

# Scaling Beyond Python

Lecture 13

Sam Madden

[madden@csail.mit.edu](mailto: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

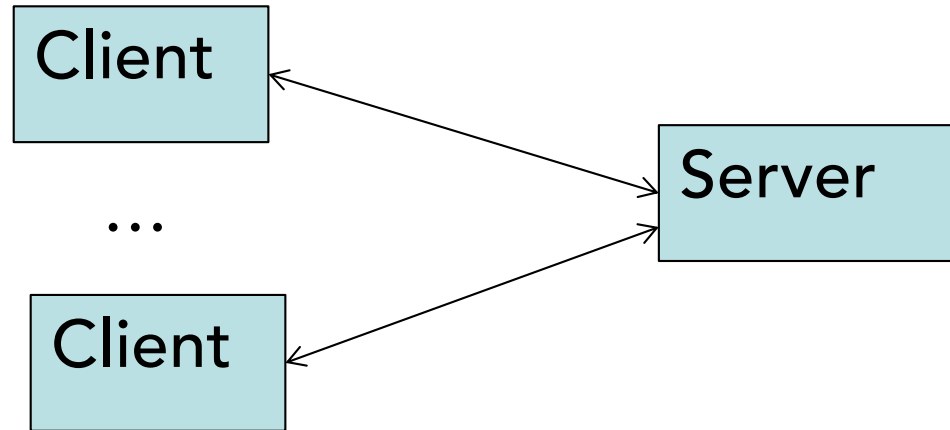
- Find the bottleneck
  - Most programs have several stages
  - Some may be I/O based, some CPU based
- Improve performance of bottleneck
- Iterate
  - Did the bottleneck change?

# 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

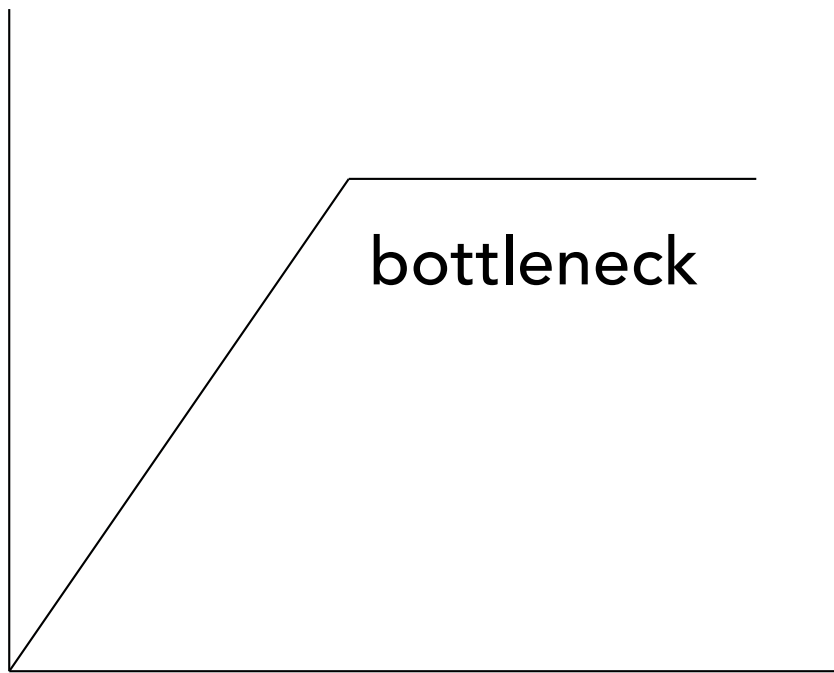
# Performance metrics



- Performance metrics:
  - **Throughput:** request/time for many requests
  - **Latency:** time / request for single request
- Latency = 1/throughput?
  - Often not; e.g., server may have two CPUs

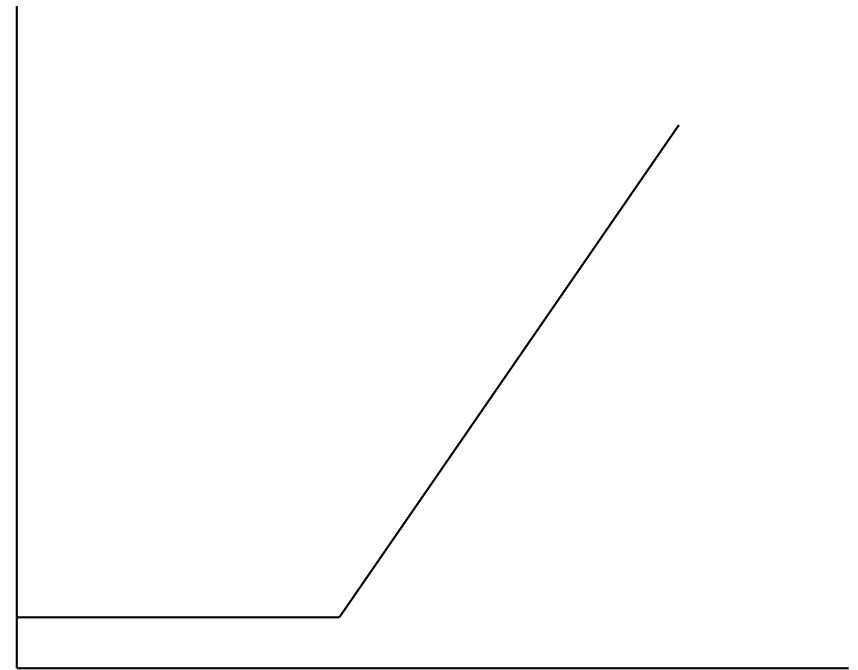
# Heavily-loaded systems

Throughput



requests

Latency




requests

- Once system busy, requests queue up

# Approaches to finding bottleneck

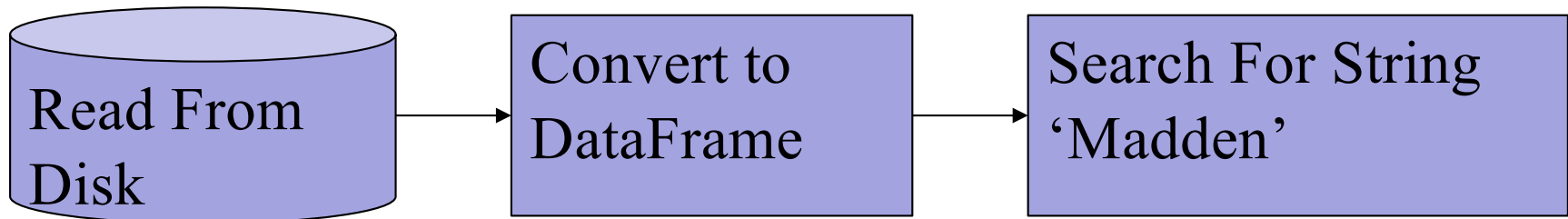
600 MB file

```
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?



# 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

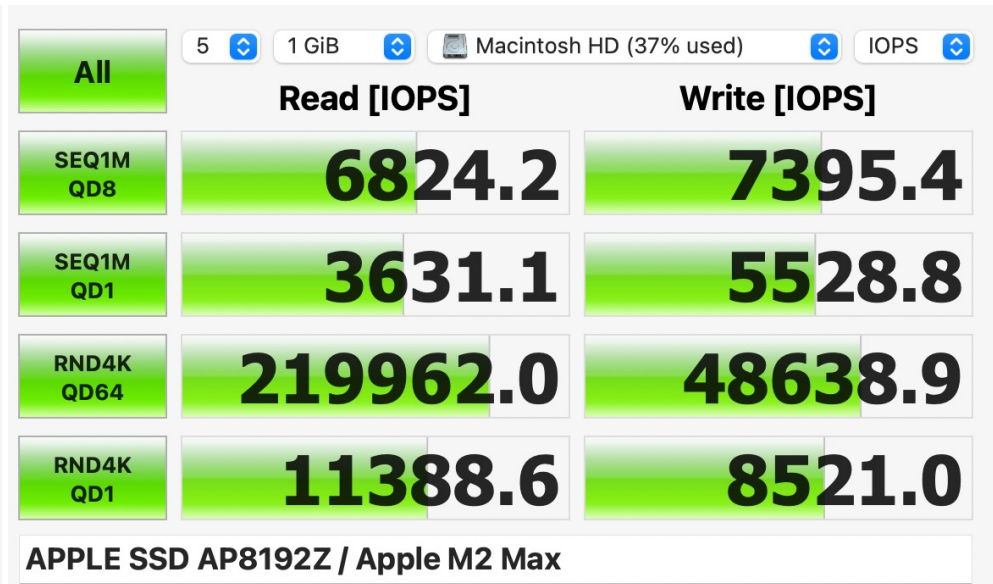
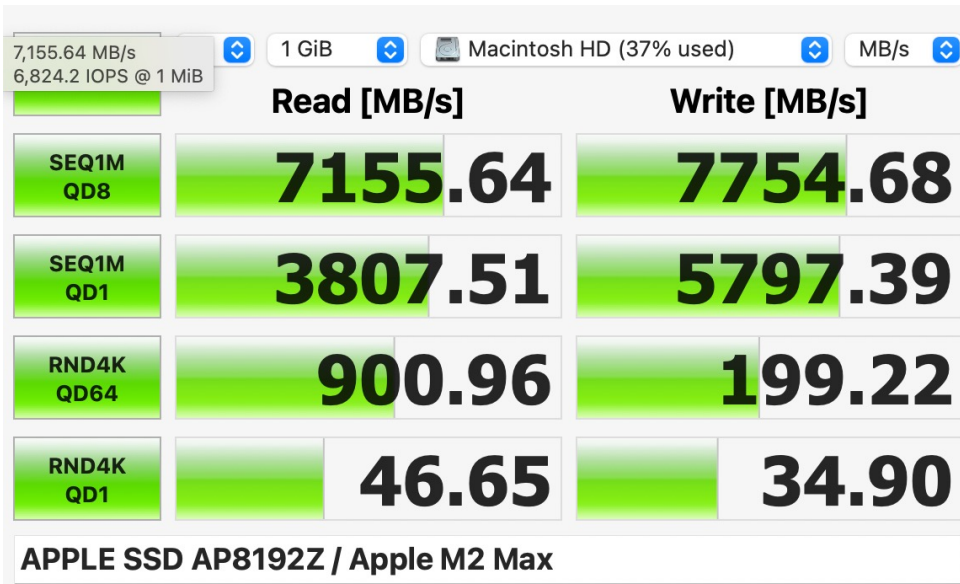
- Two main types of disks; hard disks(HDD) and solid state disks (SSD)
- Hard disks are rotating platters; cheaper and slower
- Both are block oriented, i.e., they allow reading or writing of blocks (usually a few KB)
- Unlike RAM, which is byte oriented

# 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



- 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  $\sim .075$ s. 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
  - $\sim 400$ M instructions
  - Should take  $\sim .2$  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

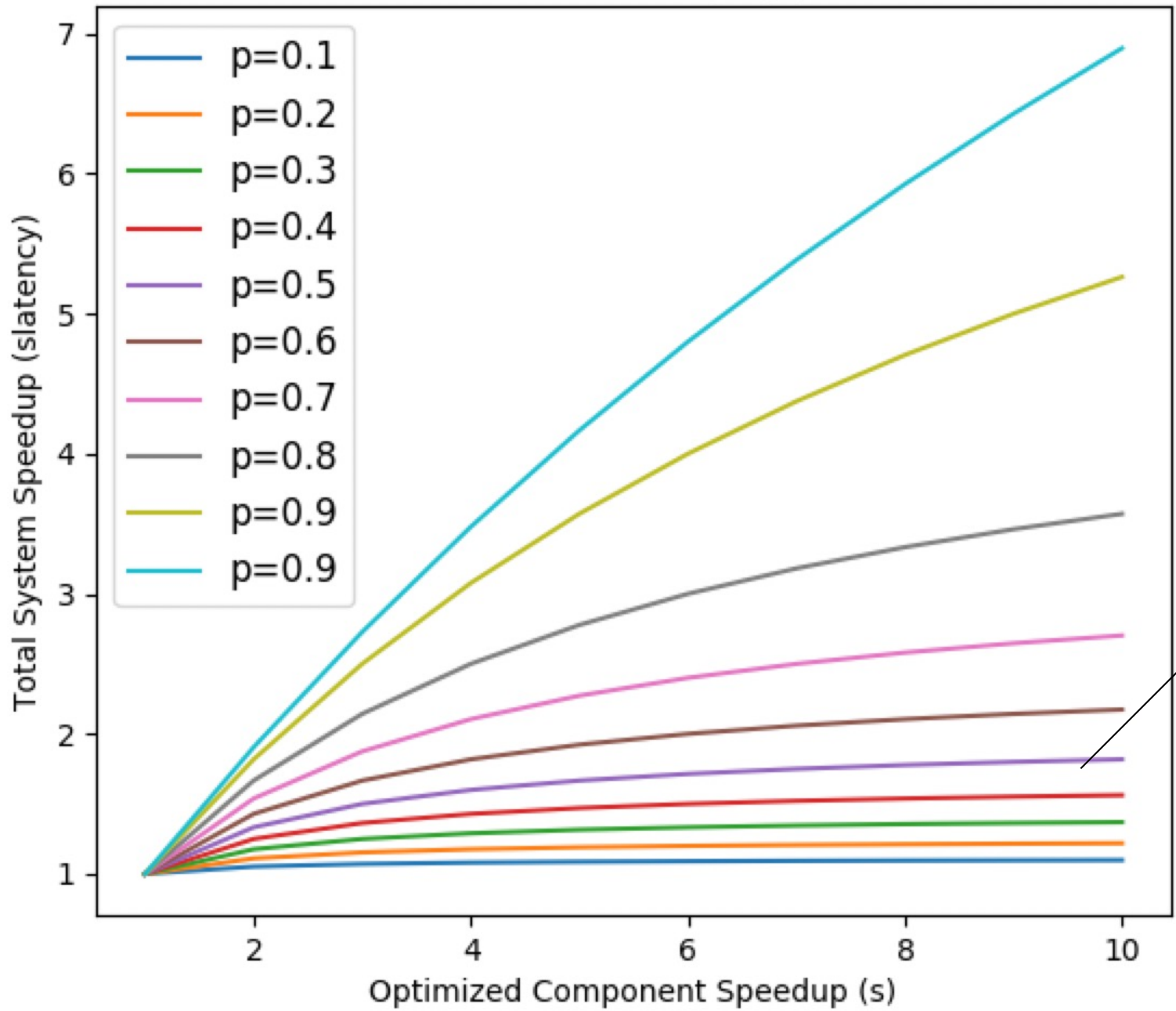
# What Improvement Can We Expect

- Always keep Amdahl's law in mind

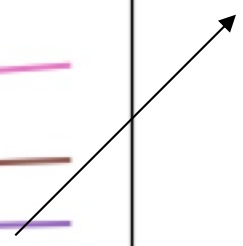
$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

$S_{\text{latency}}$  is the over all 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

Amdahl's law for varying p and s



If a component takes 50% of time, max speedup is 2x!



# 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

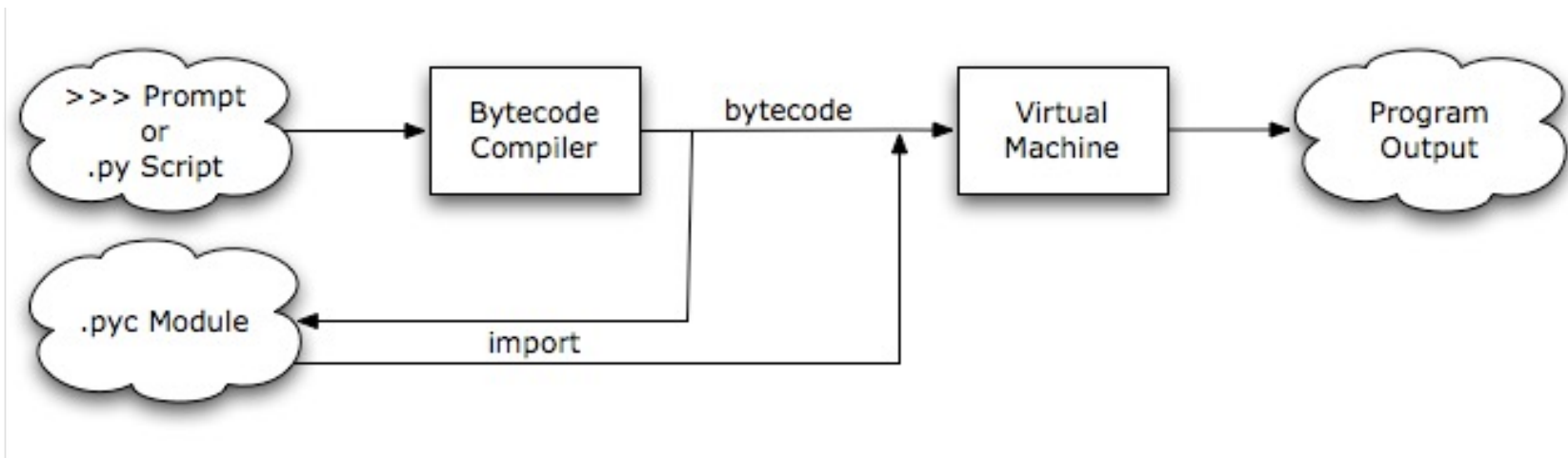
## Use a Relational DB

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

```
donations=# select NAME, EMPLOYER, TRANSACTION_AMT from donations  
where NAME ~ 'MADDEN' ;  
Time: 738.365 ms
```



# Why is Python So Slow



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

# Python In Practice

- 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 23456

- MADDEN -> MAD, ADD, DDE, DEN ...
- Index:

	Trigram	Start Offsets in Text
Sorted List	ADD	2, ...
	DDE	3, ...
	DEN	4, ...
	MAD	1, ...
	...	

Lookup: MAD -> 1, DEN -> 4

These are consecutive, so found a match

# Tree Index

A..C

D...G

G..P

P...Z

# Tree Index

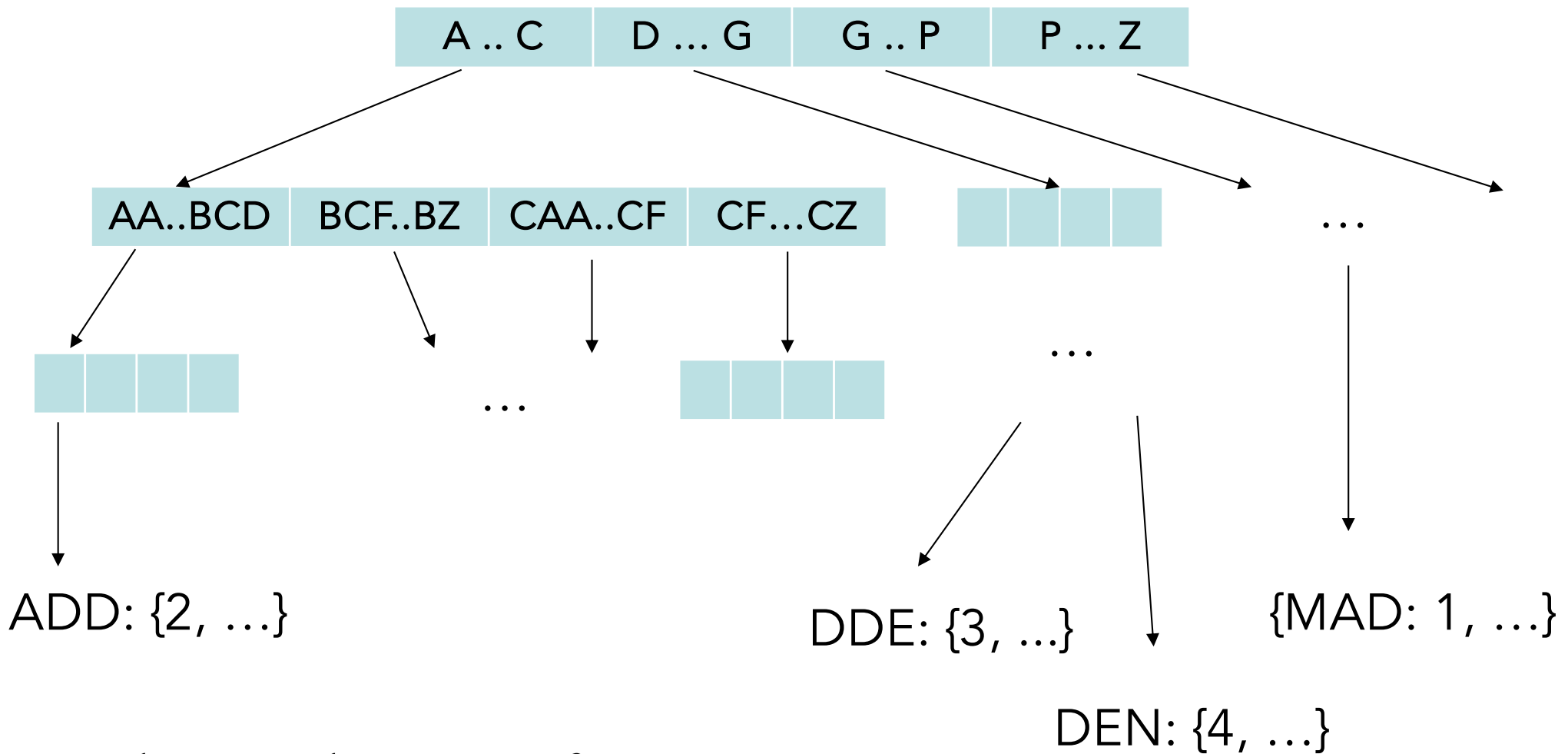


...

ADD: {2, ...}



# Tree Index



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?

```
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("read_time = %.2f, join_time = %.2f"%(read_time, join_time))
```

read\_time = 5.57, join\_time = 13.98

# Solution 1

```
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

Activity Monitor  
All Processes

CPU Memory Energy Disk Network

Process Name	Mem...	Threads	Ports	PID	User
Python	40.62 GB	8	21	84873	madden
WindowServer	4.48 GB	17	23,264	151	_windowserver
Mail	1.95 GB	9	7,555	470	madden
kernel_task	1.77 GB	309	0	0	root
Microsoft PowerPoint	1.59 GB	41	6,806	475	madden
iTerm2	1.52 GB	7	481	492	madden
https://www.boston.com	1.35 GB	20	310	29996	madden
Maps	1.29 GB	8	476	489	madden
Xcode	1.00 GB	25	1,904	71701	madden
Safari	942.8 MB	15	11,354	476	madden
Zulip Helper (GPU)	828.0 MB	10	249	1397	madden
Firefox	768.1 MB	81	9,660	32505	madden
https://www.boston.com	763.3 MB	17	296	83390	madden
Sublime Text	757.5 MB	20	654	440	madden
TextEdit	718.1 MB	6	782	503	madden
https://docs.google.com, ...	702.2 MB	7	281	44412	madden
Safari Networking	648.1 MB	11	9,627	550	madden
Microsoft Word	607.1 MB	16	1,071	78928	madden
Calendar	580.4 MB	4	3,299	9178	madden
Quip Web Content	569.0 MB	4	112	58653	madden
Dropbox	557.8 MB	156	1,451	80194	madden
Notes	548.8 MB	5	642	505	madden

MEMORY PRESSURE

Physical Memory:	64.00 GB	App Memory:	32.31 GB
Memory Used:	47.10 GB	Wired Memory:	7.14 GB
Cached Files:	15.51 GB	Compressed:	7.64 GB
Swap Used:	11.25 GB		

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