Scaling Beyond Python

Lecture 13

Sam Madden

madden@csail.mit.edu

Project Signups
Overview

• High level tools like Python are fine for many problems but may be too slow, especially as you scale up problem size
• Typically requires optimization and redesign
• Some strategies
  – Buy more hardware
  – Use a different runtime
  – Improve implementation
• Today we will focus on some simple data-oriented improvements; parallelism and algorithmic tricks in later lectures
General Approach

- Find the bottleneck
  - Most programs have several stages
  - Some may be I/O based, some CPU based
- Improve performance of bottleneck
- Iterate
  - Did the bottleneck change?
How Slow is Slow?

- Different applications have different performance demands
- In an online setting, e.g., serving a web page, 100ms may be too long
- For an interactive dashboard, 1s may be too long
- For an ML prediction, minutes may be too long
Performance metrics

- Throughput: request/time for many requests
- Latency: time / request for single request

- Often not; e.g., server may have two CPUs
Heavily-loaded systems

- Once system busy, requests queue up
Approaches to finding bottleneck

600 MB file

```python
df = pd.read_csv(PATH, delimiter='|',
                 header=None, names=header)
print df[df['NAME'].str.contains("MADDEN")]
```

• Measure utilization of each resource
  – Easy: CPU is 100% busy, disk is 20% busy
  – Hard: CPU is 50% busy, disk is 50% busy, alternating

• Model performance of your approach
  – What performance do you expect?

• Guess, check, and iterate
  – Don’t prematurely optimize
How Long Do We Expect This To Take?

- I/O vs CPU
- Which will dominate?

• Read From Disk
  • Convert to DataFrame
  • Search For String ‘Madden’
Some Tools

- print statements / timing
- top / system profilers
- code profilers
Python code profile

python3 -m cProfile -o my_program.prof slow_pandas.py
snakeviz my_program.prof
Why Is This So Slow?

- Takes 5+ seconds. Why?
- Seems to be ~4s to load data frame, ~1s to perform search
- For loading, is it I/O? How long should reading from disk take?
Model Your Code

• How long should I/O take?
• How long should data loading take?
• How long should search take?
Important numbers

• Latency:
  – 0.000001 ms: instruction time (1 ns)
  – 0.0001 ms: DRAM load (100 ns)
  – 0.1 ms: LAN network packets (100 usec)
  – 0.1 ms: SSD random I/O
  – 10 ms: random HDD I/O
  – 25 ms: Internet east -> west coast

• Throughput:
  – 100 GB/s: DRAM
  – 10 GB/s: sequential SSD
  – 10 Gbit/s: 10 Gbit LAN (or ~1 GB/s)
  – 500 MB/s: sequential HDD, or random SSD
  – 1 MB/s: random disk I/O
Disk Primer

• Two main types of disks: hard disks (HDD) and solid state disks (SSD)
• Hard disks are rotating platters; cheaper and slower
• Both are block oriented, i.e., they allow reading or writing of blocks (usually a few KB)
• Unlike RAM, which is byte oriented
Solid State Disk (SSD)

• Faster storage technology than disk
  – Flash memory that exports disk interface
  – No moving parts

• Modern Apple 2TB SSD
  – Sequential read: 8 GB/sec
  – Sequential write: 8 MB/sec
  – Random 4KB read: 200K+/-s (>800 GB/s)
    • See next slides
  – Random 4KB write: 40K+/-s (>200 MB/s)
## SSD Random Reads

### 2023 Numbers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SEQ1M</strong></td>
<td>7155.64</td>
<td>7754.68</td>
<td>6824.2</td>
<td>7395.4</td>
</tr>
<tr>
<td><strong>RND4K</strong></td>
<td>3807.51</td>
<td>5797.39</td>
<td>3631.1</td>
<td>5528.8</td>
</tr>
<tr>
<td><strong>QD1</strong></td>
<td>900.96</td>
<td>199.22</td>
<td>219962.0</td>
<td>48638.9</td>
</tr>
<tr>
<td><strong>QD64</strong></td>
<td>46.65</td>
<td>34.90</td>
<td>11388.6</td>
<td>8521.0</td>
</tr>
</tbody>
</table>

**APPLE SSD AP8192Z / Apple M2 Max**

**SEQ1M**: Sequential 1MB Transfers  
**RND4K**: Random 4KB Transfers  
**QD1**: Queue Depth 1 (1 outstanding request at a time)  
**QD64**: Queue Depth 64
SSDs and writes

• Write performance is slower:
  – Flash can erase only large units (e.g., 512 KB)

• Writing a small block:
  1. Read 512 KB
  2. Update 4KB of 512 KB
  3. Write 512 KB

• Controllers try to avoid this using aggressive caching, logging tricks
SSD versus HDD

• HDD: ~$100 for 4 TB
  – $0.025 per GB
• SSD: ~$200 for 2 TB
  – $1.00 per TB

HDD increasingly less common

• Many performance issues still the same:
  – Both SSD and Disks much slower than RAM
  – Avoid random small writes using batching
So How Much of 4s is I/O?

• Disk can read 8 GB/sec, 600 MB should take ~.075s. So disk I/O is not the issue!
  – But loading the data frame takes 4 s???

• What about CPU? 4M records, a few hundred instructions per record
  → ~400M instructions
  – Should take ~.2 seconds on a 2GHz proc
  – Actually takes 5-10x as long!
Fixing a bottleneck

• Get better hardware
• Use better execution environment
• Find better algorithm
• Write better implementation; strategies
  – Indexing
  – Predicate push down
  – Early projection
  – Caching
  – Efficient joins
  – Partitioning & parallelism  -- not today
What Improvement Can We Expect

• Always keep Amdahl’s law in mind

\[ S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}} \]

\( S_{\text{latency}} \) is the overall speedup in all stages of a task
\( s \) is the speedup on a stage of the task that we optimize
\( p \) is the original proportion of time the optimized stage took
If a component takes 50% of time, max speedup is 2x!

Amdahl's law for varying $p$ and $s$
Clicker Question

Which do you think is going to result in best performance:
A. rewrite to use lower-level python instead of pandas, e.g., loops w/ readlines  
B. rewrite in C  
C. rewrite to use a relational database  
D. none of these, pandas is best
Let’s Try It

Pandas version
read_time = 5.60, scan_time = 1.22

Python loops
read_time = 9.51, scan_time = 0.70

Rewrite in C
init_time = 0.00s, read_time = 1.29s, scan_time = 0.11s

Use a Relational DB

```sql
donations=# \copy donations from contrib.txt' delimiter 'I';
COPY 3953288
Time: 8496.195 ms (00:08.496)

donations=# select NAME, EMPLOYER, TRANSACTION_AMT from donations
where NAME ~ 'MADDEN' ;
Time: 738.365 ms
```
Virtual machine (VM) implementation is a loop that reads an instruction, and jumps to the code to execute the instruction.

On modern CPUs this is very inefficient, because it results in many branch misses and poor processor cache locality.
Loops python are very slow
- Because it is an “interpreted” language, each operation takes 100’s of CPU cycles
- Even though a CPU can run ~2B instructions per second, can only do about 5M loop iterations per second

Pandas/numpy vectorized operations generally faster
- Beware apply & co.
Summary

• Parsing data is the bottleneck
  – We will look at solutions next time
• Python is very slow
• Pandas is not bad
  – uses C implementations underneath
• Rewriting in C is painful, can be a big win
  – Can call into C from python if you have a specific algo you want to rewrite
Break
Algorithmic Bottlenecks

• Can we speed up text search?
• What about other kinds of slow algorithms?
Trigrams

1 2 3 4 5 6
• MADDEN -> MAD, ADD, DDE, DEN ...
• Index:

<table>
<thead>
<tr>
<th>Trigram</th>
<th>Start Offsets in Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>2, ...</td>
</tr>
<tr>
<td>DDE</td>
<td>3, ...</td>
</tr>
<tr>
<td>DEN</td>
<td>4, ...</td>
</tr>
<tr>
<td>MAD</td>
<td>1, ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Lookup: MAD -> 1, DEN -> 4
These are consecutive, so found a match
Tree Index

| A .. C | D ... G | G .. P | P ... Z |
Tree Index

ADD: {2, …}
What are advantages of tree organization over sorted list?
Creating Tree Index in Postgres

CREATE INDEX tbl_col_gin_trgm_idx ON donations USING gin (NAME
gin_trgm_ops);

gin is a generic interface for describing
tree indexes in Postgres
Performance

donations=# select NAME, EMPLOYER, TRANSACTION_AMT from donations where NAME ~ 'MADDEN' ;
Time: 497.880 ms

donations=# CREATE INDEX tbl_col_gin_trgm_idx ON donations USING gin (NAME gin_trgm_ops);
Time: 9020.065 ms (00:09.020)

donations=# select NAME, EMPLOYER, TRANSACTION_AMT from donations where NAME ~ 'MADDEN' ;
Time: 5.086 ms

100x speedup (20x faster than C search), high indexing cost
Other Common Algorithmic Bottlenecks

• What’s wrong with this code?

```python
start = time.time()

df = pd.read_csv(PATH, delimiter='|', header=None, names=['header']).loc[0:1000]
df2 = pd.read_csv(PATH2, delimiter='|', header=None, names=['header']).loc[0:1000]

end = time.time()
read_time = end - start

start = time.time()
matches = 0
for i, r in df.iterrows():
    for i2, r2 in df2.iterrows():
        if r.NAME == r2.NAME:
            matches = matches + 1

end = time.time()
join_time = end - start

print(f'got {matches} matches!')
print(f'read_time = {read_time:.2f}, join_time = {join_time:.2f}')
```
Solution 1

```python
matches = 0
names = {}
for i, r in df.iterrows():
    if (r.name in names):
        names[r.name] = names[r.name] + [r]
    else:
        names[r.name] = [r]
for i2, r2 in df2.iterrows():
    if r2.NAME in names:
        matches = matches + len(names[r.name])
```

read_time = 5.65, join_time = 0.02
Solution 2

10x larger

```
df = pd.read_csv(PATH, delimiter='|', header=None, names='header').loc[0:10000]
df2 = pd.read_csv(PATH2, delimiter='|', header=None, names='header').loc[0:10000]

ans = []
ans = df.merge(df2, on='NAME')
```

read_time = 5.56, join_time = 0.02
Full 2M x 2M join

read_time = 5.70,
join_time = 43.24
Let’s Try it In SQL
SQL Advantages

• Many different implementations
• Declarative Control
  – Algorithm
    • Sort merge vs Hash
  – Parallelism
• Memory conscious – able to spill to disk
Summary

• Python is often slow
• Identifying performance bottlenecks is an art
  – Figure out if you have an I/O or CPU problem
  – Estimate expected performance
  – Remember Amdahl’s law!
• Rewriting in low level languages can help
• Using more efficient data accesses can help
• Next time: How to efficiently store & access data on disk