Extracted from:

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

Alexander Dymo

edited by Michael Swaine

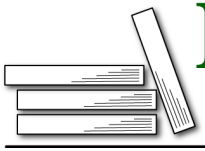
Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

Alexander Dymo

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-069-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2015

Analyze and Compare Measurements Using Statistics

Despite our best efforts to isolate our code from external factors, there will still be a variation in our measurements when we run the same code over and over again. Most of the time this won't bother us. For example, if our code takes from 10 to 15 seconds before optimization and from 1 to 2 seconds after, we won't need any statistics to tell us that our optimization worked.

But sometimes things do not look as certain. For example, say we optimized the code and the execution time went down from the 120–150 ms range to the 110–130 ms range. How can we be sure that the perceived optimization of 10–20 ms is the result of our change and not some random factor?

To answer such questions, we'll need to have some way of comparing performance measurements without knowing the true performance values before and after optimization. And statistics has the tools to do exactly that. Let me show you how to use them.

Imagine we measured the performance of the same code `n` times before optimization and `n` times after optimization. Now we want to compare these two.

Let me rephrase the same question in terms of statistics. I have two samples of random independent variables `x` and `y`. The first is the performance before optimization, the second is after. The size of my sample is `n`. And my question is: Are these two samples significantly different?

If we knew the true values of performance before and after, the numerical measure of the optimization effect would be just the difference between them.

And it turns out we can apply the same approach to the before and after samples that have a degree of uncertainty in them. We can calculate an interval within which we can confidently state the true optimization lies. Statistics calls this interval the confidence interval. The size of that interval will depend on the chosen level of confidence. In empirical science that level is usually 95%, meaning we can be 95% sure that the true optimization will lie inside that interval.

Let's say we are subtracting the number after optimization from the number before. So if the lower bound of the confidence interval is larger than zero, then we can confidently state that the optimization worked. If the interval starts with a negative number and ends with a positive number, then we can say that our optimization does nothing. If the upper bound is lower than zero, then our optimization made things worse.

So, it's simple to reach conclusions once we find the confidence interval of the optimization. The only remaining question is how to calculate one. Let me show you the algorithm.

1. Estimate the mean of before and after performance measurements with their averages:

$$\overline{x} = \frac{\sum_i x_i}{n} \qquad \overline{y} = \frac{\sum_i y_i}{n}$$

2. Calculate the standard deviation:

$$s_x = \sqrt{\frac{\sum_i (x_i - \overline{x})^2}{n-1}} \qquad s_y = \sqrt{\frac{\sum_i (y_i - \overline{y})^2}{n-1}}$$

3. Get the difference between before and after means. That would be the mean of our optimization:

$$\overline{x} - \overline{y}$$

4. Calculate the standard error of the difference, or in other words, the standard deviation of the optimization:

$$s_{\overline{x}-\overline{y}} = \sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{n}}$$

5. Finally, obtain the 95% confidence interval of the optimization. That is roughly two standard deviations away from its mean:

$$(\overline{x} - \overline{y}) \pm 2 * s_{\overline{x}-\overline{y}}$$

We want the lower bound of the interval to be larger than 0. So for example, this interval proves we indeed optimized our code:

$$0.05 \pm 0.02$$

The true difference between before and after values lies in the interval from 0.03 to 0.07 seconds. So in this imaginary case we optimized at least 30 ms.

It might seem that there's no point in calculating the confidence interval if the difference of the means is negative. But remember, we also want to know whether the optimization made things worse. For example, consider the intervals -0.05 ± 0.08 and -0.05 ± 0.02.

In both cases optimization didn't work. But in the second case it made things worse. The upper bound of the interval is –0.03. This means that with 95% confidence we can state that the code slowed down after the change, and the "optimization" must be reverted.

If you are more statistically inclined, you might frown at my confidence interval analysis. Yes, that is a shortcut, but a useful one. Of course, if you would like to be more rigorous, you can apply any of the hypothesis tests to make sure the optimization was significant. If you took 30 measurements or more, you can use the z-test. Otherwise, the t-test should work. I won't talk about these tests here because the confidence interval analysis should be good enough.

OK. Enough of formulas. It's time for an example.

The following calculates the product of the numeric array values in an idiomatic Ruby way:

chp7/before.rb
```
require 'benchmark'

data = Array.new(100000) { 10 }

GC.start
time = Benchmark.realtime do
  product = data.inject(1) { |product, i| product * i }
end
puts time
```

As we already know from Chapter 2, the inject iterator can be bad for performance. So we optimize by replacing it with each and calculating the product ourselves.

chp7/after.rb
```
require 'benchmark'

data = Array.new(100000) { 10 }

GC.start
time = Benchmark.realtime do
  product = 1
  data.each do |value|
    product *= value
  end
end
puts time
```

Let's run our before and after examples ten times each. That's not enough to make statistically sound conclusions, but we'll still do this for the sake of brevity. To make our statistics work we should take, as a rule of thumb, more than 30 measurements.

Before optimization:

```
$ for i in {1..10}; do   \
    ruby chp7/before.rb; \
  done
1.4879834910097998
1.4997473749972414
1.4694619810034055
1.4671519770054147
1.4394851910183206
1.4421958939929027
1.528489818010712
1.4666885799961165
1.4510531660052948
1.4629958330187947
```

After optimization:

```
$ for i in {1..10}; do \
    ruby chp7/after.rb; \
  done
1.4609190789924469
1.5091172570013441
1.4735914790071547
1.4498213600018062
1.445483470975887
1.46490533000906
1.434979079987388
1.4596990160061978
1.4734973890008405
1.4513041169848293
```

Just looking at the numbers it's impossible to tell whether the optimization worked. So let's use statistics.

But before we do that, we need to round our numbers off. They contain too many non-significant figures. Ruby's `Benchmark#realtime` that we use for measurements uses the operating system clock. That has microsecond precision in most cases, so we'll round our results to that.

Here are the rules for rounding to significant figures:

- If the first non-significant figure is a 5 followed by other non-zero digits, round up the last significant figure (away from zero).

  For example, 1.2459 as the result of a measurement that only allows for three significant figures should be written 1.25.

- If the first non-significant figure is a 5 not followed by any other digits or followed only by zeros, rounding requires a tie-breaking rule. For our case, use the "round half to even" rule, which rounds to the nearest even number.

  For example, to round 1.25 to two significant figures, round down to 1.2. To round 1.35, you should instead round up to 1.4.

- If the first non-significant figure is more than 5, round up the last significant figure.

- If the first non-significant figure is less than 5, just truncate the non-significant figures.

Once you start rounding to significant figures, you must continue doing so for all subsequent results of calculations.

Let's follow the rules and round our measurements to significant figures.

Before optimization:	After optimization:
1.487983	1.460919
1.499747	1.509117
1.469462	1.473591
1.467152	1.449821
1.439485	1.445483
1.442196	1.464905
1.528490	1.434979
1.466689	1.459699
1.451053	1.473497
1.462996	1.451304

Now let's follow our algorithm to get the optimization mean and its confidence interval.

1. Averages of the before and after performance measurements:

$$\overline{x} = \frac{1.487983 + 1.499747 + \cdots}{10} = 1.471525$$

$$\overline{y} = \frac{1.460919 + 1.509117 + \cdots}{10} = 1.462332$$

2. The standard deviation of the measurements:

$$s_x = \sqrt{\frac{(1.487983 - 1.471525)^2 + (1.499747 - 1.471525)^2 + \cdots}{9}} = 0.027361$$

$$s_y = \sqrt{\frac{(1.460919 - 1.462332)^2 + (1.509117 - 1.462332)^2 + \cdots}{9}} = 0.020456$$

3. The mean of our optimization:

$$1.471525 - 1.462332 = 0.009193$$

4. The standard deviation of the optimization:

$$s_{\overline{x} - \overline{y}} = \sqrt{\frac{0.027361^2}{10} + \frac{0.020456^2}{10}} = 0.010803$$

5. The 95% confidence interval of the optimization:

$$0.009193 \pm 2 * 0.010803 = (-0.012413,\ 0.030799)$$

What's the conclusion? With 95% confidence we can say that our optimization didn't work. Or, more exactly, we can't tell whether or not it worked. The difference between the `inject` and `each` iterators was not significant enough for this example.

Now let's take another look at the optimization mean and the confidence interval. The mean of our optimization was positive, about 9 ms. What invalidated our result is the standard deviation. Because it was too large, the 95% confidence interval is around plus/minus 20 ms.

What if we could reduce the variability in measurements? That would decrease the standard deviation, and, in turn, the confidence interval would be shorter. Can it be that with more precise measurements the confidence interval would lie above zero? Absolutely! That's why we spent so much time in the first part of this chapter to reduce the effect of various external and internal factors.

It is important to reduce the standard deviation of measurements as much as possible. Otherwise, you won't be able to compare your before/after results at all.

Now, in this example we are talking about milliseconds. In reality I was never able to detect such optimizations in Ruby applications. But you should definitely aim for the order of tens of milliseconds—at the very least, the lower hundreds.

You should now know enough techniques to reduce the dispersion in measurements. But if you tried them all and the standard deviation is still too high to compare results, try this: exclude outliers—measurements that are too distant from each other. Mathematicians don't like this approach, but it can help if nothing else works.

The second run of my after optimization example measured 1.509117 seconds. That was definitely an outlier. If I excluded it, both my mean and standard deviation would go down significantly.

But don't blindly exclude any results. There are a couple of statistically sound techniques of data rejection. Make sure you learn them.[2]

OK, so now you know how to compare measurements before and after optimization. Some pretty hardcore statistics are involved in that, and you probably think now, "why bother?" You'll find the answer right away in the next chapter. For now, let's summarize what you've learned.

Takeaways

There's only one way to prove that the optimization worked. You measure the performance before and after, and you compare. But the devil is in the details. Here's what you need to take care of to get the measurements right.

1. Minimize external factors to increase measurement accuracy.
2. Make sure that GC behaves as predictably as possible to decrease variability in measurements.
3. Take as many measurements as practical to make statistical analysis possible. A good default is 30.
4. Compare before and after numbers by calculating the confidence interval of the optimization effect. Conclude that optimization worked only when the lower bound of the confidence interval is higher than 0.
5. Try to reduce dispersion in measurements as much as possible. Otherwise even with statistical tools you won't be able to tell whether or not you optimized the code.

2. https://en.wikipedia.org/wiki/Truncation_(statistics) and https://en.wikipedia.org/wiki/Winsorising

Now we know how to do measurements, and how to compare them. But the goal of optimization is not to measure it, nor even to make sure that the optimized code indeed runs faster.

The real goal is to *optimize* and to make sure the slowdown never happens again. How can you do that? After optimization you'll need to measure the performance after every change, and detect even the smallest regressions from the achieved performance level.

If that smells like testing, you're right: it is testing. Performance testing. And that's exactly what we'll talk about in the next chapter.