Crunching Gigabytes Locally

[Dima Korolev]

DataFest.by, May 19 2018

Data crunching TL;DR

- **The #1 productivity booster in data science is fast iterations.**
- Use the shell for under 10MB, use Hadoop for over 100GB.
- In between there's room for creativity, and it is what this talk is about.

About me

- Machine learning, data science, and feature engineering for over 10 years.
- Competitive programming, algorithms, infra, and C/C++ for over 20 years.

About the measurements

- Timings measured on my laptop, which is:
    - ThinkPad X1 Carbon 4th gen.
    - On AC power and on battery, no significant difference.

    - Intel Core i7-7500U, 3.5 GHz, 4MB cache.
    - 16GB LPDDR3 1866.
    - 256 GB SSD, OPAL2.0 PCIe-NVMe.
- Also validated on a Hetzner box, with no surprises.
- All code snippets are on GitHub.

About the dataset

- 2016 Green Taxi Trip Data (hosted by Google for BigQuery, link).
- 2.1GB, 16.3M rides.

- Generally speaking, dataset integrity, as well as licensing, are real things.
- If you want the results reproducible, keep immutable copies of the datasets.

Data cleanup

- You would imagine the Google-hosted dataset is clean enough. Well, no.
- Two major sources of issues:
    - Timestamps.
    - Ultra-short and/or ultra-fast rides.

Issues with timestamps

- Ambiguity. Real example: `07/11/2016 01:05:09 AM`

```
$ TZ=America/New_York date -d @1478408709
Sun Nov 6 01:05:09 EDT 2016
$ TZ=America/New_York date -d @1478412309
Sun Nov 6 01:05:09 EST 2016
```

- Straight out invalid dates. Real example: `03/13/2016 02:58:40 AM`

```
$ TZ=America/New_York date -d '03/13/2016 02:58:40 AM'
date: invalid date ‘03/13/2016 02:58:40 AM’
$ TZ=America/New_York date -d @$(($(TZ=America/New_York date -d '03/13/2016 01:59:59' +%s) + 1))
Sun= Mar 13 03:00:00 EDT 2016
```

- Trip into the past.

  Example: `07/17/2016 03:16:35 PM,07/16/2016 08:30:10 AM`

Lesson learned

- Human time is hard to deal with for computers.
    - Unix epoch timestamps are the safest way.
    - If “derived” data, such as day of week, is needed, keep in mind it may change!
    - Even a dataset published by Google is not immune to this.
- **Solution**: Re-annotate timestamps in the input CSV.
    - For this one-time task, use Python to keep things simple.

Crunching tasks

- Follow the first two examples from the dataset page on Google cloud:
  - How many trips did Yellow taxis take each month in 2015?
  - What was the average speed of Yellow taxi trips in 2015?

Crunching

- How many trips did Yellow taxis take each month in 2015?

Crunching

- How many trips did Yellow taxis take each month in 2015?

```
$ ./step1_rides_by_months_python.py  | tee >(md5sum)

1445285 01/2016
1510722 02/2016
1576393 03/2016
1543925 04/2016
1536979 05/2016
1404726 06/2016
1332510 07/2016
1247675 08/2016
1162373 09/2016
1252572 10/2016
1148214 11/2016
1224158 12/2016

bd5b6d7f389e64aab50c56fed61eaded  -
```

Python timing

- The Python script runs for 24 seconds.
- Take a moment to think of this number.

Is 24 seconds fast or slow?

History

- Python is a fine scripting language, but what did people use before it?

History

- Python is a fine scripting language, but what did people use before it?

- Aho-Weinberger-Kernighan.
    - Or awk.
    - From the 70s.
    - One can say it has a built-in map-reduce.
    - And, at 16 seconds running time, it is actually faster.

- Shell classics: grep, cut, sort, uniq.
    - This approach is even faster, completing in 12 seconds total.
    - 2x the Python speed.
    - Truly, “without any programming”.

Shell

- What we should really be asking ourselves is:
    - How long does it take to look for something in the input data, such as that bad timestamp?
    - Wait, and how long does it take to count the number of lines in the file?

Shell

- What we should really be asking ourselves is:
    - How long does it take to look for something in the input data, such as that bad timestamp?
    - Wait, and how long does it take to count the number of lines in the file?

- Both are valid questions, both require full file scan, both are doable in shell. Timings:
    - `grep ...` : ~ 0.9s (for what's only found a few times in the input)
    - `wc -l ...` : < 0.5s

- In other words, **`grep`** and **`wc`** operate at over 1GB/s throughput.

All timings

Approach	Duration
python	~24s (<100MB/s)
“very naive” c++	~18s (~27s on Hetzner)
awk	~17s
cut, sort, uniq	~14s
“naive” c++, string views instead of strings	~13s
read full file, malloc() + { open()+read() / fopen()+fread() }	~3s / ~2.4s
mmap() full file, C	1.4s (>1GB/s)
“naive” C, fopen() + fgets(), analyze digits at fixed string offset	1.0s

Python, at ~24s

```
histogram = {}

with open('../cooked.csv') as f:
  for line in f:
    date = line.split(',')[1]
    yyyy_mm_dd = date.split('-')
    yyyy = yyyy_mm_dd[0]
    mm = yyyy_mm_dd[1]
    histogram_key = mm + '/' + yyyy
    if not histogram_key in histogram:
      histogram[histogram_key] = 1
    else:
      histogram[histogram_key] += 1

for mm_yyyy, count in sorted(histogram.iteritems()):
  print "%s %s" % (count, mm_yyyy)
```

awk, at ~17s

```
# Keep in mind, this code is from the 70s!

BEGIN { FS="," }

{
  split($2, date_time, " ");
  split(date_time[1], yyyy_mm_dd, "-");
  ++histogram[yyyy_mm_dd[2] "/" yyyy_mm_dd[1]];
}

END { for (i in histogram) printf "%d %s\n", histogram[i], i; }
```

Bell Laboratories

The winner, plain C, consistently at 1.0s

```
#include <stdio.h>

int MM[12];

int main(int argc, char** argv) {
  char line[10000];  // Unsafe, of course.

  FILE* f = fopen(argc >= 2 ? argv[1] : "../cooked.csv", "r");
  while (fgets(line, sizeof(line), f)) {
    ++MM[(line[7] - '0') * 10 + (line[8] - '0') - 1];  // Unsafe, of course.
  }
  fclose(f);

  for (int m = 0; m < 12; ++m) {
    printf("%d %02d/%d\n", MM[m], m + 1, 2016);
  }
}
```

Lessons learned

- Unix rocks.
  - When it comes to textual throughput, shell-grade tools are the fastest.
  - Unsophisticated old-school [unsafe] plain C is not far behind.

Lessons learned

- Unix rocks.
    - When it comes to textual throughput, shell-grade tools are the fastest.
    - Unsophisticated old-school [unsafe] plain C is not far behind.
- Scripting is ~20x slower.
    - Even splitting the row into `vector<string>` increases the running time from 13s to 18s.
    - In other words, just creating temporary strings each row takes ~33% of CPU cycles.
    - Or, to put it another way, meanwhile, the “dumb C” solution could have run five times.
    - Here goes a rant about using a “high-level language” such as Java for storage-heavy tasks.
- Python can be read by more people, but it’s a pain for the CPUs.

Going native

- The fastest code snippets are less readable, as CSV is not natural for C.
- To achieve higher speeds, machine-friendly data formats should be used.

```
void Run(const struct Ride* data, size_t n) {
  for (size_t i = 0; i < n; ++i) {
    ++MM[data[i].pickup.month - 1]
  }
  for (int m = 0; m < 12; ++m) {
    printf("%d %02d/2016\n", MM[m], m + 1);
  }
}
```

- Runs in under 0.2s, including `mmap()`-ing the entire 1.5GB binary input file.

Crunching, exercise two

- What was the average speed of Yellow taxi trips in 2015?

Crunching, exercise two

- What was the average speed of Yellow taxi trips in 2015?

```
$ gcc -O3 step3_sanitycheck_avg_speed_by_hour.c && time ./a.out | tee >(md5sum)
Total rides considered: 15592085 (95.2%)

00    15.37
01    15.85
02    16.10
03    16.67
04    17.81
05    19.72
06    19.47
07    14.82
08    12.99
09    13.55
10    13.49
11    13.51
12    13.22
13    13.18
14    12.40
15    12.02
16    11.61
17    11.29
18    11.74
19    12.70
20    13.51
21    14.00
22    14.46
23    14.93

ddba3b195052774ddc8e5f4d4ff26bec    -
```

Analysis

- Rides filtering logic blindly copied from the BigQuery example.
    - And the result is reasonable compared to the BigQuery example.
- The computation uses the same constraints as the original code snippet.
    - Including “**cost per mile is between $2 and $10**”.
    - Which is floating point.

Non-integers

- Comparison and rounding are always a path to trouble.
- Especially if we want results that are reproducible in various ways.
- Generally, if floating point can be avoided, avoid it.

Floating point surprises: three (!) different results

```c
#include "schema.h"
void Run(const struct Ride* data, int n) {
  int total_considered = 0;
#if 0
  const int two = 2;
  const int ten = 10;  // `float` and `double` give different results here. -- D.K.
#endif
  for (int i = 0; i < n; ++i) {
    const int trip_duration_seconds = data[i].dropoff.epoch - data[i].pickup.epoch;
    if (data[i].trip_distance > 0 && trip_duration_seconds > 0) {
#if 1
      const double cost_per_mile = data[i].fare_amount / data[i].trip_distance;
      if (cost_per_mile >= 2.0 && cost_per_mile <= 10.0) {
#else
      if (data[i].fare_amount >= two * data[i].trip_distance &&
          data[i].fare_amount <= ten * data[i].trip_distance) {
#endif
        ++total_considered;
        // UpdateAverageSpeed(data[i].pickup.hour,
        //                    data[i].trip_distance / trip_duration_seconds);
      }
    }
  }
}
```

Crunching, do over

- What was the average speed of Yellow taxi trips in 2015?

```
$ gcc -O3 step3_sanitycheck_avg_speed_by_dow.c && time ./a.out | tee >(md5sum)
Total rides considered: 15592077 (95.2%)

00    15.37
01    15.85
02    16.10
03    16.67
04    17.81
05    19.72
06    19.47
07    14.82
08    12.99
09    13.55
10    13.49
11    13.51
12    13.22
13    13.18
14    12.40
15    12.02
16    11.61
17    11.29
18    11.74
19    12.70
20    13.51
21    14.00
22    14.46
23    14.93

ddba3b195052774ddc8e5f4d4ff26bec    -
```

Dealing with non-integers

- In the yellow cabs case:
    - All times are in seconds, which are integers.
    - All amounts are in dollars, which are integer cents.
    - Distance can be rounded, and let's round to 1/100th of a mile to simplify division.
- **Solution**: Update the schema to use integers, and re-cook the binary file.
    - The file can even get smaller thanks to it.

But wait, there's more!

```
for (size_t i = 0; i < n; ++i) {
  const int trip_duration_s = data[i].dropoff.epoch - data[i].pickup.epoch;
  if (data[i].trip_distance_times_100 > 0 && trip_duration_s > 0) {
    if (data[i].fare_amount_cents >= 2 * data[i].trip_distance_times_100 &&
        data[i].fare_amount_cents <= 10 * data[i].trip_distance_times_100) {
      ++total_considered;
      UpdateAverageSpeed(&per_hour_counters[data[i].pickup.hour],
                         data[i].trip_distance_times_100,
                         trip_duration_seconds);
    }
  }
}

// The true average cab speed is total miles traveled divided by total hours traveled.
// The average cab speed and the average of average ride speeds are not the same!
// The original code computed something else, and rewriting the code helped locate it.
```

Recap

- A.k.a. “so what?”:
  - ***We know Python is slow, but developers understand it.***
    Their time is still more valuable than the time the machine spends running their code.
    Thanks, we’re not considering sed or awk.
  - ***We know mmap()-ing a file is fast, thanks, Captain Obvious.***
    We don’t plan on doing it though, sorry.
  - ***We know binary formats are fast.***
    Would be nice had they been some Protobuf or Thrift or Cap’n Proto or Avro.
    For pure binary, we, of course, have no plans to switch to C/C++.

The real thing

- Move code to data, not data to code.
    - In this particular example we `mmap()`-ed the file.
    - Which is OK for a few GBs, but not fun for a file that barely fits in memory.
    - Still keeping a weak laptop in mind.

The real thing

- Move code to data, not data to code.
  - In this particular example we `mmap()`-ed the file.
  - Which is OK for a few GBs, but not fun for a file that barely fits in memory.
  - Still keeping a weak laptop in mind.
- We need another superpower to complement `mmap()`, and it’s `dlopen()`.
  - Load a library dynamically, and natively call user-defined functions from it.

The real thing

- Move code to data, not data to code.
  - In this particular example we `mmap()`-ed the file.
  - Which is OK for a few GBs, but not fun for a file that barely fits in memory.
  - Still keeping a weak laptop in mind.
- We need another superpower to complement `mmap()`, and it’s `dlopen()`.
  - Load a library dynamically, and natively call user-defined functions from it.
- The production design then is:
  - The binary, that has all the data loaded up, runs as a daemon.
  - Introduce a lightweight interface to run “custom queries” against it.

Worth a thousand words!

- It's in the browser.

Script

```
FUNCTION(rides, output) {
  std::vector<int> MM(12);

  for (const IntegerRide& ride : rides) {
    ++MM[ride.pickup.month - 1];
  }

  for (int m = 0; m < 12; ++m) {
    output << Printf("%d %02d/2016\n", MM[m], m + 1);
  }
}
```

Run

Endpoint

[Link]

Result

```
1445285 01/2016
1510722 02/2016
1576393 03/2016
1543925 04/2016
1536979 05/2016
1404726 06/2016
1332510 07/2016
1247675 08/2016
1162373 09/2016
1252572 10/2016
1148214 11/2016
1224158 12/2016
```

How does it work

- An obligatory disclaimer: I am not a frontend developer.
  - The browser UX is just a demo, we never used it, really.
  - Obviously, syntax highlighting, completion, and on-the-fly compilation can be added.
- The user code is wrapped into a boilerplate for compilation.
  - The boilerplate it service-defined, hence `Printf()` and the `FUNCTION()` macro above.
  - The code is then SHA256-hashed, and each unique hash is only compiled once.
  - The user code can accept parameters, from the URL/querystring, or from HTTP headers.
- The compilation is run from a dedicated temporary directory.
  - With a symlink to the framework, to enable accessing extra headers as necessary.
  - With a dummy, empty, header file the user could `#include` locally to "fix" completion.
- A `#line` directive is injected between the boilerplate and the user code.
  - So that the IDE can jump to exact locations of the error should one occur in user code.

Also, completion! (No, this pink is not my color scheme. I code on a dark background.)

```
#include "current.h"

FUNCTION(rides, output) {
  std::vector<int> MM(12);

  for (auto const& ride : rides) {
    if (ride.pickup.

  for (int m = 0; m
    output << Print                    m], m + 1);
  }
}
~
~
~
~
~
```

```
epoch         m int
dow           m uint8_t
month         m uint8_t
Timestamp::   s
year          m uint8_t
second        m uint8_t
dst           m uint8_t
day           m uint8_t
hour          m uint8_t
minute        m uint8_t
```

How does it work cont'd

- To function flawlessly, best is to put three more files alongside the "script".
  - The dummy header file.
    - To define the same symbols as the boilerplate, such as `FUNCTION()`.
    - This enables proper syntax highlighting and completion.
    - In fact, in the demo it's a symlink to the boilerplate.
  - A symlink to the very framework.
    - So that "library" functions such as `Printf()` or `JSON()` are easy to access.
  - A `Makefile`.
    - To push the code to the service using `curl`.
    - To "build" the code and "run" it, if built successfully.
    - To set the HTTP header with the original source name, for the IDE to jump to errors.

Bonus: JSON support and auto-set Content-Type

```
$ cat snippet.txt

ENDPOINT(data, params) {
  std::map<int, int> histogram;
  for (const IntegerRide& ride : data) {
    ++histogram[ride.pickup.month - 1];
  }
  return histogram;
}
```

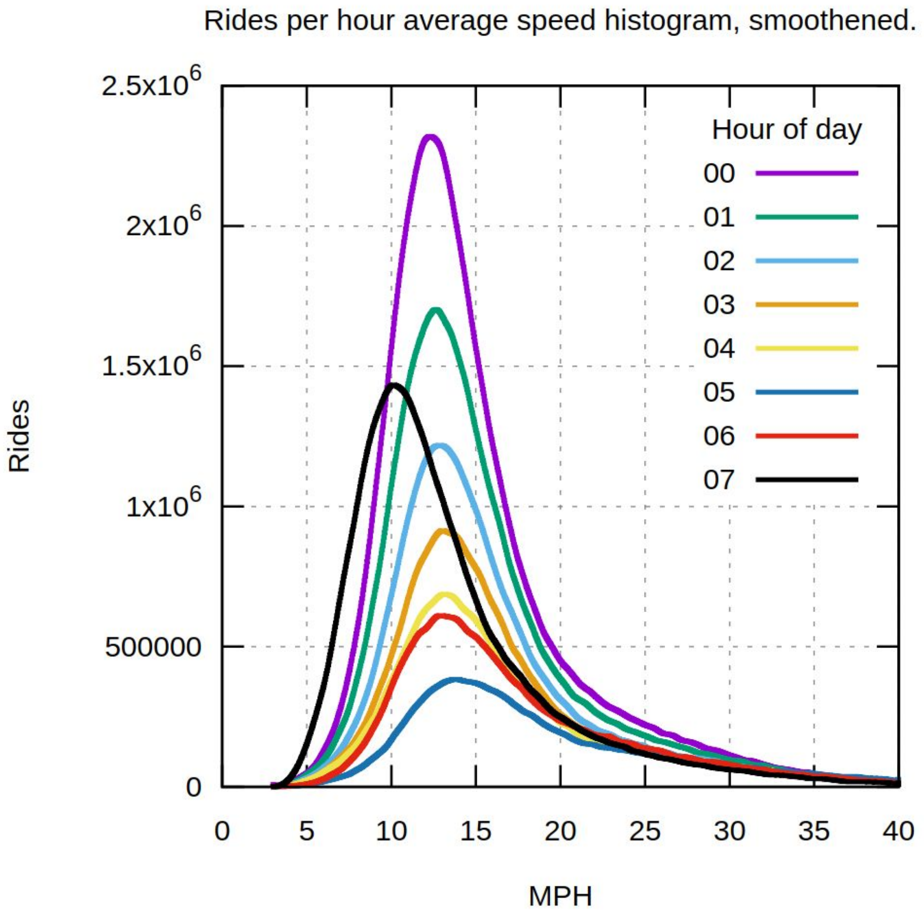
```
$ curl --data-binary @snippet.txt http://localhost:3000/full  # Takes a bit longer, as JSON support is some wild metaprogramming. -- D.K.
Compiled: @54108e84c0


$ curl -i http://localhost:3000/full/54108e84c0
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Connection: close
Access-Control-Allow-Origin: *
Content-Length: 148

[[0,1445285],[1,1510722],[2,1576393],[3,1543925],[4,1536979],[5,1404726],[6,1332510],[7,1247675],[8,1162373],[9,1252572],[10,1148214],[11,1224158]]
```

Bonus II: Visualizations

- `gnuplot`
  - Using the existing binding.
- Compile once.
  - Use URL parameters later.
  - `/e3fcdadf67?`**`from=0&to=8`**
- Ready to use endpoint.
  - For monitoring and reporting.



Pros, aside from performance and zero configuration

- Brings together the best of both worlds:
  - For the old-schoolers:
    - Code completion.
    - Build and run with one command, `Makefile`-friendly.
    - Every functional code snippet immediately becomes a `curl`-able endpoint.
  - For browser-cherishing folks:
    - Zero entry barrier.
    - Even this trivial demo is sharing-enabled.
    - User-defined logic can accept URL parameters.

- If the backend is in C/C++, the above is a prototyping framework.
  - Production-ready for internal alerts, monitoring, and reporting.
  - A playground for real-time ML features.
  - The very same code then goes through the review, and makes it as a prod API endpoint.

Cons

- Zero security.
    - Bad memory access would SEGFAULT the serving daemon.
    - I’m not sharing this very URL, as one could run `rm -rf /` on my machine.
        - Can run from under an unprivileged user.
        - Not really an issue with modern virtualization and containers.
        - Even less of an issue as data scientists tend to self-organize into friendly sub-teams.
- You have to write C++.
    - I made it as “JavaScript” as possible for you, but, face it, it is C++ behind the scenes.
    - Although one can argue whether it’s a bad thing.

Cons

- Zero security.
  - Bad memory access would SEGFAULT the serving daemon.
  - I’m not sharing this very URL, as one could run `rm -rf /` on my machine.
    - Can run from under an unprivileged user.
    - Not really an issue with modern virtualization and containers.
    - Even less of an issue as data scientists tend to self-organize into friendly sub-teams.
- You have to write C++.

Conclusions and Q&A

- “Big” data is in the eye of the *shell holder*.
  - For single- and even double-digit gigabytes, most crunching still fits a laptop.
- Streaming data in real time is also no problem whatsoever.
  - As long as the daemon binary is listening to some pub-sub bus, which most of us have.
- Don’t rush to use fancy trending toolkits until they are needed.
  - I personally have stopped people from using tools for the sake of using them.
- First principles and C++ rock.
  - And C++ is a simple and safe enough language by 2018.
    If you speak JS, you can do modern C++, and you will enjoy it.

Thanks

Q&A