team is a different task than trying to make an existing team 10x more efficient. The hardest part about building a 10x team is that who you need next is a moving target, because it's a function of who is already on the team.

The following are the top three non-technical questions we ask ourselves when considering a candidate:

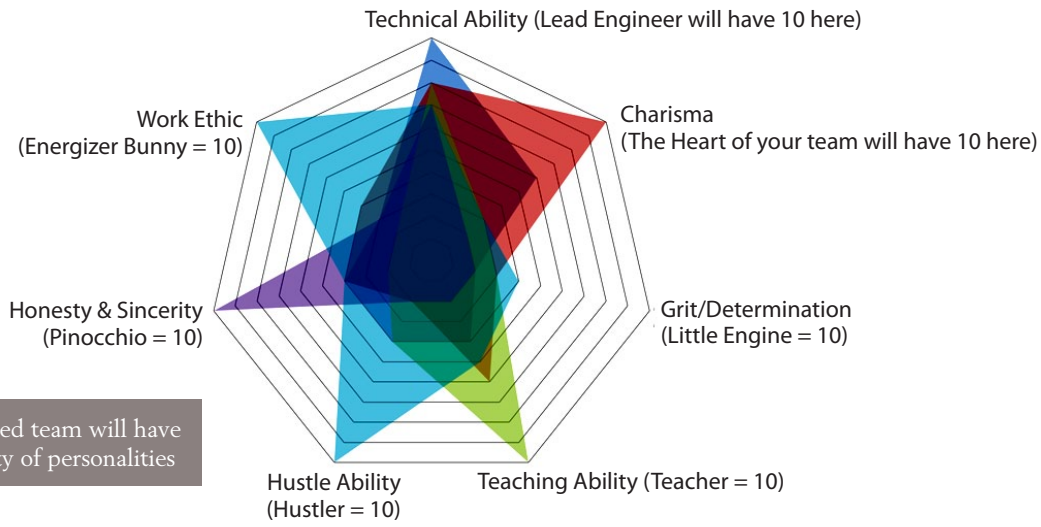
- **Does this person extend the team's one strategic advantage?** Successful startups do NOT have world class design, engineering, sales, and marketing all at once. They tend to be phenomenal at one thing and competent at the rest. Eventually they upgrade talent for "the rest." For example, Zynga first nailed virality with crappy graphics, then later upgraded their art teams.
- **Is there enough shared culture?** Communication overhead will cripple most teams. Hiring people with a common culture is the simplest way to solve this problem. For example, alumni of a university tend to use the same jargon, think similarly, know the same programming languages, etc.. They will communicate naturally and are free to focus on higher-order problems. It's not a surprise that PayPal was mostly UIUC, for example. At Spool we've consciously hired mostly Stanford alums, because Curtis and I are Stanford grads. I apologize if I gave the impression that we don't value diversity. We've gone out of our way to build a diverse team. But there are many things that don't impact your success early on that you can short-circuit by picking people who have a similar enough background. Consider it the Goldilocks Principle [hn.my/goldi].
- **Does this person make other people better?** A friend once told me that the best hire he made was a mistake. Had he properly screened this candidate's technical ability, he wouldn't have hired the candidate. But it turned out this engineer was so driven that he immediately made everyone else on the team more driven. Just by hiring him, the team became more productive, which far outweighed that individual being an average engineer. It's sometimes worth trading off some technical ability to get a multiplier for your whole team.

What sorts of people make other people better?

When we were building Spool's founding team, we looked for people who were technically solid but especially good at making other people around them better. The following are the types of people we identified that do this. There are probably others.

- **The Lead Engineer** sets the technical standard. She will conduct the hardest interviews and will generally work technical magic. She will raise everyone's technical bar. This is usually what someone says when they mean 10x developer.
- **The Hustler** will bend the rules a little when need be, find loopholes in a system, find people you need to find, hack together systems to extract data, and set the standard for just getting things done. She challenges everyone's thinking about how to get things done.
- **The Little Engine That Could** refuses to lose. She manages to do great things through sheer determination. Sometimes she will tell you about this in an interview, but many times you will need to dig into someone's background to get a read on this. She makes everyone else more driven, focused, and makes them believe great things are possible.
- **The Teacher** soaks up and disseminates information. A teacher is constantly learning new technologies or synthesizing large amounts of information. She then distills the critical points and actively shares them with others. She makes everyone more productive almost immediately. This adds up tremendously over the years.
- **The Anti-Pinocchio** is willing to call b.s. on anyone, including the CEO. She is great at spotting b.s. and willing to ask questions of anyone. This keeps a team honest and a company transparent. This is different from being an asshole or a heretic.

In this diagram, each color is a team-member rated from 1-10 on these characteristics. You can see that there's a big hole with no color. I would gladly say no to a traditional 10x engineer to get one person with tremendous grit/determination on this team.



A balanced team will have a diversity of personalities

- **The Energizer Bunny** throws herself into a task fully and doesn't have an off switch. She gets everyone to give 100% and is so enthused that everyone else becomes enthused. She sets the bar for effort and makes everyone want to work harder just so they don't disappoint her. This extends outside work, too. She'll be the first person at the party, the last one to leave, and will make everyone have more fun every day. Happy, enthusiastic teams are productive teams.
- **The Heart** — this is the person on the team that everyone misses when she's not around. She'll bring cookies in for the office, she will remember birthdays, she will make people feel better when they're down, and she will make people do great things because she's just so lovable. People want to come to work to see this person everyday. Just having people look forward to showing up every day is a huge productivity boost.

These personalities all play off each other. For example, a Teacher loves working with an Energizer Bunny because there is someone around to soak up all of that knowledge she shares. Or a Hustler and Lead Engineer can combine to uncover a new distribution channel because they iterate fast and are ruthless. As a result of having these people, you get massive productivity gains from complementary personalities and abilities. Combine these with your favorite/appropriate software development methodologies and you've got a killer team.

I'm sure there are other people who have techniques for building 10x teams. And the dynamics of what makes for a great team are going to be different across industries and stages of a company. ■

Avichal is the co-founder and CEO of Spool, a tool to capture and download any web content to any device. He was previously Co-Founder & CTO of PrepMe, an online education company acquired in 2011, and a product manager at Google on Search Quality and Ads Quality. Avichal has a B.S. and M.S. from Stanford University.

Reprinted with permission of the original author.
First appeared in *hn.my/10x* (avichal.wordpress.com)

My Dad Taught Me Cashflow with a Soda Machine

By ROB FITZPATRICK

After a brief, failed experiment paying me to do chores, my dad tried something really neat. It clearly took a bit of legwork, but maybe there are some transferrable lessons for parents who want to lay an entrepreneurial foundation.

He gave me a vending machine.

He rented the machine, found a location in a local workshop, and installed it. And then he told me two things.

1. That this would be the last time I was given allowance.
2. And that if I wanted to have any pocket money next week, I'd better spend this week's on some inventory.

I ran the machine for about four years from the time I was seven or eight. At first, my only agency was inventory management. We drove to Costco in his big van, and I decided what to buy. Stocking an empty soda machine is easy: you buy four cases of each soda you want to carry.

But then the Coca-Cola runs out first and the Sunkist is half empty and nobody has bought even a single Grape Soda. And should I cut my margins paying more per-unit for individual cans, or do I buy full cases and find somewhere to store the extras — and why am I doing algebra on the weekend!?

Looking back on it, I'm certain this whole endeavor operated at a loss. Dad subsidized it like crazy so I would have a safe — but real — environment to learn in.

At first, it was pure profit: he covered the expenses and I pocketed the take. As long as I did the work each week to buy inventory, count the revenue, and refill the change drawer, I was set.

That didn't last.

Pricing, Cashflow, Operating Costs, and Capital Expenditure

Once I was sitting pretty with my weekly soda profits, it was time for a change.

He let slip that, you know, maybe I could make more money if I raised the prices? After a week of brow-furrowing deliberation, I raised the price per can from $0.50 to $0.55. He told me that some of the customers were angry about the price increase.

I freaked out.

I tried to figure out whether I was earning more now or previously. Why hadn't I been writing all this down? And even if I was making more, how safe was I? Would competition move in and undercut me by that crucial nickel? Would my customers walk across the street to make their soda purchase? Was I being greedy?

He began charging me for the gas we used to drive to Costco. Suddenly I couldn't afford to make re-stocking trips every week — grape soda was cut for an extra column of Coca-Cola. I lost some niche customers. I invested increasing amounts into inventory to reduce gas costs. Our garage became my warehouse.

He wondered aloud if it might be worth buying one of those automated coin-counting machines to speed up my weekly bank trip. I saved up and invested.

And he gave me a taste of the joys and vanities of ownership. Watching someone drop a couple coins into my machine.

And then walking up to it, turning the key, unscrewing the lock, and opening the front. "That's right. I'll be getting my soda for free. I own the joint."

Twenty Year Retrospective

The vending machine didn't magically make me want to be an entrepreneur. I wanted to be a video game designer, then an engineer, then a video game designer again, and then an academic.

I get the impression kids are a bit slippery in that regard.

But when I stumbled into the startup world two decades later, the dots began to connect. Cashflow wasn't a new concept. Inventory tradeoffs made a bit of sense.

This thing with the internet is like that thing with the sodas.

Thanks, Dad. ■

Rob is a tech entrepreneur who moved his first company to London from the valley. He has successfully bankrupted 3 companies, is a YC alum (summer 07), has built products used globally by brands like MTV & Sony, and has raised funding in the US & UK. He blogs about learning (and doing) entrepreneurship at *thestartuptoolkit.com*

Reprinted with permission of the original author.
First appeared in *hn.my/soda* (thestartuptoolkit.com)

Illustration by Geona Demyun (gungriffon-geona.deviantart.com)

Forget Self-Improvement

By ERIC KARJALUOTO

Ever wonder how some people accomplish so much? They run marathons, write novels, start companies…without making it seem like a big deal?

Well, it is a big deal. And in spite of how effortless these accomplishments may appear, people work harder than you likely realize to make these things happen. There is, however, one thing they know — at least in practice — that you don't.

Most of us want to finish the race, but see running as a chore. A few dream about being great authors, but find the writing itself to be slow and difficult. Some of us learn all we can about starting a company, only to hit a wall when it comes time to get down to work.

Self-help books and workshops arm us with ways to trick ourselves into doing things we perhaps should do but generally don't want to. I ask whether this lack of will might actually be the universe trying to tell us something?

Maybe you aren't supposed to bother with the tedious stuff. Perhaps the reason you haven't done it yet is that you weren't meant to. Might achievement as a goal unto itself be pointless? Could this need to have done something notable, simply be greed in a more socially-acceptable form?

More than all of the rest, though: what if the missing part of the puzzle is not a lack of willpower, but instead a lack of love?

The runner discovers tranquility on the road, forgetting the pain. The writer gives in to the joy of playing with words, moving past the aggravation. The entrepreneur finds purpose in making something and stops noticing the long days.

You can spend your life fretting about how healthy, interesting, or successful you are. In fact, a whole industry depends upon this and is eager to help you make plans to change.

On the other hand, you might consider simply finding what you love and letting the rest take care of itself. ■

Eric Karjaluoto is Creative Director at smashLAB [smashlab.com] and writes at Deliberatism [deliberatism.com].

Reprinted with permission of the original author.
First appeared in *hn.my/forget* (deliberatism.com)

Minimum Viable SEO

By ROHIN DHAR

As a startup you only need to be minimally good at SEO right out of the gate. Your goal should be simply "let Google access and understand the pages on my site." You're not going to displace TripAdvisor on day one, but you can position yourself to get traffic from Google early on (or at least not hurt your chances of getting traffic later).

SEO is easiest when you think about it before you start coding. About 2 hours of upfront thought can save you weeks of frustration in the future. Think about SEO when you are sketching out the basic navigational structure of your site on a sheet of paper. There are literally hundreds of "advanced tips" that are best ignored at this stage unless you've done this before. That stuff will eventually matter, but avoid SEO "feature creep" for now.

Priceonomics just launched, and we spent a fair amount of time thinking about SEO before we wrote any code. We did this work up front not just because people look up prices on Google, but also because we remembered the painful lessons of past screw-ups. What we realized was some things matter a lot, and other things don't matter too much right in the beginning but can drive you to distraction.

As we built our site, we focused on 3 basic things to make sure our content would be accessible to Google. If you want a minimally SEO friendly site, copy these 3 things and then you can stop thinking about SEO and focus on your company's core product. It's just one way of doing things, but sometimes it's useful to hear from someone "just do it this way."

3 things that will get you to Minimum Viable SEO so you can launch

1. Decide if you're a "long tail" or "head" site.

If you're a "long tail" site, you want to show up in Google search results when people infrequently type in seemingly obscure queries like "seagate 7200.7 160gb pn#9w2734-133" If you're a head site, you're trying to show up for terms that people search for millions of times a month like "how to ride a bike."

If you're a long tail site the good news is there is a lot of room for you to get traffic early because you're targeting less competitive search terms. The bad news is that how you design your site is a lot more complex because you're going to be creating tens of thousands of pages right from the beginning. Truila, Indeed, and Yelp are long tail sites par excellence worth imitating.

If you're a "head site" you really just need to focus on optimizing your homepage and optimizing for a particular search term like "print online checks," "SF real estate," and "Christmas photocards." Typically it's very hard to rank well for these terms if you're targeting an attractive market. You should pick a term is that is achievable.

As a gross simplification, most mobile apps, social apps, and SAAS companies are "head sites." For example Bump, Farmville, and inDinero are likely focused on a few big important terms. Most "scalable web content companies" like Zillow, AirBnB, and Gogobot live and breathe on the long tail.

2. Use bread crumb navigation

Think of Google as a curious toddler who keeps clicking through each link on your site to satisfy her unquenchable desire to make sense of your website. From any one page that Googlebot is perusing, you want to make it dead simple for it to click through and discover almost all of your content.

One easy way to let Google quickly access any page on your site with the fewest clicks is using a bread crumb navigation. Using this form of navigation forces you to structure your entire website in a way that Google will like. In this form of navigation your content is organized in a browseable hierarchy like:

```
Home > Category > Sub-Category
> Product
```

You'll have clean URLs with important keywords in them, and it will be easy for both users and Googlebot to understand what's going on. If the user removes the last directory from the URL, it should take them the prior directory in the hierarchy seamlessly. Using breadcrumbs forces you to organize your content in a logical and scalable fashion. You'll be happy you did it later when it comes to basic things like maintaining your sitemap or more complicated things like how you spread your link equity across your site.

3. Page Titles

The act of titling the page is more important than the actual page titles (because it's easy to tweak page titles later). It's like learning to write an essay where you have really good topic sentences. By doing so, the reader (Google) can quickly get the gist of what's going on and dive in deeper if they're intrigued.

When you do a Google search, the Big Blue Underlined Links are almost always the site's page titles. They matter because they tell Google "this is what my site is about, please show it if you think it's relevant." Not having accurate page titles is like writing a 5,000 word essay in one paragraph: it's not advisable.

It seems today that search engine results have been taken over by the unscrupulous, but it doesn't have to be that way. You make nice web stuff and it should be accessible to the public through Google. Start with the easy things that make intuitive sense, and get you 80% of the way to being search engine friendly. After you nail that, you're off to the races with analyzing Google webmaster tools account, submitting a kickass segmented sitemap, creating a robots.txt file, building embed widgets, using rel tags where appropriate, setting uniqueness and reading level thresholds for your content, optimizing the anchor text on your inbound links, using advanced tags, and on and on and on.

Don't let perfect be the enemy of good. Go forth and conquer. ■

Rohin Dhar is the co-founder of Priceonomics [priceonomics.com]. He is also the co-founder of Personforce job boards [personforce.com] and has an MBA from Stanford and BA from Dartmouth. You can follow him @rohindhar

Reprinted with permission of the original author.
First appeared in *hn.my/mvs* (priceonomics.com)

How to Participate in Hacker News

By ED WEISSMAN

I recently received an inquiry from a Hacker News newcomer on how to best participate in the community. I was ready to reply, "Just follow the guidelines and be yourself." Then I realized that it was actually a very good question that deserved a much better answer.

So here is my more detailed answer, based upon many years of hard knocks.

First of all, follow the guidelines! [hn.my/guidelines] This is a necessary, but not sufficient condition.

There are literally hundreds of discussions about Hacker News participation just a search away, with much to learn. Hopefully, I can add something new here:

1 Be yourself.

I know that sounds lame, but think about it for a moment. Who else are you going to be? I see no need for "personas." Just be yourself. Talk to others just as if you were in the room with them. Let others see you as your genuine self, full of strengths and areas primed for learning. We can all grow together. Many of us will meet our future in this community.

2 Participate!

I never understood why people lurked so long. No need to be shy here. If you have something to say, say it. If not, then just lurk and learn. But everybody has something of value to share. This is one of the best places to do it.

3 Be positive.

This can really be hard when smart people debate, but try it anyway. Notice the difference between:

> *Person A: Water is dry.*
> *Person B: No, it's not. You're full of shit.*

and

> *Person C: Water is dry.*
> *Person D: Not in my experience. What data have you encountered to cause you to arrive at that conclusion?*

I realize that this is an extreme trivial example, but try to be more like Person D than Person B.

4 Make friends.

Harness the power of the internet! You are not restricted by geography, circumstances, or time period (to some degree). There are many incredible people here who you would likely never meet most other places. Take advantage to the opportunity to meet them, in Hacker News discussion threads, off-line via email, and even in person. Put your contact info in the "about" section of your profile (the "email" is private). Organize and participate in local Hacker News get-togethers. Who knows, your next co-founder, investor, or friend for life may be one or two clicks away.

5 Have something to add.

Again, this may sound obvious and lame, but think about it for a minute. Which comments do you like the most? The ones that add data (which very often translate into value). The key words are "add," "data," and "value". If you have something interesting to add, then please add it. It's not just your right, it's your responsibility!

Everyone wins when you do this: the community gets richer, someone gets value, and you get a bit of a following as an expert in something.

6 Know when to talk and when to listen.

If you have experience doing something being discussed, then by all means, share it! If not then read, listen, and learn. If you have a theory about something but aren't too sure, fine. Just say so. Shocking, but just because you read something on the internets doesn't necessarily mean it's true. And most of all, please never start a sentence with, "It seems to me…." Many of us already get too much of that from our PHBs.

7 The articles may be valuable, but the real gold is in the comments.

If an interesting article posted on Hacker News fell in the forest and no one commented, did it make an impact? Sometimes I post something interesting just to see what you guys will say about it.

Remember:

Good umpire: I call 'em as I see 'em.

Better umpire: I call 'em as they are.

Best umpire: They aren't anything until I call 'em.

Similarly:

Good article: I write 'em as I see 'em.

Better article: I write 'em as they are.

Best article: I'm nothing until the Hacker News community comments on me.

8 Try to focus on your work.

I know this is controversial, but our work is what makes this community what it is. There are debates about all kinds of things here and elsewhere, but remember, our work is our common thread. Frankly, I'm much more interested in what you built, what you encountered when you built it, and what you learned than your opinion about SOPA.

Another old story:

Husband: I am the head of the household! I make all of our family's critical policy decisions on the world's major economic, political, and industrial issues!

Wife: I decide the little things like where we'll live, what we'll eat, and where the kids go to school.

Similarly:

Commenter 1: This major issue can have profound impact on our technological future.

Commenter 2: I don't know much about that, but here's how it took me 9 tries to get my app just right for my audience.

Notice that everyone is right, but I still prefer reading the comments of the second person in each example.

9 Be nice.

Life's too short for anything less. There are many other places any of us could be, but we're here. When people aren't nice to me, I just close my browser and come back another day. I know that sounds silly, but dealing with not nice people is just a big waste of time, and everybody loses. Please don't be that person. ■

Ed Weissman is a programmer, widely known as "edw519" on Hacker News. Check out his free ebook, a compilation of his favorite 256 Hacker News contributions here: *hn.my/edw519*

Reprinted with permission of the original author. First appeared in *hn.my/hnhow* (edweissman.com)

Make Yourself Redundant

By ROB OUSBEY

Early on in my professional life, I was given some fairly terrible career advice: make yourself indispensable.

For decades, this has been the Standard Operating Procedure for people in a variety of roles and industries, from the developer aiming to be the only one capable of maintaining his own code, to the Project Manager who insists that certain processes couldn't run without his oversight. Half a century ago, this kind of behavior might have guaranteed one's lifelong employment. Today, it will harm their career. Worse still, the knock-on effect can undermine the company and even the industry.

I didn't take this advice, but it wasn't until a few years later that I realized why: all along, I'd been working with a different set of assumptions and towards a different goal: to make myself redundant.

My First Job

My first full-time job was working for Shell at their research facility in the North of England. My main responsibility was to run a particular type of test to assess Shell's basic gasoline for its ability to clean your engine whilst you're using it. We would try different additives at different concentrations to improve the fuel. The test that I ran gave a relatively quick indication of the formulation's performance; the best candidates would be tested in a much more rigorous, expensive, time-consuming test.

The test setup I inherited could test 4 samples at a time, and took about 4.5 hours to run. I wasn't asked to improve the process, but my precocious 18-year-old self noticed a few things, including the fact that there was always a backlog of work and that making the test more efficient wouldn't put me straight out of a job.

I pitched my ideas carefully. To the team that requested test runs, I offered the opportunity for faster turnaround. To the team that owned the test, I pitched that I could increase throughput and revenue. (Internal billing was a Big Deal at my campus, and presumably the rest of Shell as well.) Everyone was on board; my boss (let's call him "Chris") offered to provide any necessary resources I needed. I didn't know what this meant.

I loved how simple the first proposed change was. I currently ran 1 test a day (it took ~4.5 hours, remember) so I asked if I could work more flexible hours. Working just a 9-hour instead of an 8-hour day, I was able to run 2 tests per day, and take every other Friday off. I had just improved my throughput from 10 tests every two weeks to 18 tests.

Having experienced the buzz of excitement from my first process improvement, I looked for more inefficiencies. There were a few changes that made my life easier — a quicker setup, improved results analysis, etc. — which freed my time up to focus on other things, but I wanted another opportunity to make as big an impact as the flexi-time suggestion. I suggested to Chris (who, to his credit, was just letting me run with all of this) that we rebuild the test apparatus, so that it was twice the size, and could test twice as many samples per run.

He told me to get a quote from engineering, which I did, and they told us it would cost something like £4000. I forget the exact number, but it was a huge amount to me. (I was earning about £10,000 at the time. I realize now that for a company with R&D budgets like Shell, this was not an amount to trouble over.) I assumed this would never happen, but, to my surprise, Chris said he had no objection, and that I should get it built. He put me in touch with a statistician who was able to help me check that both devices gave reliably similar results.

The personal benefits to me of the changes I instigated were worthwhile. It freed up time for me to work on 'more interesting' projects. I got to spend time & budget on personal development through training (Shell was excellent at providing resources to learn about things beyond our core activities). I was awarded a prize (with cash!) by the EDT, and was invited back to work for the company in the future.

What did not happen? I didn't lose my job. I wasn't asked to take on some menial role. I wasn't told to stop being so productive because I was making other people look bad.

In large part, this is because the organization was supportive and championed this type of activity. I made my colleagues look good. Chris was an enlightened boss. I'd worked, in a small way, to make myself redundant, and had benefited from it. And now I was keen to do it again.

Radio

Next, I wanted to work in broadcasting. After university, I went to work for a small radio station. As a Production Assistant, I did some audio editing, a little research, and other "get onto the bottom rung" tasks. In my first week, I designed a batch file to bulk-convert audio files, which saved me enough time to start recording a few voiceovers and station idents.

One impressive opportunity came when our Managing Director (let's call him "Simon") walked me through the process we used for calculating and paying royalties to content owners; the process was done once per quarter and required a week or two's worth of work from the PA plus a couple of days of time from an external consultant.

"A VBA macro could do this," I said. It took me about as long to write the program as it would to have just calculated the royalties — it saved no time. However, next quarter it saved weeks of time. And the quarter after. Even the external consultant didn't mind that she wasn't needed for this process; we used the same budget to hire her to do more interesting work instead. And the time it freed up for me? I began producing my own shows. It wasn't a huge station, but there I was, still 22 and producing national radio programs with tens of thousands of listeners — all because I saw an opportunity to write a VBA script.

Perhaps obviously, the code was horrible (after all, it was VBA!), but it did the job, and I was free to pursue more ways in which to make myself redundant. I wrote a few lines of bash that removed the necessity for anyone to cover the morning shift (since the only task required was to log on to an FTP server and download a file), I bought a £20 piece of software that did lots of tiresome bulk processing for me, and I persuaded my boss to let me get a new PC (again, a huge expense in my mind) that converted audio about 5-10 times as fast as the p.o.s. I had been using so far.

I freed up enough time to end up presenting my own shows, sitting on pitch planning sessions for BBC programming, and contributing to the group's "future technologies" team.

Did I have a great boss? Yes, absolutely — and his boss (the CEO) was very supportive as well. There were a couple of people like me, and I was pleased to see how they were moved up as they deserved it.

Interestingly, throughout the company as a whole, there were plenty of people working to make themselves "indispensable." At best, their careers had stalled. At worst, they were the cause of some of the company's most fundamental problems. Had they instead worked to make themselves redundant, I have no doubt they would have been twice as happy.

Do it.

How you "make yourself redundant" depends very much on your industry, your role, your responsibilities, etc.

However, I'd offer a few general pieces of advice:

- Tell your boss what you're up to ("I'd like to eliminate this aspect of my role by doing xyz").
- Figure out who might not be happy about what you're up to, and use it as an opportunity to get them on your side.
- Tell your boss what you want to do with the time you free up, and make sure you all talk honestly about what the options are.

- Most tools (software, computers, gasoline testing apparatus) aren't that expensive in the scheme of things (even if it costs a multiple of your monthly salary), and accountants know good ways to amortize that stuff over a period of 2-5 years, so it doesn't really cost that much.

With this mindset throughout my professional life so far, I've probably ended up being underpaid for the work I was actually doing, but that work has always been more interesting and more complex than what my "business card" implied, and I've loved every minute of it.

Epilogue

The anecdotes above are ancient history. For 3 years now I've been working for Distilled. Though I came in on the bottom rung, I now run an office and a team of 13. I'm in a similar role to Chris and Simon, the middle-managers who I looked up to and who helped me realize my endeavors. (Except that I'm younger, less experienced, and have more hair than the two of them put together.)

I'm not convinced that this strategy is equally applicable as you work your way up the "corporate ladder." But I do believe it is now my turn to help my most junior employees make themselves redundant and get promoted in the process. Give it a couple of years, and I'll write about how the process goes from this side of the table. ■

Rob Ousbey is VP Operations at Distilled [distilled.net], where he is working to make himself redundant. He is a co-founder of *Linkstant.com*, an instant back-link alert tool for webmasters.

Reprinted with permission of the original author.
First appeared in *hn.my/redundant* (ousbey.com)

Producer vs. Consumer

By ACEEX

I MAKE SURE TO start every day as a producer, not a consumer. When you get up, you may start with a good routine, like showering and eating, but as soon as you find yourself with some free time you probably get that urge to check Reddit, open that game you were playing, see what you're missing on Facebook, etc.

Put all of this off until "later." Start your first free moments of the day with thoughts of what you really want to do; those long-term things you're working on or even the basic stuff you need to do today, like cooking, getting ready for exercise, etc. This keeps you from falling into the needy consumer mindset where you find yourself endlessly surfing Reddit, Facebook, etc., trying to fill a void in yourself, trying to find out what you're missing, but never feeling satisfied.

When you've started your day with doing awesome (not necessarily difficult) things for yourself, these distractions start to feel like a waste of time. You check Facebook just to make sure you're not missing anything important directed at you, but scrolling down and reading random stuff in your feed feels like stepping out into the Disneyland parking lot to listen to what's playing on the car radio — a complete waste of time compared to what you're really doing today.

It sounds subtle, but these are the only days where I find myself getting anything done. I either start my day like this and feel normal and productive, or I look up and realize it's early evening, I haven't accomplished anything, and I can't bring myself to focus no matter how hard I want to. ■

Aceex has been a redditor for over one year. He frequents /r/Paleo, /r/fitness, /r/vim and /r/GetMotivated.

Reprinted with permission of the original author.
First appeared in *hn.my/producer*

Do Things, Tell People

By CARL LANGE

These are the only things you need to do to be successful. You can get away with just doing one of the two, but that's rare, and usually someone else is doing the other part for you.

If you don't have any marketable skills, learn some. It's the future. We have Khan Academy and Wikipedia and Codecademy and almost the entire world's collective knowledge at your fingertips. Use it.

Then make something that you can talk about. Make something cool. Something interesting. Spend time on it. Go crazy. Even if it's the least useful thing you've ever made, if you can talk about it, make it. This part is easy, because you're doing something you think is cool, and interesting, and if it's useless, great, because you won't need to support it much either!

Next, find events where the people you want to work with are. Then get a drink into you (or don't) and talk to them about it. Relax. It's probably interesting to them, too. Even if it's not, because you've made it, you sound like you know what the hell you're talking about. That's the important part. This is easy, too, because you're talking about something you've made that you think is cool and interesting. As an added bonus, many people go to these events just to talk about cool and interesting things, so you'll fit right in.

You would not believe how much opportunity is out there for those who do things and tell people. It's how you travel the entrepreneurial landscape. You do something interesting and you tell everyone about it. Then you get contacts, business cards, email addresses. Then you get contracts, job offers, investors, whatever. You make friends who think what you do is cool. You make a name for yourself as "the person who did that cool thing." Then, the next time someone wants to do something in any way related to that cool thing, they come to you first.

Ciarán McCann and I (mostly him) started working on a HTML5 game engine and blog [flax.ie] when we were in first year of college. We never even finished it, but because of Flax, we landed internships at Ericsson in the summer of our second year. Now I'm on my way to Game Closure, and Ciarán is going to Demonware. We just did things and told people. ■

Carl is a computer games development student from Ireland, currently working for Game Closure in Mountain View. In his spare time, he tries to make interesting and mostly useless things, and likes to bore people on twitter as @csl_

Reprinted with permission of the original author. First appeared in *hn.my/dothings* (carl.flax.ie)

Redis Persistence Demystified

By SALVATORE SANFILIPPO

Part of my work on Redis is reading blog posts, forum messages, and the twitter time line for the "Redis" search. It is very important for a developer to understand what users and non-users think about the product he is developing. And my feeling is that there is no Redis feature that is as misunderstood as its persistence.

In this article I'll make an effort to be truly impartial: no advertising for Redis and no attempt to skip the details that may put Redis in a bad light. All I want is simply to provide a clear, understandable picture of how Redis persistence works, how reliable it is, and how it compares to other database systems.

The OS and the disk

The first thing to consider is what we can expect from a database in terms of durability. In order to do so, we can visualize what happens during a simple write operation:

1. The client sends a write command to the database (data is in client's memory).
2. The database receives the write command (data is in server's memory).
3. The database calls the system call that writes the data on disk (data is in the kernel's buffer).
4. The operating system transfers the write buffer to the disk controller (data is in the disk cache).
5. The disk controller actually writes the data into a physical media (a magnetic disk, a Nand chip, etc.).

Note: The above is an oversimplification in many ways because there are more levels of caching and buffers.

Step 2 is often implemented as a complex caching system inside the database implementation, and sometimes writes are handled by a different thread or process. However, sooner or later, the database will have to write data to disk, and this is what matters from our point of view. That is, data from memory has to be transmitted to the kernel (step 3) at some point.

Step 3 is also more complex since most advanced kernels implement different layers of caching, which is usually the file system level caching (called the page cache in Linux) and a smaller buffer cache that contains the data that waits to be committed to the disk. It is possible to bypass both using special APIs (see for instance O_DIRECT and O_SYNC flags of the open system call on Linux). However, we can consider this a unique layer of opaque caching (that is, we don't know the details). Often the page cache is disabled when the database implements its caching to avoid that both the database and the kernel try to do the same work at the same time (with bad results). The buffer cache is usually never turned off because this means that every write to the file will result in data being committed to the disk, which is too slow for almost all applications.

What databases usually do instead is call system calls that will commit the buffer cache to the disk only when absolutely needed, as we'll see later in more detail.

When is our write safe along the line?

If we consider a failure that involves just the database software (the process gets killed by the system administrator or crashes) and does not touch the kernel, the write can be considered safe just after step 3 is successfully completed (after the write (2) system call or any other system call used to transfer data to the kernel returns successfully). After this step, the kernel will take care of transferring data to the disk controller even if the database process crashes.

With a more catastrophic event, such as a power outage, we are safe only at step 5 completion, which is when the data is actually transferred to the physical device that will memorize it.

We can summarize that the important stages in data safety are steps 3, 4, and 5:

- How often the database software will transfer its user-space buffers to the kernel buffers using the write (or equivalent) system call.
- How often the kernel will flush the buffers to the disk controller.
- And how often the disk controller will write data to the physical media.

Note: When we refer to disk controller, we actually mean the caching performed by the controller or the disk itself. In environments where durability is important, system administrators usually disable this layer of caching.

Disk controllers by default only perform a write through caching for most systems (i.e. only reads are cached, not writes). It is safe to enable the write back mode (caching of writes) only when you have batteries or a super-capacitor device protecting the data in case of power shutdown.

POSIX API

The path the data follows before being actually written to the physical device is interesting, but even more interesting is the amount of control the programming API provides along the path.

Let's start from step 3. We can use the write system call to transfer data to the kernel buffers, which provides good control using the POSIX API. However, we don't have much control over how much time this system call will take before returning successfully. The kernel write buffer is limited in size; if the disk is not able to cope with the application write bandwidth, the kernel write buffer will reach its maximum size and the kernel will block our write. When the disk can receive more data, the write system call will finally return. After all, the goal is to, eventually, reach the physical media.

In step 4, the kernel transfers data to the disk controller. By default it will try to avoid doing it too often because it is faster to transfer it in bigger pieces. For instance, by default Linux will actually commit writes after 30 seconds. This means that if there is a failure, all the data written in the latest 30 seconds can potentially be lost.

The POSIX API provides a family of system calls to force the kernel to write buffers to the disk. The most well-known of the family is probably the fsync system call (see also msync and fdatasync for more information). Using fsync, the database system has a way to force the kernel to actually commit data on disk, but it is a very expensive operation. Additionally, fsync will initiate a write operation every time it is called and there is some data pending on the kernel buffer. fsync() also blocks the process for for the length of time needed to complete the write. If this is not enough, it will also block all the other threads that are writing against the same file on Linux

What we can't control

So far, we learned that we can control steps 3 and 4, but what about step 5? Well, formally speaking we don't have control from this point of view using the POSIX API. In some cases, a kernel implementation may try to tell the drive to actually commit data on the physical media. On the other hand, maybe the controller will instead re-order writes for the sake of speed and will not really write data on disk ASAP, but will instead wait a couple of milliseconds more. This is simply out of our control.

The rest of this article will simplify our scenario to two data safety levels:

- Data written to kernel buffers using the write(2) system call (or equivalent) that gives us data safety against process failure.
- Data committed to the disk using the fsync(2) system call (or equivalent) that gives us, virtually, data safety against complete system failure, like a power outage. Although there is no guarantee because of the possible disk controller caching, this is an

invariant among all the common database systems. Also, system administrators can use specialized tools to control the exact behavior of the physical device.

Note: Not all the databases use the POSIX API. Some proprietary databases use a kernel module that allows a more direct interaction with the hardware. However, the main shape of the problem remains the same. You can use user-space buffers and kernel buffers, but sooner or later there is need to write data on disk to make it safe (and this is a slow operation). A notable example of a database using a kernel module is Oracle.

Data corruption

In the previous paragraphs, we analyzed the problem of ensuring data is actually transferred to the disk by the higher level layers: the application and the kernel. However, this is not the only concern about durability. Specifically, if the internal structure of a database is unreadable or corrupted after a catastrophic event, it requires a recovery step in order to reconstruct a valid representation of data.

For instance, many SQL and NoSQL databases implement some form of on-disk tree data structure that is used to store data and indexes. This data structure is manipulated on writes. If the system stops working in the middle of a write operation, is the tree representation still valid?

In general there are three levels of safety against data corruption:

- Databases that write to the disk representation not caring about what happens in the event of failure, asking the user to use a replica for data recovery, and/or providing tools that will try to reconstruct a valid representation if possible.
- Database systems that use a log of operations (a journal) so they'll be able to recover to a consistent state after a failure.
- Database systems that never modify already written data, but only work in append only mode, so that no corruption is possible.

Now we have all the elements we need to evaluate a database system in terms of reliability of its persistence layer. It's time to check how Redis scores in this regard. Redis provides two different persistence options, which are analyzed below.

Snapshotting

Redis snapshotting is the simplest Redis persistence mode. It produces point-in-time snapshots of the dataset when specific conditions are met. For instance, if the previous snapshot was created more than 2 minutes ago and there are already at least 100 new writes, a new snapshot is created. This condition can be controlled by the user configuring the Redis instance and can also be modified at runtime without restarting the server. Snapshots are produced as a single .rdb file that contains the whole dataset.

The durability of snapshotting is limited to what the user specified as save points. If the dataset is saved every 15 minutes, then in the event of a Redis instance crash or a more catastrophic event, up to 15 minutes of writes can be lost. From the point of view of Redis transactions, snapshotting guarantees that either a MULTI/EXEC transaction is fully written into a snapshot or that it is not present at all (as already stated above, RDB files represent exactly point-in-time images of the dataset).

The RDB file cannot be corrupted because it is produced by a child process in an append-only way, starting from the image of data in the Redis memory. A new RDB snapshot is created as a temporary file and is renamed into the destination file using the atomic rename(2) system call once it is successfully generated by a child process (and only after it gets synched on disk using the fsync system call).

Redis snapshotting does NOT provide good durability guarantees if losing up to a few minutes of data is not acceptable in case of incidents. Its usage is limited to applications and environments where losing recent data is not very important.

However, even when using the more advanced persistence mode provided by Redis, called "AOF" (Append Only File), it is still advisable to turn on snapshotting. Having a single compact RDB file with the whole dataset is extremely useful to perform backups, send data to remote data centers for disaster recovery, or easily roll-back to an old version of the dataset in the event of a dramatic software bug that compromises the contents of the database in a serious way.

It's worth noting that RDB snapshots are also used by Redis when performing a master -> slave synchronization.

One of the additional benefits of RDB is the fact that for a given database size, the number of I/Os on the system is bound, despite the activity on the database. This is a property that most traditional database systems (and the Redis other persistence, the AOF) do not have.

AOF

The AOF is the main Redis persistence option. The way it works is extremely simple: every time a write operation that modifies the dataset in memory is performed, the operation gets logged. The log is produced exactly in the same format used by clients to communicate with Redis, so the AOF can even be piped via netcat to another instance or easily parsed if needed. At restart Redis re-plays all the operations to reconstruct the dataset.

To show how the AOF works in practice, I'll provide a simple example, setting up a new Redis 2.6 instance with AOF enabled:

```
./redis-server --appendonly yes
```

Now it's time to send a few write commands to the instance:

```
redis 127.0.0.1:6379> set key1 Hello
OK
redis 127.0.0.1:6379> append key1 " World!"
(integer) 12
redis 127.0.0.1:6379> del key1
(integer) 1
redis 127.0.0.1:6379> del non_existing_key
(integer) 0
```

The first three operations actually modified the dataset, but the fourth did not because there was no key with the specified name. This is what the AOF looks like:

```
$ cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
$4
key1
$5
Hello
*3
$6
append
$4
key1
$7
 World!
*2
$3
del
$4
key1
```

As you can see, the final DEL is missing because it did not produce any modification to the dataset.

New commands received will be logged into the AOF, but only if they have some effect on actual data. However, not all the commands are logged as they are received. For instance, blocking operations on lists are logged for their final effects as normal non-blocking commands. Similarly, INCRBYFLOAT is logged as SET, using the final value after the increment as payload, so that differences in the way floating points are handled by different architectures will not lead to different results after reloading an AOF file.

So far we know that the Redis AOF is an append-only business, so corruption isn't possible. However, this desirable feature can also be a problematic. In the above example, our instance is completely empty after the DEL operation, but the AOF still holds a few bytes worth of data. The AOF is an always growing file, so how do we deal with it when it gets too big?

AOF rewrite

When an AOF is too big Redis will simply rewrite it from scratch in a temporary file. The rewrite is NOT performed by reading the old one, but by directly accessing data in memory. This allows Redis to create the shortest AOF that is possible to generate and will not require read disk access while writing the new one.

Once the rewrite is terminated, the temporary file is synched on disk with fsync and is used to overwrite the old AOF file.

You may wonder what happens to data that is written to the server while the rewrite is in progress. This new data is simply written to the old (current) AOF file. At the same time, it is queued into an in-memory buffer so that when the new AOF is ready we can write this missing part to it and finally replace the old AOF file with the new one.

As you can see, everything is still append-only, and when we rewrite the AOF, we still write everything inside the old AOF file for all the time needed for the new file to be created. For our analysis, this means we

can simply avoid the fact that the AOF in Redis gets rewritten at all. So the real questions are: how often do we write(2) and how often we fsync(2)?

AOF rewrites are generated only using sequential I/O operations, so the whole dump process is efficient even with rotational disks (no random I/O is performed). This is also true for RDB snapshots generation. The complete lack of Random I/O accesses is a rare feature among databases and is possible mostly because Redis serves read operations from memory, so data on disk does not need to be organized for a random access pattern, but just for a sequential loading on restart.

AOF durability

This article was written to discuss AOF durability. I'm glad I'm here, and I'm even gladder you are still here with me.

The Redis AOF uses a user-space buffer that is populated with new data as new commands are executed. The buffer is usually flushed on disk every time we return back into the event loop, using a single write(2) call against the AOF file descriptor. However, there are actually three different configurations that will change the exact behavior of write(2), and especially, of fsync(2) calls.

These three configurations are controlled by the appendfsync configuration directive, which can have three different values: no, everysec, always. This configuration can also be queried or modified at runtime using the CONFIG SET command to alter it every time you want without stopping the Redis instance.

appendfsync no

In this configuration Redis does not perform fsync(2) calls at all. However, it will make sure that clients not using pipelining (clients that wait to receive the reply of a command before sending the next one) will receive a message that the command was executed correctly only after the change is transferred to the kernel by writing the command to the AOF file descriptor using the write(2) system call.

Because in this configuration fsync(2) is not called at all, data will be committed to disk at the kernel's wish, which is every 30 seconds in most Linux systems.

appendfsync everysec

In this configuration data will be both written to the file using write(2) and flushed from the kernel to the disk using fsync(2) once every second. Usually the write(2) call will actually be performed every time we return to the event loop, but this is not guaranteed.

If the disk can't cope with the write speed and the background fsync(2) call is taking longer than 1 second, Redis may delay the write up to an additional second (in order to avoid that the write will block the main thread because of an fsync(2) running in the background thread against the same file descriptor). If a total of two seconds elapsed, Redis finally performs a (likely blocking) write(2) to transfer data to the disk at any cost.

In this mode Redis guarantees that, in the worst case, everything you write is going to be committed to the operating system buffers and transferred to the disk within 2 seconds. In the average case, data will be committed every second.

appednfsync always

In this mode, if the client does not use pipelining, data is both written to the file and synched on disk using fsync(2) before an acknowledge is returned to the client.

This is the highest level of durability you can get, but it is also slower than the other modes.

The default Redis configuration is appendfsync everysec, which provides a good balance between speed and durability and is almost as fast as appendfsync no).

Why is pipelining different?

The reason for handling clients using pipelining differently is that clients using pipelining with writes are sacrificing the ability to read what happened with a given command before executing the next one in order to gain speed. There is no point in committing data before replying to a client uninterested in the replies before going forward. However, even if a client is using pipelining, writes and fsyncs (depending on the configuration) always happen when returning to the event loop.

AOF and Redis transactions

AOF guarantees correct MULTI/EXEC transactions semantic and will refuse to reload a file that contains a broken transaction at the end of the file. A utility shipped with the Redis server can trim the AOF file to remove the partial transaction at the end.

Note: Since the AOF is populated using a single write(2) call at the end of every event loop iteration, an incomplete transaction can only appear if the disk where the AOF resides is full while Redis is writing.

Comparison with PostgreSQL

So how durable is Redis with its main persistence engine (AOF) in its default configuration?

- Worst case: It guarantees that write(2) and fsync(2) are performed within two seconds.
- Normal case: It performs write(2) before replying to client and performs an fsync(2) every second.

What is interesting is that in this mode Redis is still extremely fast for a two reasons: fsync is performed on a background thread, and Redis only writes in append-only mode, which is a big advantage.

However, if you need maximum data safety and your write load is not high, you can still obtain the best durability in any database system using fsync always.

How does this compare to PostgreSQL, which, with good reasons, is considered a good and very reliable database?

Let's read some PostgreSQL documentation together (Note: I'm only citing the interesting pieces, you can find the full documentation here [hn.my/pgsql] in the PostgreSQL official site).

> ***fsync (boolean)***
>
> *If this parameter is on, the PostgreSQL server will try to make sure that updates are physically written to disk by issuing fsync() system calls or various equivalent methods (see wal_sync_method). This ensures that the database cluster can recover to a consistent state after an operating system or hardware crash.*
>
> *[snip]*
>
> *In many situations, turning off synchronous_commit for noncritical transactions can provide much of the potential performance benefit of turning off fsync without the attendant risks of data corruption.*

So PostgreSQL needs to fsync data in order to avoid corruptions. Fortunately with Redis AOF, we don't have this problem at all because no corruption is possible. So let's check the next parameter, which is the one that more closely compares with Redis fsync policy, even if the name is different:

> ***synchronous_commit (enum)***
>
> *Specifies whether transaction commit will wait for WAL records to be written to disk before the command returns a "success" indication to the client. Valid values are on, local, and off. The default, and safe, value is on. When off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times wal_writer_delay.) Unlike fsync, setting this parameter to off does not create any risk of database inconsistency: an operating system or database crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly.*

Here we have something much similar to what we can tune with Redis. Basically, the PostgreSQL guys are telling you that if you want speed, it is probably a good idea to disable synchronous commits. Similarly, if you want speed in Redis, it is probably a good idea to avoid using appendfsync always.

If you disable synchronous commits in PostgreSQL, you are in a very similar affair as with Redis appendfsync everysec because, by default, wal_writer_delay is set to 200 milliseconds. The PostgreSQL documentation states that you need to multiply it by three to get the actual delay of writes (600 milliseconds), very near to the 1-second Redis default.

Long story short: even if Redis is an in-memory database, it offers good durability compared to other on-disk databases.

From a more practical point of view, Redis provides both AOF and RDB snapshots, which can be enabled simultaneously (this is the advised setup, when in doubt), while offering ease of operations and data durability.

Everything discussed here about Redis durability can be applied not only when Redis is used as a datastore, but also when it is used to implement queues that need to persist on disk with good durability. ■

Salvatore Sanfilippo aka antirez is an Italian computer programmer. He is currently the lead developer of Redis and works for VMware. In the past he focused on security and programming languages.

Reprinted with permission of the original author. First appeared in *hn.my/redis* (antirez.com)

Speed Hashing

By JEFF ATWOOD

Photo credit: flickr.com/photos/cantchangerandy/2561476943

Hashes are a bit like fingerprints for data.

= 79054025 255fb1a2 6e4bc422 aef54eb4

A given hash uniquely represents a file, or any arbitrary collection of data. At least in theory. This is a 128-bit MD5 hash you're looking at above, so it can represent at most 2128 unique items, or 340 trillion trillion trillion. In reality the usable space is substantially less; you can start seeing significant collisions once you've filled the square root of the space, but the square root of an impossibly large number is still impossibly large.

Back in 2005, I wondered about the difference between a checksum [hn.my/checksums] and a hash. You can think of a checksum as a person's full name: Eubediah Q. Horsefeathers. It's a shortcut to uniqueness that's fast and simple, but easy to forge, because security isn't really the point of naming. You don't walk up to someone and demand their fingerprints to prove they are who they say they are. Names are just convenient disambiguators, a way of quickly determining who you're talking to for social reasons, not absolute proof of identity. There can certainly be multiple people in the world with the same name, and it wouldn't be too much trouble to legally change your name to match someone else's. But changing your fingerprint to match Eubediah's is another matter entirely; that should be impossible except in the movies.

Secure hashes are designed to be tamper-proof

A properly designed secure hash function changes its output radically with tiny single bit changes to the input data, even if those changes are malicious and intended to cheat the hash. Unfortunately, not all hashes were designed properly, and some, like MD5, are outright broken [hn.my/md5] and should probably be reverted to checksums.

If you could mimic another person's fingerprint or DNA at will, you could do some seriously evil stuff. MD5 is clearly compromised, and SHA-1 is not looking too great these days [hn.my/sha1].

The good news is that hashing algorithms (assuming you didn't roll your own, God forbid) were designed by professional mathematicians and cryptographers who knew what they were doing. Just pick a hash of a newer vintage than MD5 (1991) and SHA-1 (1995), and you'll be fine — at least as far as collisions and uniqueness are concerned. But keep reading.

Secure hashes are designed to be slow

Speed of a checksum calculation is important, as checksums are generally working on data as it is being transmitted. If the checksum takes too long, it can affect your transfer speeds. If the checksum incurs significant CPU overhead, that means transferring data will also slow down or overload your PC. For example, imagine the sort of checksums that are used on video standards like DisplayPort, which can peak at 17.28 Gbit/sec.

But hashes aren't designed for speed. In fact, quite the opposite: hashes, when used for security, need to be slow. The faster you can

calculate the hash, the more viable it is to use brute force to mount attacks. Unfortunately, "slow" in 1990 and 2000 terms may not be enough. The hashing algorithm designers may have anticipated predicted increases in CPU power via Moore's Law, but they almost certainly did not see the radical increases in GPU computing power coming.

How radical? Well, compare the results of CPU powered hashcat with the GPU powered oclHashcat when calculating MD5 hashes:

```
Radeon 7970     8213.6 M c/s
6-core AMD CPU  52.9 M c/s
```

The GPU on a single modern video card produces over 150 times the number of hash calculations per second compared to a modern CPU. If Moore's Law anticipates a doubling of computing power every 18 months, that's like peeking 10 years into the future. Pretty amazing stuff, isn't it?

Hashes and passwords

Let's talk about passwords, since hashing and passwords are intimately related. Unless you're storing passwords incorrectly, you always store a user's password as a salted hash, never as plain text. Right? Right? This means if your database containing all those hashes is compromised or leaked, the users are still protected — nobody can figure out what their password actually is based on the hash stored in the database. Yes, there are of course dictionary attacks that can be surprisingly effective, but we can't protect users dead-set on using "monkey1" for their password from themselves. And anyway, the real solution to users choosing crappy passwords is not to make users remember ever more complicated and longer passwords, but to do away with passwords altogether.

This has one unfortunate ramification for password hashes: very few of them were designed with such massive and commonly available GPU horsepower in mind. Here are my results on my current PC, which has two ATI Radeon 7970 cards generating nearly 16000 M c/s with MD5. I used oclHashcat-lite with the full range of a common US keyboard — that is, including uppercase, lowercase, numbers, and all possible symbols:

all 6 character password MD5s	47 seconds
all 7 character password MD5s	1 hour, 14 minutes
all 8 character password MD5s	~465 days
all 9 character password MD5s	*fuggedaboudit*

The process scales nearly perfectly as you add GPUs, so you can cut the time in half by putting four video cards in one machine. It may sound crazy, but enthusiasts have been doing it since 2008. And you could cut it in half again by building another PC with four more video cards, splitting the attack space. (Keep going if you're either crazy, or working for the NSA.) Now we're down to a semi-reasonable 117 days to generate all 8 character MD5s. But perhaps this is a worst-case scenario, as a lot of passwords have no special characters. How about if we try the same thing using just uppercase, lowercase, and numbers?

all 6 character password MD5s	3 seconds
all 7 character password MD5s	4 minutes
all 8 character password MD5s	4 hours
all 9 character password MD5s	10 days
all 10 character password MD5s	~625 days
all 11 character password MD5s	*fuggedaboudit*

If you're curious about the worst case scenario, a 12 character all lowercase password is attainable in about 75 days on this PC. Try it yourself; here's the script I used:

```
set BIN=oclHashcat-lite64
set OPTS=--gpu-accel 200 --gpu-watchdog 0 --outfile-
watch 0 --restore-timer 0 --pw-min 6 --pw-max 6
--custom-charset1 ?l?d?s?u

%BIN% %OPTS% --hash-type 0 aaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaa ?1?1?1?1?1?1?1?1?1?1?1?1
```

Just modify the pw-min, pw-max and the custom-charset as appropriate. Or, if you're too lazy to try it yourself, browse through the existing oclHashcat benchmarks others have run. This will also give you some idea how computationally expensive various known hashes are on GPUs relative to each other, such as:

MD5	23070.7 M/s
SHA-1	7973.8 M/s
SHA-256	3110.2 M/s
SHA-512	267.1 M/s
NTLM	44035.3 M/s
DES	185.1 M/s
WPA/WPA2	348.0 k/s

What about rainbow tables?

Rainbow tables are huge precomputed lists of hashes, trading off table lookups to massive amounts of disk space (and potentially memory) for raw calculation speed. They are now utterly and completely obsolete. Nobody who knows what they're doing would bother. They'd be wasting their time. I'll let Coda Hale explain [hn.my/codahale]:

> *Rainbow tables, despite their recent popularity as a subject of blog posts, have not aged gracefully. Implementations of password crackers can leverage the massive amount of parallelism available in GPUs, peaking at billions of candidate passwords a second. You can literally test all lowercase, alphabetic passwords which are ≤7 characters in less than 2 seconds. And you can now rent the hardware which makes this possible to the tune of less than $3/hour. For about $300/hour, you could crack around 500,000,000,000 candidate passwords a second.*

Given this massive shift in the economics of cryptographic attacks, it simply doesn't make sense for anyone to waste terabytes of disk space in the hope that their victim didn't use a salt. It's a lot easier to just crack the passwords. Even a "good" hashing scheme of SHA256(salt + password) is still completely vulnerable to these cheap and effective attacks.

But when I store passwords I use salts so none of this applies to me!

Hey, awesome, you're smart enough to not just use a hash, but also to salt the hash. Congratulations.

```
$saltedpassword = sha1(SALT .
$password);
```

I know what you're thinking. "I can hide the salt, so the attacker won't know it!" You can certainly try. You could put the salt somewhere else, like in a different database, or put it in a configuration file, or in some hypothetically secure hardware that has additional layers of protection. In the event that an attacker obtains your database with the password hashes, but somehow has no access to or knowledge of the salt it's theoretically possible.

This will provide the illusion of security more than any actual security. Since you need both the salt and the choice of hash algorithm to generate the hash, and to check the hash, it's unlikely an attacker would have one but not the other. If you've been compromised to the point that an attacker has your password database, it's reasonable to assume they either have or can get your secret, hidden salt.

The first rule of security is to always assume and plan for the worst. Should you use a salt, ideally a random salt for each user? Sure, it's definitely a good practice, and at the very least it lets you disambiguate two users who have the same password. But these days, salts alone can no longer save you from a person willing to spend a few thousand dollars on video card hardware, and if you think they can, you're in trouble.

I'm too busy to read all this.

If you are a user:

Make sure all your passwords are 12 characters or more, ideally a lot more. I recommend adopting pass phrases, which are not only a lot easier to remember than passwords (if not type) but also ridiculously secure against brute forcing purely due to their length.

If you are a developer:

Use bcrypt or PBKDF2 exclusively to hash anything you need to be secure. These new hashes were specifically designed to be difficult to implement on GPUs. Do not use any other form of hash. Almost every other popular hashing scheme is vulnerable to brute forcing by arrays of commodity GPUs, which only get faster and more parallel and easier to program for every year. ■

Jeff Atwood lives in Berkeley, CA with his wife, two cats, and a whole lot of computers. He is best known as the author of popular blog Coding Horror and the cofounder of Stack Overflow with Joel Spolsky.

Reprinted with permission of the original author. First appeared in *hn.my/hash* (codinghorror.com)

Learn to Speak Vim

By YAN PRITZKER

USING VIM IS like talking to your editor in "verb modifier object" sentences, turned into acronyms:

- Learn some verbs: `v` (visual), `c` (change), `d` (delete), `y` (yank/copy). Although there are others, these are the most important.
- Learn some modifiers: `i` (inside), `a` (around), `t` (till.. finds a character), `f` (find..like, till, except, including the char), `/` (search..find a string/regex).
- Learn some text objects: `w` (word), `s` (sentence), `p` (paragraph), `b` (block/parentheses), `t` (tag, works for html/xml). There are also other text objects available.

To move efficiently in vim, don't try to do anything by pressing keys numerous times. Instead, speak to the editor in sentences:

- Delete the current word: `diw` (delete inside word).
- Change current sentence: `cis` (change inside sentence).
- Change a string inside quotes: `cin"` (change inside quote).
- Change until next occurrence of "foo": `c/foo` (change search foo).
- Change everything from here to the letter X: `ctX`
- Visually select this paragraph: `vap` (visual around paragraph).

If you understand the verbs and objects you're dealing adding a new plug-in and learning a new verb or noun will exponentially increase your productivity, as you can now apply it to all the sentences you already know. It's just like learning a language.

Let's add some new text object plug-ins!

- Install surround.vim: vim-surround [hn.my/vims] — You get a new noun: "surround" (`s` or `S`).
  - Visually select a word and surround it with quotes: `viwS"`
  - Change "surround" from single quote to double quote: `cs'"`
- Install vim-textobj-rubyblock [hn.my/vimr] — You get a new noun: the "ruby block" (`r`).
  - Delete current ruby block: `dir` (delete inside ruby block).
  - Visually select a ruby function: `var` (visual around ruby block).
  - Visually select the innards of a function: `vir` (visual inside ruby block).
- Install tComment [hn.my/tcom] — You get a new verb: "go comment" (`gc`).
  - Comment the current ruby method: `gcar` (go comment around ruby).

Now go out and learn a new verb or noun every day! ■

Yan Pritzker is a Rails hacker, passionate vim user, founder of *planypus.com*, creator of the YADR curated vim/zsh dotfiles project [skwp.github.com/dotfiles], and author of the free Git Workflows book [gitworkflows.com]. You can find him at *yanpritzker.com*

Reprinted with permission of the original author.
First appeared in *hn.my/speakvim* (yanpritzker.com)

A Primer on Python Decorators

By ERIC NAESETH

Python allows you, the programmer, to do some very cool things with functions. In Python, functions are first-class objects, which means that you can do anything with them that you can do with strings, integers, or any other objects. For example, you can assign a function to a variable:

```
>>> def square(n):
...     return n * n
>>> square(4)
16
>>> alias = square
>>> alias(4)
16
```

The real power from having first-class functions, however, comes from the fact that you can pass them to and return them from other functions. Python's built-in `map` function uses this ability: you pass it a function and a list, and `map` creates a new list by calling your function individually for each item in the list you gave it. Here's an example that uses our `square` function from above:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> map(square, numbers)
[1, 4, 9, 16, 25]
```

A function that accepts other function(s) as arguments and/or returns a function is called a higher-order function. While `map` simply made use of our function without making any changes to it, we can also use higher-order functions to change the behavior of other functions.

For example, let's say we have a function which we call a lot that is very expensive:

```
>>> def fib(n):
...     "Recursively (i.e., dreadfully) calculate the nth
Fibonacci number."
...     return n if n in [0, 1] else fib(n - 2) + fib(n - 1)
```

We would like to save the results of this calculation, so that if we ever need to calculate the value for some `n`(which happens very often, given this function's call tree), we don't have to repeat our hard work. We could do that in a number of ways; for example, we could store the results in a dictionary somewhere, and every time we need a value from `fib`, we first see if it is in the dictionary.

But that would require us to reproduce the same dictionary-checking boilerplate every time we wanted a value from `fib`. Instead, it would be convenient if `fib` took care of saving its results internally, and our code that uses it could simply call it as it normally would. This technique is called memoization (note the lack of an "r").

We could build this memoization code directly into `fib`, but Python gives us another, more elegant option. Since we can write functions that modify other functions, we can write a generic memoization function that takes a function and returns a memoized version of it:

```
def memoize(fn):
    stored_results = {}

    def memoized(*args):
        try:
            # try to get the cached result
            return stored_results[args]
        except KeyError:
            # nothing was cached for those args.
            # let's fix that.
            result = stored_results[args] =
            fn(*args)
            return result

    return memoized
```

This `memoize` function takes another function as an argument, and creates a dictionary where it stores the results of previous calls to that function: the keys are the arguments passed to the function being memoized, and the values are what the function returned when called with those arguments. `memoize` returns a new function that first checks to see if there is an entry in the `stored_results` dictionary for the current arguments; if there is, the stored value is returned; otherwise, the wrapped function is called, and its return value is stored and returned back to the caller. This new function is often called a "wrapper" function, since it's just a thin layer around a different function that does real work.

Now that we have our memoization function, we can just pass `fib` to it to get a wrapped version of it that won't needlessly repeat any of the hard work it's done before:

```
def fib(n):
    return n if n in [0, 1] else fib(n - 2) +
fib(n - 1)
fib = memoize(fib)
```

By using our higher-order `memoize` function, we get all the benefits of memoization without having to make any changes to our `fib` function itself, which would have obscured the real work that function did in the midst of the memoization baggage. But you might notice that the code above is still a little awkward, as we have to write `fib` three times in the above example. Since this pattern — passing a function to another function and saving the result back under the name of the original function — is extremely common in code that makes use of wrapper functions, Python provides a special syntax for it: decorators.

```
@memoize
def fib(n):
    return n if n in [0, 1] else fib(n - 2) +
fib(n - 1)
```

Here, we say that `memoize` is acting decorating `fib`. It's important to realize that this is only a syntactic convenience. This code does exactly the same thing as the above snippet: it defines a function called `fib`, passes it to `memoize`, and saves the result of that as `fib`. The special (and, at first, a bit odd-looking) `@` syntax simply cuts out the redundancy.

You can stack these decorators on top of each other, and they will apply in bottom-out fashion. For example, let's say we also have another higher-order function to help with debugging:

```
def make_verbose(fn):
    def verbose(*args):
        # will print (e.g.) fib(5)
        print '%s(%s)' % (fn.__name__, ',
'.join(repr(arg) for arg in args))
        return fn(*args)
        # actually call the decorated function

    return verbose
```

The following two code snippets then do the same thing:

```
@memoize
@make_verbose
def fib(n):
    return n if n in [0, 1] else fib(n - 2) +
fib(n - 1)
```

```
def fib(n):
    return n if n in [0, 1] else fib(n - 2) +
fib(n - 1)
fib = memoize(make_verbose(fib))
```

Interestingly, you're not restricted to simply writing a function name after the `@` symbol: you can also call a function there, letting you effectively pass arguments to a decorator. Let's say that we aren't content with simple memoization, and we want to store the function results in memcached. If we've written a `memcached` decorator function, we could (for example) pass in the address of the server as an argument:

```
@memcached('127.0.0.1:11211')
def fib(n):
    return n if n in [0, 1] else fib(n - 2) +
fib(n - 1)
```

Written without decorator syntax, this expands to:

```
fib = memcached('127.0.0.1:11211')(fib)
```

Python comes with some functions that are very useful when applied as decorators. For example, Python has a `classmethod` function that creates the rough equivalent of a Java static method:

```
class Foo(object):
    SOME_CLASS_CONSTANT = 42

    @classmethod
    def add_to_my_constant(cls, value):
        # Here, `cls` will just be Foo, but if
        # you called this method on a
        # subclass of Foo, `cls` would be that
        # subclass instead.
        return cls.SOME_CLASS_CONSTANT + value

Foo.add_to_my_constant(10) # => 52

# unlike in Java, you can also call a
# class method on an instance
f = Foo()
f.add_to_my_constant(10) # => 52
```

Sidenote: Docstrings

Python functions carry more information than just code: they also carry useful help information, like their name and docstring:

```
>>> def fib(n):
...     "Recursively (i.e., dreadfully) calcu-
late the nth Fibonacci number."
...     return n if n in [0, 1] else fib(n - 2) +
fib(n - 1)
...
>>> fib.__name__
'fib'
>>> fib.__doc__
'Recursively (i.e., dreadfully) calculate the
nth Fibonacci number.'
```

This information powers Python's built-in `help` function. But when we wrap our function, we instead see the name and docstring of the wrapper:

```
>>> fib = memoized(fib)
>>> fib.__name__
'memoized'
>>> fib.__doc__
```

That's not particularly helpful. Luckily, Python includes a helper function that will copy this documentation onto wrappers, called `functools.wraps`:

```
import functools
def memoize(fn):
    stored_results = {}

    @functools.wraps(fn)
    def memoized(*args):
        # (as before)

    return memoized
```

There's something very satisfying about using a decorator to help you write a decorator. Now, if we were to retry our code from before with the updated `memoize`, we see the documentation is preserved:

```
>>> fib = memoized(fib)
>>> fib.__name__
'fib'
>>> fib.__doc__
'Recursively (i.e., dreadfully) calculate the
nth Fibonacci number.'
```

Eric Naeseth is a software engineer at *Thumbtack.com*, a startup of 17 employees based in San Francisco.

Surprises From Numerical Linear Algebra

By JOHN D. COOK

Here are ten things about numerical linear algebra that you may find surprising if you're not familiar with the field.

1. Numerical linear algebra applies very advanced mathematics to solve problems that can be stated with high school mathematics.

2. Practical applications often require solving enormous systems of equations, millions or even billions of variables.

3. The heart of Google is an enormous linear algebra problem. PageRank is essentially an eigenvalue problem.

4. The efficiency of solving very large systems of equations has benefited at least as much from advances in algorithms as from Moore's law.

5. Many practical problems (optimization, differential equations, signal processing, etc.) boil down to solving linear systems, even when the original problems are non-linear. Finite element software, for example, spends nearly all its time solving linear equations.

6. A system of a million equations can sometimes be solved on an ordinary PC in under a millisecond, depending on the structure of the equations.

7. Iterative methods, methods that in theory require an infinite number of steps to solve a problem, are often faster and more accurate than direct methods, methods that in theory produce an exact answer in a finite number of steps.

8. There are many theorems bounding the error in solutions produced on real computers. That is, the theorems don't just bound the error from hypothetical calculations carried out in exact arithmetic but bound the error from arithmetic as carried out in floating point arithmetic on computer hardware.

9. It is hardly ever necessary to compute the inverse of a matrix.

10. There is remarkably mature software for numerical linear algebra. Brilliant people have worked on this software for many years. ■

John D. Cook is an applied mathematician. He lives in Houston, Texas where he works for M. D. Anderson Cancer Center. His interests include numerical analysis and Bayesian statistics.

Reprinted with permission of the original author.
First appeared in *hn.my/algebra* (johndcook.com)

HARVEST

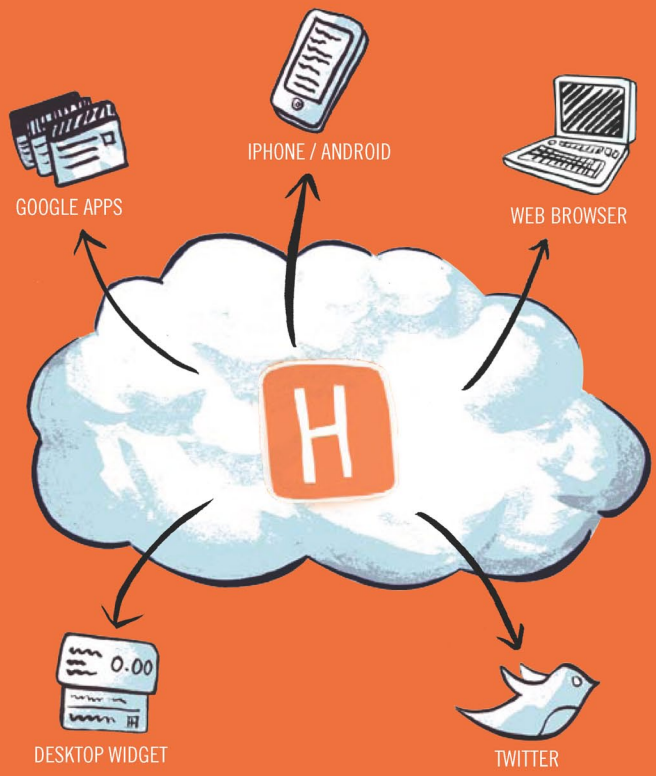
TIMESHEETS INVOICES REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.

Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.



Learn more at **www.getHarvest.com/hackers**

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com