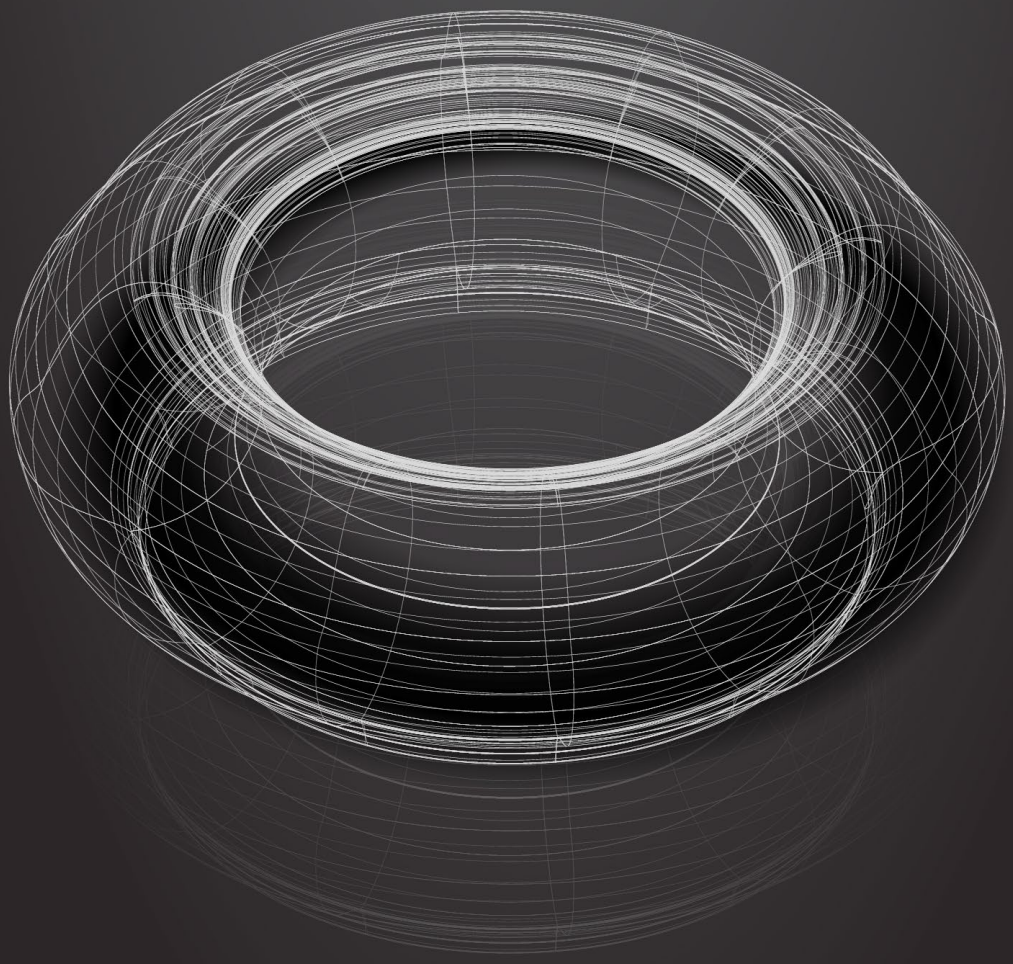


*Andrea Corbellini*

# Elliptic Curve Cryptography

HACKERMONTHLY Issue 65 October 2015

**Curator**
Lim Cheng Soon

**Contributors**
Andrea Corbellini
Matthias Wandel
Nate Berkopec
TJ
Bob Nystrom

**Proofreader**
Emily Griffin

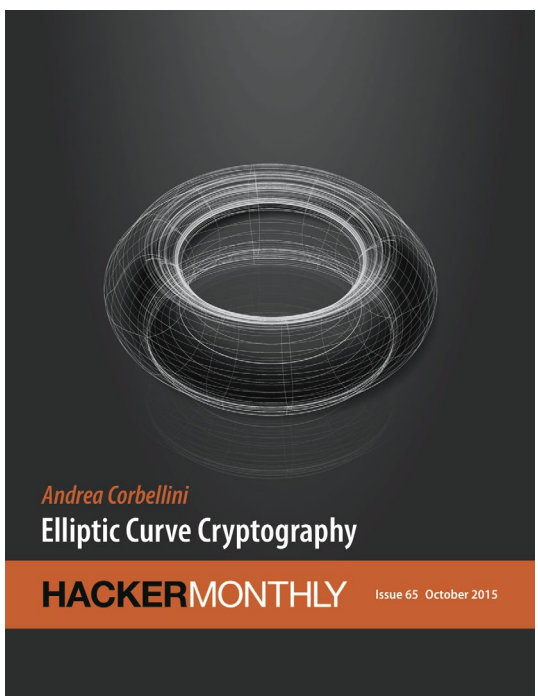
**Printer**
Blurb

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

**Advertising**
ads@hackermonthly.com

**Contact**
contact@hackermonthly.com

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



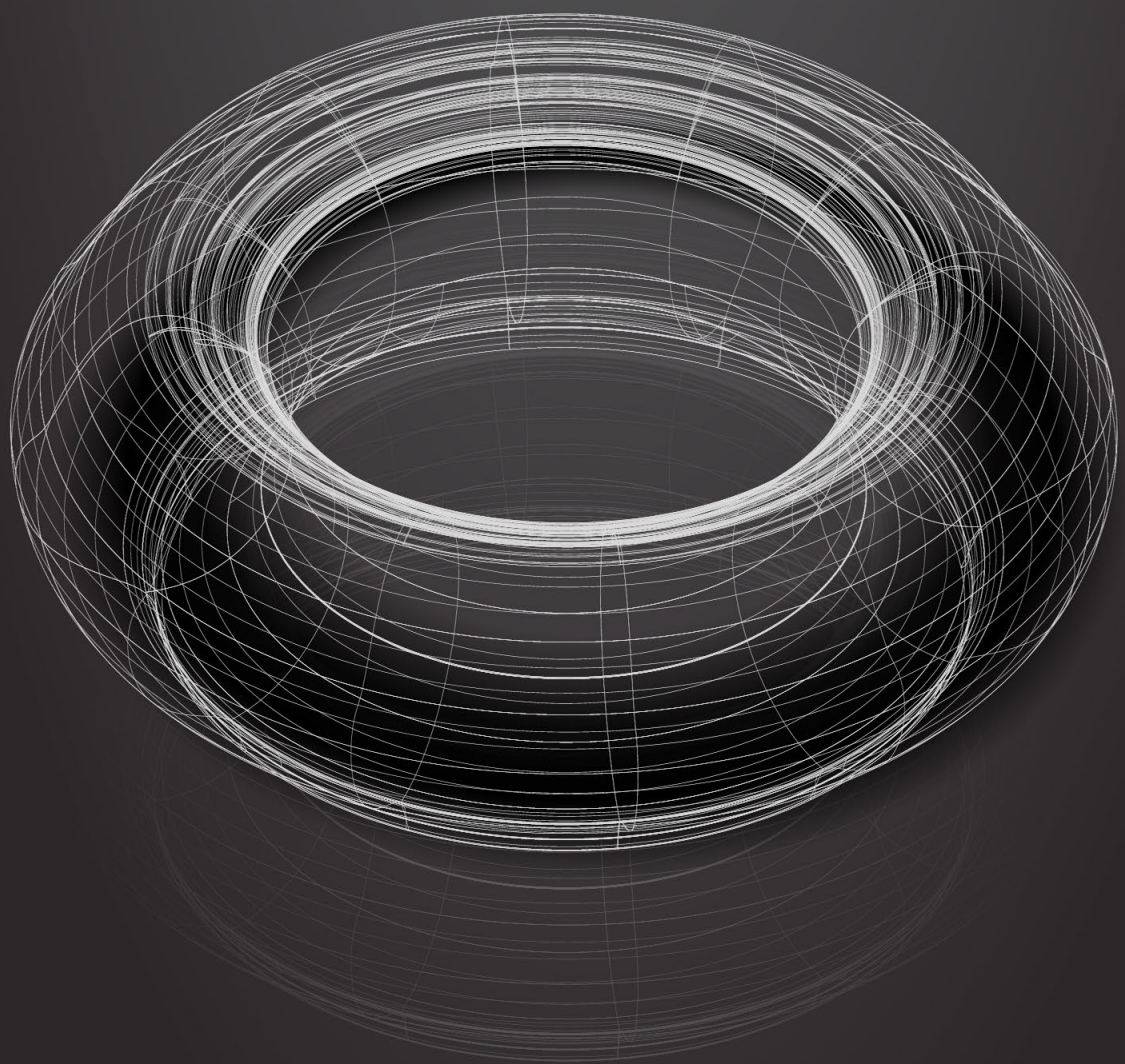
# Contents



For links to Hacker News dicussions, visit *hackermonthly.com/issue-65*

# Elliptic Curve Cryptography: a Gentle Introduction

*By* ANDREA CORBELLINI



Those of you who know what public-key cryptography is may have already heard of ECC, ECDH or ECDSA. The first is an acronym for Elliptic Curve Cryptography, the others are names for algorithms based on it.

Today, we can find elliptic curves cryptosystems in TLS, PGP and SSH, which are just three of the main technologies on which the modern web and IT world are based. Not to mention Bitcoin and other cryptocurrencies.

Before ECC become popular, almost all public-key algorithms were based on RSA, DSA, and DH, alternative cryptosystems based on modular arithmetic. RSA and friends are still very important today, and often are used alongside ECC. However, while the magic behind RSA and friends can be easily explained, is widely understood, and rough implementations can be written quite easily, the foundations of ECC are still a mystery to most.

I'm going to give you a gentle introduction to the world of elliptic curve cryptography. My aim is not to provide a complete and detailed guide to ECC (the web is full of information on the subject), but to provide a simple overview of what ECC is and why it is considered secure, without losing time on long mathematical proofs or boring implementation details. I will also give helpful examples together with visual interactive tools and scripts to play with.

Specifically, here are the topics I'll touch:

1. Elliptic curves over real numbers and the group law (covered in this article)
2. Elliptic curves over finite fields and the discrete logarithm problem
3. Key pair generation and two ECC algorithms: ECDH and ECDSA
4. Algorithms for breaking ECC security, and a comparison with RSA

In order to understand what's written here, you'll need to know some basic set theory, geometry, and modular arithmetic, and have familiarity with symmetric and asymmetric cryptography. Lastly, you need to have a clear idea of what an "easy" problem is, what a "hard" problem is, and their roles in cryptography.

Ready? Let's start!

## Elliptic Curves

First of all: what is an elliptic curve? Wolfram MathWorld gives an excellent and complete definition. But for our aims, an elliptic curve will simply be **the set of points described by the equation**:

$$y^2 = x^3 + ax + b$$

where $4a^3 + 27b^2 \neq 0$ (this is required to exclude singular curves). The equation above is what is called *Weierstrass normal form* for elliptic curves.



Different shapes for different elliptic curves ($b = 1$, $a$ varying from 2 to -3).



Types of singularities: on the left, a curve with a cusp (y2 = x3). On the right, a curve with a self-intersection (y2 = x3 – 3x + 2). None of them is a valid elliptic curve.

Depending on the value of $a$ and $b$, elliptic curves may assume different shapes on the plane. As it can be easily seen and verified, elliptic curves are symmetric about the $x$-axis.

For our aims, **we will also need a point at infinity** (also known as ideal point) to be part of our curve. From now on, we will denote our point at infinity with the symbol 0 (zero).

If we want to explicitly take into account the point at infinity, we can refine our definition of elliptic curve as follows:

$$\{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\} \cup \{0\}$$

## Groups

A group in mathematics is a set for which we have defined a binary operation that we call "addition" and indicate with the symbol +. In order for the set $\mathbb{G}$ to be a group, addition must be defined so that it respects the following four properties:

1. **Closure**: if $a$ and $b$ are members of $\mathbb{G}$, then $a + b$ is a member of $\mathbb{G}$;
2. **Associativity**: $(a + b) + c = a + (b + c)$;
3. There exists an **identity element** 0 such that $a + 0 = 0 + a = a$;
4. Every element has an **inverse**, that is: for every $a$ there exists $b$ such that $a + b = 0$.

If we add a fifth requirement:

5. Commutativity: $a + b = b + a$,

Then the group is called *abelian group*.

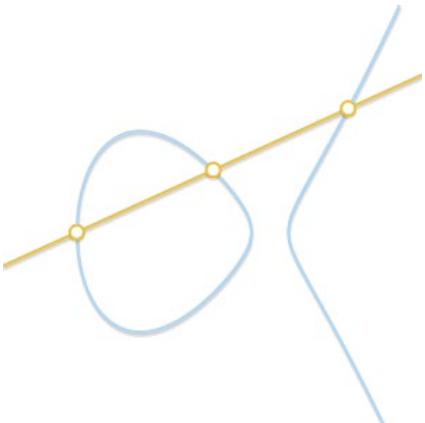
With the usual notion of addition, the set of integer numbers $\mathbb{Z}$ is a group (moreover, it's an abelian group). The set of natural numbers $\mathbb{N}$, however, is not a group, as the fourth property can't be satisfied.

Groups are nice because, if we can demonstrate that those four properties hold, we get some other properties for free. For example: the **identity element is unique**; also **the inverses are unique**, that is: for every $a$ there exists only one $b$ such that $a + b = 0$ (and we can write $b$ as $-a$). Either directly or indirectly, these and other facts about groups will be very important for us later.

## The group law for elliptic curves

We can define a group over elliptic curves. Specifically:

- The elements of the group are the points of an elliptic curve;
- The **identity element** is the point at infinity 0;
- The **inverse** of a point $P$ is the one symmetric about the $x$-axis;
- **Addition** is given by the following rule: **given three aligned, non-zero points $P$, Q and $R$, their sum is $P + Q + R = 0$.**



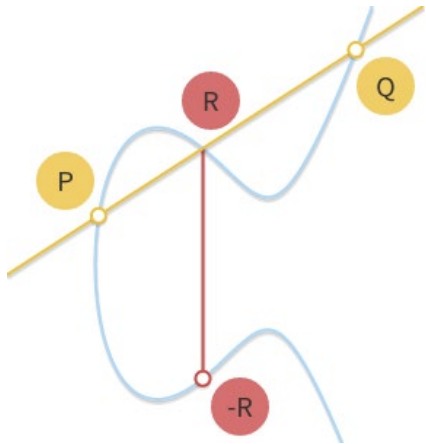
The sum of three aligned point is 0.

Note that with the last rule, we only require three aligned points, and three points are aligned without respect to order. This means that, if $P$,Q and $R$ are aligned, then $P + (Q + R) = Q + (P + R) = R + (P + Q) = \cdots = 0$. This way, we have intuitively proved that **our + operator is both associative and commutative: we are in an abelian group.**

So far, so great. But how do we actually compute the sum of two arbitrary points?

## Geometric addition

Thanks to the fact that we are in an abelian group, we can write $P + Q + R = 0$ as $P + Q = -R$. This equation, in this form, lets us derive a geometric method to compute the sum between two points $P$ and Q: **if we draw a line passing through $P$ and Q, this line will intersect a third point on the curve**, $R$ (this is implied by the fact that $P$, Q and $R$ are aligned). **If we take the inverse of this point, $-R$, we have found the result of $P + Q$.**



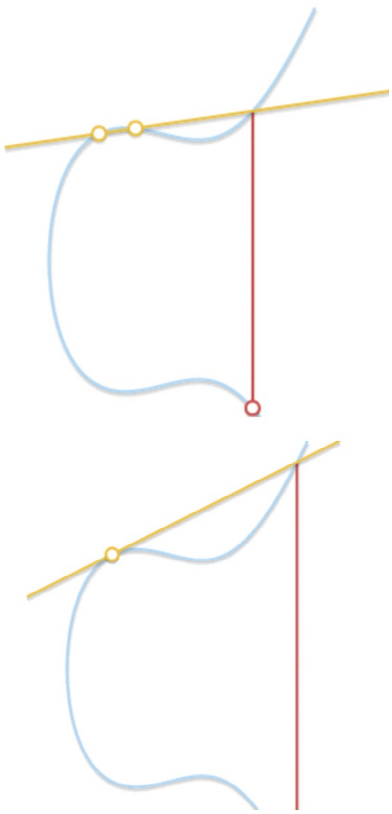

Draw the line through $P$ and Q. The line intersects a third point $R$. The point symmetric to it, $-R$, is the result of $P + Q$.

This geometric method works but needs some refinement. Particularly, we need to answer a few questions:

- **What if $P = 0$ or $Q = 0$?** Certainly, we can't draw any line (0 is not on the xy-plane). But given that we have defined 0 as the identity element, $P + 0 = P$ and $0 + Q = Q$, for any $P$ and for any Q.
- **What if $P = -Q$?** In this case, the line going through the two points is vertical, and does not intersect any third point. But if $P$ is the inverse of Q, then we

have $P + Q = P + (-P) = 0$ from the definition of inverse.

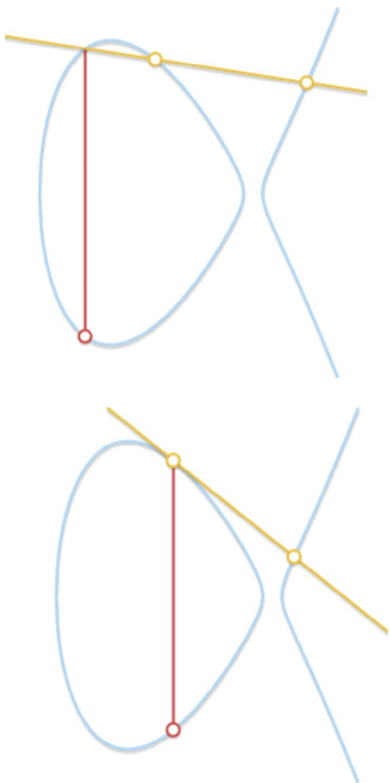
- **What if $P = Q$?** In this case, there are infinitely many lines passing through the point. Here things start getting a bit more complicated. But consider a point $Q' \neq P$. What happens if we make $Q'$ approach $P$, getting closer and closer to it?



As the two points become closer together, the line passing through them becomes tangent to the curve.

As $Q'$ tends towards $P$, the line passing through $P$ and $Q'$ becomes tangent to the curve. In the light of this we can say that $P + P = -R$, where $R$ is the point of intersection between the curve and the line tangent to the curve in $P$.

- **What if $P \neq Q$, but there is no third point $R$?** We are in a case very similar to the previous one. In fact, we are in the case where the line passing through $P$ and $Q$ is tangent to the curve.



If our line intersects just two points, then it means that it's tangent to the curve. It's easy to see how the result of the sum becomes symmetrical to one of the two points.

Let's assume that $P$ is the tangency point. In the previous case, we would have written $P + P = -Q$. That equation now becomes $P + Q = -P$. If, on the other hand, $Q$ were the tangency point, the correct equation would have been $P + Q = -Q$.

The geometric method is now complete and covers all cases. With a pencil and a ruler we are able to perform addition involving every point of any elliptic curve. If you want to try, take a look at the HTML5/JavaScript visual tool I've built for computing sums on elliptic curves! [hn.my/ectool]

## Algebraic addition

If we want a computer to perform point addition, we need to turn the geometric method into an algebraic method. Transforming the rules described above into a set of equations may seem straightforward, but actually it can be really tedious because it requires solving cubic equations. For this reason, here I will report only the results.

First, let's get rid of the most annoying corner cases. We already know that $P + (-P) = 0$, and we also know that $P + 0 = 0 + P = P$. So, in our equations, we will avoid these two cases and we will only consider **two non-zero, non-symmetric points** $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$.

**If $P$ and $Q$ are distinct** ($x_P \neq x_Q$), the line through them has slope:

$$m = \frac{y_P - y_Q}{x_P - x_Q}$$

The intersection of this line with the elliptic curve is a third point $R = (x_R, y_R)$:

$$\begin{aligned} x_R &= m^2 - x_P - x_Q \\ y_R &= y_P + m(x_R - x_P) \end{aligned}$$

Or, equivalently:

$$y_R = y_Q + m(x_R - x_Q)$$

Hence $(x_P, y_P) + (x_Q, y_Q) = (x_R, -y_R)$ (pay attention at the signs and remember that $P + Q = -R$).

If we wanted to check whether this result is right, we would have had to check whether $R$ belongs to the curve and whether $P$, $Q$ and $R$ are aligned. Checking whether the points are aligned is trivial, checking that $R$ belongs to the curve is not,

as we would need to solve a cubic equation, which is not fun at all.

Instead, let's play with an example: according to our visual tool, given $P = (1, 2)$ and $Q = (3, 4)$ over the curve $y^2 = x^3 - 7x + 10$, their sum is $P + Q = -R = (-3, 2)$. Let's see if our equations agree:

$$\begin{aligned} m &= \frac{y_P - y_Q}{x_P - x_Q} = \frac{2-4}{1-3} = 1 \\ x_R &= m^2 - x_P - x_Q = 1^2 - 1 - 3 = -3 \\ y_R &= y_P + m(x_R - x_P) = 2 + 1 \cdot (-3 - 1) = -2 \\ &= y_Q + m(x_R - x_Q) = 4 + 1 \cdot (-3 - 3) = -2 \end{aligned}$$

Yes, this is correct!

Note that these equations work even if **one of $P$ or $Q$ is a tangency point**. Let's try with $P = (-1, 4)$ and $Q = (1, 2)$.

$$\begin{aligned} m &= \frac{y_P - y_Q}{x_P - x_Q} = \frac{4-2}{-1-1} = -1 \\ x_R &= m^2 - x_P - x_Q = (-1)^2 - (-1) - 1 = 1 \\ y_R &= y_P + m(x_R - x_P) = 4 + -1 \cdot (1 - (-1)) = 2 \end{aligned}$$

We get the result $P + Q = (1, -2)$, which is the same result given by the visual tool.

**The case $P = Q$ needs to be treated a bit differently**: the equations for $x_R$ and $y_R$ are the same, but given that $x_P = x_Q$, we must use a different equation for the **slope**:

$$m = \frac{3x_P^2 + a}{2y_P}$$

Note that, as we would expect, this expression for $m$ is the first derivative of:

$$y_P = \pm\sqrt{x_P^3 + ax_P + b}$$

To prove the validity of this result it is enough to check that $R$ belongs to the curve and that the line passing through $P$ and $R$ has only two intersections with the curve. But again, we don't prove this fact, but instead try with an example: $P = Q = (1, 2)$.

$$\begin{aligned} m &= \frac{3x_P^2 + a}{2y_P} = \frac{3 \cdot 1^2 - 7}{2 \cdot 2} = -1 \\ x_R &= m^2 - x_P - x_Q = (-1)^2 - 1 - 1 = -1 \\ y_R &= y_P + m(x_R - x_P) = 2 + (-1) \cdot (-1 - 1) = 4 \end{aligned}$$

Which gives us $P + P = -R = (-1, -4)$. Correct!

Although the procedure to derive them can be really tedious, our equations are pretty compact. This is thanks to Weierstrass normal form: without it, these equations could have been really long and complicated!

## Scalar multiplication

Other than addition, we can define another operation: **scalar multiplication**, that is:

$$nP = \underbrace{P + P + \cdots + P}_{n \text{ times}}$$

Where $n$ is a natural number. I've written a visual tool for scalar multiplication, too, if you want to play with that. [hn.my/ectool2]

Written in that form, it may seem that computing $nP$ requires $n$ additions. If $n$ has $k$ binary digits, then our algorithm would be $O(2^k)$, which is not really good. But there exist faster algorithms.

One of them is the **double and add** algorithm. Its principle of operation can be better explained with an example. Take $n = 151$. Its binary representation is $10010111_2$. This binary representation can be turned into a sum powers of two:

$$151 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

(We have taken each binary digit of $n$ and multiplied it by a power of two.)

In view of this, we can write:

$$151 \cdot P = 2^7P + 2^4P + 2^2P + 2^1P + 2^0P$$

What the double and add algorithm tells us to do is:

- Take $P$.
- *Double* it, so that we get $2P$.
- *Add* $2P$ to $P$ (in order to get the result of $2^1P + 2^0P$).
- *Double* $2P$, so that we get $2^2P$.
- *Add* it to our result (so that we get $2^2P + 2^1P + 2^0P$).
- *Double* $2^2P$ to get $2^3P$.
- Don't perform any addition involving $2^3P$.
- *Double* $2^3P$ to get $2^4P$.
- *Add* it to our result (so that we get $2^4P + 2^2P + 2^1P + 2^0P$).
- …

In the end, we can compute $151 \cdot P$ performing just seven doublings and four additions.

If this is not clear enough, here's a Python snippet that implements the algorithm:

```
def bits(n):
    """
    Generates the binary digits
    of n, starting from the
    least significant bit.

    bits(151) -> 1, 1, 1, 0, 1,
    0, 0, 1
    """
    while n:
        yield n & 1
        n >>= 1


def double_and_add(n, x):
    """
    Returns the result of n *
    x, computed using the
    double and add algorithm.
    """
    result = 0
    addend = x
```

```
    for bit in bits(n):
        if bit == 1:
            result += addend
        addend *= 2

    return result
```

If doubling and adding are both O(1) operations, then this algorithm is O(log $n$) (or O($k$) if we consider the bit length), which is pretty good. Surely much better than the initial O($n$) algorithm!

## Logarithm

Given $n$ and $P$, we now have at least one polynomial time algorithm for computing $Q = nP$. But what about the other way round? **What if we know $Q$ and $P$ and need to find out $n$?** This problem is known as the **logarithm problem**. We call it "logarithm" instead of "division" for conformity with other cryptosystems (where instead of multiplication we have exponentiation).

I don't know of any "easy" algorithm for the logarithm problem. However, playing with multiplication, it's easy to see some patterns. For example, take the curve $y^2 = x^3 - 3x + 1$ and the point $P = (0, 1)$. We can immediately verify that, if $n$ is odd, $nP$ is on the curve on the left semiplane; if $n$ is even, $nP$ is on the curve on the right semiplane. If we experimented more, we could probably find more patterns that eventually could lead us to write an algorithm for computing the logarithm on that curve efficiently.

But there's a variant of the logarithm problem: the *discrete* logarithm problem. As we will see in the next post, if we reduce the domain of our elliptic curves, **scalar multiplication remains "easy," while the discrete logarithm becomes a "hard" problem**. This duality is the key brick of elliptic curve cryptography. ■

*Read the next article of the series here: [hn.my/finite]*

---

Andrea is a software developer with a passion for the web and the cloud. Over the years, he has had the opportunity to work with a large set of technologies, in many different fields. Nowadays, he enjoys designing cloud and cloud-based services with focus on security, quality, scalability and continuous delivery.

Reprinted with permission of the original author.
First appeared in *hn.my/ecc* (andrea.corbellini.name)

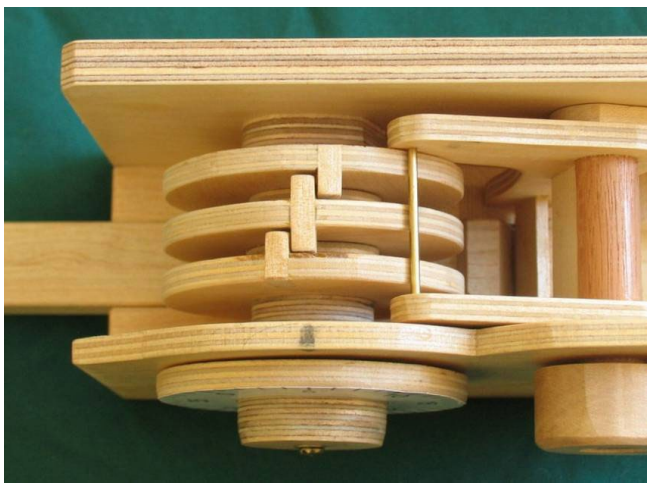
# Wooden Combination Lock

*By* MATTHIAS WANDEL

In terms of neat mechanical things to build out of wood, I figured a single dial sequential combination lock would be a neat thing to make. It would be relatively simple, involve movement, and also show people how a combination lock actually works.


I spent some time thinking about it, and I came up with the simplest design that would also be visually appealing. Unlike a real lock, my priority was to show how it actually works.

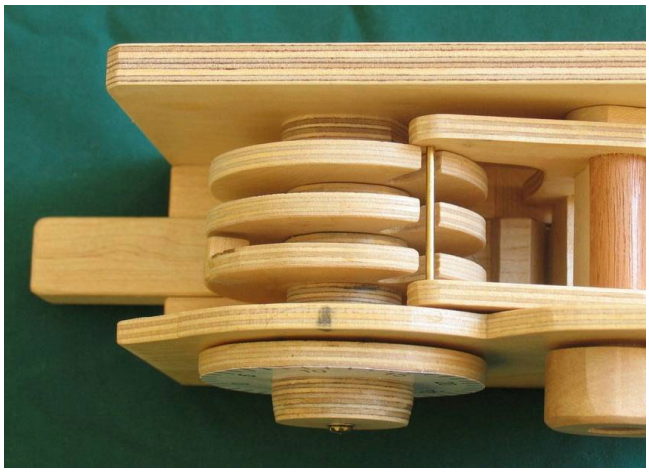
Like most real combination locks such as a Dudley or master Combination locks, the core of this lock consists of three rotors. Each rotor has a notch in it, and when the three notches line up, some sort of bar can drop into them, allowing the lock to be opened.



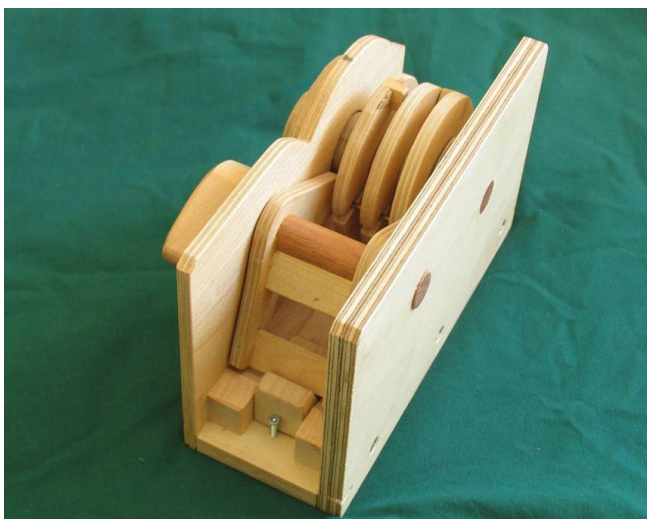
The front-most rotor is directly coupled to the dial on the front of the lock. All the rotors have tabs sticking out the front and back, so that once the front rotor is turned by one turn, its tab hits the tab on the middle rotor. The middle rotor's tab in turn sticks out the back, and with another clockwise turn of the dial catches the tab in the last rotor. So with two turns in the same direction, the back rotor is also engaged.

To open the lock, one turns the dial to the right until the notch on the rear-most rotor is aligned with the bar. The first number of the combination corresponds to the position that the dial needs to be turned to accomplish this.

After the back rotor is lined up, rotation is reversed. By turning counterclockwise one turn, one will catch the middle rotor with the tab pushing it in the opposite direction. As long as one doesn't turn too far, only the front and middle rotors now turn with the dial. The dial is turned far enough to align the middle rotor.



After that, the rotation is again reversed, now turning clockwise again until the notch in the front most rotor is lined up with the bar. The picture at left shows all three notches lined up. The lock is ready to open.



This view shows the bar in the slots. The lock is constructed in such a way that the bar getting into the slots actually turns the rotors a little bit.

This view with the back panel removed better illustrates how the lock opens. The L-shaped part on the left turns clock wise to open the lock. This pulls the bolt (on the bottom) back to the left at the same time.

The notches are not quite lined up in this shot. You can barely see the end of the bar that needs to drop into the notches protruding through the L-shaped bracket.

The lock isn't actually functional without the back panel. The shaft that the rotors sit on is part of the back panel, so this shot just has the rotors carefully stacked in place where they would go without their shaft, just to illustrate.

And here's the lock disassembled, showing all of its parts. Really, there is not very much to it. Note that the shaft that the rotors turn on is glued into the back piece of the plywood. This shaft itself doesn't turn. The rotors are semi-loose on this shaft. Ideally, they would turn with a bit of friction so they wouldn't overshoot with momentum when dialing a combination quickly.

The most complicated part is the part with the bar that drops into the notches, and that part is really only as complicated as it is to make the internal workings easier to see... On a real lock, the corresponding part would be much simpler and more robust. ■

---

Matthias writes about woodworking and makes woodworking. The amount of woodworking I do is more related to a hobby, because the bulk of the time is taken up by making videos, writing, and running the website.

Reprinted with permission of the original author.
First appeared in *hn.my/woodlock* (woodgears.ca)

# The Complete Guide to Rails Caching

*Speed Up Your Rails App by 66%*

*By* NATE BERKOPEC

Caching in a Rails app is a little bit like that one friend you sometimes have around for dinner, but should really have around more often. Nearly every Rails app that's serious about performance could use more caching, but most Rails apps eschew it entirely! And yet, intelligent use of caching is usually the only path to achieving fast server response times in Rails — easily speeding up ~250ms response times to 50-100ms.

A quick note on definitions: this post will only cover "application"-layer caching. I'm leaving HTTP caching (which is a whole other beast, and not even necessarily implemented in your application) for another day.

## Why don't we cache as much as we should?

Developers, by our nature, are very different from end-users. We understand a lot about what happens behind the scenes in software and web applications. We know that when a typical webpage loads, a lot of code is run, database queries executed, and sometimes services pinged over HTTP. That takes time. We're used to the idea that when you interact with a computer, it takes a little while for the computer to come back with an answer.

End-users are completely different. Your web application is a magical box. End-users have no idea what happens inside of that box. Especially these days, end-users expect near-instantaneous response from our magical boxes. Most end-users wanted whatever they're trying to get out of your web-app yesterday.

This rings of a truism. Yet, we never set hard performance requirements in our user stories and product specifications. Even though server response time is easy to measure and target, and we know users want fast webpages, we fail to ever say for a particular site or feature: "This page should return a response within 100ms." As a result, performance often gets thrown to the wayside in favor of the next user story, the next great big feature. Performance debt, like technical debt, mounts quickly. Performance never really becomes a priority until the app is basically in flames every time someone makes a new request.

In addition, caching isn't always easy. Cache expiration especially can be a confusing topic. Bugs in caching behavior tend to happen at the integration layer, usually the least-tested layer of your application. This makes caching bugs insidious and difficult to find and reproduce.

To make matters worse, caching best practices seem to be frequently changing in the Rails world. Key-based what? Russian mall caching? Or was it doll?

## Benefits of Caching

So why cache? The answer is simple. Speed. With Ruby, we don't get speed for free because our language isn't very fast to begin with.

We have to get speed from executing less Ruby on each request. The easiest way to do that is with caching. Do the work once, cache the result, serve the cached result in the future.

But how fast do we need to be, really?

Guidelines for human-computer interaction have been known since computers were first developed in the 1960s. The response-time threshold for a user to feel as if they are freely navigating your site, without waiting for the site to load, is 1 second or less. That's not a 1-second response time, but 1 second "to glass" — 1 second from the instant the user clicked or interacted with the site until that interaction is complete (the DOM finishes painting).

One second "to-glass" is not a very long time. First, figure about 50 milliseconds for network latency (this is on desktop, latency on mobile is a whole other discussion). Then, budget another 150ms for loading your JS and CSS resources, building the render tree and painting. Finally, figure at least 250 ms for the execution of all the JavaScript you've downloaded, and potentially much more than that if your JavaScript has a lot of functions tied to the DOM being ready. So before we're even ready to consider how long the server has to respond, we're already about ~500ms in the hole. In order to consistently achieve a 1 second to glass webpage, server responses should be kept below 300ms. For a 100-ms-to-glass webpage, server responses must be kept at around 25-30ms.

Three hundred ms per request is not impossible to achieve without caching on a Rails app, especially if you've been diligent with your SQL queries and use of ActiveRecord. But it's a heck of a lot of easier if you do use caching. Most Rails apps I've seen have at least a half dozen pages in the app that consistently take north of 300ms to respond, and could benefit from some caching. In addition, using heavy frameworks in addition to Rails, like Spree, the popular e-commerce framework, can slow down responses significantly due to all the extra Ruby execution they add to each request. Even popular heavyweight gems, like Devise or ActiveAdmin, add thousands of lines of Ruby to each request cycle.

Of course, there will always be areas in your app where caching can't help — your POST endpoints, for example. If whatever your app does in response to a POST or PUT is extremely complicated, caching probably won't help you. But if that's the case, consider moving the work into a background worker instead.

## Getting started

First, Rails' official guide on caching [hn.my/rcache] is excellent regarding the technical details of Rails' various caching APIs. If you haven't yet, give that page a full read-through.

Later on in the article, I'm going to discuss the different caching backends available to you as a Rails developer. Each has their advantages and disadvantages; some are slow but offer sharing between hosts and servers, some are fast but can't share the cache at all, not even with other processes. Everyone's needs are different. In short, the default cache store, `ActiveSupport::Cache::FileStore` is OK, but if you you're going to follow the techniques used in this guide (especially key-based cache expiration), you need to switch to a different cache store eventually.

As a tip to newcomers to caching, my advice is to ignore action caching and page caching. The situations where these two techniques can be used is so narrow that these features were removed from Rails as of 4.0. I recommend instead getting very comfortable with fragment caching, which I'll cover in detail now.

## Profiling Performance

### Reading the Logs

Alright, you've got your cache store set up and you're ready to go. But what to cache?

This is where profiling comes in. Rather than trying to guess "in the dark" what areas of your application are performance hotspots, we're going to fire up a profiling tool to tell us exactly what parts of the page are slow.

My preferred tool for this task is the incredible `rack-mini-profiler`. [hn.my/profiler] `rack-mini-profiler` provides an excellent line-by-line breakdown of where exactly all the time goes during a particular server response.

However, we don't even have to use `rack-mini-profiler` or even any other profiling tools if we're too lazy and don't want to. Rails provides a total time for page generation out of the box in the logs. It'll look something like this:

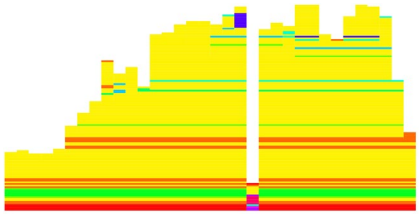
```
Completed 200 OK in 110ms
(Views: 65.6ms | ActiveRecord:
19.7ms)
```

The total time (110ms in this case) is the important one. The amount of time spent in Views is a total of the time spent in your template files (index.html.erb for example). But this can be a little misleading, thanks to how ActiveRecord::Relations lazily loads your data. If you're defining an instance variable with an ActiveRecord::Relation, such as `@users = User.all`, in the controller, but don't do anything with that variable until you start using its results in the view (e.g., `@users.each do ...`), then that query (and reification into ActiveRecord objects), will be counted in the Views number. ActiveRecord::Relations are lazily loaded, meaning the database query isn't executed until the results are actually accessed (usually in your view).

The ActiveRecord number here is also misleading. As far as I can tell from reading the Rails source, this is not the amount of time spent executing Ruby in ActiveRecord (building the query, executing the query, and turning the query results into ActiveRecord objects), but only the time spent querying the database (the actual time spent in DB). Sometimes, especially with very complicated queries that use a lot of eager loading, turning the query result into ActiveRecord objects takes a lot of time, and that may not be reflected in the ActiveRecord number here.

And where'd the rest of the time go? Rack middleware and controller code mostly. But to get a millisecond-by-millisecond breakdown of exactly where your time goes during a request, you'll need `rack-mini-profiler` and the `flamegraph` extension.

Using that tool, you'll be able to see exactly where every millisecond of your time goes during a request on a line-by-line basis. I'm working on a guide for using `rack-mini-profiler`.



What the flamegraph looks like in rack-mini-profiler

## Production Mode

Whenever I profile Rails apps for performance, I always do it in production mode. Not on production, of course, but with `RAILS_ENV=production`. Running in production mode ensures that my local environment is close to what the end-user will experience, and also disables code reloading and asset compilation, two things which will massively slow down any Rails request in development mode. Even better if you can use Docker to perfectly mimic the configuration of your production environment. For instance, if you're on Heroku, Heroku recently released some Docker images to help you, but usually virtualization is a mostly unnecessary step in achieving production-like behavior. Mostly, we just need to make sure we're running the Rails server in production mode.

As a quick refresher, here's what you usually have to do to get a Rails app running in production mode on your local machine:

```
export RAILS_ENV=production
rake db:reset
rake assets:precompile
SECRET_KEY_BASE=test rails s
```

In addition, where security and privacy concerns permit, I always test with a copy of production data. All too often, database queries in development (like User.all) return just 100 or so sample rows, but in production, trigger massive 100,000 row results that can bring a site crashing to its knees. Either use production data or make your seed data as realistic as possible. This is especially important when you're making extensive use of `includes` and Rails' eager loading facilities.

## Setting a Goal

Finally, I suggest setting a maximum acceptable average response time, or MAART, for your site. The great thing about performance is that it's usually quite measurable — and what gets measured, gets managed! You may need two MAART numbers: one that is achievable in development with your developer hardware, and one that you use in production with production hardware.

Unless you have an extremely 1-to-1 production/development setup, using virtualization to control cpu and memory access, you simply will not be able to duplicate performance results across those two environments (though you can come close). That's OK. Don't get tripped up by the details. You just need to be sure that your page performance is in the right ballpark.

As an example, let's say we want to build a 100ms-to-glass web app. That requires server response times of 25-50ms. So I'd set my MAART in development to be 25ms, and in production, I'd slacken that to about 50ms. My development machine is a little faster than a Heroku dyne (my typical

deployment environment), so I give it a little extra time on production.

I'm not aware of any tools yet to do automated testing against your maximum acceptable average response time. We have to do that (for now) manually using benchmarking tools.

### Apache Bench

So, how do we decide what our site's actual average response time is in development? I've only described to you how to read response times from the logs, so is the best way to hit "refresh" in your browser a few times and take your best guess at the average result? Nope.

This is where benchmarking tools like `wrk` and `Apache Bench` come in. `Apache Bench`, or `ab`, is my favorite, so I'll quickly describe how to use it. You can install it on Homebrew with `brew install ab`.

Start your server in production mode, as described earlier. Then fire up Apache Bench with the following settings:

```
ab -t 10 -c 10 http://
localhost:3000/
```

Obviously, you'll need to change that URL out as appropriate. The -t option controls how long we're going to benchmark for (in seconds), and -c controls the number of requests that we'll try at the same time. Set the -c option based on your production load. If you have more than an average of 1 request per second (per server), it would be good to increase the -c option approximately according to the formula of (Production requests per minute / production servers or dynes) * 2. I usually test with at least -c 2 to flush out any weird threading/concurrency errors I might have accidentally committed.

Here's some example output from Apache Bench, abridged for clarity:

```
...
Requests per second:    161.04
[#/sec] (mean)
Time per request:       12.419
[ms] (mean)
Time per request:       6.210
[ms] (mean, across all concur-
rent requests)
...

Percentage of the requests
served within a certain time
(ms)
  50%     12
  66%     13
  75%     13
  80%     13
  90%     14
  95%     15
  98%     17
  99%     18
 100%     21 (longest request)
```

The "time per request" would be the number we compare against our MAART. If you also have a 95th percentile goal (95 percent of requests must be faster than X), you can get the comparable time from the chart at the end, next to "95%". Neat, huh?

For a full listing of things you can do with Apache Bench, check out the man page. Notable other options include SSL support, KeepAlive, and POST/PUT support.

Of course, the great thing about this tool is that you can also use it against your production server. If you want to benchmark heavy loads though, it's probably best to run it against your staging environment instead, so that your customers aren't affected.

From here, the workflow is simple: I don't cache anything unless I'm not meeting my MAART. If my page is slower than my set MAART, I dig in with `rack-mini-profiler` to see exactly which parts of the page are slow.

In particular, I look for areas where a lot of SQL is being executed unnecessarily on every request, or where a lot of code is executed repeatedly.

## Caching techniques

### Key-based cache expiration

Writing and reading from the cache is pretty easy. Again, if you don't know the basics of it, check out the Rails Guide on this topic. The complicated part of caching is knowing when to expire caches.

In the old days, Rails developers used to do a lot of manual cache expiration, with Observers and Sweepers. Nowadays, we try to avoid these entirely, and instead use something called key-based expiration.

Recall that a cache is simply a collection of keys and values, just like a Hash. In fact, we use hashes as caches all the time in Ruby. Key-based expiration is a cache expiration strategy that expires entries in the cache by making the cache key contain information about the value being cached, such that when the object changes (in a way that we care about), the cache key for the object also changes. We then leave it to the cache store to expire the (now unused) previous cache key. We never expire entries in the cache manually.

In the case of an ActiveRecord object, we know that every time we change an attribute and save the object to the database, that object's `updated_at` attribute

changes. So we can use `updated_at` in our cache keys when caching ActiveRecord objects. Each time the ActiveRecord object changes, it's updated_at changes, busting our cache. Thankfully, Rails knows this and makes it very easy for us.

For example, let's say I have a Todo item. I can cache it like this:

```
<% todo = Todo.first %>
<% cache(todo) %>
  ... a whole lot of work here
...
<% end %>
```

When you give an ActiveRecord object to `cache`, Rails realizes this and generates a cache key that looks a lot like this:

```
views/todos/123-
20120806214154/7a1156131a6928cb
0026877f8b749ac9
```

The `views` bit is self-explanatory. The `todos` part is based on the Class of the ActiveRecord object. The next bit is a combination of the `id` of the object (123 in this case) and the `updated_at` value (some time in 2012). The final bit is what's called the template tree digest. This is just an md5 hash of the template that this cache key was called in. When the template changes (e.g., you change a line in your template and then push that change to production), your cache busts and regenerates a new cache value. This is super convenient, otherwise we'd have to expire all of our caches by hand when we changed anything in our templates!

Note here that changing anything in the cache key expires the cache. So if any of the following items change for a given Todo item, the cache will expire and new content will be generated:

- The class of the object (unlikely)
- The object's id (also unlikely, since that's the object's primary key)
- The object's `updated_at` attribute (very likely, because that changes every time the object is saved)
- Our template changes (possible between deploys)

Note that this technique doesn't actually expire any cache keys, it just leaves them unused. Instead of manually expiring entries from the cache, we let the cache itself push out unused values when it begins to run out of space. Or the cache might use a time-based expiration strategy that expires our old entries after a period of time.

You can give an Array to `cache` and your cache key will be based on a concatenated version of everything in the Array. This is useful for different caches that use the same ActiveRecord objects. Maybe there's a todo item view that depends on the current_user:

```
<% todo = Todo.first %>
<% cache([current_user, todo])
%>
 ...a whole lot of work here...
<% end %>
```

Now if the current_user gets updated or if our todo changes, this cache key will expire and be replaced.

## Russian Doll Caching

Don't be afraid of the fancy name; the DHH-named caching technique isn't very complicated at all.

We all know what Russian dolls look like: one doll contained inside another. Russian doll caching is just like that. We're going to stack cache fragments inside each other. Let's say we have a list of Todo elements:

```
<% cache('todo_list') %>
  <ul>
    <% @todos.each do |todo| %>
      <% cache(todo) do %>
        <li class="todo"><%=
todo.description %></li>
      <% end %>
    <% end %>
  </ul>
<% end %>
```

But there's a problem with my above example code. Let's say I change an existing todo's description from "walk the dog" to "feed the cat." When I reload the page, my todo list will still show "walk the dog" because, although the inner cache has changed, the outer cache (the one that caches the entire todo list) has not! That's not good. We want to re-use the inner fragment caches, but we also want to bust the outer cache at the same time.

Russian doll caching is simply using key-based cache expiration to solve this problem. When the "inner" cache expires, we also want the outer cache to expire. If the outer cache expires, though, we don't want to expire the inner caches. Let's see what that would like in our todo_list example above:

```
<% cache(["todo_list", @
todos.map(&:id), @todos.
maximum(:updated_at)]) %>
  <ul>
    <% @todos.each do |todo| %>
      <% cache(todo) do %>
        <li class="todo"><%=
todo.description %></li>
      <% end %>
    <% end %>
  </ul>
<% end %>
```

Now, if any of the @todos change (which will change @todos.maximum(:updated_at)) or an Todo is deleted or added to @todos (changing @todos.map(&:id)), our outer cache will be busted. However, any Todo items which have not changed will still have the same cache keys in the inner cache, so those cached values will be re-used. Neat, right? That's all there is to it!

In addition, you may have seen the use of the `touch` option on ActiveRecord associations. Calling the `touch` method on an ActiveRecord object updates' the record's `updated_at` value in the database. Using this looks like:

```
class Corporation < ActiveRecord::Base
  has_many :cars
end

class Car < ActiveRecord::Base
  belongs_to :corporation, touch: true
end

class Brake < ActiveRecord::Base
  belongs_to :car, touch: true
end

@brake = Brake.first

# calls the touch method on @brake, @brake.car,
# and @brake.car.corporation.
# @brake.updated_at, @brake.car.updated_at and
# @brake.car.corporation.updated_at
# will all be equal.
@brake.touch

# changes updated_at on @brake and saves as
# usual. @brake.car and @brake.car.corporation
# get "touch"ed just like above.
@brake.save

@brake.car.touch
# @brake is not touched. @brake.car.corporation
# is touched.
```

We can use the above behavior to elegantly expire our Russian Doll caches:

```
<% cache @brake.car.corporation %>
  Corporation: <%= @brake.car.corporation.name
%>
  <% cache @brake.car %>
    Car: <%= @brake.car.name %>
    <% cache @brake %>
      Brake system: <%= @brake.name %>
    <% end %>
  <% end %>
<% end %>
```

With this cache structure (and the `touch` relationships configured as above), if we call `@brake.car.save`, our two outer caches will expire (because their `updated_at` values changed) but the inner cache (for `@brake`) will be untouched and reused.

### Which cache backend should I use?

There are a few options available to Rails developers when choosing a cache backend:

- **ActiveSupport::FileStore** This is the default. With this cache store, all values in the cache are stored on the filesystem.
- **ActiveSupport::MemoryStore** This cache store puts all of the cache values in, essentially, a big thread-safe Hash, effectively storing them in RAM.
- **Memcache and dalli** `dalli` is the most popular client for Memcache cache stores. Memcache was developed for LiveJournal in 2003 and is explicitly designed for web applications.
- **Redis and redis-store** `redis-store` is the most popular client for using Redis as a cache.
- **LRURedux** is a memory-based cache store, like ActiveSupport::MemoryStore, but it was explicitly engineered for performance by Sam Saffron, co-founder of Discourse.

Let's dive into each one one-by-one, comparing some of the advantages and disadvantages of each. At the end, I've prepared some performance benchmarks to give you an idea of some of the performance trade-eoffs associated with each cache store.

### ActiveSupport::FileStore

FileStore is the default cache implementation for all Rails applications for as far back as I can tell. If you have not explicitly set `config.cache_store` in production.rb (or whatever environment), you are using FileStore.

FileStore simply stores all of your cache in a series of files and folders (in `tmp/cache` by default).

#### Advantages

**FileStore works across processes.** For example, if I have a single Heroku dyne running a Rails app with Unicorn and I have 3 Unicorn workers, each of those 3 Unicorn workers can share the same cache. So if worker 1 calculates and stores my todolist cache from an earlier example, worker 2 can use that cached value. However, this does not work across hosts (since, of course, most hosts don't have access to the same filesystem). So, again, on Heroku, while all of the processes on each dyne can share the cache, they cannot share across dynos.

**Disk space is cheaper than RAM.** Hosted Memcache servers aren't cheap. For example, a 30MB Memcache server will run you a few bucks a month. But a 5GB cache? That'll be $290/month, please. Ouch. But disk space is a heckuva lot cheaper than RAM, so if you access to a lot of disk space and have a huge cache, FileStore might work well for that.

#### Disadvantages

**Filesystems are slow(ish).** Accessing the disk will always be slower than accessing RAM. However, it might be faster than accessing a cache over the network (which we'll get to in a minute).

**Caches can't be shared across hosts.** Unfortunately, you can't share the cache with any Rails server that doesn't also share your filesystem (across Heroku dynes, for example). This makes FileStore inappropriate for large deployments.

**Not an LRU cache.** This is FileStore's biggest flaw. FileStore expires entries from the cache based on the time they were written to the cache, not the last time they were recently used/accessed. This cripples FileStore when dealing with key-based cache expiration. Recall from our examples above that key-based expiration does not actually expire any cache keys manually. When using this technique with FileStore, the cache will simply grow to maximum size (1GB!) and then start expiring cache entries based on the time they were created. If, for example, your todo list was cached first, but is being accessed 10 times per second, FileStore will still expire that item first! Least-Recently-Used cache algorithms (LRU) work much better for key-based cache expiration because they'll expire the entries that haven't been used in a while first.

**Crashes Heroku dynos.** Another nail in FileStore's coffin is its complete inadequacy for the ephemeral filesystem of Heroku. Accessing the filesystem is extremely slow on Heroku for this reason and actually adds to your dynes' "swap memory." I've seen Rails apps slow to a total crawl due to huge FileStore caches on Heroku that take ages to access. In addition, Heroku restarts all dynes every 24 hours. When that happens, the filesystem is reset, wiping your cache!

#### When should I use ActiveSupport::FileStore?

Reach for FileStore if you have low request load (1 or 2 servers) and still need a very large cache (>100MB). Also, don't use it on Heroku.

### ActiveSupport::MemoryStore

MemoryStore is the other main implementation provided for us by Rails. Instead of storing cached values on the filesystem, MemoryStore stores them directly in RAM in the form of a big Hash.

ActiveSupport::MemoryStore, like all of the other cache stores on this list, is thread-safe.

#### Advantages

- **It's fast.** One of the best-performing caches on my benchmarks (below).
- **It's easy to set up.** Simply change `config.cache_store` to `:memory_store`. Tada!

#### Disadvantages

- **Caches can't be shared across processes or hosts.** Unfortunately, the cache cannot be shared across hosts (obviously), but it also can't even be shared across processes (for example, Unicorn workers or Puma clustered workers).
- **Caches add to your total RAM usage.** Obviously, storing data in memory adds to your RAM usage. This is tough on shared environments like Heroku where memory is highly restrained.

#### When should I use ActiveSupport::MemoryStore?

If you have one or two servers, with a few workers each, and you're storing very small amounts of cached data (<20MB), MemoryStore may be right for you.

## Memcache and dalli

Memcache is probably the most frequently used and recommended external cache store for Rails apps. Memcache was developed for LiveJournal in 2003, and is used in production by sites like Wordpress.org, Wikipedia, and YouTube.

While Memcache benefits from having some absolutely enormous production deployments, it is under a somewhat slower pace of development than other cache stores (because it's so old and well-used, if it ain't broke, don't fix it).

### Advantages

**Distributed, so all processes and hosts can share.** Unlike FileStore and MemoryStore, all processes and dynos/hosts share the exact same instance of the cache. We can maximize the benefit of caching because each cache key is only written once across the entire system.

### Disadvantages

- **Distributed caches are susceptible to network issues and latency**. Of course, it's much, much slower to access a value across the network than it is to access that value in RAM or on the filesystem. Check my benchmarks below for how much of an impact this can have. In some cases, it's extremely substantial.
- **Expensive**. Running FileStore or MemoryStore on your own server is free. Usually, you're either going to have to pay to set up your own Memcache instance on AWS or via a service like Memcachier.
- **Cache values are limited to 1MB**. In addition, cache keys are limited to 250 bytes.

### When should I use Memcache?

If you're running more than 1-2 hosts, you should be using a distributed cache store. However, I think Redis is a slightly better option, for the reasons I'll outline below.

## Redis and redis-store

Redis, like Memcache, is an in-memory, key-value data store. Redis was started in 2009 by Salvatore Sanfilippo, who remains the project lead and sole maintainer today.

In addition to redis-store, there's a new Redis cache gem on the block: readthis. It's under active development and looks promising.

### Advantages

- **Distributed, so all processes and hosts can share**. Like Memcache, all processes and dynos/hosts share the exact same instance of the cache. We can maximize the benefit of caching because each cache key is only written once across the entire system.
- **Allows different eviction policies beyond LRU**. Redis allows you to select your own eviction policies, which gives you much more control over what to do when the cache store is full. For a full explanation of how to choose between these policies, check out the excellent Redis documentation.
- **Can persist to disk, allowing hot restarts**. Redis can write to disk, unlike Memcache. This allows Redis to write the DB to disk, restart, and then come back up after reloading the persisted DB. No more empty caches after restarting your cache store!

### Disadvantages

- **Distributed caches are susceptible to network issues and latency**. Of course, it's much, much slower to access a value across the network than it is to access that value in RAM or on the filesystem. Check my benchmarks below for how much of an impact this can have. In some cases, it's extremely substantial.
- **Expensive**. Running FileStore or MemoryStore on your own server is free. Usually, you're either going to have to pay to set up your own Redis instance on AWS or via a service like Redis.
- **While Redis supports several data types, redis-store only supports**. Strings This is a failure of the redis-store gem rather than Redis itself. Redis supports several data types, like Lists, Sets, and Hashes. Memcache, by comparison, only can store Strings. It would be very interesting to be able to use the additional data types provided by Redis (which could cut down on a lot of marshaling/serialization).

### When should I use Redis?

If you're running more than 2 servers or processes, I recommend using Redis as your cache store.

## Cache Benchmarks

Who doesn't love a good benchmark? All of the benchmark code is available here on GitHub. [hn.my/cachebench]

### Fetch

The most often-used method of all Rails cache stores is `fetch`. If this value exists in the cache, read the value. Otherwise, we write the value by executing the given block. Benchmarking this method tests both read and write performance. `i/s` stands for "iterations/second."

```
LruRedux::ThreadSafeCache:   337353.5 i/s
ActiveSupport::Cache::MemoryStore:    52808.1
i/s - 6.39x slower
ActiveSupport::Cache::FileStore:    12341.5 i/s
- 27.33x slower
ActiveSupport::Cache::DalliStore:     6629.1 i/s
- 50.89x slower
ActiveSupport::Cache::RedisStore:     6304.6 i/s
- 53.51x slower
ActiveSupport::Cache::DalliStore at pub-mem-
cache-13640.us-east-1-1.2.ec2.garantiadata.
com:13640:       26.9 i/s - 12545.27x slower
ActiveSupport::Cache::RedisStore at pub-
redis-11469.us-east-1-4.2.ec2.garantiadata.com:
25.8 i/s - 13062.87x slower
```

Wow! So here's what we can learn from those results:

- LRURedux, MemoryStore, and FileStore are so fast as to be basically instantaneous.
- Memcache and Redis are still very fast when the cache is on the same host.
- When using a host far away across the network, Memcache and Redis suffer significantly, taking about ~50ms per cache read (under extremely heavy load). This means two things: when choosing a Memcache or Redis host, choose the one closest to where your servers are and benchmark its performance. Second, don't cache anything that takes less than ~10-20ms to generate by itself.

### Full-stack in a Rails app

For this test, we're going to try caching some content on a webpage in a Rails app. This should give us an idea of how much time reading/writing a cache fragment takes when we have to go through the entire request cycle as well.

Essentially, all the app does is set `@cache_key` to a random number between 1 and 16, and then render the following view:

```
<% cache(@cache_key) do %>
  <p><%= SecureRandom.base64(100_000) %></p>
<% end %>
```

### Average response time in ms - less is better

The below results were obtained with Apache Bench. The result is the average of 10,000 requests made to a local Rails server in production mode.

- Redis/redis-store (remote) 47.763
- Memcache/Dalli (remote) 43.594
- With caching disabled 10.664
- Memcache/Dalli (localhost) 5.980
- Redis/redis-store (localhost) 5.004
- ActiveSupport::FileStore 4.952
- ActiveSupport::MemoryStore 4.648

Some interesting results here, for sure! Note that the difference between the fastest cache store (MemoryStore) and the uncached version is about 6 milliseconds. We can infer, then, that the amount of work being done by `SecureRandom.base64(100_000)` takes about 6 milliseconds. Accessing the remote cache, in this case, is actually slower than just doing the work!

The lesson? **When using a remote, distributed cache, figure out how long it actually takes to read from the cache.** You can find this out via benchmarking, like I did, or you can even read it from your Rails logs. Make sure you're not caching anything that takes longer to read than it does to write!

## Conclusions

Hopefully, this article has given you all you need to know to get out there and use caching more in your Rails apps. It really is the key to extremely performant Rails sites. ■

---

Nate Berkopec is a Ruby on Rails developer in NYC. Over the years, he's seen a lot of slow Rails apps, so nowadays he writes about Rails performance at *nateberkopec.com*

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

# The Ultimate OpenBSD Router

*By* TJ

Friends don't let friends use consumer networking equipment. The consumer-grade home routers are particularly bad. They're proprietary, have security issues, and offer very little flexibility. Why would you let something like that sit between you and the internet? This tutorial will show you how to build your own gateway, based on OpenBSD and PF, and take back control of the network. Let's get started.

## Hardware

This is a list of hardware I'll be using (although nothing in this tutorial is specific to it).

- A Soekris net6501 with power supply
- A low profile USB drive or mSATA SSD for the OS
- Four CAT6 cables, green is my favorite color
- A USB to serial converter and null modem cable (if you want to install via serial console)

Buy whatever hardware you want, just make sure the network cards are supported beforehand. The board I chose uses Intel NICs that are known to have good BSD support.

## Background

First, let's define what a router is, since everyone has different requirements. I've got three computers that need to share my internet connection. One of them is a server that I'd like to be able to SSH into remotely, but otherwise I don't want any of the systems exposed to the internet. The router will be doing the following things:

- Performing Network Address Translation
- Giving my server and laptop static IPs, based on their MAC address
- Handing out IP addresses via DHCP to everyone else
- Doing local DNS caching for the LAN and encrypting all outgoing DNS lookups
- Allowing incoming SSH connections to my server and the router itself
- Automatically emailing me when there's a security patch I need to apply

This ultimate router will be running nothing more than OpenBSD. Almost everything I'm using is included in the base system. I'm going to assume you're capable of installing the OS on your machine. If you want a fully-encrypted installation, see our tutorial for that. [hn.my/fde] If you have a serial cable, install it that way. [hn.my/sercon] Combining FDE and serial requires some additional steps though. [hn.my/serconx] You can also install over PXE if that's your thing. The hardware I chose has four NICs, which show up in the OS as em0, em1, em2 and em3. I'm going to be using the first one as the external interface and the other three as the internal interfaces for the LAN. This particular board also has a PCIe x1 slot that can be used for expansion.

## Networking

The setup detailed here is an "all-in-one" solution. All the clients will be directly connected to the gateway, without the need for a switch. This is done by bridging your internal NICs with the virtual ethernet interface.

```
# echo dhcp > /etc/hostname.em0
# echo up > /etc/hostname.em1
# echo up > /etc/hostname.em2
# echo up > /etc/hostname.em3
# echo 'inet 192.168.1.1 255.255.255.0
192.168.1.255' > /etc/hostname.vether0
# vi /etc/hostname.bridge0
```

Add the following:

```
add vether0
add em1
add em2
add em3
blocknonip vether0
blocknonip em1
blocknonip em2
blocknonip em3
up
```

We need to allow IP forwarding and adjust a couple other values for network throughput. Note that we're not actually enabling any of these options just yet. They'll be applied after our first reboot.

```
# vi /etc/sysctl.conf
```

Add the following:

```
net.inet.ip.forwarding=1
net.inet.ip.redirect=0
kern.bufcachepercent=50
net.inet.ip.ifq.maxlen=1024
net.inet.tcp.mssdflt=1440
kern.securelevel=2
```

- **net.inet.ip.forwarding** lets traffic pass through the interfaces when needed. This is the only required sysctl change, the others are just recommendations.
- **net.inet.ip.redirect** disables sending IP redirects.
- **kern.bufcachepercent** tells the kernel how much memory it can use for cache.
- **net.inet.ip.ifq.maxlen** should generally be 256 times the number of NICs you have — four in this case.
- **net.inet.tcp.mssdflt** should match the "max-mss" value in our firewall config. A value of 1440 is a good general rule for most networks, but you can adjust it to be higher or lower depending on your needs (or disable it entirely).
- **kern.securelevel** locks the securelevel to the highest setting and prevents changes to the firewall rules. You might want to hold off on this until you have a firewall configuration in place that you're happy with.

## DHCP

Users need to have IP addresses, so we'll need to tell the DHCP server to start on boot and give them one.

```
# echo 'dhcpd_flags="vether0"' >> /etc/rc.conf.
# local
# vi /etc/dhcpd.conf
```

Take this example and modify it for your needs:

```
option domain-name-servers 192.168.1.1;
subnet 192.168.1.0 netmask 255.255.255.0 {
    option routers 192.168.1.1;
    range 192.168.1.4 192.168.1.254;
    host meimei {
        fixed-address 192.168.1.2;
        hardware ethernet 00:00:00:00:00:00;
        }
    host suigintou {
        fixed-address 192.168.1.3;
        hardware ethernet 11:11:11:11:11:11;
        }
}
```

You can specify any IP range you want to use and any DNS servers you want to use. By default, I want all clients to query the local DNS resolver that we'll set up in just a minute. This will speed up repeated lookups and is handy to have. Use the MAC addresses of your computers if you want static IPs.

## DNS

Setting up a local DNS caching server is pretty easy. We'll be using unbound, which is part of the base system, along with DNSCrypt to keep our lookups private.

```
# echo 'unbound_flags=""' >> /etc/rc.conf.local
# vi /var/unbound/etc/unbound.conf
```

Something like this should do:

```
server:
    interface: 192.168.1.1
    interface: 127.0.0.1
    do-ip6: no
    access-control: 192.168.1.0/24 allow
    do-not-query-localhost: no
    hide-identity: yes
    hide-version: yes

forward-zone:
        name: "."
        forward-addr: 127.0.0.1@40
```

Now we'll set up dnscrypt-proxy. It's not part of the base system, so we'll need to install it from ports or packages.

```
# export PKG_PATH=http://ftp.openbsd.org/pub/
OpenBSD/`uname -r`/packages/`uname -m`/
# pkg_add dnscrypt-proxy
# echo 'pkg_scripts="dnscrypt_proxy"' >> /etc/
rc.conf.local
# echo 'dnscrypt_proxy_flags="-l /dev/null -R
opendns -a 127.0.0.1:40"' >> /etc/rc.conf.local
# echo 'nameserver 127.0.0.1' > /etc/resolv.conf
```

You can edit /etc/dhclient.conf's "supersede domain-name-servers" section so it doesn't overwrite your local nameserver...

```
# echo 'supersede domain-name-servers
127.0.0.1;' >> /etc/dhclient.conf
```

...or use a more "forceful" approach:

```
# chflags schg /etc/resolv.conf
```

The dnscrypt-proxy port won't use any server by default; you need to specify one with the "-R" flag. I'm using OpenDNS in this example, but you can check the included documentation [hn.my/dnscrypt] for a list of supported resolvers.

## Firewall Rules

The centerpiece of this entire guide is the file `/etc/pf.conf`. Like in some of the previous sections, there's another tutorial [hn.my/pf] for more in-depth explanation and advanced rulesets.

```
# vi /etc/pf.conf
```

For my needs, I ended up with:

```
int_if="{ vether0 em1 em2 em3 }"
broken="224.0.0.22 127.0.0.0/8 192.168.0.0/16
172.16.0.0/12 \
        10.0.0.0/8 169.254.0.0/16 192.0.2.0/24 \
        198.51.100.0/24, 203.0.113.0/24, \
        169.254.0.0/16 0.0.0.0/8 240.0.0.0/4
255.255.255.255/32"
set block-policy drop
set loginterface egress
set skip on lo0
match in all scrub (no-df random-id max-mss
1440)
match out on egress inet from !(egress:network)
to any nat-to (egress:0)
antispoof quick for (egress)
block in quick on egress from { $broken no-route
urpf-failed } to any
block in quick inet6 all
block out quick inet6 all
block return out quick log on egress proto { tcp
udp } from any to any port 53
block return out quick log on egress from any to
{ no-route $broken }
block in all
pass out quick inet keep state
pass in on $int_if inet
pass in on $int_if inet proto { tcp udp } from
any to ! 192.168.1.1 port 53 rdr-to 192.168.1.1
pass in on egress inet proto tcp to (egress)
port 222 rdr-to 192.168.1.2
pass in on egress inet proto tcp from any to
(egress) port 2222
```

In this example, we would be running SSH on port 2222 and the server would be running SSH on port 222, both open to the internet. Adjust to your needs.

## Final Tweaks

Since I'm using a flash drive for the OS, I want to minimize the number of writes to it. I'll append the "noatime" flag to the mount point and enable soft updates. You may also want to consider using mfs or tmpfs for /var and /tmp.

```
# vi /etc/fstab
```

Assuming my root device is "sd0" (yours might not be), my fstab will look like this:

```
/dev/sd0a / ffs rw,noatime,softdep 1 1
```

Finally, a sound server isn't the most useful thing for a router, so let's disable it.

```
# echo 'sndiod_flags=NO' >> /etc/rc.conf.local
# reboot
```

At this point, you should be able to plug in some computers to the other ethernet ports and everything will work. They'll be assigned IP addresses and granted access to the internet, while being protected by the firewall. If that's all you want, you're done! I'd recommend subscribing to the openbsd-announce mailing list to get notifications when a new security patch or version of the OS is released.

## Outgoing SMTP

It's possible to configure the router to send you nightly emails using nothing but smtpd and an email account. I'm using a throwaway gmail account for this example, but you can obviously use any mail server. Make sure your system's hostname is in present in /etc/hosts:

```
# grep 127 /etc/hosts
127.0.0.1       localhost bsdnow.tv
```

Add the email account you'll be sending the mail from.

```
# echo 'gmail youruser@gmail.com:yourpassword' >
/etc/mail/secrets
# chmod 640 /etc/mail/secrets
# chown root:_smtpd /etc/mail/secrets
# makemap /etc/mail/secrets
```

Move the default smtpd configuration to a backup file and create a new one.

```
# mv /etc/mail/smtpd.conf /etc/mail/smtpd.conf.
orig
# vi /etc/mail/smtpd.conf
```

Add the following, changing the server to whatever you used:

```
listen on lo0
table aliases db:/etc/mail/aliases.db
table secrets db:/etc/mail/secrets.db
accept for local alias <aliases> deliver to mbox
accept for any relay via tls+auth://gmail@smtp.
gmail.com:587 auth <secrets>
```

Finally, enable it on startup.

```
# echo 'smtpd_flags=""' >> /etc/rc.conf.local
# /etc/rc.d/smtpd start
```

Now you should be able to take the output of any command and send it with your email account. We can test it by doing something like this:

```
# echo 'Woah, my router can send emails! Nice
tutorials as always dude!' | mail feedback@
bsdnow.tv
```

You can pipe any command or script's output to an email, send it off and then check what's going on in the morning. It can be used for firewall logs, automatically checking for updates and patches, or really anything you can think of.

## Bandwidth Throttling

If you have to share your internet connection with other users, it's quite possible that they will hog all your bandwidth if you let them. Fortunately, pf provides a way to assign packets to different queues, giving them specific bandwidth limitations. There are two approaches to throttling: setting a maximum limit a connection can use, or reserving a minimum amount that it will always have access to. Since we're nice, we'll just reserve a minimum amount for certain types of traffic that are particularly annoying to use while all the bandwidth is being hoarded. You probably don't want your interactive SSH session lagging because of your friend torrenting. Being forced to wait ten seconds for a website to load because someone is uploading a video of their cat is also unacceptable. For this example, we'll just focus on SSH, FTP(S) and HTTP(S) traffic. The pf.conf manpage has additional details for all the different things you can do with queueing. We'll assume you have a twenty megabit connection, but you can adjust the numbers up or down to suite your needs.

```
queue limits on em0 bandwidth 20M
queue shell parent limits bandwidth 1M min 1M
queue ftp parent limits bandwidth 8M max 8M
queue web parent limits bandwidth 5M min 5M max
10M default
```

Our example protocols each get assigned to their own queue. One megabit will always be reserved for SSH traffic to prevent any frustrating delays between typing in an interactive session and getting a response. FTP(S) traffic will be choked to a max of eight megabits. Web traffic will always have five megabits reserved for it, but will also be able to use up to ten megabits, depending on how much is available. Add the pass lines for them:

```
pass out quick inet proto tcp from any to any
port 22 set queue shell
pass out quick inet proto tcp from any to any
port { 20 21 989 990 } set queue ftp
pass out quick inet proto tcp from any to any
port { 80 443 } set queue web
```

With throttling enabled, the full pf.conf would look something like this:

```
int_if="{ vether0 em1 em2 em3 }"
broken="224.0.0.22 127.0.0.0/8 192.168.0.0/16
172.16.0.0/12 \
        10.0.0.0/8 169.254.0.0/16 192.0.2.0/24
\
        198.51.100.0/24, 203.0.113.0/24, \
        169.254.0.0/16 0.0.0.0/8 240.0.0.0/4
255.255.255.255/32"
set block-policy drop
set loginterface egress
set skip on lo0
match in all scrub (no-df random-id max-mss
1440)
match out on egress inet from !(egress:network)
to any nat-to (egress:0)
queue limits on em0 bandwidth 20M
queue shell parent limits bandwidth 1M min 1M
queue ftp parent limits bandwidth 8M max 8M
queue web parent limits bandwidth 5M min 5M max
10M default
antispoof quick for (egress)
block in quick on egress from { $broken no-route
urpf-failed } to any
block in quick inet6 all
block out quick inet6 all
block return out quick log on egress proto { tcp
udp } from any to any port 53
block return out quick log on egress from any to
{ no-route $broken }
block in all
pass out quick inet proto tcp from any to any
port 22 set queue shell
pass out quick inet proto tcp from any to any
port { 20 21 989 990 } set queue ftp
pass out quick inet proto tcp from any to any
port { 80 443 } set queue web
pass out quick inet keep state
pass in on egress inet proto tcp to (egress)
port 222 rdr-to 192.168.1.2
pass in on egress inet proto tcp from any to
(egress) port 2222 flags S/SA synproxy state
pass in on $int_if inet
pass in on $int_if proto { tcp udp } from any to
! 192.168.1.1 port 53 rdr-to 192.168.1.1
```

If you want to throttle both download and upload, you could do it on a per-interface basis. For example, limit the outbound interface (em0) for download and the internal interfaces (em1-3) for upload. With pf's powerful syntax, you can get creative and combine this with certain IPs or hostnames. It's possible to limit a specific user's connection to a video streaming site based on their UID and destination address, add groups that have no limits whatsoever or really anything else you can think of.

## Bandwidth Statistics

Monitoring how much bandwidth is being used is a common feature of many routers. The same thing can be done on an OpenBSD box quite easily. We'll install the "vnstat" daemon and tell it to monitor each interface.

```
# pkg_add vnstat
# vnstat -u -i em0
# vnstat -u -i em1
# vnstat -u -i em2
# vnstat -u -i em3
# chown _vnstat /var/db/vnstat/*
```

If you only care about WAN traffic statistics, just enable it for the egress interface, which is em0 in my case. Next, make any changes you want to the configuration file:

```
# vi /etc/vnstat.conf
```

I like to make things a bit more human-readable:

```
--- vnstat.conf     Sat May  2 21:15:35 2015
+++ vnstat.conf     Sat May  2 21:13:32 2015
@@ -28,7 +28,7 @@
 # how units are prefixed when traffic is shown
 # 0 = IEC standard prefixes (KiB/MiB/GiB/TiB)
 # 1 = old style binary prefixes (KB/MB/GB/TB)
-UnitMode 0
+UnitMode 1

 # output style
 # 0 = minimal & narrow, 1 = bar column visible
@@ -37,11 +37,11 @@
 OutputStyle 3

 # used rate unit (0 = bytes, 1 = bits)
-RateUnit 1
+RateUnit 0

 # maximum bandwidth (Mbit) for all interfaces,
0 = disable feature
 # (unless interface specific limit is given)
-MaxBandwidth 100
+MaxBandwidth 0

 # interface specific limits
 #  example 8Mbit limit for 'ethnone':
```

Be sure to add the rc.d script to your startup items, alongside dnscrypt-proxy.

```
# grep scripts /etc/rc.conf.local

pkg_scripts="dnscrypt_proxy vnstatd"
```

Finally, start the daemon.

```
# /etc/rc.d/vnstatd start
```

Wait a few minutes and it should start collecting data.

## Power Management

You may also want to enable apmd to save power if your hardware supports it. It will scale the CPU down during idle times and turn it up when the load reaches a certain point. Check the man page for a few different options.

```
# echo 'apmd_flags="-A"' >> /etc/rc.conf.local
# /etc/rc.d/apmd start
```

You can check what level (with 0 being the lowest, 100 being the highest) the CPU is running at with:

```
# sysctl hw.setperf
```

Try different levels and apmd settings to find the balance you're most comfortable with. Always running it on the lowest setting might limit the data throughput too much, but it will really depend on what hardware you're using.

Thanks for reading. ■

---

TJ is the writer and producer of BSD NOW, a weekly BSD Postcast.

Reprinted with permission of the original author.
First appeared in *hn.my/bsdr* (bsdnow.tv)

# The Hardest Program I've Ever Written

*By* BOB NYSTROM

The hardest program I've ever written, once you strip out the whitespace, is 3,835 lines long. That handful of code took me almost a year to write. Granted, that doesn't take into account the code that didn't make it. The commit history shows that I deleted 20,704 lines of code over that time. Every surviving line has about three fallen comrades.

If it took that much thrashing to get it right, you'd expect it to do something pretty deep right? Maybe a low-level hardware interface or some wicked graphics demo with tons of math and pumping early-90s-style techno? A likely-to-turn-evil machine learning AI Skynet thing?

Nope. It reads in a string and writes out a string. The only difference between the input and output strings is that it modifies some of the whitespace characters. I'm talking, of course, about an automated code formatter. [hn.my/dartstyle]

## Introducing dartfmt

I work on the Dart programming language. [dartlang.org] Part of my job is helping make more Dart code, readable, idiomatic, and consistent, which is why I ended up writing our style guide. That was a good first step, but any style guide written in English is either so brief that it's ambiguous, or so long that no one reads it.

Go's "gofmt" tool showed a better solution: automatically format everything. Code is easier to read and contribute to because it's already in the style you're used to. Even if the output of the formatter isn't great, it ends those interminable soul-crushing arguments on code reviews about formatting.

Of course, I still have to sell users on running the formatter in the first place. For that, having great output really does matter. Also, I'm pretty picky with the formatting in my own code, and I didn't want to tell users to use a tool that I didn't use myself.

Getting that kind of quality means applying pretty sophisticated formatting rules. That in turn makes performance difficult. I knew balancing quality and speed would be hard, but I didn't realize just how deep the rabbit hole went.

I have finally emerged back into the sun, and I'm pleased with what I brought back. I like the output, and the performance is solid. On my laptop, it can blow through over two million lines of code in about 45 seconds, using a single core.

## Why is formatting hard?

At this point, you're probably thinking, "Wait. What's so hard about formatting?" After you've parsed, can't you just walk the AST and pretty-print it with some whitespace?

If every statement fit within the column limit of the page, yup. It's a piece of cake. (I think that's what gofmt does.) But our formatter also keeps your code within the line length limit. That means adding line breaks (or "splits" as the formatter calls them), and determining the best place to add those is famously hard. [hn.my/k27]

Check out this guy:

```
experimentalBootstrap = document.
querySelectorAll('link').any((link) =>
    link.attributes['rel'] == 'import' &&
        link.attributes['href'] ==
POLYMER_EXPERIMENTAL_HTML);
```

There are thirteen places where a line break is possible here according to our style rules. That's 8,192 different combinations if we brute force them all [1]. The search space we have to cover is exponentially large, and even ranking different solutions is a subtle problem. Is it better to split before the `.any()`? Why or why not?

In Dart, we made things harder on ourselves. We have anonymous functions, lots of higher-order functions, and — until we added `async` and `await` — used futures for concurrency. That means lots of callbacks and lots of long method chains. Some Dart users really dig a functional style and appear to be playing a game where whoever crams the most work before a single semicolon wins.

Here's real code from an amateur player:

```
_bindAssignablePropsOn.forEach((String event-
Name) => node
    .addEventListener(eventName, (_) => zone.
run(() => bindAssignableProps
        .forEach((propAndExp) => propAndExp[1].
assign(
            scope.context,
jsNode[propAndExp[0]])))));
```

Yeah, that's four nested functions. 1,048,576 ways to split that one. Here's one of the best that I've found. This is what a pro player brings to the game:

```
return doughnutFryer
    .start()
    .then((_) => _frostingGlazer.start())
    .then((_) => Future.wait([
          _conveyorBelts.start(),
          sprinkleSprinkler.start(),
          sauceDripper.start()
        ]))
    .catchError(cannotGetConveyorBeltRunning)
    .then((_) => tellEveryoneDonutsAreJustAbout-
Done())
    .then((_) => Future.wait([
          croissantFactory.start(),
          _giantBakingOvens.start(),
          butterbutterer.start()
        ])
            .catchError(_handleBakingFailures)
            .timeout(scriptLoadingTimeout,
onTimeout: _handleBakingFailures)
            .catchError(cannotGetConveyorBeltRu
nning))
    .catchError(cannotGetConveyorBeltRunning)
    .then((_) {
  _logger.info("Let's eat!");
});
```

That's a single statement, all 565 characters of it. There are about 549 billion ways we could line break it.

Ultimately, this is what the formatter does. It applies some fairly sophisticated ranking rules to find the best set of line breaks from an exponential solution space. Note that "best" is a property of the entire statement being formatted. A line break changes the indentation of the remainder of the statement, which in turn affects which other line breaks are needed. Sorry, Knuth. No dynamic programming this time [2].

I think the formatter does a good job, but how it does it is a mystery to users. People get spooked when robots surprise them, so I thought I would trace the inner workings of its metal mind. And maybe try to justify to myself why it took me a year to write a program whose behavior in many ways is indistinguishable from cat.

---

[1] Yes, I really did brute force all of the combinations at first. It let me focus on getting the output correct before I worried about performance. Speed was fine for most statements. The other few wouldn't finish until after the heat death of the universe.

[2] For most of the time, the formatter did use dynamic programming and memoization. I felt like a wizard when I first figured out how to do it. It worked fairly well, but was a nightmare to debug.

It was highly recursive, and ensuring that the keys to the memoization table were precise enough to not cause bugs but not so precise that the cache lookups always fail was a very delicate balancing act. Over time, the amount of data needed to uniquely identify the state of a sub problem grew, including things like the entire expression nesting stack at a point in the line, and the memoization table performed worse and worse.

## How the formatter sees your code

As you'd expect from a program that works on source code, the formatter is structured much like a compiler. It has a front end that parses your code and converts that to an intermediate representation [3]. It does some optimization and clean up on that [4], and then the IR goes to a back end [5] that produces the final output. The main objects here are chunks, rules, and spans.

### Chunks

A chunk is an atomic unit of formatting. It's a contiguous region of characters that we know will not contain any line breaks. Given this code:

```
format /* comment */ this;
```

We break it into these chunks: `format /* comment */ this;`.

Chunks are similar to a token in a conventional compiler, but they tend to be, well, chunkier. Often, the text for several tokens ends up in the same chunk, like `this` and `;` here. If a line break can never occur between two tokens, they end up in the same chunk [6].

Chunks are mostly linear. For example, given an expression like:

```
some(nested, function(ca + 11))
```

We chunk it to the flat list: `some( nested, function( ca + 11))`.

We could treat an entire source file like a single flat sequence of chunks, but it would take forever and a day to line break the whole thing [7]. With things like long chains of asynchronous code, a single "statement" may be hundreds of lines of code containing several nested functions or collections that each contain their own piles of code.

We can't treat those nested functions or collection literals entirely independently because the surrounding expression affects how they are indented. That in turn affects how long their lines are. Indent a function body two more spaces and now its statements have two fewer spaces before they hit the end of the line.

Instead, we treat nested block bodies as a separate little list of chunks to be formatted mostly on their own but subordinate to where they appear. The chunk that begins one of these literals, like the `{` preceding a function or map, contains a list of child block chunks for the contained block. In other words, chunks do form a tree, but one that only reflects block nesting, not expressions.

The end of a chunk marks the point where a split may occur in the final output, and the chunk has some data describing it [8]. It keeps track of whether a blank line should be added between the chunks (like between two class definitions), how much the next line should be indented, and the expression nesting depth at that point in the code.

---

[3] The IR evolved constantly. Spans and rules were later additions. Even the way chunks tracked indentation changed frequently. Indentation used to be stored in levels, where each level was two spaces, then directly in spaces. Expression nesting went through a number of representations.

In all of this, the IR's job is to balance being easy for the front-end to produce while being efficient for the back end to consume. The back end really drives this. The IR is structured to be the right data structure for the algorithm the back end wants to use.

[4] Comments were the one of the biggest challenges. The formatter initially assumed there would be no newlines in some places. Who would expect a newline, say, between the keywords in `abstract` class? Alas, there's nothing preventing a user from doing:

```
abstract // Oh, crap. A line comment.
class Foo {}
```

So I had to do a ton of work to make it resilient in the face of comments and newlines appearing in all sorts of weird places. There's no single clean solution for this, just lots of edge cases and special handling.

[5] The back end is where all of the performance challenges come from, and it went through two almost complete rewrites before it ended up where it is today.

[6] I started from a simpler formatter written by a teammate that treated text, whitespace, and splits all as separate chunks. I unified those so that each chunk included non-whitespace text, line split information, and whitespace information if it didn't split. That simplified a lot.

[7] When I added support for better indentation of nested functions that broke the code that split source into separately splittable regions. For a while, a single top-level statement would be split as a single unit, even if it contained nested functions with hundreds of lines of code. It was… not fast.

[8] Ideally, the split information in a chunk would describe the split before the chunk's text. This would avoid the pointless split information on the last chunk, and also solve annoying special-case handling of the indentation before the very first chunk.

I've tried to correct this mistake a number of times, but it causes a near-infinite number of off-by-one bugs and I just haven't had the time to push it all the way through and fix everything.

The most important bit of data about the split is the rule that controls it [9].

## Rules

Each potential split in the program is owned by a rule. A single rule may own the splits of several chunks. For example, a series of binary operators of the same kind like `a + b + c + d` uses a single rule for the splits after each `+` operator.

A rule controls which of its splits break and which don't. It determines this based on the state that the rule is in, which it calls its value. You can think of a rule like a dial and the value is what you've turned it to. Given a value, the rule will tell you which of its chunks get split.

The simplest rule is a "hard split" rule. It says that its chunk always splits, so it only has one value: `0`. This is useful for things like line comments where you always need to split after it, even in the middle of an expression [10].

Then there is a "simple" split rule. It allows two values: `0` means none of its chunks split and `1` means they all do. Since most splits are independent of the others, this gets used for most of the splits in the program.

Beyond that, there are a handful of special-case rules. These are used in places where we want to more precisely control the configuration of a set of splits. For example, the positional argument list in a function list is controlled by a single rule. A function call like:

```
function(first, second, third)
```

Will have splits after `function(`, `first,`, `second,`, and `third)`. They are all owned by a single rule that only allows the following configurations:

```
// 0: Don't split at all.
function(first, second, third)
// 1: Split before the first.
function(
    first, second, third)

// 2: Split before only the last argument.
function(first, second,
    third)

// 3: Split before only the middle argument.
function(first,
    second, third)

// 4: Split before all of them.
function(
    first,
    second,
    third)
```

Having a single rule for this instead of individual rules for each argument lets us prohibit things like:

```
function(
    first, second,
    third)
```

## Constraints

Grouping a range of splits under a single rule helps us prevent split configurations we want to avoid like this, but it's not enough. There are more complex constraints we want to enforce like: "if a split occurs inside a list element, the list should split too". That avoids output like this:

```
[first, second +
    third, fourth]
```

[9] Rules are a relatively recent addition. Originally each chunk's split was handled independently. You could specify some relations between them like "if this chunk splits then this other one has to as well", but you could not express things like "only one of these three chunks may split"

Eventually, I realized the latter is what I really needed to get argument lists formatting well, so I conceived of rules as a separate concept and rewrote the front and line splitter to work using those.

[10] At first, I thought hard splits weren't needed. Any place a mandatory newline appears (like between two statements) is a place where you could just break the list of chunks in two and line split each half independently. From the line splitter's perspective, there would be no hard splits.

Which would work… except for line comments:

```
some(expression,
   // with a line comment
   rightInTheMiddleOfIt);
```

This has to be split as a single unit to get the expression nesting and indentation correct, but it also contains a mandatory newline after the line comment.

Here, the list and the + expression have their own rules, but those rules need to interact. If the + takes value `1`, the list rule needs to as well. To support this, rules can constrain each other. Any rule can limit the values another rule is allowed to take based on its own value. Typically, this is used to make a rule inside a nested expression force the rules surrounding itself to split when it does [11].

Finally, each rule has a cost. This is a numeric penalty that applies when any of that rule's chunks are split. This helps us determine which sets of splits are better or worse than others [12].

Rule costs are only part of how overall fitness is calculated. Most of the cost calculation comes from spans.

### Spans

A span marks a series of contiguous chunks that we want to avoid splitting. I picture it like a rubber band stretching around them. If a split happens in any of those chunks, the span is broken. When that happens, the solution is penalized based on the cost of the span.

Spans can nest arbitrarily deeply. In an expression like:

```
function(first(a, b), second(c, d))
```

There will be spans around `a, b` and `c, d` to try to keep those argument lists from splitting, but also another span around `first(a, b), second(c, d)` to keep the outer argument list from splitting.

If a split occurs between `a,` and `b,` the `a, b` span splits, but so does the `first(a, b), second(c, d)` one. However, if a split occurs after `first(a, b)`, then the `a, b` span is still fine. In this way, spans teach the formatter to prefer splitting at a higher level of nesting when possible since it breaks fewer nested spans.

## Parsing source to chunks

Converting your raw source code to this representation is fairly straightforward. The formatter uses the wonderful analyzer package to parse your code to an AST. This gives us a tree structure that represents every single byte of your program. Unlike many ASTs, it even includes comments.

Once we have that, the formatter does a top-down traversal of the tree. As it walks, it writes out chunks, rules, and spans for the various grammar productions. This is where the formatting "style" is determined.

There's no rocket science here, but there are a lot of hairy corner cases. Comments can appear in weird places. We have to handle weird things like:

```
function(argument, // comment
    argument)
```

Here, we normally would have a split after the first argument owned by an argument list rule. But the line comment adheres to the , and has a hard split after it, so we need to make sure the argument list rule handles that.

Whitespace is only implicitly tracked by the AST so we have to reconstitute it in the few places where your original whitespace affects the output. Having a detailed test suite really helps here.

Once we've visited the entire tree, the AST has been converted to a tree of chunks and a bunch of spans wrapped around pieces of it.

## Formatting chunks

We've got ourselves a big tree of chunks owned by a slew of rules. Earlier, I said a rule is like a knob. Now we get to dial them in.

Doing this naïvely is infeasible. Even a small source file contains hundreds of individual rules and the set of possible solutions is exponential in the number of rules.

---

[11] There used to be a separate class for a "multisplit" to directly handle forcing outer expressions to split when inner ones did. Once rules came along, they also needed to express constraints between them, and eventually those constraints were expressive enough to be able to handle the multisplit behavior directly and multisplits were removed.

[12] I spent a lot of time tuning costs for different grammar productions to control how tightly bound different expressions were. The goal was to allow splits at the places where the reader thought code was "loosest", so stuff like higher precedence expressions would have higher costs.

Tuning these costs was a nightmare. It was like a hanging mobile where tweaking one cost would unbalance all of the others. On more than one occasion, I found myself considering making them floating point instead of integers, a sure sign of madness.

It turns out spans are what you really want in order to express looseness. Nested infix operators then fall out naturally because you have more spans around the deeper nested operands. The parse tree gives it to you for free.

These days, almost every chunk and span has a cost of 1, and it's the quantity of nested spans and contained chunks that determine where it splits.

The first thing we do is divide the chunk list into regions we know can't interfere with each other. These are roughly "lines" of code. So with:

```
first(line);
second(line);
```

We know that how we split the first statement has no effect on the second one. So we run through the list of chunks and break them into shorter lists whenever we hit a hard split that isn't nested inside an expression.

Each of these shorter chunk lists is fed to the line splitter. Its job is to pick the best set of values for all the rules used by the chunks in the line. In most cases, this is trivial: if the whole line fits on the page, every rule gets set to zero — no splits — and we're done.

When a line doesn't fit, the splitter has to figure out which combination of rule values produces the best result. That is:

1. The one with the fewest characters that go over the column limit.
2. The one with the lowest cost, based on which rules and spans were split.

Calculating the cost for a set of rule values is pretty easy, but there are still way too many permutations to brute force it. If we can't brute force it, how do we do it?

## How line splitting works

I'm a college dropout so my knowledge of algorithms was fairly, um, rudimentary. So before I interviewed at Google, I spent two days in a hotel room cramming as many of them — mostly graph traversal — in my head as I could. At the time, I thought graphs would never come up in the interviews…

Then I had multiple interview questions that reduced down to doing the right kind of traversal over a graph. At the time, I thought this stuff would never be relevant to my actual job…

Then I spent the past few years at Google discovering that damn near every program I have to write can be reduced down to some kind of graph search. I wrote a package manager where dependencies are a transitive closure and version constraint solving is graph based. My hobby rogue like uses graphs for path finding. Graphs out the wazoo. I can do BFS in my sleep now.

Naturally, after several other failed approaches, I found that line splitting can be handled like a graph search problem [13]. Each node in the graph represents a solution — a set of values for each rule. Solutions can be partial: some rules may be left with their values unbound.

From a given partial solution (including the initial "no rules bound" one), there are edges to new partial solutions. Each binds one additional rule to a value. By starting from an empty solution and walking this graph, we eventually reach complete solutions where all of the rules have been bound to values.

Graph search is great if you know where your destination is and you're trying to find the best path. But we don't actually know that. We don't know what the best complete solution is. (If we did, we've been done already!)

Given this, no textbook graph search algorithm is sufficient. We need to apply some domain knowledge — we need to take advantage of rules and conditions implicit in the specific problem we're solving.

After a dozen dead ends, I found three (sort of four) that are enough to get it finding the right solution quickly:

### Bailing early

We are trying to minimize two soft constraints at the same time:

1. We want to minimize the number of characters that overflow the line length limit. We can't make this a hard constraint that there is no overflow because it's possible for a long identifier or string literal to overflow in every solution. In that case, we still need to find the one that's closest to fitting.
2. We want to find the lowest cost — the fewest split rules and broken spans.

[13] I had known that clang-format worked this way for a long time, but I could never wrap my head around how to apply it to dartfmt's richer chunk/rule/span system.

I took a lot of walks along the bike trail next to work trying to think through a way to get graph search working when the two numbers being optimized (overflow characters and cost) are in direct opposition, and we don't even know what the goal state looks like. It took a long time before it clicked. Even then, it didn't work at all until I figured out the right heuristics to use to optimize it.

The first constraint dominates the second: we'll prefer a solution with any cost if it fits one more character in. In practice, there is almost always a solution that does fit, so it usually comes down to picking the lowest cost solution [14].

We don't know a priori what the cost of the winning solution will be, but we do know one useful piece of information: forcing a rule to split always increases the cost.

If we treat any unbound rule as being implicitly unsplit [15], that means the starting solution with everything unbound always has the lowest cost (zero). We can then explore outward from there in order of increasing cost by adding one rule at a time.

This is a basic best-first search: we keep a running queue of all of the partial solutions we've haven't explored yet, sorted from lowest cost to highest. Each iteration, we pop a solution off.

If the solution completely fits in the page width, then we know we've won the overflow constraint. Since we're exploring in order of increasing cost, we also know it's the lowest cost. So, ta-da!, we found the winner and can stop exploring. Otherwise, if it has any unbound rules, we enqueue new solutions, each of which binds one of those to a value.

We basically explore the entire solution space in order of increasing cost. As soon as we find a solution that fits in the page, we stop.

### Avoiding dead ends

The above sounds pretty promising, but it turns out that there can be an imperial ton of "low-cost but overflowing" solutions. When you're trying to format a really long line, there are plenty of ways it cannot fit, and this algorithm will try basically all of them. After all, they're low cost since they don't have many splits.

We need to avoid wasting time tweaking rules that aren't part of the problem. For example, say we're looking at a partial solution like this:

```
// Blog-friendly 40-char line limit:    |
function(
    firstCall(a, b, c, d, e, f, g, h),
    secondCall("very long argument string
here"));
```

There are a bunch of ways we can split the arguments to `firstCall()`, but we don't need to. Its line already fits. The only line we need to worry about is the `secondCall()` one.

So, when we are expanding a partial solution, we only bind rules that have chunks on overflowing lines. If all of a rule's chunks are on lines that already fit, we don't mess with it. In fact, we don't even worry about rules on any overflowing line but the first. Since tweaking the first line will affect the others, there's no reason to worry about them yet [16].

This dramatically cuts down "branchiness" of the graph. Even though a partial solution may have dozens of unbound rules, usually only a couple are on long lines and only those get explored.

### Pruning redundant branches

This gets us pretty far, but the splitter can still go off the deep end in some cases. The problem is that within large statements, you still run into cases where how you format part of the statement is mostly independent of later parts.

Take something like:

---

[14] 14 For a long time, overflow and cost were treated as a single fitness function. Every overflow character just added a very high value to the cost to make the splitter strongly want to avoid them.

Splitting overflow out as a separate metric turned out to be key to getting the graph search to work because it let us order the solutions by cost independently of overflow characters.

[15] I went back and forth on how an unbound rule should implicitly behave. Treating it as implicitly split gives you solutions with fewer overflow characters sooner. Treating it as unsplit gives you lower costs.

[16] Oh, God. I tried a million different ways to reduce the branchiness before I hit on only looking at rules in the first long line. I'm still amazed that it works.

I could also talk about how controlling branchiness lets us avoid reaching the same state from multiple different paths. After all, it's a graph, but everything I've described talks about it like it's a tree. By carefully controlling how we extend partial solutions, we ensure we only take a single path to any given complete solution.

Before I got that working, I had to keep a "visited" set to make sure we didn't explore the same regions twice, but just maintaining that set was a big performance sink.

```
// Blog-friendly 40-char line limit:    |
new Compiler(
    assertions: options.has('checked-mode'),
    annotations: options.has('annotations'),
    primitives: options.has('primitives'),
    minify: options.has('minify'),
    preserve: options.has('preserve'),
    liveAnalysis: check(options.has('live'),
options.has('analysis')),
    multi: options.has('multi'),
    sourceMap: options.has('source-map'));
```

Each of those named arguments can be split in a few different ways. And, since those are less nested — which means fewer split spans — than that nasty `liveAnalysis:` line, it will try every combination of all of them before it finally gets down to the business of splitting that `check()` call.

The best way to split the `liveAnalysis:` line is the best way to split it regardless of how we split `assertions:` or `annotations:`. In other words, there are big branches of the solution space that initially differ in irrelevant ways, but eventually reconvene to roughly the same solution. We traverse every single one of them.

What we need is a way to prune entire branches of the solution space. Given two partial solutions A and B, if we could say not just "A is better than B" but "every solution we can get to from A will be better than every solution we can get to from B" then we can discard B and the entire branch of solutions stemming from it.

It took some work, but I finally figured out that you could do this in many cases. Given two partial solutions, if one has a lower cost than the other and:

- They have the same set of unbound rules (but their bound rules have different values, obviously).
- None of their bound rules are on the same line as an unbound rule.
- None of their bound rules place constraints on an unbound rule.

If all of those are true, then the one with a lower cost will always lead to solutions that also have a lower cost. Its entire branch wins. We can discard the other solution and everything that it leads to. Once I got this working, the formatter could line split damn near anything in record time [17].

### An escape hatch

Alas, that "damn near" is significant. There are still a few cases where the formatter takes a long time. I've only ever seen this on machine-generated code. Stuff like:

[17] Discarding overlapping branches is the last macro-optimization I did and its behavior is very subtle. Correctly detecting when two partial solutions overlap took a lot of iteration. Every time I thought I had it, one random weird test would fail where it accidentally collapsed two branches that would eventually diverge.

That bullet list was paid for in blood, sweat, and tears. I honestly don't think I could have figured them out at all until late in the project when I had a comprehensive test suite.

```
class ResolutionCopier {
  @override
  bool visitClassDeclaration(ClassDeclaration node) {
    ClassDeclaration toNode = this._toNode as ClassDeclaration;
    return javaBooleanAnd(
        javaBooleanAnd(
            javaBooleanAnd(
                javaBooleanAnd(javaBooleanAnd(javaBooleanAnd(
                        javaBooleanAnd(javaBooleanAnd(
                            javaBooleanAnd(javaBooleanAnd(javaBooleanAnd(
                                    _isEqualNodes(node.documentationComment,
                                        toNode.documentationComment),
                                    _isEqualNodeLists(
                                        node.metadata, toNode.metadata)),
                                _isEqualTokens(node.abstractKeyword,
                                    toNode.abstractKeyword)), _isEqualTokens(
                                node.classKeyword, toNode.classKeyword)),
                            _isEqualNodes(
                                node.name, toNode.name)), _isEqualNodes(
                            node.typeParameters, toNode.typeParameters)),
                        _isEqualNodes(
                            node.extendsClause, toNode.extendsClause)),
                    _isEqualNodes(
                        node.withClause, toNode.withClause)), _isEqualNodes(
                    node.implementsClause, toNode.implementsClause)),
                _isEqualTokens(node.leftBracket, toNode.leftBracket)),
            _isEqualNodeLists(
                node.members,
                toNode.members)),
        _isEqualTokens(
            node.rightBracket,
            toNode.rightBracket));
  }
}
```

Yeah, welcome to my waking nightmare. Unsurprisingly, code like this bogs down the formatter. I want dartfmt to be usable in things like presubmit scripts where it will have a ton of weird code thrown at it and it must complete in a reliable amount of time.

So there is one final escape hatch. If the line splitter tries, like, 5,000 solutions and still hasn't found a winner yet, it just picks the best it found so far and bails.

In practice, I only see it hit this case on generated code. Thank God.

## Finally, output

Once the line splitter has picked values for all of the rules, the rest is easy. The formatter walks the tree of chunks, printing their text. When a rule forces a chunk to split, it outputs a newline (or two), updates the indentation appropriately and keeps trucking.

The end result is a string of (I hope!) beautifully formatted Dart code. So much work just to add or remove a few spaces! ■

---

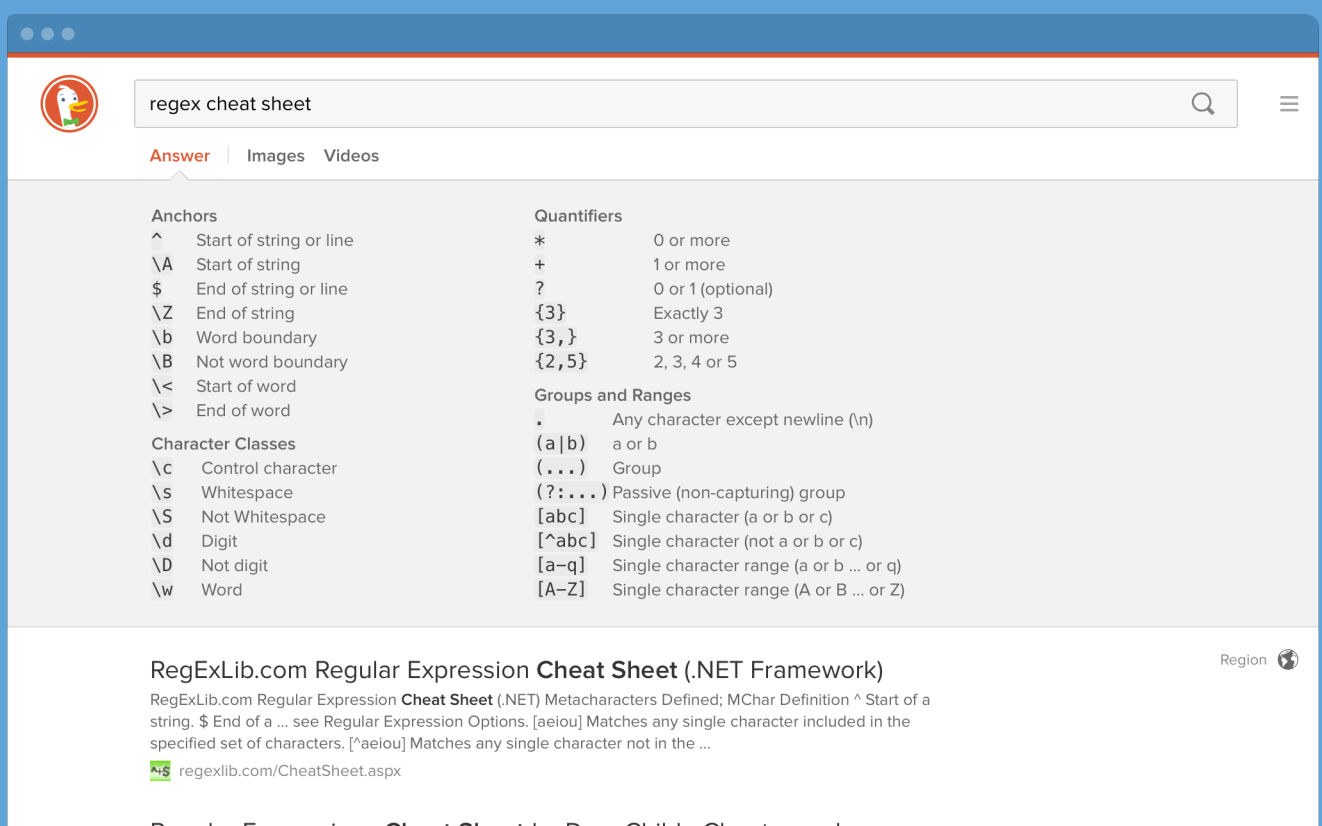
Robert Nystrom has programmed professionally for twenty years. He's worked on games, music applications, the web, and programming languages. The common thread, if there is one, is that he's most excited by making software that magnifies the creativity of others, whether that's other programmers using his code, or end users using his apps. Robert lives with his wife and two daughters in Seattle where you are most likely to find him cooking for his friends and plying them with good beer.

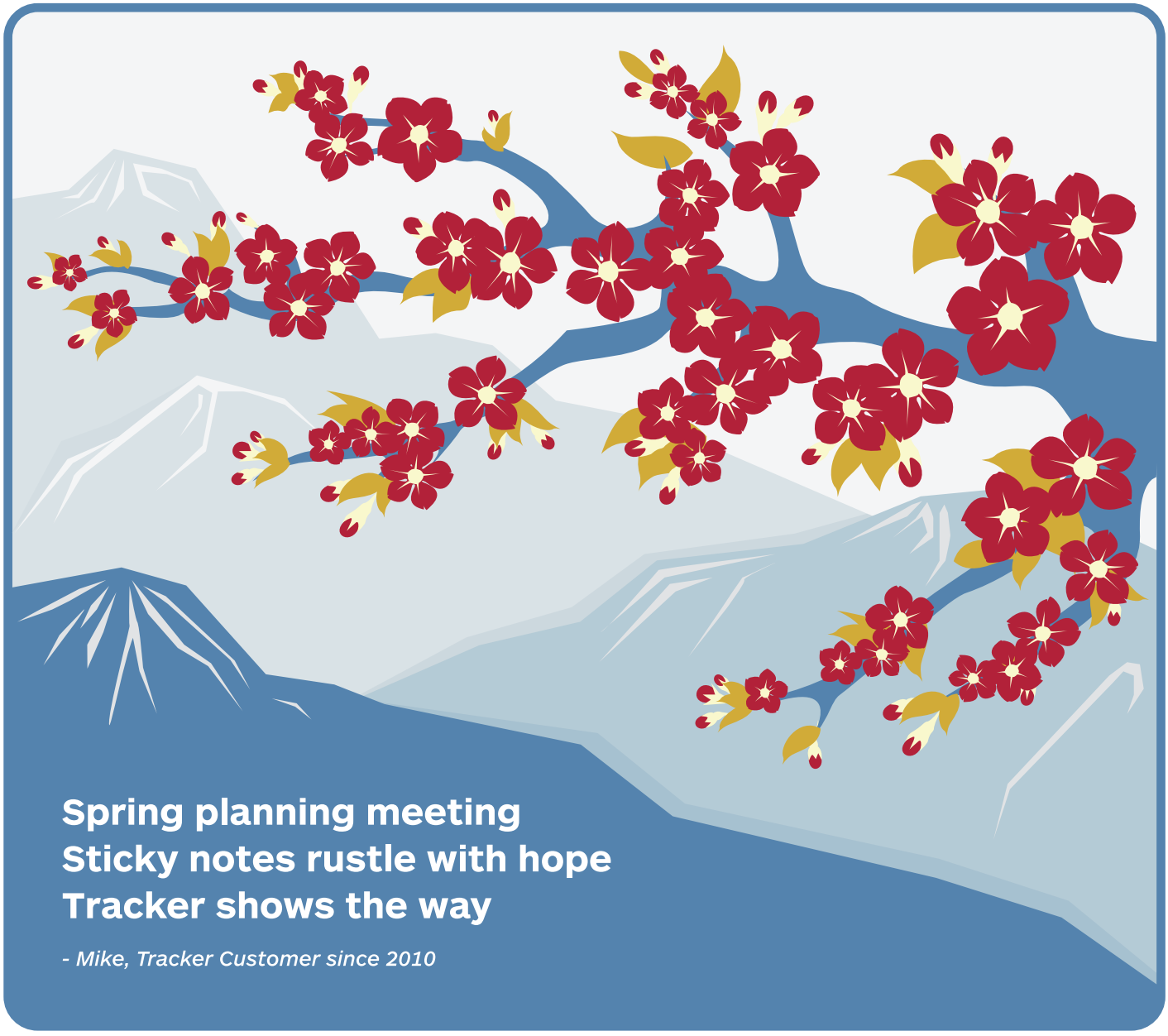
Reprinted with permission of the original author. First appeared in *hn.my/hard* (stuffwithstuff.com)

# Join the DuckDuckGo Open Source Community.

Create Instant Answers or share ideas and help change the future of search.

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com






# Discover the newly redesigned Pivotal Tracker

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project funcionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com.

PivotalTracker

Build better software, faster.