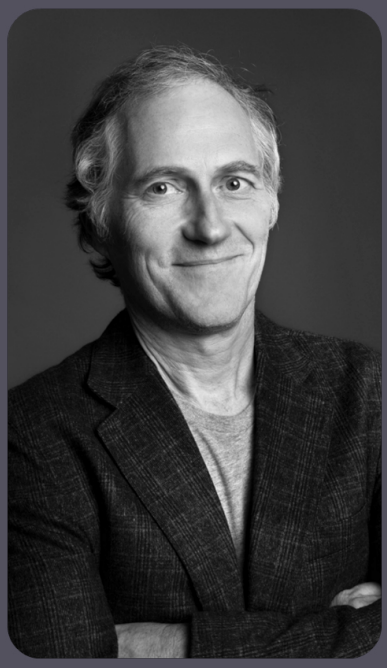









hacker bits

September 2016

new bits

Hello from Redmond!

Welcome to the September issue of *Hacker Bits*! We hope you've had an awesome summer!

In this issue of *Hacker Bits*, we are so thrilled to bring you Tim O'Reilly, who tells us all about the future of work and why we shouldn't fear technology. You can check out the details for an event that explores the same topic at Tim's The Next Economy Summit.

And in the most highly-upvoted article, industry veteran Derek Sivers has some wise advice for those looking to give constructive feedback.

Also, as part of our design revamp, the magazine now features a consistent layout for every article, which we hope would improve readability, especially for those of you who read on mobile devices. Let us know if it's working for you…

Enjoy the much cooler weather!

Peace and see you next issue!

— Maureen and Ray

us@hackerbits.com

content bits

September 2016

6 Don't replace people, augment them

12 Ten rules for negotiating a job offer

22 Hitchhiker trees: functional, persistent, off-heap sorted maps

28 Don't add your 2 cents

30 Why Angular 2 switched to TypeScript

36 Senior engineers reduce risk

40 The truth about deep learning

44 A beginners guide to thinking in SQL

52 AI, Apple and Google

60 The dreaded weekly status email

contributor bits

Tim O'Reilly
Tim is the founder and CEO of O'Reilly Media, Inc. Considered by many to be the best event producer, computer book and video publisher in the world. Visit O'Reilly's platform at www.safaribooksonline.com and www.oreilly.com.

Haseeb Qureshi
Haseeb is a software engineer at Airbnb. He's a former professional poker player, a bootcamp grad, an effective altruist, an English major, and an unreasonably cool guy.

David Greenberg
David is an independent consultant who is interested in high performance software and distributed systems. He's the author of the O'Reilly book *Building Applications on Mesos*. You can find out more about his talks and services at dgrnbrg.com.

Derek Sivers
Derek has been a musician, producer, circus performer, entrepreneur, TED speaker, and book publisher. See his (incredibly useful) site at sivers.org for more.

Victor Savkin
Victor makes Angular. He also toys with eclectic programming technologies and obsesses over fonts and keyboard layouts. You can reach him @victorsavkin.

Zach Tellman
Zach writes code and sometimes not-code. Most recently, he's writing a book called *Elements of Clojure*. You can find information about his open source libraries, talks, and other writing at ideolalia.com.

Clay McLeod
Clay is the co-founder of CM Technology, a company focused on improving supply chain management in the construction industry. He also works on building cutting edge tools machine learning for genomic pipelining.

Soham Chetan Kamani
Soham is a full stack developer passionate about the web. Especially interested in JavaScript, Python, and IOT. You can reach him @sohamkamani.

Benedict Evans
Benedict works at Andreessen Horowitz ('a16z'), a venture capital firm in Silicon Valley that invests in technology companies. He tries to work out what's going on and what will happen next.

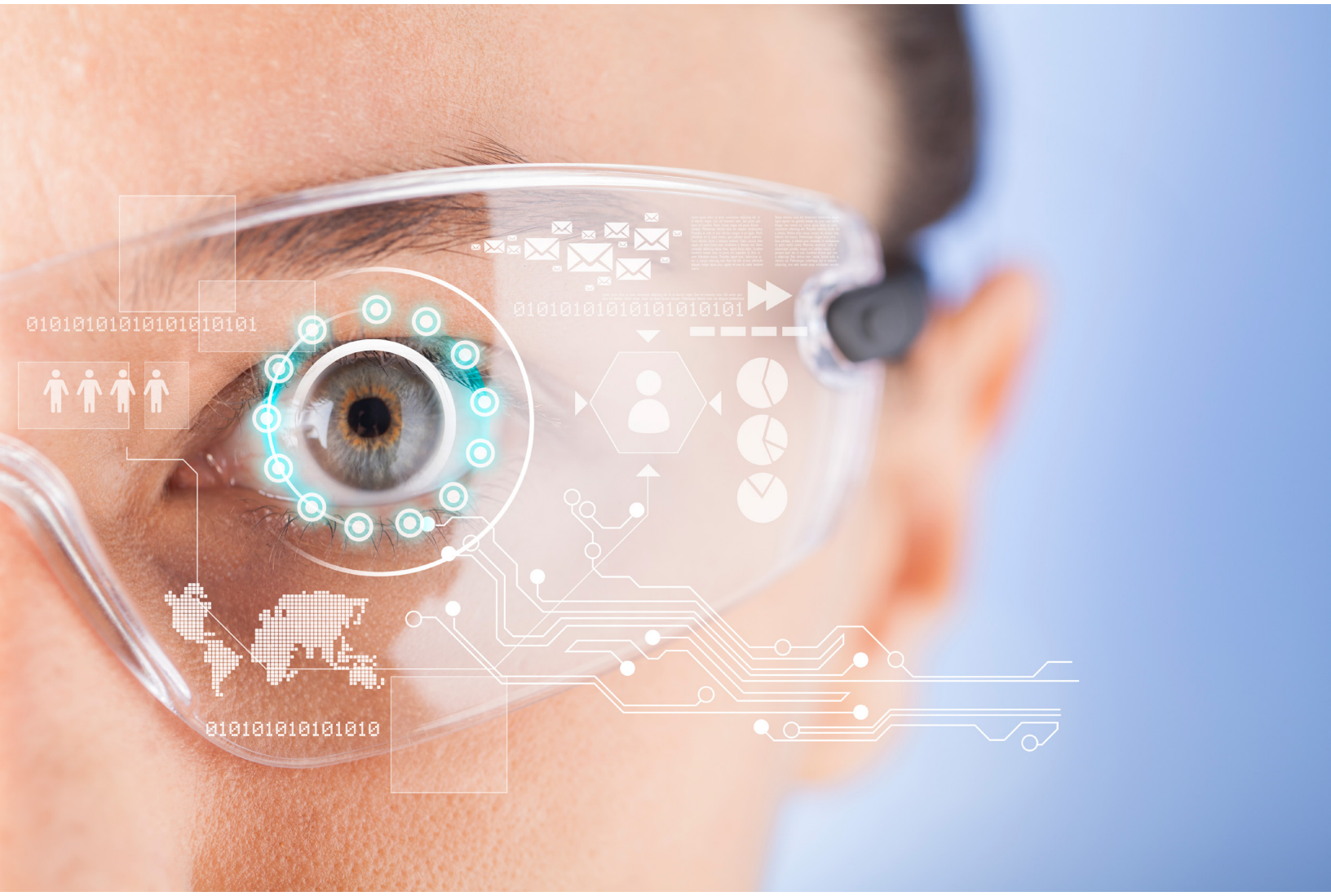
Christina Wodtke
Christina is an author, professor and speaker who teaches techniques to create high performing teams. Her latest book, *Radical Focus* shows how to set better goals and create a rhythm of execution in order to achieve great things.

Ray Li
Curator

Ray is a software engineer and data enthusiast who has been blogging at rayli.net for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.

Maureen Ker
Editor

Maureen is an editor, writer, enthusiastic cook and prolific collector of useless kitchen gadgets. She is the author of 3 books and 100+ articles. Her work has appeared in the *New York Daily News*, and various adult and children's publications.



Don't replace people, augment them

By TIM O'REILLY

If we let machines put us out of work, it will be because of a failure of imagination and the will to make a better future!

I'm exploring technology and the future of work in a new event called The Next:Economy Summit. If you're building technology that augments humans so they can do work that was previously impossible, I want to hear from you.

"Could a machine do your job?" asked Michael Chui, James Manyika, and Mehdi Miremadi in a recent *McKinsey Quarterly* article, "Where Machines Could Replace Humans and Where They Can't Yet." The authors try to put the current worries about this question in perspective:

> *"As automation technologies such as machine learning and robotics play an increasingly great role in everyday life, their potential effect on the workplace has, unsurprisingly, become a major focus of research and public concern. The discussion tends toward a Manichean guessing game: which jobs will or won't be replaced by machines?*
>
> *In fact, as our research has begun to show, the story is more nuanced. While automation will eliminate very few occupations entirely in the next decade, it will affect portions of almost all jobs to a greater or lesser degree, depending on the type of work they entail."*

Instead of the binary question of which jobs will be eliminated, the authors instead wisely point out that it is tasks that are being automated, and that automation doesn't simply destroy jobs. It changes them.

But they don't go far enough in their analysis. They assess the potential for job change in terms of the technical feasibility of automating various activities, the economics of labor supply and demand, and whether the savings from automation will justify the cost.

They also note that "a fourth factor to consider is the benefits beyond labor substitution, including higher levels of output, better quality, and fewer errors. These are often larger than those of reducing labor costs."

But they don't ask what, in my opinion, is the key question.

What will new technology let us do that was previously impossible?

Those weavers who smashed machine looms in Ned Ludd's rebellion of 1811 didn't realize that descendants of those machines would make unbelievable things possible.

We'd tunnel through mountains and under the sea, we'd fly through the air, crossing continents in hours, we'd build cities in the desert with buildings a half mile high, we'd more than double average human lifespan, we'd put spacecraft in orbit

What is impossible today, but will become possible with the technology we are now afraid of?

around Jupiter, we'd smash the atom itself!

What is impossible today, but will become possible with the technology we are now afraid of?

As Google chief economist Hal Varian has said, "My grandfather wouldn't recognize what I do as work." What are the new jobs of the 21st century that aren't going to be replaced or changed, but invented out of whole cloth?

Let me explain with a personal anecdote. I used to be legally blind without huge coke-bottle glasses. My eyes were fixed by an augmented surgeon who was able to do something that had been previously impossible.

Ten years ago, in my column for *Make* magazine, I gave an account of my surgery:

> *I had laser eye surgery the other day, and after more than forty years of wearing glasses so strong that I was legally blind without them, I can see clearly on my own. I had a perfect outcome: 20/20 for far vision, yet still able to read and do other close work as well. I keep saying to myself: I'm seeing with my own eyes!*
>
> *But in order to remove my need for prosthetic vision, the surgeon ended up relying on prosthetics of her own, performing the surgery with the aid of a complex of high tech equipment and a team of specialized technicians.*
>
> *First they mapped my eyes with a device called a corneal topographer, and came up with a modification plan. Then they used a laser to blister the surface of my cornea, and twenty minutes later, the surgeon used a micro-keratome to lift the flap of the blister so another laser could do the real mods to the deeper layers of the cornea.*
>
> *During the actual surgery, apart from lifting the flap and smoothing it back into place after the laser was done, her job was to clamp open my eyes, hold my head, utter reassuring words, and tell me, sometimes with urgency, to keep looking at the red light!*
>
> *Afterwards, I asked what happened if my eyes drifted, and I didn't stay focused on the light. "Oh, the laser would stop. It only works when your eyes are tracking."*
>
> *In short, surgery this sophisticated could never be done by an unaugmented human being. The human touch of my superb doctor was paired with the inhuman accuracy of complex*

This is the true opportunity of technology: it extends human capability.

> *machines, a twenty-first century hybrid freeing me from the tyranny of assistive devices first invented in thirteenth century Italy.*
>
> *Whether or not we're heading for a Kurzweil-style singularity, in which humans merge with machines, an increasing number of our activities are only possible with the aid of computers and other complex devices. My eye surgery is only one example.*
>
> *The revolution in sensors, computers and control technologies is going to make many of the daily activities of the twentieth century seem quaint as, one by one, they are reinvented in the twenty-first.*

This is the true opportunity of technology: it extends human capability. There is way too much handwringing about the possibility of technology eliminating human jobs, and way too little imagining new jobs that could only be done with the help of technology.

There's a profound failure of imagination and will in much of today's economy. For every Elon Musk who wants to reinvent the world's energy infrastructure, build revolutionary new forms of transport, go to Mars, and forge ahead with self-driving cars, there are far too many companies that are simply cutting costs and pulling money out of the economy.

I sometimes think it will take a great crisis to pull us out of our current malaise, in much the same way that World War II helped to end the Great Depression. Climate change or a pandemic or another vast war impelled by anger and hopelessness may be the trigger. But one would hope that we could avoid that dire contingency!

Economic historian Louis Hyman was recently interviewed about the gig economy (one symptom of the technological displacement that Chui, Manyika, and Miremadi explored.)

Hyman reflected on what the history of the Great Depression and World War II teaches us about the kind of investment necessary to make a better future.

> *"There's a great unraveling, and there's a great forgetting what made possible the post-war life and forgetting that there's a deep connection between security in our economic lives, and security and us as a democratic society, as people who are happy, and vote reasonably and don't, you know, worry about the future as much.*
>
> *Across lines of class, across lines of education, we are all movingtowards this freelance economy, this unstable economy. And for some people, it works out great if you're a consultant. You can make*

That we utterly transformed the US economy during a period of only five years should inspire us to ask what is holding us back today.

tons of money in just one day. And if you're an undocumented worker who's working in very dangerous conditions outside the surveillance of labor law, it's not so good...

Back in the 1930s, when there were homeless encampments in Washington, D.C., very much like the homeless encampments that are now under the I-280 in San Francisco, the federal government invested capital in new industries to create jobs for millions of people. They created tax codes that redistributed from the rich to the poor..."

But redistribution of income and the beginnings of the modern social safety net were only part of the story. Hyman continued:

"The New Deal's Reconstruction Finance Corporation not only helped light up America—moving it from 10 percent of homes having electricity in 1930 to more than 60 percent a decade later—it also funded research in the Defense Plant Corporation.

It was fundamentally about investment in edgy technology, so things like aerospace, aluminum extraction, synthetic rubber were all brought to scale,' Hyman says. 'Aerospace before 1939 had fewer people working in it than worked in candy manufacturing. And after World War II, the aerospace industry was four times the size of the pre-war car industry. This is incredible scale and scope of an endeavor, to utterly transform the economy in about five years, by using idle capital..."

"There's so much capital out there, and they don't know where to put it. It's hard for us to imagine as normal people, but for the big players in the global economy, the pension funds, the hedge funds the bazillion billionaires, they're desperate to find an outlet for capital. And right now, the best outlook for capital is our home mortgages and our credit card loans. And until we provide them with better outlets, like we did during the 1930s and '40s, to invest in aerospace and more cutting edge technology in industries that employ millions, it's going be very hard to get our economy going again. And fundamentally, this is how capitalism has to work. It has to be a virtuous cycle, where capital comes into businesses, is invested and creates new jobs."

What might we be astonished by if we have the courage to invest in the possibilities of a better future?

Hyman's reminder that during World War II we utterly transformed the US economy during a period of only five years should inspire us to ask what is holding us back today. Do we need a crisis, or can we make bold moves without one?

How we frame the future matters! If we create an attitude of fear towards technology, we miss the huge opportunity to put it to work solving problems that bedevil us today. It's our responsibility as entrepreneurs and technologists to rethink what is possible!

Technology lets us rethink the very structure of how we do things. Consider, for example, the way that Uber and Lyft have transformed urban transportation. There were connected taxicabs long before Uber—but all they did was to recreate the old process.

What we got for our connectivity was a credit card reader in the back, and a small screen showing us ads. What Garrett Camp and Travis Kalanick realized was that humans were now augmented by location-aware smartphones, and so you could completely rethink the way you summoned a car.

It would be utter magic to someone from the past — that you can click on your phone, and summon a car to whereever you are, and to know just how long it will take for a car to pick you up.

But when Uber started talking about self-driving cars, they lost the plot and started talking only about cutting costs and eliminating workers. Rather than crowing about how they'd finally get rid of those pesky drivers, they should have been talking about an experiment that they've run since 2014, delivering flu shots.

"Sure, we won't always have drivers. But just imagine how many other jobs we can restructure and make more magical and on demand once the transportation is even cheaper and more convenient!"

Zipline is completely rethinking how healthcare could be delivered in an on-demand world. Their pilot project in Rwanda looks to address one of the leading causes of death — postpartum hemorrhage — by delivering blood on demand, via high speed drone, to locations without modern transportation or healthcare infrastructure.

But if you think about it, on demand technology could be transforming healthcare everywhere—if we think big, and use technology not just to cut costs and improve profits but to deliver previously impossible services!

If you'd told the weavers of Ned Ludd's time that those machines they were smashing would mean that ordinary people would have more changes of clothing than the richest nobles of their day, they would have shaken their heads in astonishment. *What might we be astonished by if we have the courage to invest in the possibilities of a better future?* ■

Reprinted with permission of the original author. First appeared at medium.com/the-wtf-economy.

Ten rules for negotiating a job offer

By HASEEB QURESHI

Negotiation is a skill that can be learned, just like any other.

When the story of how I landed a job at Airbnb went viral, I was surprised at how infatuated people were with my negotiations. Media stories portrayed me as some kind of master negotiator—a wily ex-poker-player who was able to con the tech giants into a lucrative job offer.

This is silly. It's silly for a lot of reasons, but one of the main ones is that in reality, my negotiation skills are nothing special. There are lots of job candidates who are better negotiators than I, to speak nothing of recruiters and other professional negotiators.

It just so happens that most people don't negotiate at all, or if they do, they negotiate just enough to satisfy themselves that they did.

Worse yet, most of the advice out there on negotiation is borderline useless. Almost anything you read on the subject will be a vague and long-winded exhortation to "make sure you negotiate" and "never say the first number." Beyond those two morsels of advice, you're pretty much on your own.

I thought to myself: why is there so little actionable advice out there about negotiation? I suspect it's because deep down, many people believe that negotiation is inexplicable, that it's something some people can do and others can't, and that there's no real way to break it down so anyone can learn it.

I say that's BS. Negotiation is a skill that can be learned, just like any other. I don't believe it's particularly elusive or hard to understand. So I'm going to try to explain how anyone can do it.

Three caveats.

First: I'm not an expert. There are people who really are experts at this, and when my advice contradicts theirs, you should assume I'm wrong.

Second: negotiation is tricky to generalize about because it's deeply intertwined with social dynamics and power. The appropriate advice for an Asian male in Silicon Valley may not be appropriate for a black woman in Birmingham, Alabama. Racial, sexual, and political dynamics accompany you to the negotiating table.

At the same time, I want to caution against overemphasizing these factors. Being afraid to negotiate out of fear of discrimination can often be just as deleterious as discrimination itself.

Ceteris paribus, negotiate aggressively.

Third: I'm the first to admit that negotiation is stupid. It's a practice that inherently benefits those who are good at it, and is an absurd axis on which to reward people. But it's a reality of our economic system. And like most collective action problems, we're probably not going to be able to abolish it any time soon. In which case, you might as well improve at it.

So here's my guide to negotiation. It's going to be split into two parts: the first part will be about conceptualizing the negotiating process, about how to begin the process and set yourself up for maximal success.

The second part will be advice on the actual back-and-forth portion of negotiating and how to ask for what you want.

Let's take it from the top.

What it means to "get a job"

In our culture we call entering the employment market "trying to get a job." This is an unfortunate turn of phrase. "Getting a job" implies that jobs are a resource out in the world, and you're attempting to secure one of these resources.

But that's completely backwards. What you are actually doing is selling your labor, and a company is bidding for it.

Employment is just striking a mutual deal in the labor market.

Like any market, the labor market only functions well if it's competitive. This is the only way to ensure fair and equitable pricing.

At the risk of spouting truisms: always, always negotiate.

Imagine you were a farmer selling watermelons. Would you just sell your watermelons to the first buyer who agreed to purchase them?

Or would you survey the marketplace of buyers, see the best price (and business partner) you could get, and then make an informed decision on which buyer to sell to?

And yet, when people talk about the labor market, they think "oh, a company wants to *give me a job*! What a relief!" As though having a job were in itself some special privilege for which a company is the gatekeeper.

Dispel yourself of this mindset.

A job is just a deal. It is a deal between you and a company to exchange labor for money (and other things you value).

This might sound like an abstract point, but you should absolutely approach negotiation from this perspective.

The role of negotiation

Negotiating is a natural and expected part of the process of trying to make a deal. It's also a signal of competence and seriousness.

Companies generally respect candidates who negotiate, and most highly attractive candidates negotiate (if for no other reason, because they often have too many options to choose from).

At the risk of spouting truisms: always, always negotiate. It doesn't matter how good or bad you think you are. You never damage a relationship by negotiating.

In all my time as an instructor at App Academy, out of hundreds of offers negotiated, only once or twice were offers ever rescinded in negotiations. It basically never happens. And when it does, it was usually because the candidate was being an unconscionable asshole, or the company was imploding and needed an excuse to rescind the offer.

You might think to yourself: "*well, I don't want to set high expectations, and the offer is already generous, so I ought to just take it.*"

No. Negotiate.

Or maybe: "*I don't want to start off on the wrong foot and look greedy with my future employer.*"

No. Negotiate.

"But this company is small and—"

No. Shut up. Negotiate.

We'll talk more in the next section about why a lot of these objections are BS, and fundamentally misapprehend the dynamics of hiring. But for now, just trust me that you should always negotiate.

The ten rules of negotiating

I've tried to boil down negotiation to ten rules. The rules, in order of appearance, are:

1. Get everything in writing
2. Always keep the door open
3. Information is power
4. Always be positive
5. Don't be the decision maker
6. Have alternatives
7. Proclaim reasons for everything
8. Understand what they value
9. Be motivated by more than just money
10. Be winnable

We'll only get through some of these in this blog post, and the rest will appear in the second part. But I'll explain each rule as we get to it.

So let's start from the top and try to walk through a negotiation process from the very beginning. For most, that starts when you receive an offer.

Rule #1 of negotiating: have everything in writing.

The offer conversation

You've just received the phone call: your interview went well, and after much deliberation they decided they like you. They want to make you an offer. Congratulations!

Don't get too excited though. The fun is just getting started.

Thank your recruiter. Sound excited — hopefully this won't be hard. Before jumping into details, try to ask for specific feedback on your interview performance.

If they give it to you, this will help you gauge how much they want you, as well as tell you things you can improve on in your next interview(s).

Now, time to explore the offer.

Rule #1 of negotiating: have everything in writing.

Eventually, they'll give you information about the offer. Write it all down. Doesn't matter if they're going to send you a written version later, *write everything down*.

Even if there are things that are not directly monetary, if they relate to the job, write them down. If they tell you "we're working on porting the front-end to Angular," write that down.

If they say they have 20 employees, write that down. You want as much information as you can. You'll forget a lot of this stuff, and it's going to be important in informing your final decision.

Depending on the company, they'll also tell you about the equity package. We'll look more specifically at equity in part II, but be sure to write everything down.

The rule from here on out is that everything significant you discuss will have some kind of a paper trail. Often, the company won't even send you an official offer letter until a deal is finalized. So it falls to you to confirm all of the important details in subsequent emails.

So yadda yadda, lots of details, writing stuff down, oh there's a joke, time to laugh. Now the recruiter is done talking and you're done asking all of your questions.

Your recruiter will now say something along the lines of "*so what do you think?*"

This seems innocuous, but your reply here is critical, because there's a lot you can say to weaken your position. This is your first decision point.

A decision point is a moment in the negotiation where your interlocutor wants to compel you to make a decision. If they succeed in tying you to a position, they will close the door on further negotiating.

Of course "what do you think?" is a subtle prod. But it is the beginning of many attempts to get you to make a premature commitment.

This leads to rule #2 of negotiating: always keep the door open.

Never give up your negotiating power until you're absolutely ready to make an informed, deliberate final decision.

This means your job is to traverse as many of these decision points as possible without giving up the power to continue negotiating.

Very frequently, your interlocutor will try to trick you into making a decision, or tie you to a decision you didn't commit to. You must keep verbally jiu-jitsu-ing out of these antics until you're actually ready to make your final decision.

Protecting information

There's an uncomfortable silence by now, and their "*what do you think?*" is hanging in the air.

If you say "*yes, that sounds amazing, when do I start?*" you'd have implicitly accepted the offer and completely closed the door on the negotiation.

Bottom line, you want them to be uncertain on exactly what it would take to sign you.

This is your recruiter's number one favorite thing to hear. It stands to reason you probably shouldn't do this.

But their second favorite thing to hear you say is "*can you do 90K instead of 85K?*" This also closes the door, but for a different and more subtle reason. And it's the number one reason why most people suck at negotiation.

Rule #3 of negotiating: information is power.

To protect your power in the negotiation, you must protect information as much as possible.

A company doesn't give you insight into what it's thinking. It doesn't tell you its price range, how much it paid the previous candidate with your experience, or anything like that.

It intentionally obfuscates those things. But it wants you not to do the same.

A company wants to be like a bidder in a secret auction. But unlike the other bidders, it wants to know exactly how high all of the other bids are. It then openly intends to exploit that knowledge, often by bidding one cent more than the second highest bid.

Yeah, no. Screw that. It's a silent auction, and to keep it that way, you must protect information.

In many situations, the only reason why you have any negotiating power at all is because the employer doesn't actually know what you're thinking.

They might not know how good your other offers are, or how much you were making in your last job, or how you weigh salary vs equity, or even how rational you are as a decision-maker.

Bottom line, you want them to be uncertain on exactly what it would take to sign you.

When you say "*can you do 90K instead of 85K,*" you've told them exactly what it will take to make you sign. The sheet's pulled back, the secret auction is up, and they're going to bid 90K (or more likely, 87K). And they know there's almost no risk in doing so, because you'll probably accept.

What if you were the kind of person who wouldn't even consider an offer below 110K? Or the kind of person who wouldn't consider an offer below 120K?

If you were, you wouldn't ask for 90K, and if they offered it as conciliation, you'd tell them to stop wasting your time.

By staying silent, *they don't actually know which of those kinds of people you are.* In their mind, you could be any of the three.

A corollary of this rule is that you should not reveal to companies what you're currently making. There are some exceptions, but as a rule you should assume this.

If you must divulge what you're making, you should be liberal in noting the total value of your package (incorporate bonuses, unvested stock, nearness to promotion etc.), and always mention it in a context like "*[XYZ] is what I'm currently making, and I'm definitely looking for a step up in my career for my next role.*"

Companies will ask about your current compensation at different stages in the process—some before they ever interview you, some after they decide to make you an offer. But be mindful of this, and protect information.

So given this offer, don't ask for more money or equity or anything of the sort. Don't comment on any specific details of the offer except to clarify them.

Give away nothing. Retain your power.

Say instead:

> *"Yeah, [COMPANY_NAME] sounds great! I really thought this was a good fit, and I'm glad that you guys agree. Right now I'm talking with a few other companies so I can't speak to the specific details of the offer until I'm done with the process and get closer*

Remember, you are the product!

> *to making a decision. But I'm sure we'll be able to find a package that we're both happy with, because I really would love to be a part of the team."*

Think like the watermelon farmer. This offer is just the first businessman who's stopped by your watermelon patch, glanced over your crops, and announced "I'll take all of these right now for $2 a melon."

Cool. It's a big market, and you're patient—you're a farmer after all. Just smile and tell them you'll keep their offer in mind.

And this is super important: always be unequivocally positive.

The importance of positivity

Staying positive is rule #4 of negotiation.

Even if the offer sucks, it's extremely important to remain positive and excited about the company.

This is because *your excitement is one of your most valuable assets in a negotiation.*

A company is making you an offer because they think you'll do hard work for them if they pay you. If you lose your excitement for the company during the interview process, then they'll lose confidence that you'll actually want to work hard or stay there for a long time.

Each of those makes you less attractive as an investment. Remember, you are the product! If you become less excited, then the product you're selling actually loses value.

Imagine you were negotiating with someone over buying your watermelons, but the negotiation took so long that by the time you'd reached an agreement, your watermelons had gone bad.

Companies are terrified of that. They don't want their candidates to go bad during a negotiation. Hence they hire professional recruiters to manage the process and make sure they remain amicable.

You and the recruiter share the same interest in that regard. If a company feels like you've gone bad, suddenly they're a lot less willing to pay for you.

So despite whatever is happening in the negotiation, give the company the impression that 1) you still like the company, and that 2) you're still excited to work there, even if the numbers or the money or the timing is not working out.

Generally the most convincing thing to signal this is to reiterate you love the mission, the team, or the problem they're working on, and really want to see things work out.

Don't be the decision-maker

You can wrap up the conversation now by saying:

> *"I'll look over some of these details and discuss it with my [FAMILY / CLOSE_FRIENDS / SIGNIFICANT_OTHER]. I'll reach out to you if I have any questions. Thanks so much for sharing the good news with me, and I'll be in touch!"*

So not only are you ending the conversation with the power all in your hands, but note there's another important move here: you're roping in other decision-makers.

Rule #5 of negotiation: don't be the decision-maker.

Even if you don't particularly care what your friends/family/husband/mother thinks, by mentioning them, you're no longer the only person the recruiter needs to win over.

There's no point in them trying to bully and intimidate you; the "true decision-maker" is beyond their reach.

This is a classic technique in customer support and remedia-

Just having an offer in hand will get the engine running.

tion. It's never the person on the phone's fault, they're just some poor schmuck doing their job. It's not their decision to make. This helps to defuse tension and give them more control of the situation.

It's much harder to pressure someone if they're not the final decision-maker. So take advantage of that.

Okay!

We have our first offer. Send a follow-up email confirming all of the details you discussed with your recruiter so you have a paper trail. Just say "*just wanted to confirm I had all the details right.*"

Groovy. Next step is to leverage this to land other offers and find the best deal we can find in the job market.

Getting other offers

Turns out, it doesn't matter that much where your first offer is from, or even how much they're offering you. Just having an offer in hand will get the engine running.

If you're already in the pipeline with other companies (which you should be if you're doing it right), you should proactively reach out and let them know that you've just received an offer.

Try to build a sense of urgency. Regardless of whether you know the expiration date, all offers expire at some point, so take advantage of that.

> *"Hello [PERSON],*
>
> *I just wanted to update you on my own process. I've just received an offer from [COMPANY] which is quite strong. That said, I'm really excited about [YOUR AMAZING COMPANY] and really want to see if we can make it work. Since my timeline is now compressed, is there anything you can do to expedite the process?"*

Should you specifically mention the company that gave you an offer? Depends. If it's a well-known company or a competitor, then definitely mention it. If it's a no-name or unsexy company, you should just say you received an offer. If it's expiring soon, you should mention that as well.

Either way, send out a letter like this to every single company you're talking to. No matter how hopeless or pointless you think your application is, you want to send this signal to everyone who is considering you in the market.

Second, if there are any other companies you are looking to apply to (whether through referral or cold application), or even companies at which you've already applied but haven't heard back, I would also follow up with a similar email.

So why do this? Isn't this tacky, annoying, or even desperate?

None of the above. It is the oldest method in history to galvanize a marketplace—show that supplies are limited and build urgency. Demand breeds demand. Not every company will respond to this, but many will.

Isn't it stupid that companies respond to this though?

Why companies care about other offers

When I wrote about the story of my own job search, I mentioned how having an offer from Google made companies turn around and expedite me through their funnels. Many commentators lamented at the capriciousness of these companies.

If Uber or Twitch only talked to me because of Google and until then weren't willing to look at me, what did that say about their hiring processes? What legitimately are they evaluating, if anything at all?

I think this response is totally backwards. The behavior of tech companies here is actually very rational, and you would do well to understand it.

The single strongest determinant of your final offer will be the number and strength of offers that you receive.

First, you must realize what a company's goal is. A company's goal is to hire someone who will become an effective employee and produce more value than their cost.

How do you figure out who will do that? Well, you can't know for certain without actually hiring them, but there are a few proxies.

Pedigree is the strongest signal; if they did it at other companies, they can probably do it at yours. And if someone trusted within the organization can vouch for them, that's often a strong signal as well.

But turns out, almost everything else is a weak signal. Weak in the sense that it's just not very reliable. Interviews, if you think about it, are long, sweaty, uncomfortable affairs that only glancingly resemble actual employment.

They're weird and can't tell you that much about whether an individual will be a good at their job. There's no way around this. There are a few stronger signals, like bringing someone in for a week or two on a contract-to-hire position, but strong candidates won't consider this. So candidates as a whole have effectively forced companies to assume almost all of the risk in hiring.

The truth is, knowing that someone has passed your interview just doesn't say *that much* about whether they'll be a good employee. It's as though you knew nothing about a student other than their SAT score. It's just not a lot of data to go off.

Nobody has solved this problem. Not Google or anyone else.

And this is precisely why it's rational for companies to care that you've received other offers. They care because each company knows that their own process is noisy, and the processes of most other companies are also noisy.

But a candidate having multiple offers means that they have multiple weak signals in their favor. Combined, these converge into a much stronger signal than any single interview.

It's like knowing that a student has a strong SAT score, and GPA, and won various scholarships. Sure, it's still possible that they're a dunce, but it's much harder for that to be true.

This is not to say that companies respond proportionally to these signals, or that they don't overvalue credentials and brands. They do. But caring about whether you have other offers and valuing you accordingly is completely rational.

So this is all to say—tell other companies that you've received offers. Give them more signal so that they know you're a valued and compelling candidate. And understand why this changes their mind about whether to interview you.

As you continue interviewing, remember to keep practicing your interview skills. The single strongest determinant of your final offer will be the number and strength of offers that you receive.

Some advice on timing

You want to be strategic about the timing of your offers. Generally, you should try to start interviewing at larger companies earlier. Their processes are slower and their offer windows are wider (meaning they allow you more time to decide). Startups are the other way around.

Your goal should be to have as many offers overlapping at the same time as possible. This will maximize your window for negotiating.

When you receive an offer, often the first thing you should ask for is more time to make your decision. Especially in your first offer, more time is by far the most valuable thing you can ask for. It's time that enables you to activate other companies and end up with the strongest possible offer. So be prepared to fight for time.

Exploding offers are anathema to your ability to effectively navigate the labor market.

How to approach exploding offers

Hoo boy.

Exploding offers are offers that expire within 24–72 hours. You won't see this much at big companies, but they're becoming increasingly common among startups and mid-sized companies.

Exploding offers suck, and I share most people's disdain for this practice. But I do understand it. Exploding offers are a natural weapon for employers to combat a strong hiring market for tech workers.

Companies know exactly what they're doing with exploding offers—they play on fear and limit your ability to seek out counteroffers.

In a sense, it's unsurprising that if startups have more difficulty attracting and securing talent, they'd resort to this practice. What I don't like is the dishonesty about it.

Employers often justify this by saying:

> *"If you need more time than this, then that's a sign you're not the kind of person we're looking for."*

Please don't buy this crap or feel guilty over it. They're simply doing this to improve their chance of closing candidates. Needing more than three days to make a life decision isn't a sign of anything other than thoughtfulness.

So what should you do if you receive an exploding offer?

Exploding offers are anathema to your ability to effectively navigate the labor market. Thus, there is only one thing to do. Treat the offer as a non-offer unless the expiration window is widened.

In no uncertain terms, convey that if the offer is exploding, it's useless to you.

Example conversation:

> *"I have one big concern. You mentioned that this offer explodes in 48 hours. I'm afraid this doesn't work at all for me. There's no way that I can make a decision on this offer within a 48 hour window. I'm currently wrapping up my interview process at a few other companies, which is likely to take me another week or so. So I'm going to need more time to make an informed decision."*

If they push back and say this is the best they can do, then politely reply:

> *"That's really unfortunate. I like [YOUR COMPANY] and was really excited about the team, but like I said, there's no way I can consider this offer. 48 hours just too unreasonable of a window. The next company I join will be a big life decision for me, and I take my commitments very seriously. I also need to consult with my [EXTERNAL_DECISION_MAKER]. There's no way that I can make a decision I'm comfortable with in this short an amount of time."*

Pretty much any company will relent at this point. If they persist, don't be afraid to walk away over it. (They probably won't let that happen, and will come grab you as you're walking out the door. But if they don't, then honestly, screw 'em.)

I was given several exploding offers during my job search. And every time, I did essentially this. Every single offer immediately widened to become more reasonable, sometimes by several weeks.

I want to emphasize, lest I be misunderstood here—what I'm saying is not to just silently let an exploding offer expire,

Keep an open mind, and remember that a job search is a two-sided process.

and assume that everything will be fine and they'll still hire you. They won't.

For exploding offers to be a credible weapon, a company has to have a reputation of enforcing them. I'm saying explicitly call this out as an issue when they make the offer.

Don't let a company bully you into giving away your negotiating power.

The negotiating mindset

Before we enter into the actual back-and-forth, I want to examine the mindset you should have as a negotiator. This applies not just to how you approach the conversation, but also to how you think about the company.

Do not fall into the trap of valuing companies solely along one dimension. That means don't just value companies based on salary, equity, or even on prestige.

Those are all important dimensions, but so are cultural fit, the challenge of the work, learning potential, later career options, quality of life, growth potential, and just overall happiness.

None of these inherently trump any of the other. Anyone who tells you "just choose wherever you think you'll be happiest" is being just as simplistic as someone who says "just choose the one that offers the most money."

All of these things matter, and your decision should be genuinely multi-dimensional.

Be open to being surprised as you explore different companies.

It's also important to understand that companies don't all value you along the same dimension either. That is, different companies are genuinely looking for different skills, and there are some companies at which you will be more and less valuable. Even at peer companies this is true, especially so if you have a specialized skill-set.

The more companies you talk to, the more likely you are to find a company to which you are significantly more valuable than the rest. Chances are this is where you'll be able to negotiate your strongest offer.

It might surprise you which company this turns out to be; keep an open mind, and remember that a job search is a two-sided process.

One of the most valuable things you can do for yourself in this process is to really try to understand how employers think and what motivates them. Understanding your interlocutor is extremely important in negotiation, and we'll be exploring that a lot in the next blog post.

But most of all I want to emphasize: be curious about the other side. Try to understand why employers think the way they do. Be sympathetic toward them. Care about what they want and help them try to get it. Adopting this mindset will make you a much stronger negotiator, and accordingly, a much better employee and team member.

Okay. That's as far as we're going for today. In the next blog post, I'm going to cover the last four rules of negotiation. I'll also go over the actual back-and-forth process—how to ask for what you want, how to strengthen offers, and how to dismantle the tricks that companies will try to pull on you. Also a lot more on the theory of negotiation, which I really dig.

Do share this post if you found it useful! And follow me on Twitter for updates on when I post part 2 (which I promise will be soonish!) ■

Reprinted with permission of the original author. First appeared at medium.freecodecamp.com.

Hitchhiker trees: functional, persistent, off-heap sorted maps

By DAVID GREENBERG

The goal of the hitchhiker tree is to wed three things...

This document will attempt to sketch out the big ideas in the hitchhiker tree. It will also attempt to call out various locations in the implementation where features are built.

High-level understanding

The goal of the hitchhiker tree is to wed three things: the query performance of a B+ tree, the write performance of an append-only log, and the convenience of a functional, persistent data structure. Let's look at each of these in some detail.

B+ trees win at queries

You might remember an important theorem from data structures: the best-performing data structure for looking up sorted keys cannot do those queries faster than $O(\log(n))$.

Since sorted trees provide a solution for this, we'll start with them. Now, a common sorted tree for this purpose is the Red-Black tree, whose actual query performance is between $\log_2(n)$ and $2*\log_2(n)$ (the write performance is $\log_2(n)$).

The factor of 2 comes from the partial imbalances (which are still asymptotically balanced) that the algorithm allows, and the base 2 of the log comes from the fact that it's a binary search tree.

A less popular sorted tree is the AVL tree — this tree achieves $\log_2(n)$ query performance, at the cost of always paying $2*\log_2(n)$ for inserts. We can already see a pattern — although many trees reach the asymptotic bound, they differ in their constant factors.

The tree that the hitchhiker tree is based off of is the B+ tree, which achieves $\log_b(n)$ query performance. Since b can be very large (on the order of 100s or 1000s), these trees are especially great when each node is stored on higher latency media, like remote storage or a disk.

This is because each node can contain huge numbers of keys, meaning that by only keeping the index nodes in memory, we can access most keys with fewer, often just one, data accesses.

Unlike the above sorted trees (and B trees, which we won't discuss), B+ trees only store their data (i.e. the values) in their leaves—internal nodes only need to store keys.

Event logs win at writing data

Do you know the fastest way to

The answer is that we're going to "overlay" an event log on the B+ tree!

write data? Append it to the end of the file. There's no pointers, no updating of data structures, no extra IO costs incurred.

Unfortunately, to perform a query on an event log, we need to replay all the data to figure out what happened. That replay costs `O(n)`, since it touches every event written. So, how can we fix this?

Unifying B+ trees and event logs

The first idea to understand is this: how can we combine the write performance of an event log with the query performance of a B+ tree? The answer is that we're going to "overlay" an event log on the B+ tree!

The idea of the overlay is this: each index node of the B+ tree will contain an event log. Whenever we write data, we'll just append the operation (insert or delete) to the end of the root index node's event log.

In order to avoid the pitfall of appending every operation to an ever-growing event log (which would leave us stuck with linear queries), we'll put a limit on the number of events that fit in the log.

Once the log has overflowed in the root, we'll split the events in that log towards their eventual destination, adding those events to the event logs of the children of that node.

Eventually, the event log will overflow to a leaf node, at which point we'll actually do the insertion into the B+ tree.

This process gives us several properties:

- Most inserts are a single append to the root's event log
- Although there are a linear number of events, nodes are exponentially less likely to overflow the deeper they are in the tree
- All data needed for a query exists along a path of nodes between the root and a specific leaf node. Since the logs are constant in size, queries still only read `log(n)` nodes.

Thus we dramatically improve the performance of insertions without hurting the IO cost of queries.

Functional persistence

Now that we get the sketch of how to combine event logs and B+ trees, let's see the beauty of making the whole thing functional and persistent!

Since the combined B+/log data structure primarily only

Let's see the beauty of making the whole thing functional and persistent!

modifies nodes near the root, we can take advantage of the reduced modification to achieve reduced IO when persisting the tree.

We can use the standard path-copying technique from functional, persistent data structures. This gives great performance, since the structure is designed to avoid needing to copy entire paths — most writes will only touch the root.

Furthermore, we can batch many modifications together, and wait to flush the tree, in order to further batch IO.

Code structure

The hitchhiker tree's core implementation lives in 2 namespaces: `hitchhiker.tree.core` and `hitchhiker.tree.messaging.hitchhiker.tree.core` implements the B+ tree and its extensibility hooks; `hitchhiker.tree.messaging` adds the messaging layer (aka log) to the B+ tree from `core`.

Protocols

In `hitchhiker.tree.core`, we have several important protocols:

`hitchhiker.tree.core/IKeyCompare`

This protocol should be extended to support custom key comparators. It's just like `clojure.core/compare`.

`hitchhiker.tree.core/IResolve`

This protocol is the functionality for a minimal node. Not only will every node implement this, but also backends will use this to implement stubs, which automatically and lazily load the full node into memory during queries.

`last-key` is used for searches, so that entire nodes can remain unloaded from memory when we only need their boundary key.

`dirty?` is used to determine whether the IO layer would need to flush this node, or whether it already exists in the backing storage.

`resolve` loads the node into memory—this could return itself (in the case of an already loaded node), or it could return a new object after waiting on some IO.

`hitchhiker.tree.core/INode`

This protocol implements a node fully in-memory. Generally, this shouldn't need to be

re-implemented; however, if the hitchhiker was to be enhanced with key and value size awareness during splits and merges, you'd want to adjust the methods of this protocol.

`hitchhiker.tree.core/Config`

This structure must be passed to the `hitchhiker.tree.core/b-tree` constructor, which is the only way to get a new tree. The `index-b` is the fanout on index nodes; the `data-b` is the key & value fanout on data nodes, and the `op-buf-size` is the size of the log at each index node. The Internet told me that choosing `sqrt(b)` for the `op-buf-size` and `b-sqrt(b)` for the `index-b` was a good idea, but who knows?

`hitchhiker.tree.core/IBackend`

This protocol implements a backend for the tree.

`new-session` returns a "session" object, which is a convenient way to capture backend-specific stats.

`write-node` will write a node to storage, returning the stub object which implements `IResolve` for that backend. It can record stats by mutating or logging to the session.

`anchor-root` is called by the persistence functionality to ensure that the backend knows which nodes are roots; this is a hint to any sort of garbage collectors.

`delete-addr` removes the given node from storage.

`TestingBackend` is a simple implementation of a backend which bypasses serialization and is entirely in memory. It can be a useful reference for the bare minimum implementation of a backend.

`hitchhiker.tree.messaging/IOperation`

This protocol describes an operation to the tree. Currently, there are only InsertOp and DeleteOp, but arbitrary mutation is supported by the data structure.

Useful APIs

`hitchhiker.tree.core/flush-tree`

This takes a tree, does a depth-first search to ensure each node's children are durably persisted before flushing the node itself. It returns the updated tree and the session under which the IO was performed. `flush-`

We'd actually implemented fractal reads identically!

`tree` does block on the writing IO—a future improvement would be to make that non-blocking.

`hitchhiker.tree.messaging/enqueue`

This is the fundamental operation for adding to the event log in a hitchhiker tree. enqueue will handle the appending, overflow, and correct propagation of operations through the tree.

`hitchhiker.tree.messaging/apply-ops-in-path`

This is the fundamental operation for reading from the event log in a hitchhiker tree. This finds all the relevant operations on the path to a leaf node, and returns the data that leaf node would contain if all the operations along the path were fully committed. This is conveniently designed to work on entire leaf nodes, so that iteration is as easy as using the same logic as a non-augmented B+ tree, and simply expanding each leaf node from the standard iteration.

`lookup, insert, delete, lookup-fwd-iter`

These are the basic operations on hitchhiker trees. There are implementations in `hitchhiker.tree.core` and `hitchhiker.tree.messaging`, which leverage their respective tree variants. They correspond to `get`, `assoc`, `dissoc`, and `subseq` on sorted maps.

`hitchhiker.core.b-tree`

This is how to make a new hitchhiker or B+ tree. You should either use the above mutation functions on it from one or the other namespace; it probably won't work if you mix them.

Related work

Hitchhiker trees are made persistent with the same method, path copying, as used by Okasaki. The improved write performance is made possible thanks to the same buffering technique as a fractal tree index. As it turns out, after I implemented the fractal tree, I spoke with a former employee of Tokutek, a company that commercialized fractal tree indices. That person told me that we'd actually implemented fractal reads identically! This is funny because there's no documentation anywhere about how exactly you should structure your code to compute the query. ■

Reprinted with permission of the original author. First appeared at github.com/datacrypt-project/hitchhiker-tree.

Don't add your 2 cents

By DEREK SIVERS

Photograph by Maura Teague via flickr.com

Once you become the boss, unfortunately, your opinion is dangerous because it's not just one person's opinion anymore.

My friend Simone in Korea just became the big boss at work, managing other people for the first time.

Usually we talk about Debussy, Steven Pressfield, and Tavira, but today she asked if I had any non-obvious advice on management.

My only advice: *don't add your two cents to their ideas.*

"My two cents" is American slang for adding a small opinion or suggestion to someone else's thing.

Here's how it plays out at work:

> *Employee:*
> *"I've been working for the past two weeks on this new design. What do you think?"*
>
> *Boss:*
> *"I like it! Really good. Maybe just a darker shade of blue there, and change the word 'giant' to 'huge'. Other than that, it's great!"*

Now, because the boss said so, the creator of that design will have to change it just a little bit.

Because of that small change, that person no longer feels full ownership of their project. (Then you wonder why they're not motivated!)

Imagine this instead:

> *Employee:*
> *"I've been working for the past two weeks on this new design. What do you think?"*
>
> *Boss:*
> *"It's perfect. Great work!"*

This slight change made a huge difference in the psychology of motivation. Now that person can feel full ownership of this project, which is more likely to lead to more involvement and commitment for future projects.

The boss's opinion is no better than anyone else's. But once you become the boss, unfortunately, *your opinion is dangerous* because it's not just one person's opinion anymore — it's a command! So adding your two cents can really hurt morale.

A business should not focus on the boss, so this restraint is healthy. You shouldn't give your opinion on everything just because you can.

Obviously, if there's more than "2 cents" worth of stuff that needs to change, then this rule does not apply.

But if your contribution is small and probably just a meaningless opinion, just let it go. Let the other person feel full ownership of the idea, instead. ■

For more thoughts along this line, read the great book "What Got You Here Won't Get You There".

Why Angular 2 switched to TypeScript

By VICTOR SAVKIN

Typescript provides advanced autocompletion, navigation, and refactoring. Having such tools is almost a requirement for large projects.

Angular 2 is written in TypeScript. In this article I will talk about why we made the decision. I'll also share my experience of working with TypeScript: how it affects the way I write and refactor my code.

I like TypeScript, but you don't have to

Even though Angular 2 is written in TypeScript, you don't have to use it to write Angular 2 applications. The framework also works great with ES5, ES6, and Dart.

TypeScript has great tools

The biggest selling point of TypeScript is tooling. *It provides advanced autocompletion, navigation, and refactoring. Having such tools is almost a requirement for large projects.* Without them the fear of changing the code puts the code base in a semi-read-only state, and makes large-scale refactorings very risky and costly.

TypeScript is not the only typed language that compiles to JavaScript. There are other languages with stronger type systems that in theory can provide absolutely phenomenal tooling. But in practice most of them do not have anything other than a compiler.

This is because building rich dev tools has to be an explicit goal from day one, which it has been for the TypeScript team. That is why they built language services that can be used by editors to provide type checking and autocompletion. If you have wondered why there are so many editors with great TypeScript supports, the answer is the language services.

The fact that intellisense and basic refactorings (e.g., rename a symbol) are reliable makes a huge impact on the process of writing and especially refactoring code. Although it is hard to measure, I feel that the refactorings that would have taken a few days before can now be done in less than a day.

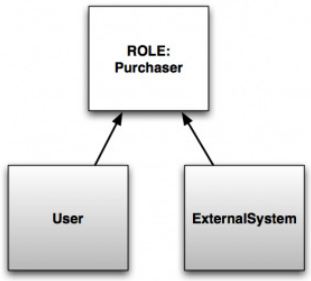
While TypeScript greatly improves the code editing experience, it makes the dev setup more complex, especially comparing to dropping an ES5 script on a page. In addition, you cannot use tools analyzing JavaScript source code (e.g., JSHint), but there are usually adequate replacements.

TypeScript is a superset of JavaScript

Since TypeScript is a superset of JavaScript, you don't need to go through a big rewrite to migrate to it. You can do it gradually, one module at a time.

Just pick a module, rename the .js files into .ts, then incrementally add type annotations. When you are done with this module, pick the next one. Once the whole code base is typed, you can start tweaking the compiler settings to make it more strict.

This process can take some time, but it was not a big problem for Angular 2, when we were migrating to TypeScript. Doing it gradually allowed us to keep developing new functionality and fixing bugs during the transition.



▲ Book-selling application

TypeScript makes abstractions explicit

A good design is all about well-defined interfaces. And it is much easier to express the idea of an interface in a language that supports them.

For instance, imagine a book-selling application where a purchase can be made by either a registered user through the UI or by an external system through some sort of an API (see Figure above).

As you can see, both classes play the role of a purchaser. Despite being extremely important for the application, the notion of a purchaser is not clearly expressed in the code. There is no file named purchaser.js. And as a result, it is possible for someone modifying the code to miss the fact that this role even exists.

```
function processPurchase(purchaser, details){ }

class User { }

class ExternalSystem { }
```

It is hard, just by looking at the code to tell what objects can play the role of a purchaser, and what methods this role has. We do not know for sure, and we will not get much help from our tools. We have to infer this information manually, which is slow and error-prone.

Now, compare it with a version where we explicitly define the Purchaser interface.

```
interface Purchaser {
  id: int; bankAccount: Account;
}

class User implements Purchaser {}

class ExternalSystem implements Purchaser {}
```

The typed version clearly states that we have the Purchaser interface, and the User and ExternalSystem classes implement it. So TypeScript interfaces allow us to define abstractions/protocols/roles.

It is important to realize that TypeScript does not force us to introduce extra abstractions. The Purchaser abstraction is present in the JavaScript version of the code, but it is not explicitly defined.

In a statically-typed language, boundaries between subsystems are defined using interfaces. Since JavaScript lacks interfaces, boundaries are not well expressed in plain JavaScript. Not being able to clearly see the boundaries, developers start depending on concrete types instead of abstract interfaces, which lead to tight coupling.

My experience of working on Angular 2 before and after our transition to TypeScript reinforced this belief. Defining an interface forces me to think about the API boundaries, helps me define the public interfaces of subsystems, and exposes incidental coupling.

TypeScript makes code easier to read and understand

Yes, I know it does not seem intuitive. Let me try to illustrate what I mean with an example. Let's look at this function jQuery.ajax(). What kind of information can we get from its signature?

Defining an interface forces me to think about the API boundaries, helps me define the public interfaces of subsystems, and exposes incidental coupling.

```
jQuery.ajax(url, settings)
```

The only thing we can tell for sure is that the function takes two arguments. We can guess the types. Maybe the first one is a string and the second one is a configuration object. But it is just a guess, and we might be wrong. We have no idea what options go into the settings object (neither their names nor their types), or what this function returns.

There is no way we can call this function without checking the source code or the documentation. Checking the source code is not a good option—the point of having functions and classes is to be able to use them without knowing how they are implemented.

In other words, we should rely on their interfaces, not on their implementation. We can check the documentation, but it is not the best developer experience—it takes additional time, and the docs are often out-of-date.

So although it is easy to read jQuery.ajax(url, settings), to really understand how to call this function, we need to either read its implementation or its docs.

Now, contrast it with a typed version.

```
ajax(url: string,
     settings?: JQueryAjaxSettings): JQueryXHR;

interface JQueryAjaxSettings {
  async?: boolean;
  cache?: boolean;
  contentType?: any;
  headers?: { [key: string]: any; };
  //...
}

interface JQueryXHR {
  responseJSON?: any; //...
}
```

It gives us a lot more information.

- The first argument of this function is a string.
- The settings argument is optional. We can see all the options that can be passed into the function, and not only their names, but also their types.
- The function returns a JQueryXHR object, and we can see its properties and functions.

The typed signature is certainly longer than the untyped one, but :string, :JQueryAjaxSettings, and JQueryXHR are not clutter. They are important documentation that improves the comprehensibility of the code.

We can understand the code to a much greater degree without having to dive into the implementation or reading the docs. My personal experience is that I can read the typed code faster because types provide more context to understand the code. But if any of the readers can find a study on how types affect code readability, please leave a comment.

One thing that is different about TypeScript comparing to many other languages compiled to JavaScript is that its type annotations are optional, and jQuery.ajax(url, settings) is still valid TypeScript.

So instead of an on-off switch, TypeScript's

Using TypeScript makes your code more rigid, but to a much lesser degree than people think.

types are more of a dial. If you find that the code is trivial to read and understand without type annotations, do not use them. Use types only when they add value.

Does TypeScript limit expressiveness?

Dynamically-typed languages have inferior tooling, but they are more malleable and expressive. I think using TypeScript makes your code more rigid, but to a much lesser degree than people think.

Let me show you what I mean. Let's say I use ImmutableJS to define the Person record.

```
const PersonRecord = Record({name:null,
age:null});

function createPerson(name, age) {
  return new PersonRecord({name, age});
}

const p = createPerson("Jim", 44);

expect(p.name).toEqual("Jim");
```

How do we type the record? Let's start with defining an interface called Person.

```
interface Person { name: string, age: number };
```

If we try to do the following:

```
function createPerson(name: string,
                      age: number): Person {
  return new PersonRecord({name, age});
}
```

The TypeScript compiler will complain. It does not know that PersonRecord is actually compatible with Person because PersonRecord is created reflectively.

Some of you with the FP background are probably saying: "If only TypeScript had dependent types!" But it does not. TypeScript's type system is not the most advanced one. But its goal is different. It is not here to prove that the program is 100% correct. It is about giving you more information and enable greater tooling. So it is OK to take shortcuts when the type system is not flexible enough.

So we can just cast the created record, as follows:

```
function createPerson(name: string,
                      age: number): Person {
  return <any>new PersonRecord({name, age});
}
```

The typed example:

```
interface Person { name: string, age: number };

const PersonRecord = Record({
  name:null, age:null});

function createPerson(name: string,
                      age: number): Person {
```

TypeScript takes 95% of the usefulness of a good statically-typed language and brings it to the JavaScript ecosystem.

```
  return <any>new PersonRecord({name, age});
}

const p = createPerson("Jim", 44);

expect(p.name).toEqual("Jim");
```

The reason why it works is because the type system is structural. As long as the created object has the right fields—name and age—we are good.

You need to embrace the mindset that it is OK to take shortcuts when working with TypeScript. Only then will you find using the language enjoyable.

For instance, don't try to add types to some funky metaprogramming code—most likely you won't be able to express it statically. Type everything around that code, and tell the typechecker to ignore the funky bit. In this case you will not lose a lot of expressiveness, and the bulk of your code will remain toolable and analyzable.

This is similar to trying to get 100% unit test code coverage. Whereas getting 95% is usually not that difficult, getting 100% can be challenging, and may negatively affect the architecture of your application.

The optional type system also preserves the JavaScript development workflow. Large parts of your application's code base can be "broken", but you can still run it. TypeScript will keep generating JavaScript, even when the type checker complains. This is extremely useful during development.

Why TypeScript?

There are a lot of options available to frontend devs today: ES5, ES6 (Babel), TypeScript, Dart, PureScript, Elm, etcSo why TypeScript?

Let's start with ES5. ES5 has one significant advantage over TypeScript: it does not require a transpiler. This allows you to keep your build setup simple. You do not need to set up file watchers, transpile code, and generate source maps. It just works.

ES6 requires a transpiler, so the build setup will not be much different from TypeScript. But it is a standard, which means that every single editor and build tool either supports ES6 or will support it. This is a weaker argument that it used to be as most editors at this point have excellent TypeScript support.

Elm and PureScript are elegant languages with powerful type systems that can prove a lot more about your program than TypeScript can. The code written in Elm and PureScript can be a lot terser than similar code written in ES5.

Each of these options has pros and cons, but I think TypeScript is in a sweet spot that makes it a great choice for most projects.

TypeScript takes 95% of the usefulness of a good statically-typed language and brings it to the JavaScript ecosystem. You still feel like you write ES6: you keep using the same standard library, same third-party libraries, same idioms, and many of the same tools (e.g., Chrome dev tools). It gives you a lot without forcing you out of the JavaScript ecosystem. ■

Senior engineers reduce risk

By ZACH TELLMAN

A senior engineer understands these risks, and mitigates them where possible.

If you read articles on career development in software, you'll already know there are many things a senior engineer is *not*.

They are not, for instance, just someone with ten years of experience. They are not just someone who maintains a popular open source project. They are not just someone who wrote the initial prototype for their company's flagship product.

Instead, senior engineers possess many disparate skills, the exact composition of which varies from author to author. These laundry lists are detailed, nuanced, and typically leave the reader feeling as if they have a long way to go.

Empirically, though, there are many people who have the title "senior engineer" who do not possess all these skills. Unless all of them can be explained by title inflation or poor management, this is a serious disconnect with reality.

We have to treat these articles as aspirational: a description of the industry as it should be. For someone wanting to make a significant, recognized impact on their company, today, these articles are more distracting than helpful.

Senior engineers reduce risk, in every sense. Often, "risk" is used to describe technical risk, which is that the software doesn't function properly, or is never completed at all.

But there are other issues that can prevent the business atop the software from succeeding—risks around process, or product design, or sales, or the company's culture.

A senior engineer understands these risks, and mitigates them where possible.

Senior engineers affect the larger system

Most startups die because they build the wrong product. The core risks are rarely technical; if no one wants the product, building it well won't change the outcome.

The most important thing an engineer can do during this time is either speed up the search, or narrow the search space. Building prototypes, gaining domain expertise, and generating metrics all lower the chance the company will run out of money.

Once a company finds product-market fit, the risks change. The product has to work, scale, and easily adapt to anything learned by the customer-facing employees.

New users introduce technical risk but, more importantly, new hires introduce risk into the company's processes and culture. If the company can't make new hires productive, it will collapse under the weight of its payroll.

The software will need to be broken into manageable pieces, or else new engineers will need to hold the entire growing system in their head to be productive. Where before there may have been one engineer who

In this environment, a senior engineer is an engineer who works within the processes, and even refines them, to ensure reliable output.

handled “talking to customers”, the planning process may now need to include salespeople.

Above all, each new hire dilutes any clarity that existed around what the company values, and what it doesn’t. Left alone, the company will become whatever people were used to at their last job.

As the company continues to grow, any new project will require collaboration across teams. After a few projects get delayed because of one missing piece, the leadership will start to value predictable results above all else.

Processes will be established, recurring “sync” meetings will be scheduled. Variance in productivity, above or below the mean, will be minimized. The largest risk is perceived to be a small component which is never completed, obviating all the other related work. In this environment, a senior engineer is an engineer who works within the processes, and even refines them, to ensure reliable output.

This narrative arc describes the average growing software startup, but no company is exactly average. The precise risks will vary depending on who founded the company, what it does, and who they hire.

The senior engineers may reduce risk deliberately, or simply have the right skill set at the right time. A fresh graduate with the right domain expertise can have a huge impact on the larger system without ever understanding it.

Senior engineers find leverage

Impact can be serendipitous, but the greatest impact comes from a deliberate attempt to effect change with the least effort. If production is unstable, precluding a common failure mode will have a more immediate impact than volunteering for pager duty.

If the company is rapidly hiring, improving and standardizing interview practices will have a broader impact than conducting a dozen phone screens every week.

If there are any junior engineers, spending an hour to talk over unfamiliar topics will have a longer lasting impact than writing more code.

Where possible, solving the general form of any problem is preferable to solving a single instance. However, this is only possible if the problem space is well understood.

Premature abstraction, in code or elsewhere, introduces risk. If the failure modes in production are poorly understood, volunteering for pager duty may be the only way to really help.

Senior engineers are storytellers

Most risks, especially where managed effectively, never come to pass. For this reason, it’s often impossible to tell the difference between real risk and

The expectation that "senior" is a fungible title is both widespread and harmful, leading to unrealistic expectations from both engineers and companies.

perceived risk.

By the same token, it can be difficult to tell the difference between real impact and perceived impact. If a senior engineer identifies a significant risk, they have to be able to concisely explain and prioritize it for a non-expert audience.

Some see this as a form of corporate politics; a skilled storyteller can protect their company from non-existent risks, and be seen as a hero. But the fact that storytelling can be misused doesn't make it invalid, just something that should be approached with a critical eye.

Senior engineers choose companies with the right risks

Every company has different risks, and so every company expects something different from their senior engineers. An engineer who has spent the last five years making small, continuous improvements to the processes in a larger company may not enjoy or even understand the sort of role expected by a three-person startup.

The expectation that "senior" is a fungible title is both widespread and harmful, leading to unrealistic expectations from both engineers and companies.

Through trial and error, engineers can identify the sorts of problems they enjoy solving. A senior engineer should find a company that both has those problems, and knows it. Anything else will lead to frustration, and eventually apathy.

Likewise, companies should try to communicate their risks, early and often. If we were to judge by what's asked in interviews, most companies believe their only risks are technical, but that's absurd.

The interviewers know the real problems facing the company, but the polite fiction is that they're only temporary, and soon everyone will be able to focus entirely on the algorithms, which are what really matter.

Since most companies tell the same story, candidates have to read between the lines to see if there's a good fit. Honesty would make things much simpler.

Senior engineers know titles don't mean much

Without context, knowing someone was a "senior engineer" tells you almost nothing. Titles can matter a lot in some environments, but in those cases titles are just another tool for reducing risk; if being a senior, or staff, or principal engineer is the only way to make your voice heard, then pursuing those titles is worthwhile. More often, though, it's just a distraction. ■

Reprinted with permission of the original author. First appeared at medium.com/@ztellman.

The truth about deep learning

By CLAY MCLEOD

Even the most academic among us mistakenly merge two very different schools of thought in our discussions on deep learning.

Come on people — let's get our act together on deep learning. I've been studying and writing about DL for close to two years now, and it still amazes me the misinformation surrounding this relatively complex learning algorithm.

This post is not about how deep learning is or is not overhyped, as that is a well documented debate. Rather, it's a jumping off point for a (hopefully) fresh, concise understanding of deep learning and its implications. This discussion/rant is somewhat off the cuff, but the whole point is to encourage those of us in the machine learning community to *think clearly* about deep learning.

Let's be bold and try to make some claims based on actual science about whether or not this technology will or will not produce artificial intelligence. After all, aren't we supposed to be the leaders in this field and the few that understand its intricacies and implications?

With all of the news on artificial intelligence breakthroughs and non-industry commentators making rash conclusions about how deep learning will change the world, don't we owe it to the world to at least have our shit together? It feels like most of us are just sitting around waiting for others to figure that out for us.

[Note added on 05/05/16: Keep in mind that this is a blog post, not an academic paper. My goal was to express my thoughts and inspire some discussion about how we should contextualize deep learning, not to lay out a deeply technical argument. Obviously, a discussion of that magnitude could not be achieved in a few hundred words, and this post is aimed at the machine learning layman nonetheless. I leave that technical discussion as an exercise to the readers (feel free to email me). One article cannot be all things to all people.]

The problem

Even the most academic among us mistakenly merge two very different schools of thought in our discussions on deep learning:

1. The benefits of neural networks over other learning algorithms.
2. The benefits of a "deep" neural network architecture over a "shallow" architecture.

Much of the debate going on is surprisingly still concerned with the first point instead of the second. Let's be clear —

What does increasing computing power and adding layers to a neural network actually allow us to do better than a normal neural network?

the inspiration for, benefits of, and detriments against neural networks *are all well documented in the literature*. Why are we still talking about this like the discussion is new?

Nothing is more frustrating when discussing deep learning than someone explaining their views on why deep neural networks are "modeled after how the human brain works" (much less true than the name suggests) and thus are "the key to unlocking true artificial intelligence." This is an obvious straw man, since this discussion is essentially the same as was produced when plain old neural networks were introduced.

The idea I'd like for you to take away here is that we are not asking the right question for the answer which we desire. If we want to know how one can contextualize deep neural networks in the ever-increasing artificially intelligent world, we must answer the following question: *what does increasing computing power and adding layers to a neural network actually allow us to do better than a normal neural network?*

Answering this question could yield a truly fruitful discussion on deep learning.

The answer (?)

Here is my personal answer to the second question — deep neural networks are more useful than traditional neural networks for two reasons:

1. The automatic encoding of features which previously had to be hand engineered.
2. The exploitation of structurally/spatially associated features.

At the risk of sounding bold, that's it — if you believe there is another benefit which is not somehow encompassed by these two traits, please let me know. These are the only two that I have come across in all my time working with deep learning.

[Note added 05/05/16: In the responses, the most common possible third benefit is the configurability of layers within a model. Although this is quite true, it's not new, nor is it philosophically unique to deep learning. Fundamentally, this is just a more frictionless version of pipelining, which we have been doing for a while. I have not (yet) heard a good argument against this proof by decomposition.]

If this was true, what would we expect to see in the academic landscape? We might expect that deep neural networks would be useful in situations where the

Start viewing deep learning for what it truly is: a tool that is useful in assisting a computer's ability to perceive.

data has some spatial qualities that can be exploited, such as image data, audio data, natural language processing, etc.

Although we might say there are many areas that could benefit from that spatial exploitation, we would certainly not find that this algorithm is a magical cure for any data that you throw at it.

The words "deep learning" will not magically cure cancer (unless you find some way to spatially exploit data associated with cancer, as had been done with the human genome), there is no reason to believe it will start thinking and suddenly become sentient.

We might see self-driving cars that assist in simply keeping the car between the lines, but not one which can decide whether to protect its own driver or the pedestrian walking the street.

Hell, even those that actually read the papers on AlphaGo will realize that deep learning was simply a tool used by traditional AI algorithms. Lastly, we might find that, once again, that the golden mean is generally spot on, and that deep learning is not the answer to all machine learning problems, but also not completely baseless.

Since I am feeling especially bold, I will make another prediction: *deep learning will not produce the universal algorithm*. There is simply not enough there to create such a complex system.

However, deep learning *is* an extremely useful tool. Where will it be most useful in AI? I predict it will be as a sensory learning system (vision, audio, etc.) that exploits some spatially [MK2] characteristics in data that would otherwise go unaccounted for, which, like in AlphaGo, must be used by a truly artificially intelligent system as an input.

Stop studying deep learning thinking it will lay all other algorithms to waste no matter the scenario. Stop throwing deep learning at every dataset you see. Start experimenting with these technologies outside of the "hello world" examples in the packages you use — you will quickly learn what they are actually useful for.

Most of all, let's stop viewing deep learning as the "almost there!!!" universal algorithm and start viewing it for what it truly is: a tool that is useful in assisting a computer's ability to perceive. ■

Reprinted with permission of the original author. First appeared at blog.claymcleod.io.

Photograph by fruity monkey via flickr.com

A beginners guide to thinking in SQL

By SOHAM CHETAN KAMANI

```
SELECT members.firstname || ' ' || members.lastname
AS "Full Name"
FROM borrowings
JOIN members
ON members.memberid=borrowings.memberid
JOIN books
ON books.bookid=borrowings.bookid
WHERE borrowings.bookid IN (SELECT bookid
  FROM books
  WHERE stock>(SELECT avg(stock)
    FROM books))
GROUP BY members.firstname, members.lastname;
```

▲ Figure 1: SQL gibberish

*Is it "SELECT * WHERE a=b FROM c" or "SELECT WHERE a=b FROM c ON *" ?*

If you're anything like me, SQL is one of those things that may look easy at first (it reads just like regular English!), but for some reason you can't help but google the correct syntax for every silly query.

Then, you get to joins, aggregation, and subqueries and everything you read just seems like gibberish. Something like Figure 1.

Yikes! This would scare any newcomer, or even an intermediate developer looking at SQL for the first time. It shouldn't have to be like this.

It's always easy to remember something which is intuitive, and through this guide, I hope to ease the barrier of entry for SQL newbies, and even for people who have worked with SQL, but want a fresh perspective.

All queries used in this post are made for PostgreSQL, although SQL syntax is very similar across databases, so some of these would work on MySQL, or other SQL databases as well.

1. The three magic words

Although there are lots of keywords used in `SQL`, `SELECT`, `FROM`, and `WHERE` are the ones you would be using in almost every query that you make. After reading ahead, you would realize that these key words represent the most fundamental aspect of querying a database, and other, more complex queries are simply extensions of them.

2. Our database

Let's take a look at the sample data we will be using throughout the rest of this article (see Figure 2).

We have a library, with books and members. We also have another table for borrowings made.

- Our "books" table has information about the title, author, date of publication, and stock available. Pretty straightforward.
- Our "members" table only has the first and last name of all registered members.
- The "borrowings" table has information on the books borrowed by the members. The `bookid` column refers to the id of the book in the "books" table that was borrowed, and the `memberid` column corresponds to the member in the "members" table that borrowed the book. We also have the dates when the books were borrowed, and when they are expected to be returned.

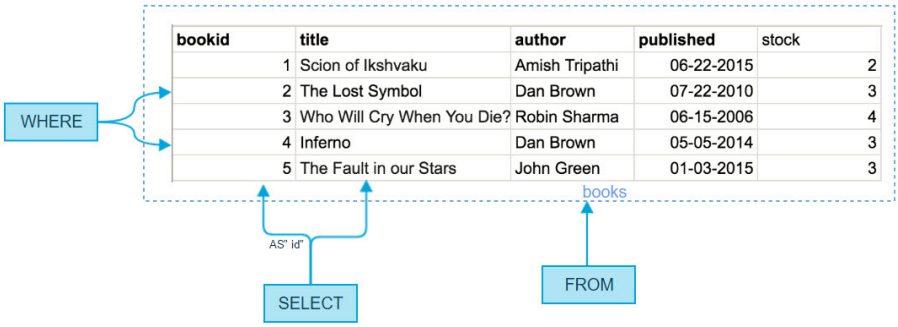
3. Simple Query

Let's get started with our first query: We want the *names* and *ids* of all books written by "Dan Brown".

Our query would be :

```
SELECT bookid AS "id", title
FROM books
WHERE author='Dan Brown';
```

Which would give us:



▲ Figure 2: Sample data

id	title
2	The Lost Symbol
4	Inferno

Simple enough. However, let's try to dissect the query to really understand what's happening.

3.1 FROM – Where do we get the data from?

This might seem obvious now, but it actually matters a lot when we get to joins and subqueries. `FROM` is there to point our query to its table, the place where it has to look for the data. This table can simply be one that already exists (like the previous example), or a table which we generate through joins or subqueries.

3.2 WHERE – What data should we show?

`WHERE`, quite simply acts to filter out the *rows* that we want to show. In our case the only rows we want to consider are those where the value of the `author` column is "Dan Brown."

3.3 SELECT – How should we show it?

Now that we got all the rows we wanted from the table that we wanted, the question that arises is what exactly do we want to show out of the data that we got? In our case we only need the name and id of the book, so that's what we `SELECT`. We can also rename the columns we want to show with `AS`.

In the end, you can represent the entire query as a simple diagram (see Figure 3).

4. Joins

We would now like to see the names of all books (not unique) written by "Dan Brown" that were borrowed, along with the date of return:

```
SELECT books.title AS "Title",
borrowings.returndate AS "Return Date"

FROM borrowings JOIN books ON
borrowings.bookid=books.bookid

WHERE books.author='Dan Brown';
```

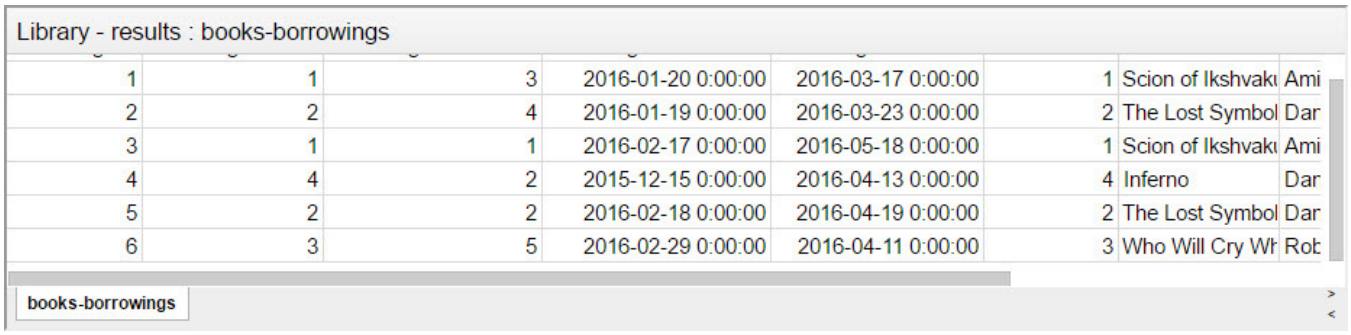
Which would give us:

Title	Return Date
The Lost Symbol	2016-03-23 00:00:00
Inferno	2016-04-13 00:00:00
The Lost Symbol	2016-04-19 00:00:00

Most of the query looks similar to our previous example *except* for the `FROM` section. What this means is that *the table we are querying from has changed*. We are neither querying the "books" table nor the "borrowings" table. Instead, we are querying a *new table* formed by joining these two tables.

`borrowings JOIN books ON borrowings.bookid=books.bookid` can be considered as another table formed by combining all entries from the books table and the borrowings table, as long as these entries have the same `bookid` in each table. The resultant table would be (see Figure 4).

Now, we just query this table like we did in the simple example above. This means that every time we join a table, we just have to worry about how we join our tables. After that, the query is reduced



▲ Figure 3: Query as a simple diagram

in complexity to the level of the "Simple Query" example.

Let's try a slightly more complex join where we join 2 tables.

We now want the first name and last name of everyone who has borrowed a book written by "Dan Brown".

This time, we'll take a bottom-up approach to get our result:

- *Step 1* — Where do we get the data from? To get the result we want, we would have to join the "member" table, as well as the "books" table with the "borrowings" table. The join section of the query would look like:

```
borrowings
JOIN books ON borrowings.bookid=books.bookid
JOIN members ON members.memberid=borrowings.
memberid
```

See Figure 5 for the resulting table.

- *Step 2* – What data should we show? We are only concerned with data where the author is "Dan Brown".

```
WHERE books.author='Dan Brown'
```

- *Step 3* – How should we show it? Now that we got the data we want, we just want to show the first name and the last name of the members who borrowed it:

```
SELECT
members.firstname AS "First Name",
members.lastname AS "Last Name"
```

Awesome! Now we just have to combine the 3 components of our query and we get:

```
SELECT
members.firstname AS "First Name",
members.lastname AS "Last Name"
FROM borrowings
JOIN books ON borrowings.bookid=books.bookid
JOIN members ON members.memberid=borrowings.
memberid
WHERE books.author='Dan Brown';
```

Which gives us:

First Name	Last Name
Mike	Willis
Ellen	Horton
Ellen	Horton

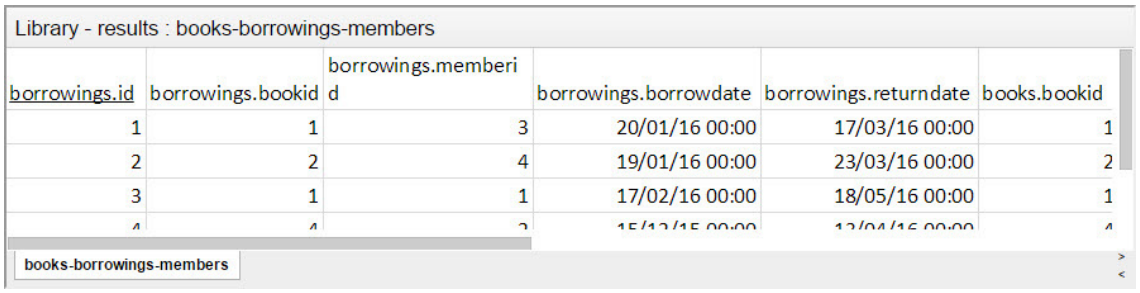
Awesome! However, the names are repeating (non-unique). We'll get to solving that in a bit...

5. Aggregations

In a nutshell, *aggregations are used to convert many rows into a single row*. The only thing that changes is the logic used on each column for its aggregation.

Let's continue with our previous example, where we saw that there were repetitions in the results we got from our query. We know that Ellen Horton borrowed more than one book, but this is not really the best way to show this information. We can write another query:

```
SELECT
members.firstname AS "First Name",
```



▲ Figure 4: Resultant table from JOIN

```
members.lastname AS "Last Name",
count(*) AS "Number of books borrowed"
FROM borrowings
JOIN books ON borrowings.bookid=books.bookid
JOIN members ON members.memberid=borrowings.
memberid
WHERE books.author='Dan Brown'
GROUP BY members.firstname, members.lastname;
```

Which would give us our required result:

First Name	Last Name	Number of books borrowed
Mike	Willis	1
Ellen	Horton	2

Almost all aggregations we do come with the `GROUP BY` statement. What this does is convert the table otherwise returned by the query into groups of tables. Each group corresponds to a unique value (or group of values) of columns, which we specify in the `GROUP BY` statement.

In this example, we are converting the result we got in the previous exercise into groups of rows. We also perform an aggregation in this case `count`, which converts multiple rows into a single value (which in our case is the number of rows). This value is then attributed to each group.

Each row in the result represents the aggregated result of each of our groups (see Figure 5).

We can also logically come to the conclusion that all fields in the result must either be specified in the `GROUP BY` statement, or have an aggregation done on them. This is because all other fields will vary row wise, and if they were `SELECT`ed, we wouldn't know which of their possible values to take.

In the above example, the `count` function worked on all rows (since we are only counting the number of rows). Other functions like `sum` or `max` would work on only a specific row. For example, if we want the total stock of all books written by each author, we would query:

```
SELECT author, sum(stock)
FROM books
GROUP BY author;
```

And get the result:

author	sum
Robin Sharma	4
Dan Brown	6
John Green	3
Amish Tripathi	2

Here the `sum` function, only works on the `stock` column, summing all values for each group.

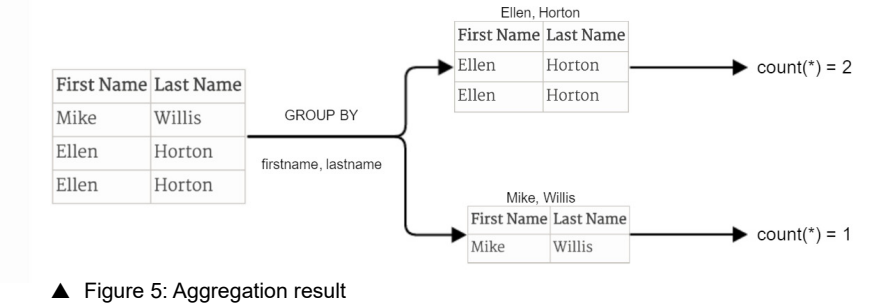
6. Subqueries

Sub queries are regular SQL queries that are embedded inside larger queries.

There are 3 different types of subqueries, based on what they return.

6.1 Two-dimensional table

These are queries that return more than one column. A good example is the query we performed in the previous aggregation exercise. As a subquery, these simply return another table, which can be queried further. From the previous exercise, if we only want the stock of books written by "Robin



▲ Figure 5: Aggregation result

Sharma", one way of getting that result would be to use sub-queries:

```
SELECT *
FROM (SELECT author, sum(stock)
  FROM books
  GROUP BY author) AS results
WHERE author='Robin Sharma';
```

Result:

author	sum
Robin Sharma	4

6.2 One-dimensional array

Queries which return multiple rows of a single column can be used as arrays, in addition to being used as two-dimensional tables.

For example, let's say we want to get the titles and ids of all books written by an author, whose total stock of books is greater than 3.
We can break this down into 2 steps:

1. Get the list of authors with total stock of books greater than 3. Building on top of our previous example, we can write:

```
SELECT author
FROM (SELECT author, sum(stock)
  FROM books
  GROUP BY author) AS results
WHERE sum > 3;
```

Which gives us:

author
Robin Sharma
Dan Brown

Which can also be written as: `['Robin Sharma', 'Dan Brown']`

2. We then use this result in our next query:

```
SELECT title, bookid
FROM books
WHERE author IN (SELECT author
  FROM (SELECT author, sum(stock)
  FROM books
  GROUP BY author) AS results
  WHERE sum > 3);
```

Which gives us:

title	bookid
The Lost Symbol	2
Who Will Cry When You Die?	3
Inferno	4

This is equivalent to writing:

```
SELECT title, bookid
FROM books
WHERE author IN ('Robin Sharma', 'Dan Brown');
```

6.3 Single values

These are queries whose results have only one row and one column. These can be treated as a constant value, and can be used anywhere a value is used, like for comparison operators. They can also be used like two-dimensional tables, as well

Most of the write operations in a database are pretty straightforward, as compared to the more complex read queries.

as an array containing 1 element.

As an example, let's find out information about all books having stock above the average stock of books present.

The average stock can be queried using:

```
select avg(stock) from books;
```

Which gives us:

avg
3.000

Which can also be used as the scalar value `3`. So now, we can finally write our query:

```
SELECT *
FROM books
WHERE stock>(SELECT avg(stock) FROM books);
```

Which is equivalent to writing:

```
SELECT *
FROM books
WHERE stock>3.000
```

And which gives us:

bookid	title	author	published	stock
3	Who Will Cry When You Die?	Robin Sharma	2006-06-15 00:00:00	4

7. Write operations

Most of the write operations in a database are pretty straightforward, as compared to the more complex read queries.

7.1 Update

The syntax of `UPDATE` queries is semantically similar to read queries. The only difference however, is that instead of `SELECT`ing columns from a bunch of rows, we `SET` those columns instead.

If we suddenly lost all of our books written by "Dan Brown", we would like to update the stock to make it 0. For this we would write:

```
UPDATE books
SET stock=0
WHERE author='Dan Brown';
```

`WHERE` still performs the same function here, which is that of choosing rows. Instead of `SELECT` which we used in our read queries, we now use `SET`. However, now, in addition to mentioning the column names, you also have to mention the new value of the columns in the selected rows as well.

7.2 Delete

A `DELETE` query is simply a `SELECT`, or an `UPDATE` query without the column names. Seriously. As in `SELECT` and `UPDATE`, the `WHERE` clause remains as it is, selecting rows to be deleted. Since a delete operation removes an entire row, there is no such thing as specifying column names to delete. Hence, instead of updating the stock to 0, if we just deleted the entries from Dan Brown all together, we would write:

Possibly the only outlier from the other query types is the INSERT query.

```
DELETE FROM books
WHERE author='Dan Brown';
```

7.3 Insert

Possibly the only outlier from the other query types is the `INSERT` query. Its format is:

```
INSERT INTO x
  (a,b,c)
VALUES
  (x, y, z);
```

Where `a`, `b`, `c` are the column names and `x`, `y`, and `z` are the values to be inserted into those columns, in the same order in which they are specified. That's pretty much all there is to it.

For a more concrete example, here is the `INSERT` query to input the entire data of the "books" table:

```
INSERT INTO books
  (bookid,title,author,published,stock)
VALUES

  (1,'Scion of Ikshvaku','Amish Tripa-
thi','06-22-2015',2),

  (2,'The Lost Symbol','Dan
Brown','07-22-2010',3),

  (3,'Who Will Cry When You Die?','Robin Shar-
ma','06-15-2006',4),

  (4,'Inferno','Dan Brown','05-05-2014',3),

  (5,'The Fault in our Stars','John
Green','01-03-2015',3);
```

8. Feedback

Now that we have come to the end of the guide, it's time for a small test. Take a look at the first query at the very beginning of this post. Can you try to figure out what it does? Try breaking it down into its `SELECT`, `FROM`, `WHERE`, `GROUP BY`, and subquery components.

Here it is written in a more readable way:

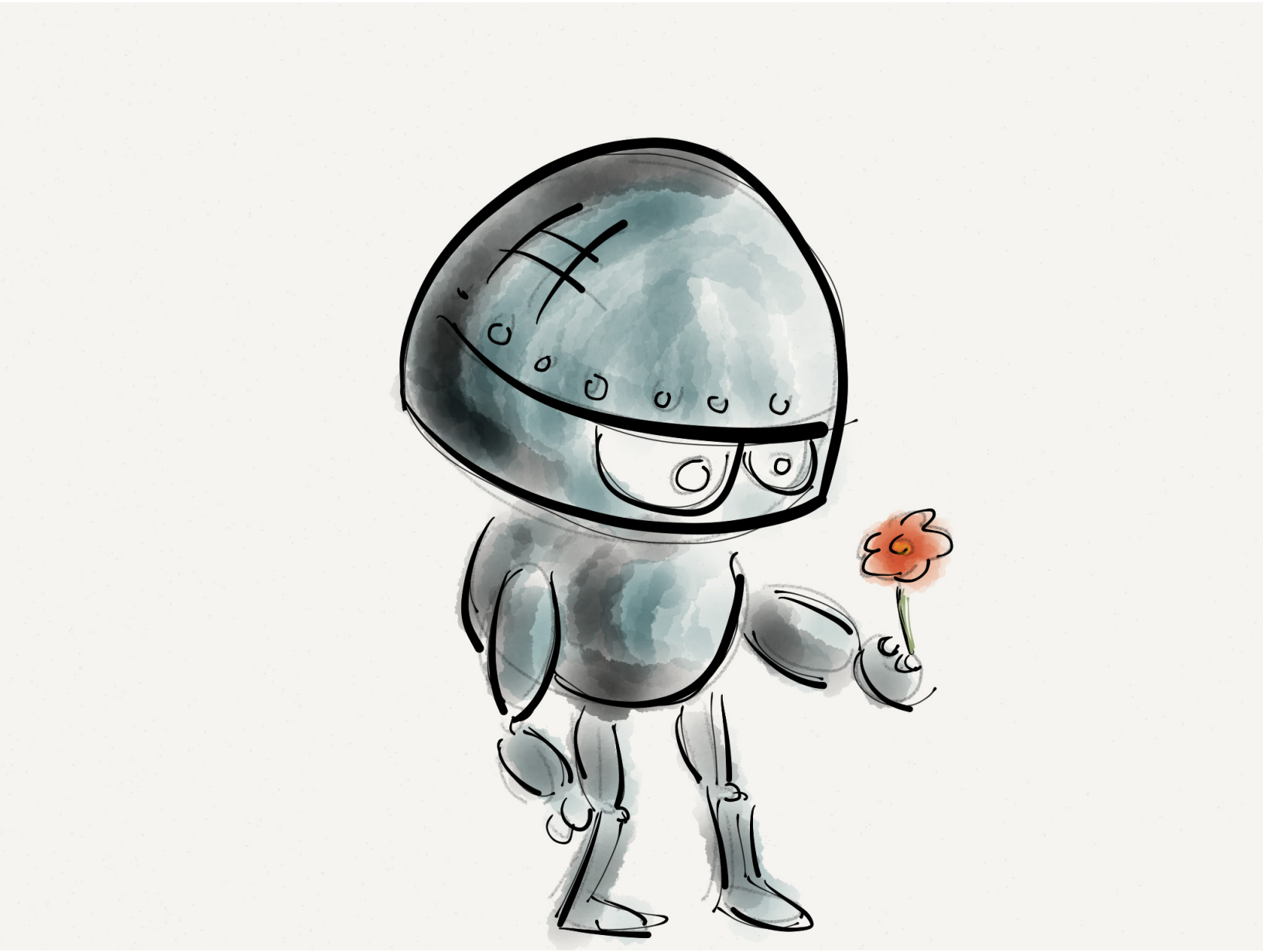
```
SELECT members.firstname || ' ' || members.last-
name AS "Full Name"

FROM borrowings
JOIN members
ON members.memberid=borrowings.memberid
JOIN books
ON books.bookid=borrowings.bookid

WHERE borrowings.bookid IN (SELECT bookid FROM
books WHERE stock>  (SELECT avg(stock) FROM
books)  )

GROUP BY members.firstname, members.lastname;
```

It's actually the list of all members who borrowed any book with a total stock that was above average.

Result:

Full Name
Lida Tyler

Hopefully, you were able to get the answer no sweat, but if not, I would love your feedback or comments on how I could make this post better. Cheers! ■



AI, Apple and Google

By BENEDICT EVANS

Machine learning offers the promise that a lot of very large and very boring analyses of data can be automated.

(Note: for a good introduction to the history and current state of AI, see my colleague Frank Chen's presentation here*.)*

In the last couple of years, magic started happening in AI. Techniques started working, or started working much better, and new techniques have appeared, especially around machine learning ('ML'), and when those were applied to some long-standing and important use cases, we started getting dramatically better results.

For example, the error rates for image recognition, speech recognition and natural language processing have collapsed to close to human rates, at least on some measurements.

So you can say to your phone: 'show me pictures of my dog at the beach' and a speech recognition system turns the audio into text, natural language processing takes the text, works out that this is a photo query and hands it off to your photo app, and your photo app, which has used ML systems to tag your photos with 'dog' and 'beach', runs a database query and shows you the tagged images. Magic.

There are really two things going on here – you're using voice to fill in a dialogue box for a query, and that dialogue box can run queries that might not have been possible before. Both of these are enabled by machine learning, but they're built quite separately, and indeed the most interesting part is not the voice but the query.

In fact, the important structural change behind being able to ask for 'pictures with dogs at the beach' is not that the computer can find it but that the computer has worked out, itself, how to find it. You give it a million pictures labelled 'this has a dog in it' and a million labelled 'this doesn't have a dog' and it works out how to work out what a dog looks like.

Now, try that with 'customers in this data set who were about to churn', or 'this network had a security breach', or 'stories that people read and shared a lot'. Then try it without labels ('unsupervised' rather than 'supervised' learning).

Today you would spend hours or weeks in data analysis tools looking for the right criteria to find these, and you'd need people doing that work – sorting and re-sorting that Excel table and eyeballing for the weird result, metaphorically speaking, but with a million rows and a thousand columns.

Machine learning offers the *promise* that a lot of very large and very boring analyses of data can be automated – not just running the search, but working out what the search should be to find the result you want.

That is, the eye-catching demos of speech interfaces or image recognition are just the most visible demos of the underlying techniques, but those have much broader applications – you can also apply them to a keyboard, a music recommendation system, a network security model or a self-driving car. Maybe.

This is clearly a fundamental change for Google. Narrowly,

While we have magic, we do not have HAL 9000 – we do not have a system that is close to human intelligence.

image and speech recognition mean that it will be able to understand questions better and index audio, images and video better. But more importantly, it will answer questions better, and answer questions that it could never really answer before at all. Hence, as we saw at Google IO, the company is being re-centred on these techniques.

And of course, all of these techniques will be used in different ways to varying degrees for different use cases, just as AlphaGo uses a range of different techniques. The thing that gets the attention is 'Google Assistant — a front-end using voice and analysis of your behaviour to try both to capture questions better and address some questions before they're asked.

But that's just the tip of the spear — the real change is in the quality of understanding of the corpus of data that Google has gathered, and in the kind of queries that Google will be able to answer in all sorts of different products. That's really just at the very beginning right now.

The same applies in different ways to Microsoft, which (having missed mobile entirely) is creating cloud-based tools to allow developers to build their own applications on these techniques, and for Facebook (what is the newsfeed if not a machine learning application?), and indeed for IBM. Anyone who handles lots of data for money, or helps other people do it, will change, and there will be a whole bunch of new companies created around this.

On the other hand, while we have magic we do not have HAL 9000 — we do not have a system that is close to human intelligence (so-called 'general AI'). Nor really do we have a good theory as to what that would mean — whether human intelligence is the sum of techniques and ideas we already have, but more, or whether there is something else. Rather, we have a bunch of tools that need to be built and linked together.

I can ask Google or Siri to show me pictures of my dog on a beach because Google and Apple have linked together tools to do that, but I can't ask it to book me a restaurant unless they've added an API integration with OpenTable. This is the fundamental challenge for Siri, Google Assistant or any chat bot (as I discussed here) — what can you ask?

This takes us to a whole class of jokes often made about what does and does not count as AI in the first place:

- "Is that AI or just a bunch of IF statements?"
- "Every time we figure out a piece of it [AI], it stops being magical; we say, 'Oh, that's just a computation'
- "AI is whatever hasn't been done yet"

These jokes reflect two issues. The first is that it's not totally apparent that human intelligence itself is actually more than 'a bunch of IF statements', of a few different kinds and at very large scale, at least at a conceptual level.

But the second is that this

A foundational point here is Eric Raymond's rule that a computer should 'never ask the user for any information that it can auto detect, copy, or deduce.'

movement from magic to banality is a feature of all technology and all computing, and it doesn't mean that it's not working but that it is. That is, *technology* is in a sense anything that hasn't been working for very long. We don't call electricity technology, nor a washing machine a robot, and you could replace "is that AI or just computation?" with "is that technology or just engineering?"

I think a foundational point here is Eric Raymond's rule that a computer should 'never ask the user for any information that it can auto detect, copy, or deduce' — especially, here, deduce. One way to see the whole development of computing over the past 50 years is by removing questions that a computer needed to ask, and adding new questions that it could ask.

Lots of those things didn't necessarily look like questions as they're presented to the user, but they were, and computers don't ask them anymore:

- Where do you want to save this file?
- Do you want to defragment your hard disk?
- What interrupt should your sound card use?
- Do you want to quit this application?
- Which photos do you want to delete to save space?
- Which of these 10 search criteria do you want to fill in to run a web search?
- What's the PIN for your phone?
- What kind of memory do you want to run this program in?
- What's the right way to spell that word?
- What number is this page?
- Which of your friends' updates do you want to see?

It strikes me sometimes, as a reader of very old science fiction, that sci-fi did indeed mostly miss computing, but it talked a lot about 'automatic'. If you look at that list, none of the items really look like 'AI' (though some might well use it in future), but a lot of them are 'automatic'.

And that's what any 'AI' short of HAL 9000 really is — the automatic pilot, the automatic spell checker, the automatic hardware configuration, the automatic image search or voice recogniser, the automatic restaurant-booker or cab-caller...

They're all clerical work your computer doesn't make you do anymore, because it gained the intelligence, artificially, to do them for you.

This takes me to Apple.

Apple has been making computers that ask you fewer questions since 1984, and people have been complaining about that for just as long — one user's question is another user's free choice (something you can see clearly in the contrasts between iOS and Android today).

Steve Jobs once said that the interface for iDVD should just have one button: 'BURN'. It launched Data Detectors in 1997 — a framework that tried

to look at text and extract structured data in a helpful way — appointments, phone numbers or addresses. Today you'd use AI techniques to get there, so was that AI? Or a 'bunch of IF statements'? Is there a canonical list of algorithms that count as AI? Does it matter? To a user who can tap on a number to dial instead of copy and pasting, is that a meaningful question?

This certainly seems to be one way that Apple is looking at AI on the device. In iOS 10, Apple is sprinkling AI through the interface. Sometimes this is an obviously new thing, such as image search, but more often it's an old feature that works better or a small new feature to an existing application. Apple really does seem to see 'AI' as 'just computation'.

Meanwhile (and this is what gets a lot of attention) Apple has been very vocal that companies should not collect and analyse user data, and has been explicit that it is not doing so to provide any of these services. Exactly what that means varies a lot.

Part of the point of neural networks is that training them is distinct from running them. You can train a neural network in the cloud with a vast image set at leisure, and then load the trained system onto a phone and run it on local data without anything leaving the device. This, for example, is how Google Translate works on mobile — the training is done in advance in the cloud but the analysis is local.

Apple says it's doing the same for Apple Photos — 'it turns out we don't need *your* photos of mountains to train a system to recognize mountains. We can get our own pictures of mountains'. It also has APIs to allow developers to run pre-trained neutral networks locally with access to the GPU. For other services it's using 'differential privacy', which uses encryption to obfuscate the data such that though it's collected

A common thread...is that eventually many 'AI' techniques will be APIs and development tools across everything...

by Apple and analyses at scale, you can't (in theory) work out which users it relates to.

The sheer variety of places and ways that Apple is doing this, and the different techniques, makes it pretty hard to make categorical statements on the lines of 'Apple is missing this'. Apple has explicitly decided to do at least some of this with one hand tied behind its back, but it's not clear how many services that really affects, or how much.

Maybe you don't need my photos of mountains, but how about training to recognize my son — is that being done on the device? Is the training data being updated? How much better is Google's training data? How much would it benefit from that?

Looking beyond just privacy, this field is moving so fast that it's not easy to say where the strongest leads necessarily are, or to work out which things will be commodities and which will be strong points of difference.

Though most of the primary computer science around these techniques is being published and open-sourced, the implementation is not trivial — these techniques are not necessarily commodities, yet.

But there's definitely a contrast with Apple's approach to chip design. Since buying PA Semi in 2008 (if not earlier) Apple has approached the design of the SOCs in its devices as a fundamental core competence and competitive advantage, and it now designs chips for itself that are unquestionably market-leading (which, incidentally, will be a major advantage when it launches a VR product). It's not clear whether Apple looks at 'AI' in the same way.

There's also, perhaps cynically, a 'power of defaults' issue here — if Google Photos is always 10-15% better than Apple Photos at object classification, will users notice beyond a certain level of shared accuracy? After all, Apple Maps has 3x more users than Google Maps on the iPhone and Google Maps is *definitely* better. And is any Google lead offset by, say, Apple's Photo Stream or other features layered on top? Again, little of this is clear yet.

A common thread for both Apple and Google, and the apps on their platforms, is that eventually many 'AI' techniques will be APIs and development tools across everything, rather like, say, location.

15 years ago geolocating a mobile phone was witchcraft and mobile operators had revenue forecasts for 'location-based services'. GPS and Wi-Fi-lookup made LBS just another API call: 'where are you?' became another question that a computer never has to ask you.

But although location became just an API — just a database lookup — *just another IF statement* — the services created with it sit on a spectrum. At one end are things like Foursquare — products that are only possible with real-time location and use it to do magic. Slightly behind are Uber or Lyft — it's useful for Lyft to know where you are when you call a car, but not essential (it *is* essential for

It will try to give you the answer itself, and it will also try to give you that answer before you asked the question.

the drivers' app, of course).

But then there's something like Instagram, where location is a free nice-to-have — it's useful to be able to geotag a photo automatically, but not essential and you might not want to anyway. (Conversely, image recognition is going to transform Instagram, though they'll need a careful taxonomy of different types of coffee in the training data). And finally, there is, say, an airline app, that *can* ask you what city you're in when you do a flight search, but really needn't bother.

In the same way, there will be products that are only possible because of machine learning, whether applied to images or speech or something else entirely (no one looked at location and thought 'this could change taxis").

There will be services that are enriched by it but could do without, and there will be things where it may not be that relevant at all (that anyone has realised yet). So, Apple offers photo recognition, but also a smarter keyboard and venue suggestions in the calendar app — it's sprinkled 'AI' all over the place, much like location. And, like any computer science tool, there will be techniques that are commodities and techniques that aren't, yet.

All of this, so far, presumes that the impact of AI forms a sort of T-shaped model: there will be a vertical, search, in which AI techniques are utterly transformative, and then a layer across everything where it changes things (much as location did).

But there's another potential model in which AI becomes a new layer for the phone itself — that it changes the interaction model and relocates services from within app silos to a new runtime of some sort. Does it change the layer of aggregation on the phone — it makes apps better, but does it change what apps are? That's potentially much more destabilizing to the model that Apple invented.

Clearly in some cases the answer is 'yes'. At the very least, structural changes in what search means change the competitive landscape and destabilize the mix between Google's general purposes search and its vertical competitors: a Yelp search might become a Google question, or an answer offered before you've even asked.

This is another case of removing a question — instead of Google offering you ten search results that it thinks might answer your question, it will try to give you the answer itself, and it will also try to give you that answer before you asked the question.

More interestingly, an Uber or Lyft request, or an OpenTable booking, might also be re-aggregated from an app to a suggestion or answer within a voice UI or, say, Maps.

An app with one button — that asks one simple question — could become a request pretty easily, whether in Google Assistant, Siri, Apple or Googles Maps or a messaging app. In fact, one way to look at Apple's opening of APIs into Maps, Siri, Messen-

We have excitement and bullshit, skepticism and vision, and a bunch of amazing companies being created.

ger and so on is as a counter to Google.

Where Google will find you a cinema, restaurant or hotel itself, Apple will lean on developers to solve the same use case. Google Allo suggests a restaurant where Apple's iMessage gives you an OpenTable plugin.

How broad is this, though? Yes, you could tell Siri or Google Assistant to 'show me all the new Instagram posts', but why is putting that inside to a feed of responses to all your questions a better UI? Why is Google's ML interface a better place to see this than Chrome designed by Instagram?

ML might (indeed, will) make the Facebook newsfeed better, but does it remove the difference between one-to-many and one-to-one communication channels? Why is the general-purpose rendering layer better than the special-purpose one? Does being subsumed into a general purpose ML layer change this?

One could propose a rebundling because it allows an easier interface — your home screen could show you documents, emails and meetings of the day instead of you having to go into each app to deal with them.

Perhaps 'which app do you want to open next?' is a question that can be moved — the car is ordered, the meeting accepted, the expense report approved. This is already what Facebook does for a whole section of interaction, and before ML — what shared posts to see, who to talk to, what news to read. But it's not the only thing on the phone. And, again, we don't have HAL 9000.

We don't actually have a system that knows you, and everything you want, and everything inside all of your apps, and we're not anywhere close to that. So the idea that Google can subsume everything you do on your phone into a single unified AI-based layer that sits on top looks rather like what one might call a 'Dr. Evil plan' — it's too clever by half and needs technology (the killer laser satellite!) that doesn't actually exist.

It seems to me that there are two things that make it hard to talk about the AI explosion. The first is that 'AI' is an impossibly broad term that implies we have a new magic hammer that turns every problem into a nail.

We don't — we have a bunch of new tools that solve some, but not all, problems, and the promise of extracting new insight from all sorts of data pools will not always be met. It might be the wrong data, or the wrong insight.

The second is that this field is growing and changing very fast, and things that weren't working now are, and new things are being discussed all the time. So we have excitement and bullshit, skepticism and vision, and a bunch of amazing companies being created. Some of this stuff will be in everything and you won't even notice it, and some of it will be the next Amazon. ■

Reprinted with permission of the original author. First appeared at ben-evans.com.

The dreaded weekly status email

By CHRISTINA WODTKE

I did what everyone does: I listed every single thing my reports did, and made a truly unreadable report.

I remember the first time I had to write one of these puppies.

I had just been promoted to manager at Yahoo back in 2000, and was running a small team. I was told to "write a status email covering what your team has done that week, due friday." Well, you can easily imagine how I felt. I had to prove my team was getting things done! Not only to justify our existence, but to prove we needed more people. Because, you know, more people ami-right?*

So I did what everyone does: I listed every single thing my reports did, and made a truly unreadable report. Then I started managing managers, and had them send me the same, which I collated into an even longer and more horrible report. This I sent to my design manager, Irene Au and my GM, Jeff Weiner (who sensibly requested I put a summary at the top. Go Jeff!)

And so it went, as I moved from job to job, writing long tedious reports that at best got skimmed. At one job, I stopped authoring them. I had my managers send them to my project manager, who collated them, sent it to me for review, and after checking for anything embarrassing, I forwarded it on to my boss. One week I forgot to read it, and didn't hear anything about it. It was a waste of everybody's time.

Then I got to Zynga in 2010. Now say what you want about Zynga (and much of it was true) but they were really good at some critical things that make an organization run well. One was the status report. All reports were sent to the entire management team, and I enjoyed reading them. Yes, you heard me right: *I enjoyed reading them*, even if when there were 20 of them.

Why? Because they had important information laid out in a digestible format. I used them to understand what I needed to do, and learn from what was going right. Please recall that Zynga, in the early days, grew faster than any company I've seen. I suspect the efficiency of communication was a big part of that. When I left Zynga, I started to consult. I adapted the status mail to suit the various companies I worked with, throwing in some tricks from Agile. Now I have a simple, solid format that works across any org, big or small.

1. **Lead with your team's OKRs, and how much confidence you have that you are going to hit them this quarter**.

You list OKR's to remind everyone (and sometimes yourself) WHY you are doing the things you did.

If you don't use OKRs, use any goal you set quarterly. If you don't set goals, you have more problems than your status report. Your confidence is your guess of how likely you feel you will meet your key results, on a scale from 1 to 10. 1 is never going to happen and 10 is in the bag. Mark your confidence red when it falls below 5, green as it heads toward ten. Color makes it scannable, making your boss and teammates happy.** Listing confidence helps you and your teammates track progress, and correct early if needed.

2. **List last week's prioritized tasks, and if they were achieved.** If they were not, a few words to explain why. The goal here is to learn what keeps the organization from accomplishing what it needs to accomplish. See below for format.
3. **Next, list next week's priorities.** Only list three P1's, and make them meaty accomplishments that encompass multiple steps. "Finalize spec for project xeno" is a good P1. It probably encompasses writing, reviews with multiple groups and sign off. It also gives a heads up to other teams and your boss that you'll be coming by. "Talk to legal" is a bad P1. This priority takes about half hour, has no clear outcome, feels like a subtask and not only that, you didn't even tell us what you were talking about! You can add a couple P2's, but they should also be meaty, worthy of being next week's P2's. You want fewer, bigger items.
4. **List any risks or blockers.** Just as in an Agile stand-up, note anything you could use help on that you can't solve yourself. Do NOT play the blame game. Your manager does not want to play mom, listening to you and a fellow executive say "it's his fault." Also, list anything you know of that could keep you from accomplishing what you set out to do— a business partner playing hard-to-schedule, or a tricky bit of technology that might take longer than planned to sort out. Bosses do not like to be surprised. Don't surprise them.

Change your status email today, and transform your teams.

5. **Notes.** Finally, if you have anything that doesn't fit in these categories, but you absolutely want to include, add a note. "Hired that fantastic guy from Amazon that Jim sent over. Thanks, Jim!" is a decent note, as is "Reminder: team out Friday for offsite to Giant's game." Make them short, timely and useful. Do not use notes for excuses, therapy or novel writing practice.

This format also fixes another key challenge large organizations face: coordination.

To write a status report the old way, I had to have team status in by Thursday night in order to collate, fact check and edit. But with this system, I know what my priorities are, and I use my reports' status only as a way to make sure their priorities are mine. I send out my report Friday, as I receive my report's. We stay committed to each other, honest and focused.

Work should not be a chore list, but a collective push forward toward shared goals. The status email reminds everyone of this fact, and helps us avoid slipping into checkbox thinking. This system is so effective in helping me get things done, I've even adapted it for my personal life.

Coordinating organizational efforts is critical to a company's ability to compete and innovate. Giving up on the status email is a strategic error. It can be a task that wastes key resources, or it can be a way that teams connect and support each other. Change your status email today, and transform your teams. ■

Footnotes

*Later, I learned to say "the problem isn't that we don't have too few people, the problem is that we have too many priorities." But that's another topic for another day.

**Do not use more colors than red, green and black in your reports. Red and green are cultural symbols for go and stop, bad and good. Black is easy to read for body text. If you need to add a legend to your status reports, you're doing it wrong. And while we're on the topic, Georgia is a lovely font for anything you want to do. Just sayin'.

Finally I advise a company called Workboard that automates status! Makes life easier…

Reprinted with permission of the original author. First appeared at eleganthack.com.

food bit *

By DryPot - Own work, CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=14929630

Hardtack

For the ultimate survival food, look no further than hardtack. Also known as pilot bread, sea biscuits, molar breakers and a slew of unsavory names, hardtack is undoubtedly one of the greatest survival foods ever created.

A mainstay of expeditions, long ship journeys and earthquake emergency kits, these plain hard crackers were widely used during the American Civil War because they were cheap, filling and as non-perishable as it gets. Made from flour, salt and water, hardtack is admittedly dry and flavorless, but its blandness means that it goes superbly with everything from peanut butter and cheese to soups and whiskey.

* FOOD BIT is where we, enthusiasts of all edibles, sneak in a fun fact about food.