Rust > Go > Python ...to parse millions of dates in CSV files

15 May 2018 0 comments Python

Peterbe.com Menu▾

It all started so innocently. The task at hand was to download an inventory of every single file ever uploaded to a public AWS S3 bucket. The way that works is that you download the root `manifest.json`. It references a boat load of `.csv.gz` files. So to go through every single file uploaded to the bucket, you read the `manifest.json`, the download each and every `.csv.gz` file. Now you can parse these and do something with each row. An example row in one of the CSV files looks like this:

```
"net-mozaws-prod-delivery-firefox","pub/firefox/nightly/latest-mozilla-central-l10n/linux-i68(
```

In the Mozilla Buildhub what we do is we periodically do this, in Python (with `asyncio`), to spot if there are any files in the S3 bucket have potentially missed to record in an different database.
But ouf the 150 or so `.csv.gz` files, most of the files are getting old and in this particular application we can be certain it's unlikely to be relevant and can be ignored. To come to that conclusion you parse each `.csv.gz` file, parse each row of the CSV, extract the `last_modified` value (e.g. `2017-09-21T13:08:25.000Z`) into a `datetime.datetime` instance. Now you can quickly decide if it's too old or recent enough to go through the other various checks.

So, the task is to parse 150 `.csv.gz` files totalling about 2.5GB with roughly 75 million rows. Basically parsing the date strings into `datetime.datetime` objects 75 million times.

Python

What this script does is it opens, synchronously, each and every `.csv.gz` file, parses each records date and compares it to a constant ("Is this record older than 6 months or not?")

This is an extraction of a bigger system to just look at the performance of parsing all those `.csv.gz` files to figure out *how many* are old and how many are within 6 months. Code looks like this:

```
import datetime
import gzip
import csv
from glob import glob

cutoff = datetime.datetime.now() - datetime.timedelta(days=6 * 30)

def count(fn):
```

```
    count = total = 0
    with gzip.open(fn, 'rt') as f:
        reader = csv.reader(f)
        for line in reader:
            lastmodified = datetime.datetime.strptime(
                line[3],
                '%Y-%m-%dT%H:%M:%S.%fZ'
            )
            if lastmodified > cutoff:
                count += 1
            total += 1

    return total, count


def run():
    total = recent = 0
    for fn in glob('*.csv.gz'):
        if len(fn) == 39:  # filter out other junk files that don't fit the pattern
            print(fn)
            t, c = count(fn)
            total += t
            recent += c

    print(total)
    print(recent)
    print('{:.1f}%'.format(100 * recent / total))


run()
```

Code as a gist here.

Only problem. This is horribly slow.

To reproduce this, I took a sample of 38 of these `.csv.gz` files and ran the above code with CPython 3.6.5. **It took 3m44s** on my 2017 MacBook Pro.

Let's try a couple low-hanging fruit ideas:

- PyPy 5.10.1 (based on 3.5.1): **2m30s**
- Using asyncio on Python 3.6.5: **3m37s**
- Using a thread pool on Python 3.6.5: **7m05s**
- Using a process pool on Python 3.6.5: **1m5s**

Hmm... Clearly this is CPU bound and using multiple processes is the ticket. But what's really holding us back is the date parsing. From the "Fastest Python datetime parser" benchmark the trick is to use `ciso8601`. Alright, let's try that. Diff:

```
6c6,10
< cutoff = datetime.datetime.now() - datetime.timedelta(days=6 * 30)
---
> import ciso8601
>
> cutoff = datetime.datetime.utcnow().replace(
>     tzinfo=datetime.timezone.utc
```

```
> ) - datetime.timedelta(days=6 * 30)
14,17c18
<             lastmodified = datetime.datetime.strptime(
<                 line[3],
<                 '%Y-%m-%dT%H:%M:%S.%fZ'
<             )
---
>             lastmodified = ciso8601.parse_datetime(line[3])
```

Version with ciso8601 here.

- Plain Python 3.6.5 with `ciso8601`: **1m08s**
- Using a process pool *and* `ciso8601`: **18.4s**

So what originally took 3 and a half minutes now takes 18 seconds. I suspect that's about as good as it gets with Python.
But it's not too shabby. Parsing 12,980,990 date strings in 18 seconds. Not bad.

Go

My Go is getting rusty but it's quite easy to write one of these so I couldn't resist the temptation:

```go
package main

import (
    "compress/gzip"
    "encoding/csv"
    "fmt"
    "log"
    "os"
    "path/filepath"
    "strconv"
    "time"
)

func count(fn string, index int) (int, int) {
    fmt.Printf("%d %v\n", index, fn)
    f, err := os.Open(fn)
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()
    gr, err := gzip.NewReader(f)
    if err != nil {
        log.Fatal(err)
    }
    defer gr.Close()

    cr := csv.NewReader(gr)
    rec, err := cr.ReadAll()
    if err != nil {
        log.Fatal(err)
    }
    var count = 0
    var total = 0
    layout := "2006-01-02T15:04:05.000Z"
```

```
    minimum, err := time.Parse(layout, "2017-11-02T00:00:00.000Z")
    if err != nil {
        log.Fatal(err)
    }

    for _, v := range rec {
        last_modified := v[3]

        t, err := time.Parse(layout, last_modified)
        if err != nil {
            log.Fatal(err)
        }
        if t.After(minimum) {
            count += 1
        }
        total += 1
    }
    return total, count
}

func FloatToString(input_num float64) string {
    // to convert a float number to a string
    return strconv.FormatFloat(input_num, 'f', 2, 64)
}

func main() {
    var pattern = "*.csv.gz"

    files, err := filepath.Glob(pattern)
    if err != nil {
        panic(err)
    }
    total := int(0)
    recent := int(0)
    for i, fn := range files {
        if len(fn) == 39 {
            // fmt.Println(fn)
            c, t := count(fn, i)
            total += t
            recent += c
        }
    }
    fmt.Println(total)
    fmt.Println(recent)
    ratio := float64(recent) / float64(total)
    fmt.Println(FloatToString(100.0 * ratio))
}
```

Code as as gist here.

Using `go1.10.1` I run `go make main.go` and then `time ./main`. This takes just **20s** which is about the time it took the Python version that uses a process pool and `ciso8601`.

I showed this to my colleague @mostlygeek who saw my scripts and did the Go version more properly with its own repo.
At first pass (`go build filter.go` and `time ./filter`) this one clocks in at **19s** just like my naive

initial hack. However if you run this as `time GOPAR=2 ./filter` it will use 8 workers (my MacBook Pro as 8 CPUs) and now it only takes: **5.3s**.

By the way, check out @mostlygeek's `download.sh` if you want to generate and download yourself a bunch of these `.csv.gz` files.

Rust

First @mythmon stepped up and wrote two versions. One single-threaded and one using `rayon` which will use all CPUs you have.

The version using `rayon` looks like this (single-threaded version here):

```
extern crate csv;
extern crate flate2;
#[macro_use]
extern crate serde_derive;
extern crate chrono;
extern crate rayon;

use std::env;
use std::fs::File;
use std::io;
use std::iter::Sum;

use chrono::{DateTime, Utc, Duration};
use flate2::read::GzDecoder;
use rayon::prelude::*;

#[derive(Debug, Deserialize)]
struct Row {
    bucket: String,
    key: String,
    size: usize,
    last_modified_date: DateTime<Utc>,
    etag: String,
}

struct Stats {
    total: usize,
    recent: usize,
}

impl Sum for Stats {
    fn sum<I: Iterator<Item=Self>>(iter: I) -> Self {
        let mut acc = Stats { total: 0, recent: 0 };
        for stat in  iter {
            acc.total += stat.total;
            acc.recent += stat.recent;
        }
        acc
    }
}

fn main() {
    let cutoff = Utc::now() - Duration::days(180);
```

```
    let filenames: Vec<String> = env::args().skip(1).collect();

    let stats: Stats = filenames.par_iter()
        .map(|filename| count(&filename, cutoff).expect(&format!("Couldn't read {}", &filename
        .sum();

    let percent = 100.0 * stats.recent as f32 / stats.total as f32;
    println!("{} / {} = {:.2}%", stats.recent, stats.total, percent);
}

fn count(path: &str, cutoff: DateTime<Utc>) -> Result<Stats, io::Error> {
    let mut input_file = File::open(&path)?;
    let decoder = GzDecoder::new(&mut input_file)?;
    let mut reader = csv::ReaderBuilder::new()
        .has_headers(false)
        .from_reader(decoder);

    let mut total = 0;
    let recent = reader.deserialize::<Row>()
        .flat_map(|row| row)  // Unwrap Somes, and skip Nones
        .inspect(|_| total += 1)
        .filter(|row| row.last_modified_date > cutoff)
        .count();

    Ok(Stats { total, recent })
}
```

I installed it like this (I have `rustc 1.26` installed):

```
▸ cargo build --release --bin single_threaded
▸ time ./target/release/single_threaded ../*.csv.gz
```

That finishes in **22s**.

Now let's try the one that uses all CPUs in parallel:

```
▸ cargo build --release --bin rayon
▸ time ./target/release/rayon ../*.csv.gz
```

That took **5.6s**

That's rougly 3 times faster than the best Python version.

When chatting with my teammates about this, I "nerd-sniped" in another colleague, Ted Mielczarek who forked Mike's Rust version.

Compile and running these two I get **17.4s** for the single-threaded version and **2.5s** for the `rayon` one.

In conclusion

1. Simplest Python version: **3m44s**
2. Using PyPy (for Python 3.5): **2m30s**
3. Using `asyncio`: **3m37s**
4. Using `concurrent.futures.ThreadPoolExecutor`: **7m05s**

5. Using `concurrent.futures.ProcessPoolExecutor`: **1m5s**
6. Using `ciso8601` to parse the dates: **1m08s**
7. Using `ciso8601` and `concurrent.futures.ProcessPoolExecutor`: **18.4s**
8. Novice Go version: **20s**
9. Go version with parallel workers: **5.3s**
10. Single-threaded Rust version: **22s**
11. Parallel workers in Rust: **5.6s**
12. (Ted's) Single-threaded Rust version: **17.4s**
13. (Ted's) Parallel workers in Rust: **2.5s**

Most interesting is that **this is not surprising**. Of course it gets faster if you use more CPUs in parallel. And of course a C binary to do a critical piece in Python will speed things up. What I'm personally quite attracted to is how easy it was to replace the date parsing with `ciso8601` in Python and get a more-than-double performance boost with very little work.

Yes, I'm perfectly aware that these are not scientific conditions and the list of disclaimers is long and boring. However, it was fun! It's fun to compare and constrast solutions like this. Don't you think?

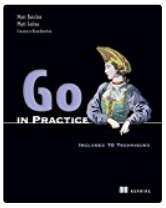
Follow @peterbe on Twitter

Relevant Amazon.com Products

The Go Bestiary: Bugs, gotchas and guidelines for go programmers

$12.99 By **Kenneth Grant**

Publication Date: 2017-10-09
Binding: Paperback
Number of Pages: 121



Go in Practice: Includes 70 Techniques

$44.99 By **Matt Butcher, Matt Farina**

Publication Date: 2016-10-01
Binding: Paperback
Number of Pages: 312

Amazon Web Services in Action

$49.99 By **Andreas Wittig, Michael Wittig**

Publication Date: 2015-10-17
Binding: Paperback
Number of Pages: 424

Refresh products

Comments

What do you think?

Your full name

Your email (never shown, never shared)

Your email will never ever be published

Preview first | Post comment

Related posts

Previous:

- Webpack Bundle Analyzer for create-react-app 14 May 2018

Related by Keyword:

- Fastest Python datetime parser 02 May 2018
- Unzip benchmark on AWS EC2 c3.large vs c4.large 29 November 2017
- Fastest way to find out if a file exists in S3 (with boto3) 16 June 2017
- Autocompeter is Dead. Long live Autocompeter! 09 January 2017
- How to slice a rune in Go 16 March 2015

Related by Text:

- jQuery and Highslide JS 08 January 2008
- I'm back! Peterbe.com has been renewed 05 June 2005
- Anti-McCain propaganda videos 12 August 2008
- I'm Prolog 01 May 2007
- Ever wondered how much $87 Billion is? 04 November 2003

Check out my latest side project Song Search