# An introduction to 99 percentile for programmers.

With examples in Go (Golang).

Ankur Anand Follow
Sep 29 · 5 min read

Whether you're just getting started with programming, or you have years of experience, you've likely at some point of the time will do or have done some performance test of your application to measure the Latency (*Latency is defined as* ***the time it took one operation to happen***).

> *When it comes to latency result analysis,* ***It's tempting to reach for basic statistics like mean, median or max. But this ignores relevance problems*** *— turning the focus away from latency behavior distribution over time toward others, which often hide the truth.*


Actual Vs Mean(Avg), Median, Max report.

The objective of this blog post is to explain the importance of the analysis of 99 percentile, their notations, differences with basic statistics like mean, median or max, and by the end turn this newly learned knowledge into a real-world scenario and write an Application in Go(Golang) around this.

So let's focus on understanding these basic elements of statistics, as well and then slowly move toward the other topics as mentioned above.

## Mean(Average), Median

Both of these are measures of **central tendency** (which refers to the measure used to determine the "centre" of a distribution of data).

> It's used to find a single score that is most representative of an entire data set.

Consider this sample data set of latency number in seconds.
`[2, 2, 2, 4, 6, 2, 3, 3]`

If we could pick a **single most representative** value to represent this sample data set, what ways we could do it?

**Mean:** Add up all values, then divide by the total number of values.
24/**8 = 3**

**Median:** Sort the value in order, and find out what lies in the middle. `[2, 2, 2, 2, 3, 3, 4, 6]` . So here we have `2` and `3` both as the middle element, but we can pick only one, so we take the average of `2` and `3`. **2.5**

Imagine this **single most representative** value **3** or **2.5** given to you from the above latency data set, which you may consider a "normal" Latency number. **But what this single most representative value has done is, it has hidden the outliers** (outlier is a data point that differs significantly from other observations), which may seem insignificant in this small dataset.

These outliers which are usually hidden by the *mean* and *median,* are really important because "THIS HAS HAPPENED" within your application and by not having the value of these data you don't have any insight into what actually happening. **But then, the max is easily distorted by a single outlier**.

> So these single value plotting latency by Mean, Median or Max for Measuring performance is ambiguous and unreliable.

So how do we measure performance more responsibly?. We Look at quantiles, not Mean, Median or Max, which brings us to our next element percentiles.

## Percentiles

Before going to how do Percentiles deal with the above of variables, let us focus on having a basic understanding of percentiles through an example.

Consider the same data set, `[2, 2, 2, 2, 3, 3, 4, 6]` . How many numbers are even? — **6**. What is the percentage of even number? = (6/8)*100 = **75%.** But that percentage,

> A percentile is a value below which a certain percentage of observation lie. Percentile is NOT Percentage.

Top highlight

So what is the percentile value of `4` in the above dataset? - **(number of value below 4/ total number) * 100**

(6/8) *100 = 75 percentile. i.e 75% of data in the dataset is below `4` .

What is the index in the dataset which marks where the percentile just about begin? — **(percentile/100) * (total number n + 1)**

(50/100) * (9) = 4.5 ~ 5

The value at Index 5 (not zero indexes) is `3` i.e 50% of the dataset is below 3.

> Plotting latency by percentile distribution is an excellent way to see what your performance behavior actually looks like because latency isn't going to be uniform, They often exhibit things like GC pauses and other "hiccups," and central tendency tends to hide these.

**Measuring performance isn't all that easy, but if you do it, do it responsibly.**

## Using the percentile

Consider you've been assigned a task to write a service that makes a call to external service, and you need to Measure the Latency for DNS resolution of a system where the DNS Cache has not been enabled.

### HTTP tracing

Go supports **HTTP tracing** with the help of the `net/http/httptrace` standard package! This package allows you to trace the phases of an HTTP request.

```
trace := &httptrace.ClientTrace{
        DNSStart: func(info httptrace.DNSStartInfo) {
                dnsStart = now()
        },
        DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
                dnsDuration = now() - dnsStart
        },
}
```

httptrace

The preceding code is all about tracing HTTP requests. The `httptrace.ClientTrace` object defines the events that interest us. When such an event occurs, the relevant code is executed. As we are interested in measuring the DNS latency we will trace when DNS resolution starts and when it ends.

Also since calculating percentiles can be a trivial task in a large dataset. In a naive implementation, we simply store all the values in a sorted array. To find the 75 percentile, you simply find the value that is at `my_array[count(my_array) * 0.75]`.

As the sorted array grows linearly with the number of values in your dataset — clearly, the naive implementation does not scale. So to calculate percentiles across a potentially large dataset of values, *approximate* percentiles are calculated using **T-Digest: a data structure to estimate quantiles accurately**. It's an efficient online algorithm. To balance the memory utilization with estimation accuracy there is a compression parameter. By increasing the compression value, you can increase the accuracy of your percentiles at the cost of more memory, and it also makes the algorithm slower since the underlying tree data structure grows in size, resulting in more expensive operations.

We will use compression value as `100` for our demonstration purpose.

```
td := tdigest.NewWithCompression(100)
```

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "net/http/httptrace"
        "sync"
        "time"

        "github.com/influxdata/tdigest"
)

// n number of request we are going to make
var n = 1000

var startTime = time.Now()

// now returns time.Duration using stdlib time
func now() time.Duration { return time.Since(startTime) }

// results holds a individual dns lookup duration upto n
var results = make(chan time.Duration, n)

func main() {

        var wg sync.WaitGroup
        wg.Add(n)
        for i := 0; i < n; i++ {
                go func() {
                        var dnsStart, dnsDuration time.Duration

                        trace := &httptrace.ClientTrace{
                                DNSStart: func(info httptrace.DNSStartInfo) {
                                        dnsStart = now()
                                },
                                DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
                                        dnsDuration = now() - dnsStart
                                },
                        }

                        req, _ := http.NewRequest("GET", "https://ankuranand.com", nil)
                        req = req.WithContext(httptrace.WithClientTrace(req.Context(), t
                        _, err := http.DefaultTransport.RoundTrip(req)
```

```
                        results <- dnsDuration
                        wg.Done()
                        if err != nil {
                                fmt.Println(err)
                                return
                        }
                }()
        }
        wg.Wait()
        td := tdigest.NewWithCompression(100)
        var avgDNS float64
        for i := 0; i < n; i++ {
                dns := <-results
                avgDNS += dns.Seconds()
                td.Add(dns.Seconds(), 1)
        }

        log.Println("Avg", avgDNS/float64(n))
        // Compute Quantiles
        log.Println("50th", td.Quantile(0.5))
        log.Println("75th", td.Quantile(0.75))
        log.Println("90th", td.Quantile(0.9))
        log.Println("99th", td.Quantile(0.99))
}
```

**dnsMetrics.go** hosted with ❤ by **GitHub** **view raw**

DNS percentile metrics in Go

Output:

```
2019/09/30 00:02:34 Avg 0.4358035023000002
2019/09/30 00:02:34 50th 0.45583694643874645
2019/09/30 00:02:34 75th 0.5710421958333333
2019/09/30 00:02:34 90th 0.5991279228758171
2019/09/30 00:02:34 99th 0.6658698944444444
```

The example above is not a great one, but it serves its purpose of demonstration.

While the Average latency number `0.43s` from the above output may seem normal, you can see it doesn't give you the complete truth and is probably *lying to your face* because with percentile value it can be observed that not even 500 DNS was resolved within that time frame.

With percentile value we can intuitively characterize this system:

1. Around 900 DNS resolution will happen below `0.59s`
2. Around 90 DNS resolution will take between `0.59s` to `0.66s`
3. Around 10 DNS resolution will go beyond the `0.66s` .

> It's Important to understand percentiles to understand latency because you have to consider the entire distribution. Simply looking at the Mean(Avg), Median or even 90th or 99th percentile is not sufficient.

Where to go from here? Gil Tene has a great talk on "How NOT to Measure

Latency . Be aware of the tools that you use for monitoring and benchmarking and the data they report. Many of the tools we use do *not* do a good job of capturing and representing this data.

. . .

Learned something? Clap your 👏 to help others find this article. Or you can say thank you by becoming a patreon

Go Golang Performance Benchmark Programming

268 claps

WRITTEN BY

**Ankur Anand**

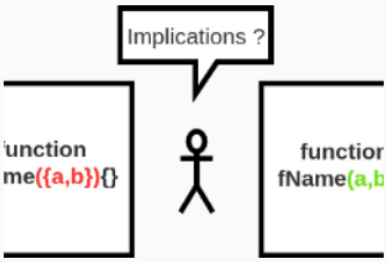
Learning to Learn in Feynman Technique. - https://ankuranand.com

Follow

See responses (3)

## More From Medium

More from Ankur Anand




### ES6's Function Destructuring Assignment Is Not A Free Lunch.

Ankur Anand in... Jun 29, 2018 · 6 min read 1.5K

Also tagged Golang

### Easy Guide to Unit Testing in Golang

Arindam Roy Oct 3 · 4 min read ★ 1

Related reads

### Working with Compressed Tar Files in Go

Vladimir Vivien in... Jul 19 · 8 min read ★ 105

**Discover Medium**

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

**Make Medium yours**

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

**Become a member**

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

**Medium**

About Help Legal