ANDREW MELO

VANDERBILT UNIVERSITY

BIG DATA AND SCIENCE

INTRODUCTION

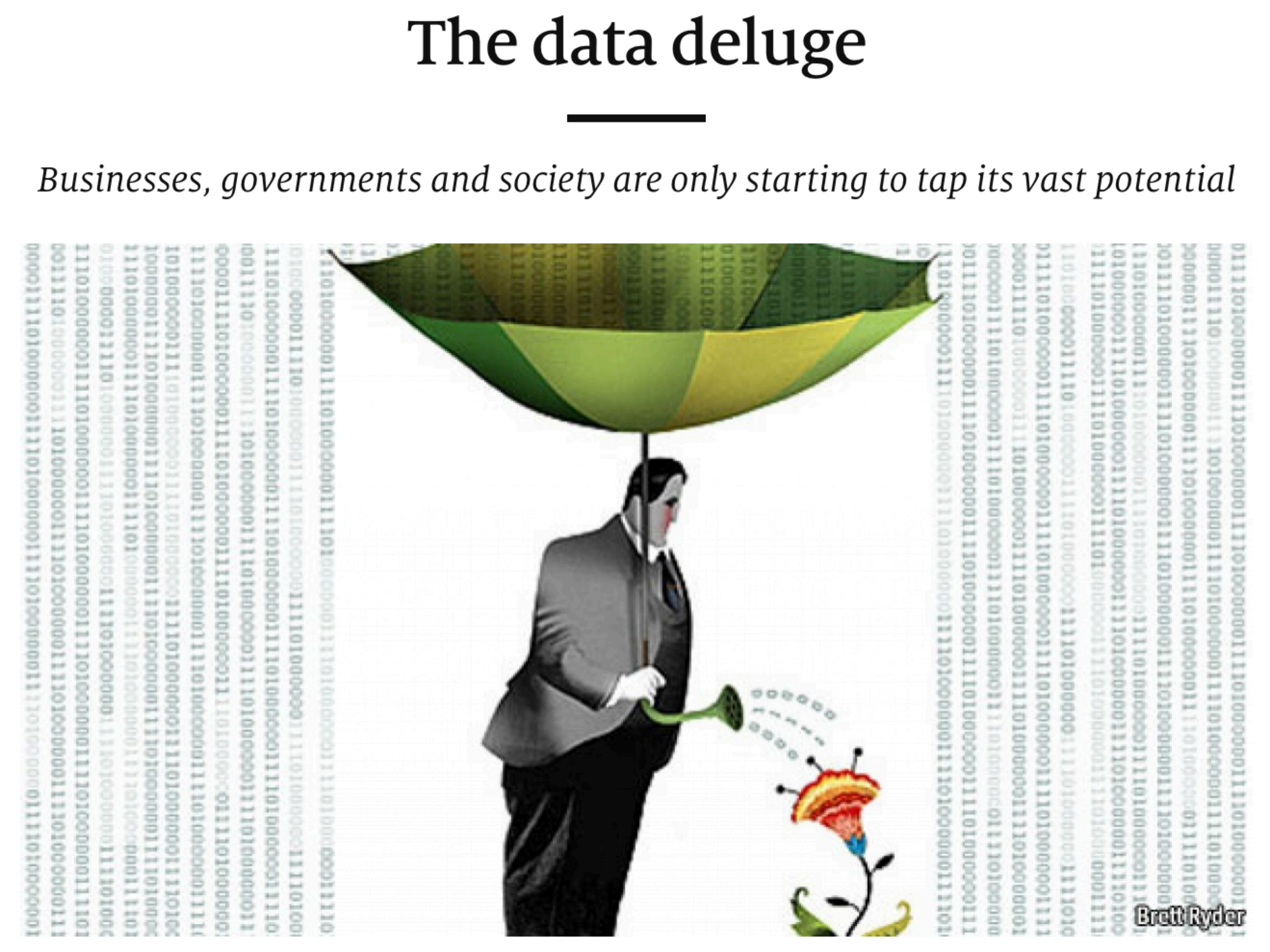
- About Me
  - Postdoc at Vanderbilt University
  - Member of CMS
  - Focused on CMS Computing & SUSY searches

CONTENT

1. History of Big Data
2. Apache Spark, Dask, Scientific Pythion
3. Big Data and HEP

LECTURE 1

HISTORY OF BIG DATA



The data deluge

Businesses, governments and society are only starting to tap its vast potential

https://www.economist.com/leaders/2010/02/25/the-data-deluge

2010: 1.2 ZETTABYTES

*1 ZETTABYTE = 1BILLION TERABYTES

https://www.seagate.com/our-story/data-age-2025/

2010: 1.2 ZETTABYTES

2018: 33 ZETTABYTES

*1 ZETTABYTE = 1BILLION TERABYTES

https://www.seagate.com/our-story/data-age-2025/

2010: 1.2 ZETTABYTES

2018: 33 ZETTABYTES

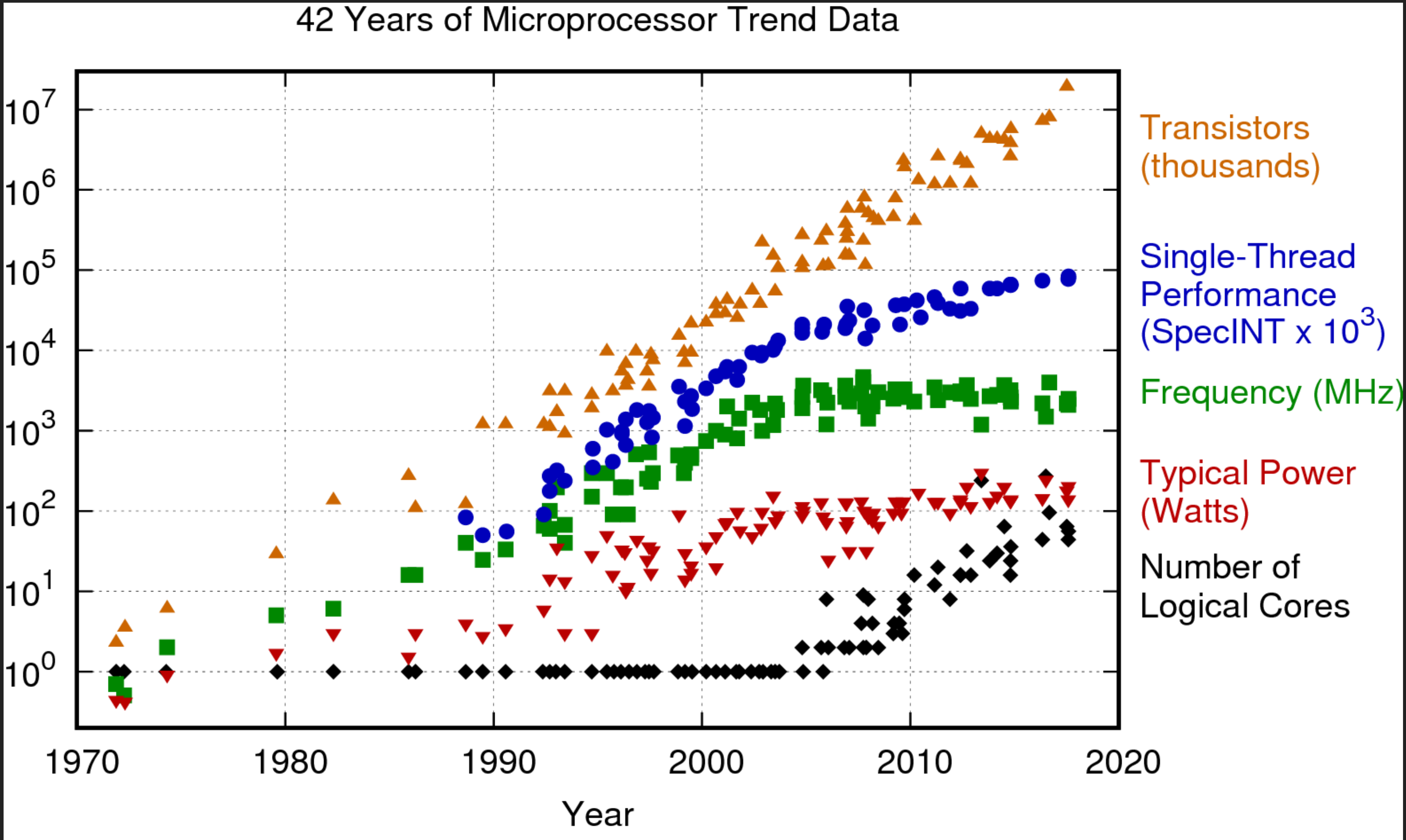
2025: 178 ZETTABYTES

A 61% annual increase!

*1 ZETTABYTE = 1BILLION TERABYTES

https://www.seagate.com/our-story/data-age-2025/

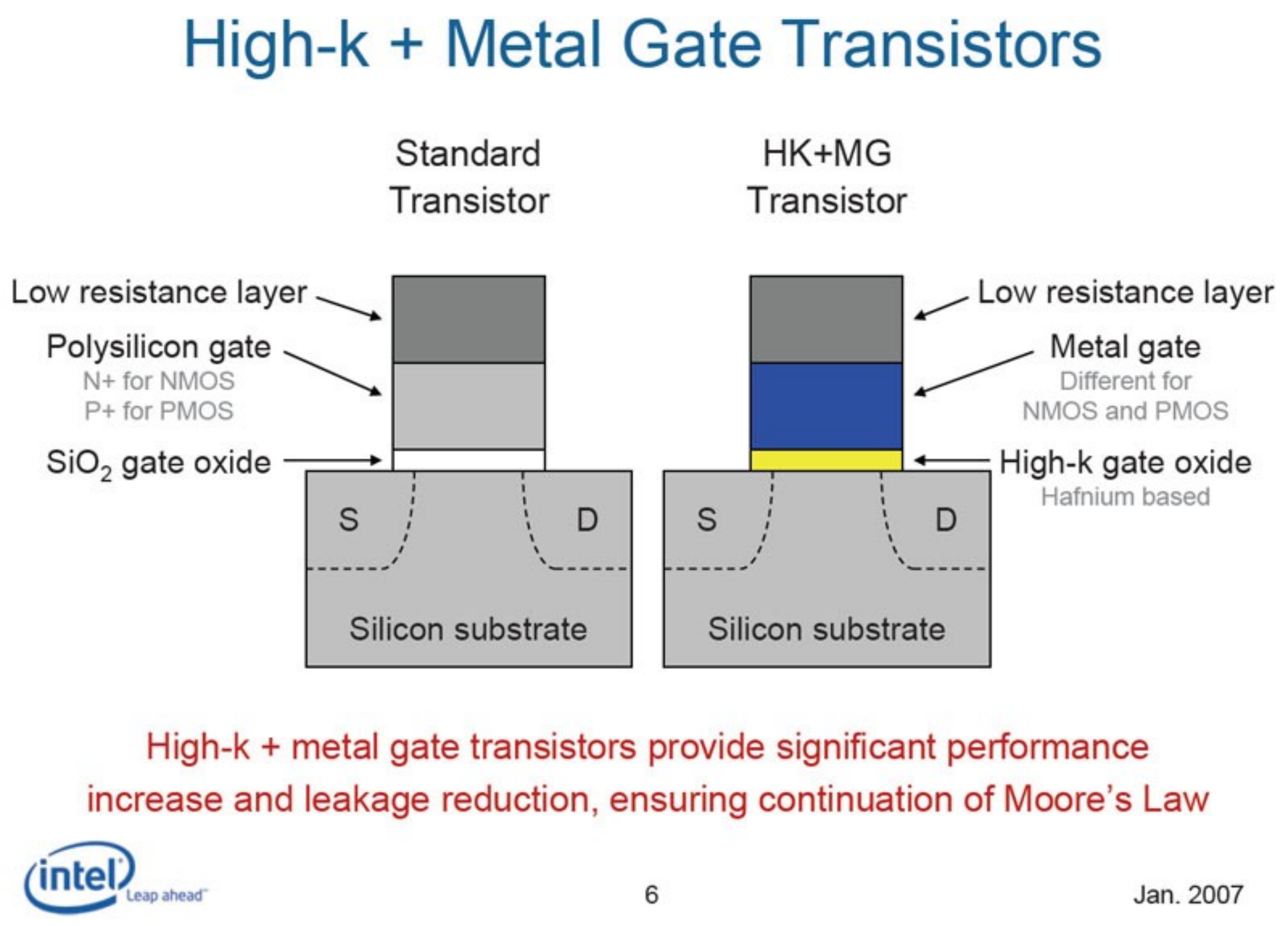
INDIVIDUAL THREAD PERFORMANCE IS ~FLATTENING



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

CURRENT LEAKAGE



High-k + Metal Gate Transistors

Standard
Transistor

HK+MG
Transistor

Low resistance layer

Polysilicon gate
N+ for NMOS
P+ for PMOS

SiO_2 gate oxide

S

D

Silicon substrate

Low resistance layer

Metal gate
Different for
NMOS and PMOS

High-k gate oxide
Hafnium based

S

D

Silicon substrate

High-k + metal gate transistors provide significant performance increase and leakage reduction, ensuring continuation of Moore's Law

intel Leap ahead

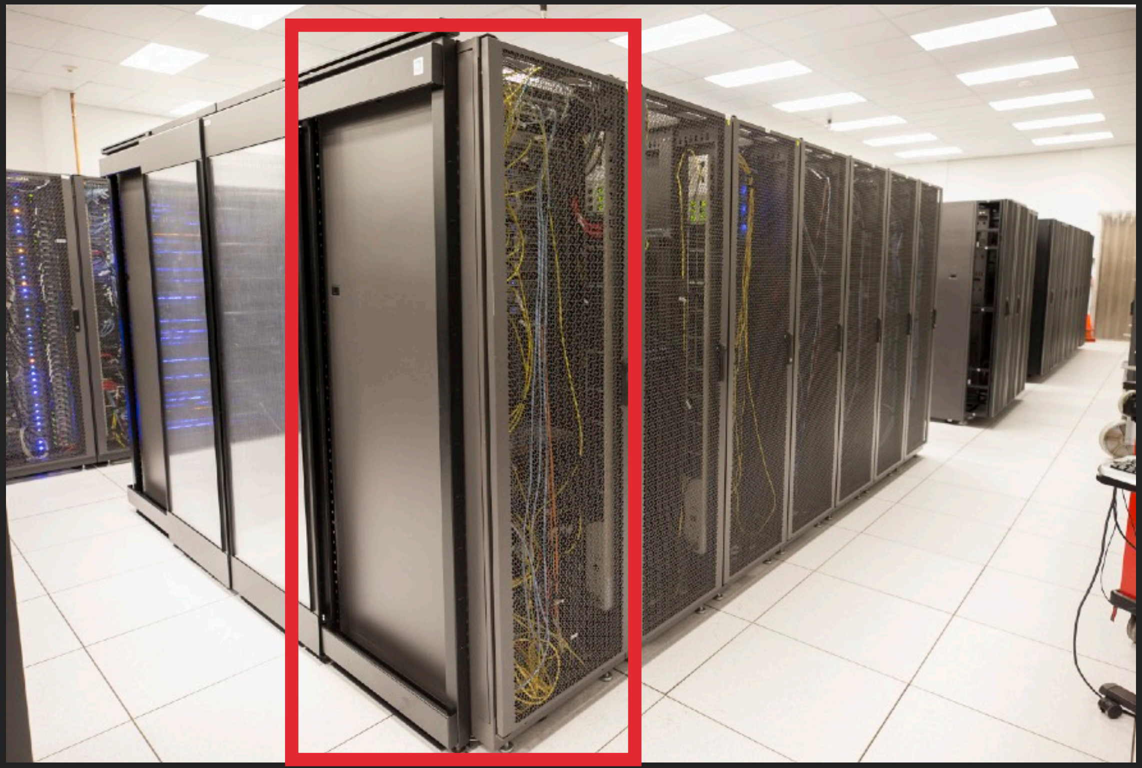
6

Jan. 2007

https://www.hardwaresecrets.com/details-on-intels-forthcoming-45-nm-manufacturing-technology/

POWER IS EXPENSIVE

- Electricity costs
- Switches, PDUs
- Cooling



Each rack at full load consumes 10KW

A typical (American) house consumes 1.2KW!

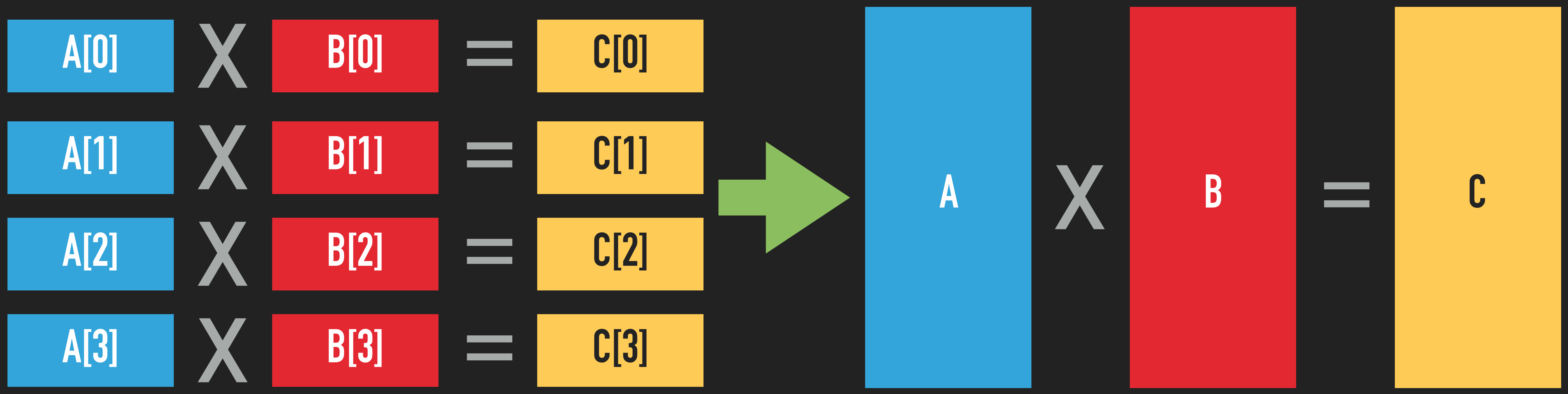
SINGLE-THREAD PERFORMANCE

- Cache utilisation
- IPC/Pipelining
- SIMD

Tightly-written code can take advantage of these, to gain more operations per GHz but hard to get right!

SIMD - SINGLE INSTRUCTION MULTIPLE DATA

- Performing an operation over multiple input data simultaneously



Traditional

SIMD/Vectorized

Intel CPU supports up to 16-way SIMD, GPUs support many many more

RECAP

- An explosion in data
- CPUs aren't getting faster
  - Architecture hitting fundamental power/physics limitations
- Individual disks are slowly increasing capacity
- Simultaneously, individual "consumer grade" hardware becoming cheaper

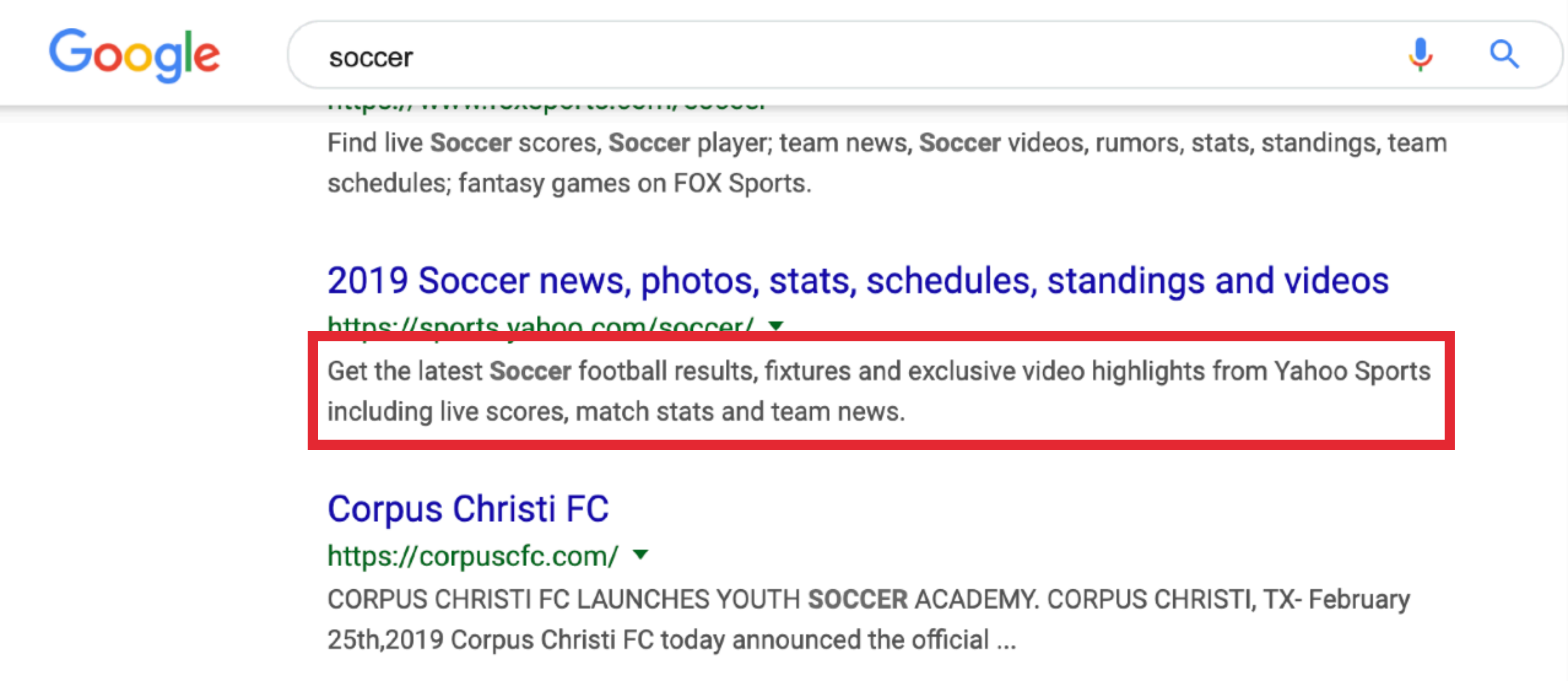
A PERFECT STORM FOR A FUNDAMENTAL CHANGE

BIG DATA -- NOT (ONLY) ABOUT VOLUME

- The "three Vs" of Big Data
    - Volume
    - Velocity
    - Variability

VELOCITY

- Want to lessen "time to insight"
- Moving from batch-oriented to streaming-like frameworks



Find live **Soccer** scores, **Soccer** player; team news, **Soccer** videos, rumors, stats, standings, team schedules; fantasy games on FOX Sports.

2019 Soccer news, photos, stats, schedules, standings and videos
https://sports.yahoo.com/soccer/ ▾
Get the latest **Soccer** football results, fixtures and exclusive video highlights from Yahoo Sports including live scores, match stats and team news.

Corpus Christi FC
https://corpuscfc.com/ ▾
CORPUS CHRISTI FC LAUNCHES YOUTH **SOCCER** ACADEMY. CORPUS CHRISTI, TX- February 25th,2019 Corpus Christi FC today announced the official ...

Hard-disk seek is ~3msec - only way to load this page faster is to store the entire Internet in RAM

VARIABILITY

- Unstructured data is quickly becoming dominant

48 hours of video uploaded every minute

3m "likes"/day

10m images uploaded/day

EXISTING RDBMS TOOLS A POOR FIT FOR THESE DATA

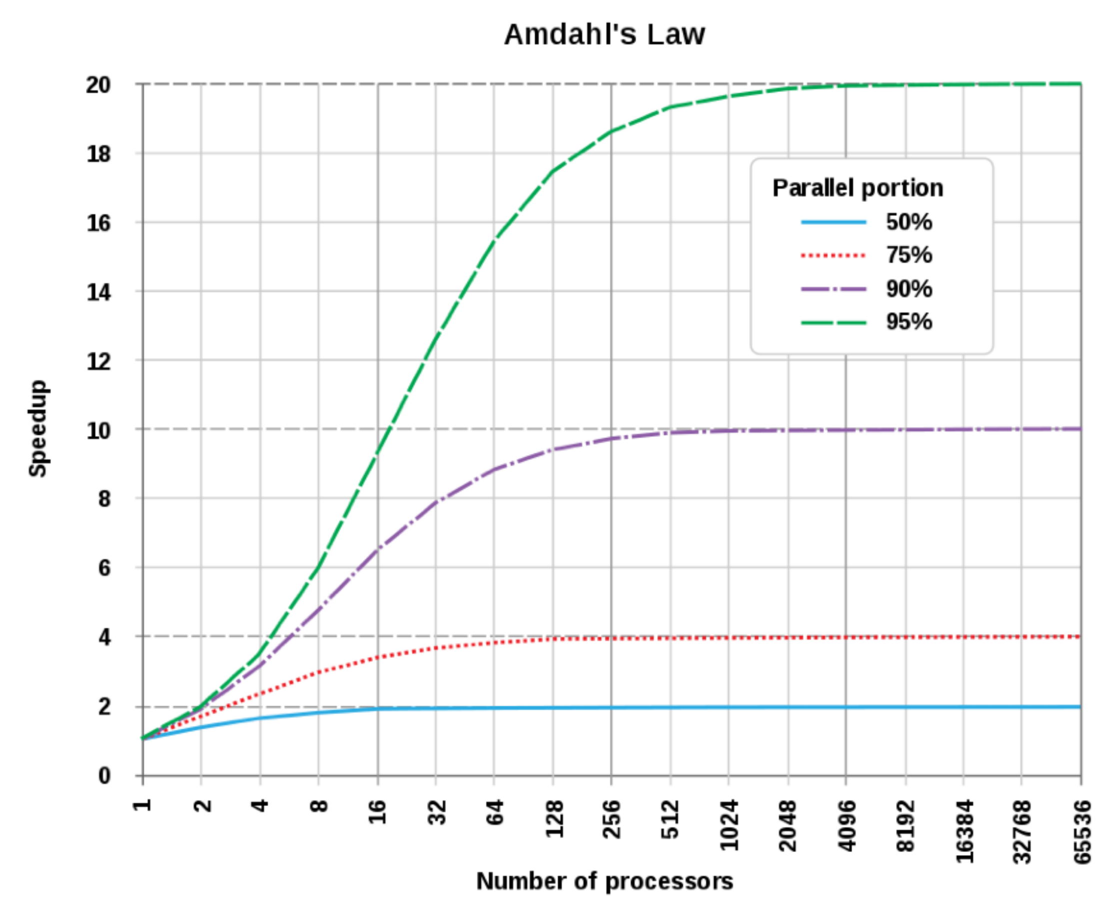
PROCESSING THESE DATA VOLUMES

TARGET PARALLELIZATION FROM THE OUTSET

- Look at a whole cluster as an execution unit
  - Not just a single CPU or node
- Handle resiliency
  - More hardware - something guaranteed to fail
- Minimize shared state, synchronization
  - Amdahl's Law

MAXIMIZING SPEEDUP W/INCREASED CONCURRENCY

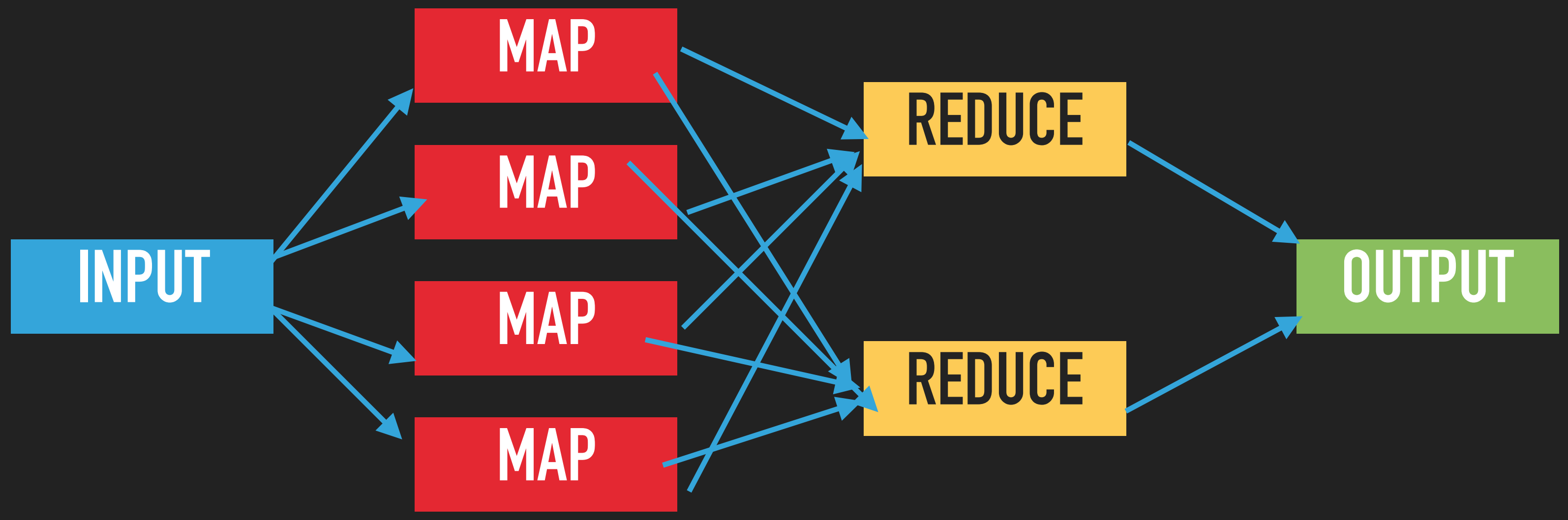
- Performance scales sub-linearly with resources
    - 2x the CPUs necessarily produces <2x the performance



IT'S IMPORTANT MINIMIZE SYNCHRONIZATION, SINCE IT'S INHERENTLY SERIAL

DIVIDE AND CONCUR

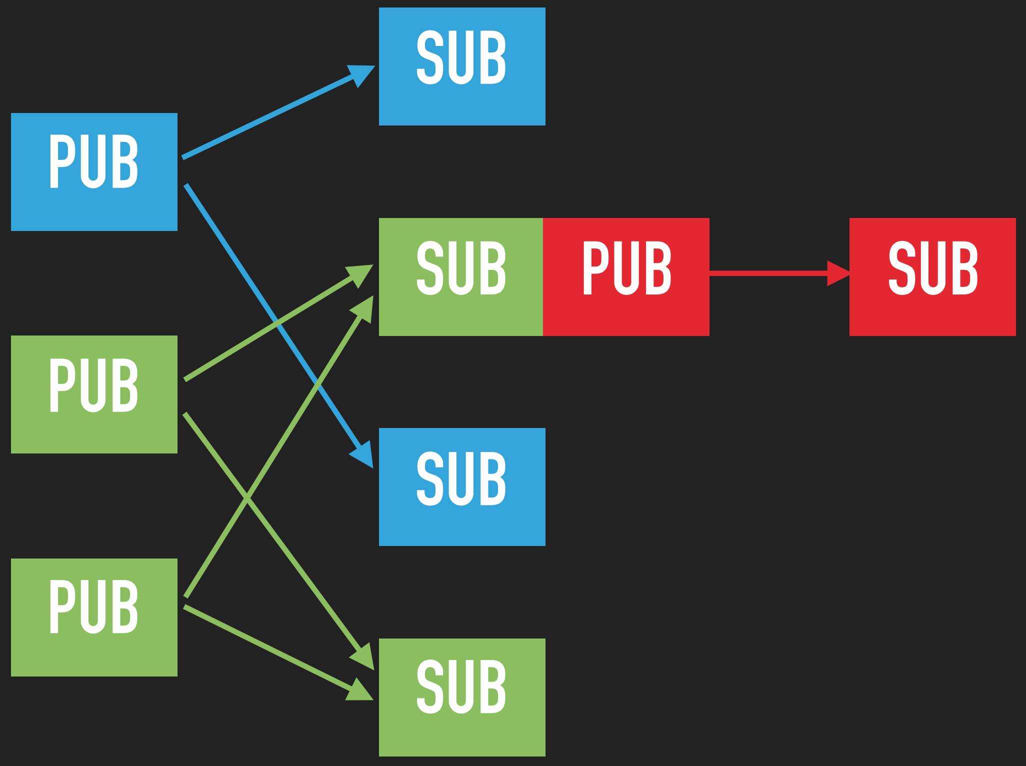
- Split input into N pieces
- Do "something" over each piece
- Shuffle/sort/combine intermediate outputs
- Generate final output

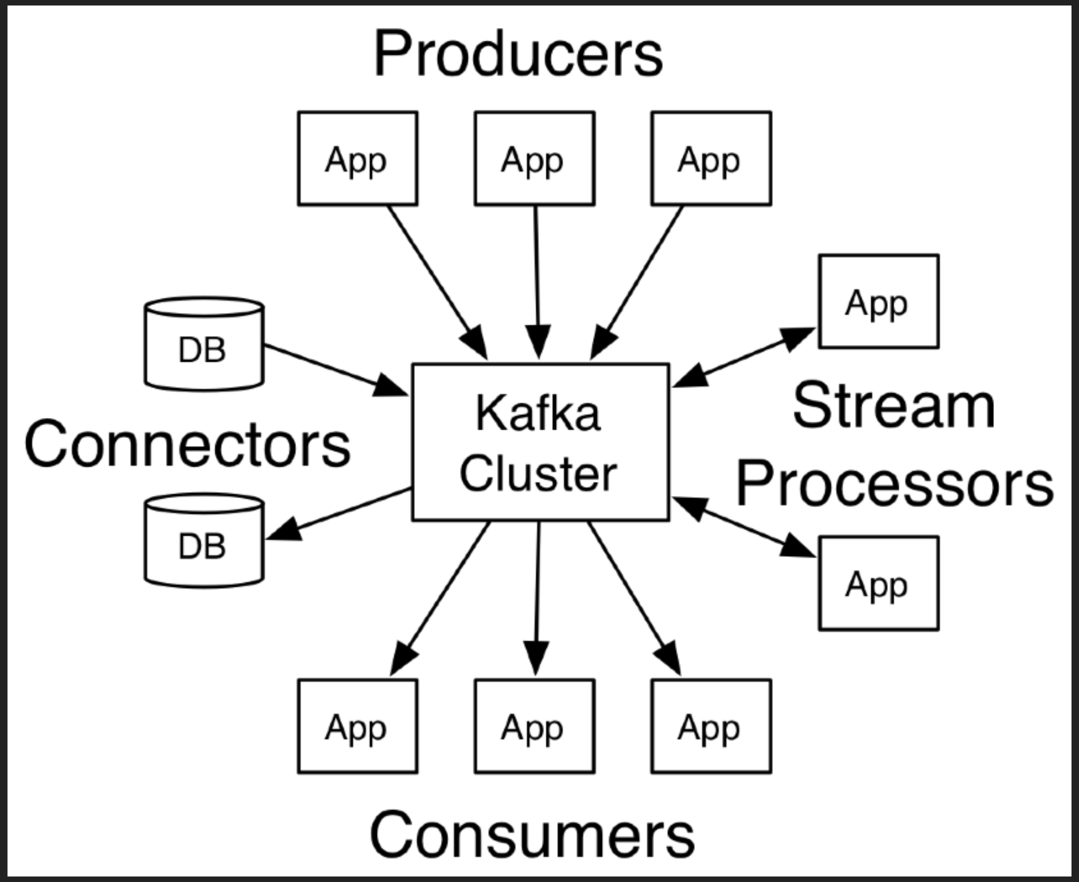


hadoop

PRODUCER/CONSUMER AKA PUBLISH/SUBSCRIBE

- Independent actors can produce data, which is consumed by some number of consumers
- Framework ensures each consumer receives their requested data





COMMON ISSUES

- What if a worker dies?
- How do we aggregate partial/complete results
  - Shared filesystem? IPC?
- How do we know when the workers are done?
- What if workers need to share large static data?
- Which worker should run which task?

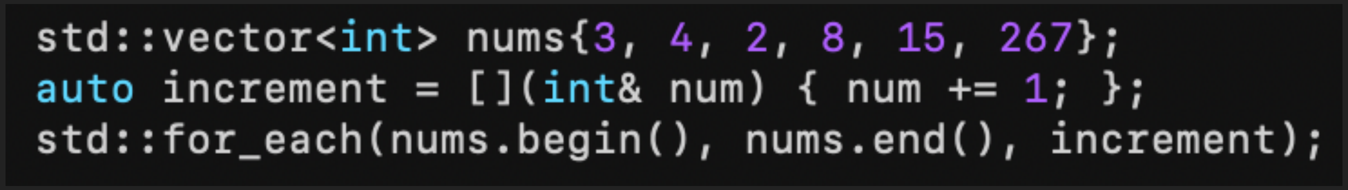
MANAGING WORKERS IS DIFFICULT

- Run asynchronously on possibly many nodes
- No guarantee of ordering
  - Or completion! Hardware can fail in subtle ways
- Need some type of shared state/synchronization
  - :(

SHARED STATE IS HARD

- Concurrent programming on a single machine is hard
  - Distributed concurrent programming is even harder
  - Even a "distributed clock" is hard!
- Barriers, semaphores, counters, etc.. are all difficult to reason about and get correct
- Separate "what to do" from "how to do it"
  - Let the experts handle the sticky details

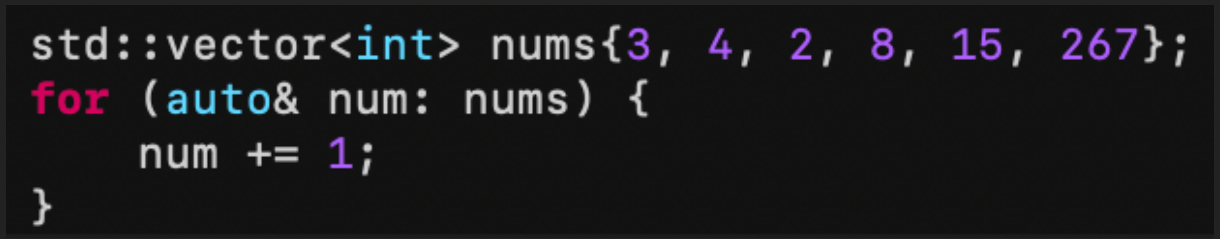
DECLARATIVE VS IMPERATIVE



```
std::vector<int> nums{3, 4, 2, 8, 15, 267};
auto increment = [](int& num) { num += 1; };
std::for_each(nums.begin(), nums.end(), increment);
```

Declarative

By swapping for_each out with a different implementation, this code can be parallelized w/o any changes to end-user code



```
std::vector<int> nums{3, 4, 2, 8, 15, 267};
for (auto& num: nums) {
    num += 1;
}
```

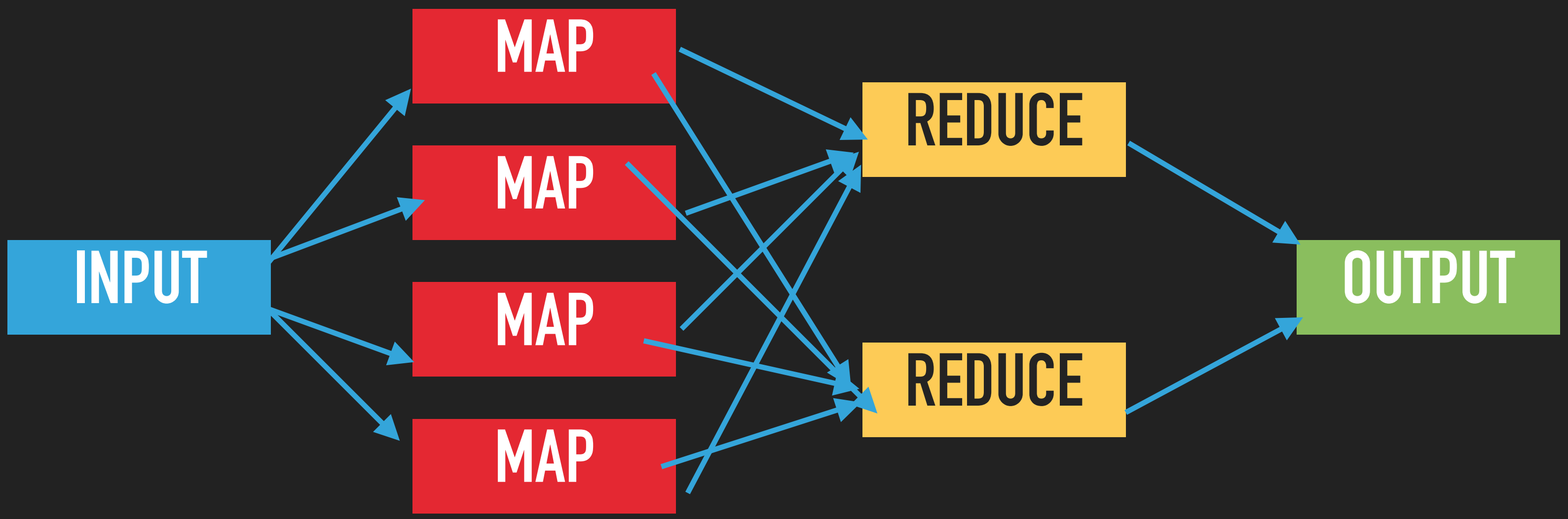
Imperative

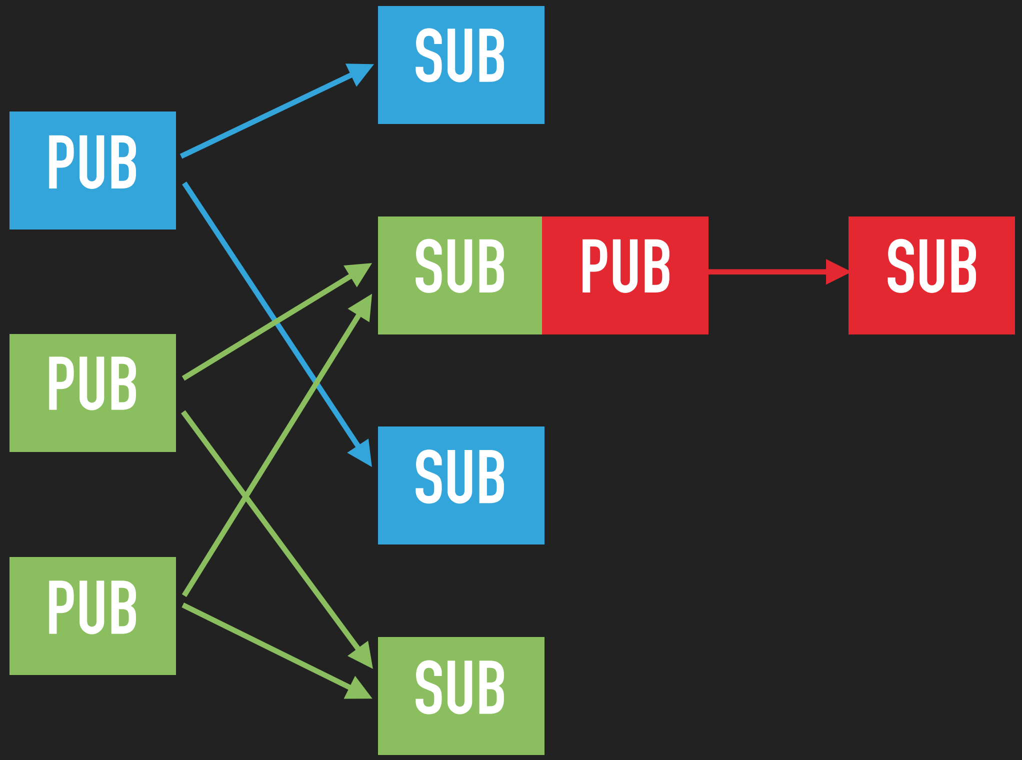
The for loop is "baked in", and adapting this code to run in parallel would involve significant changes/boiler plate

FUNCTIONAL PROGRAMMING

```
std::vector<int> nums{3, 4, 2, 8, 15, 267};
auto increment = [](int& num) { num += 1; };
std::for_each(nums.begin(), nums.end(), increment);
```

A declarative, functional style allows for separation of interests - users provide the function to be executed, and the framework provides the execution engine





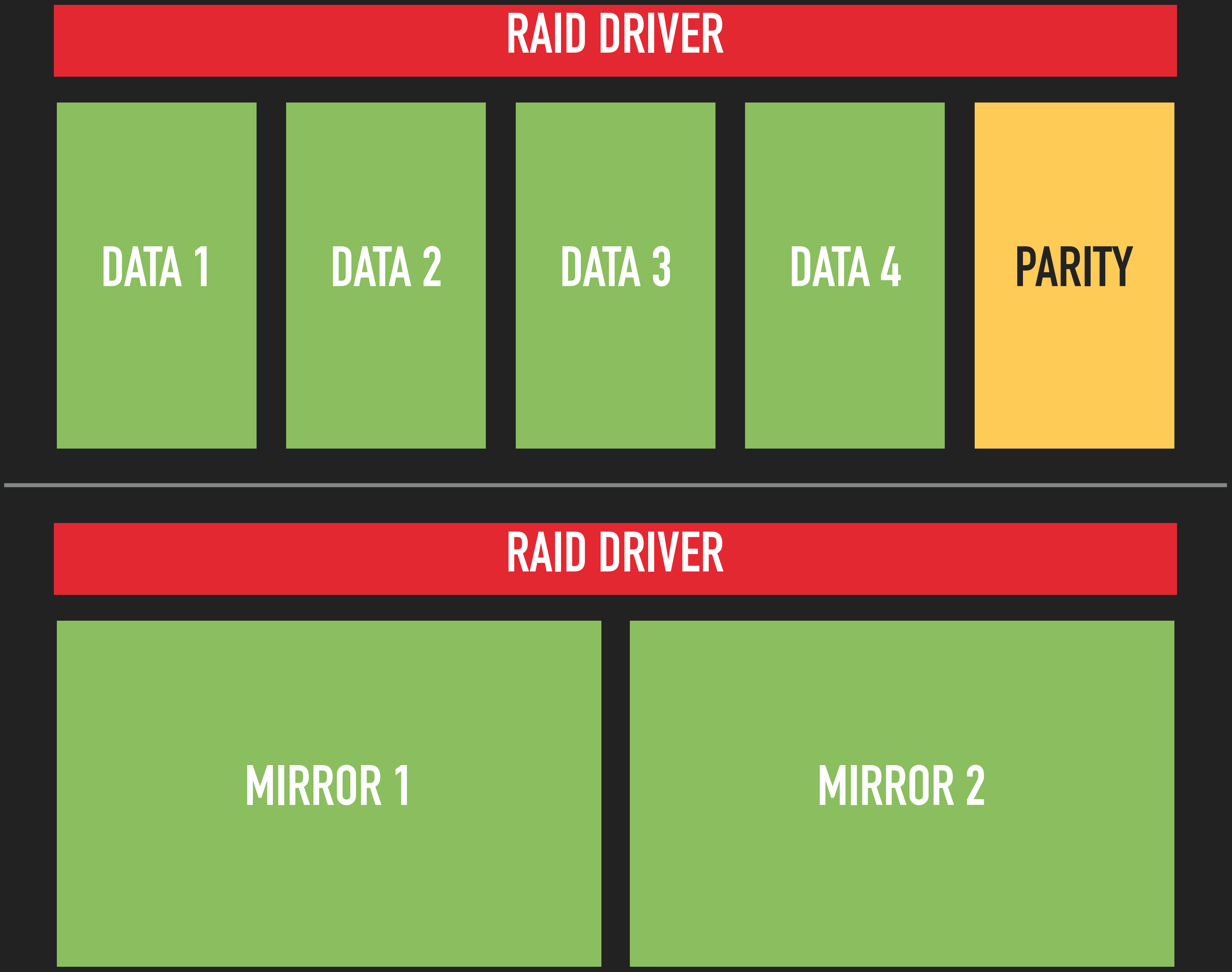
DATA
MANAGEMENT

DATA MANAGEMENT

- The ability to reliably store extremely large datasets is important
- Cost is key
  - Cheap hardware
  - Enhance reliability via software

RAID-1/5

- Common on workstation-class machines



RAID DRIVER

DATA 1

DATA 2

DATA 3

DATA 4

PARITY

RAID DRIVER

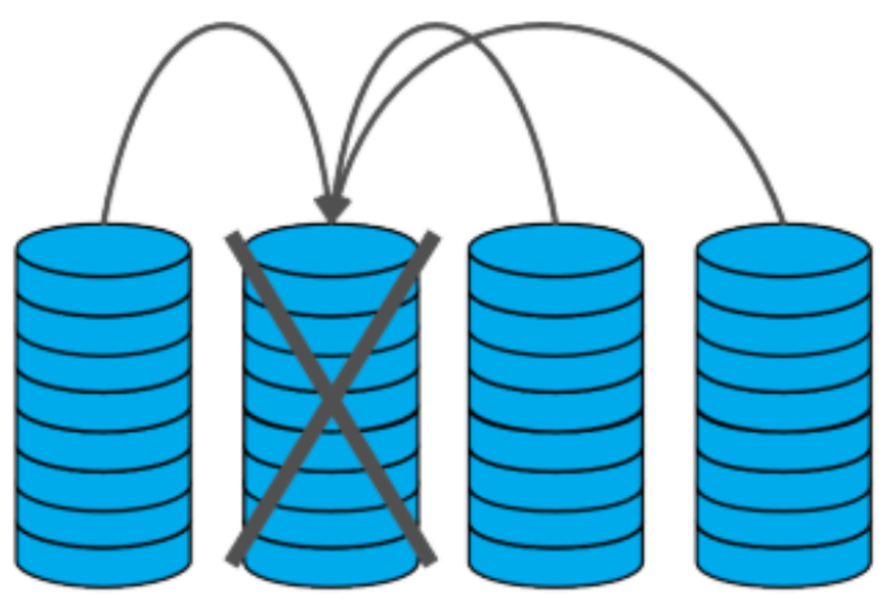
MIRROR 1

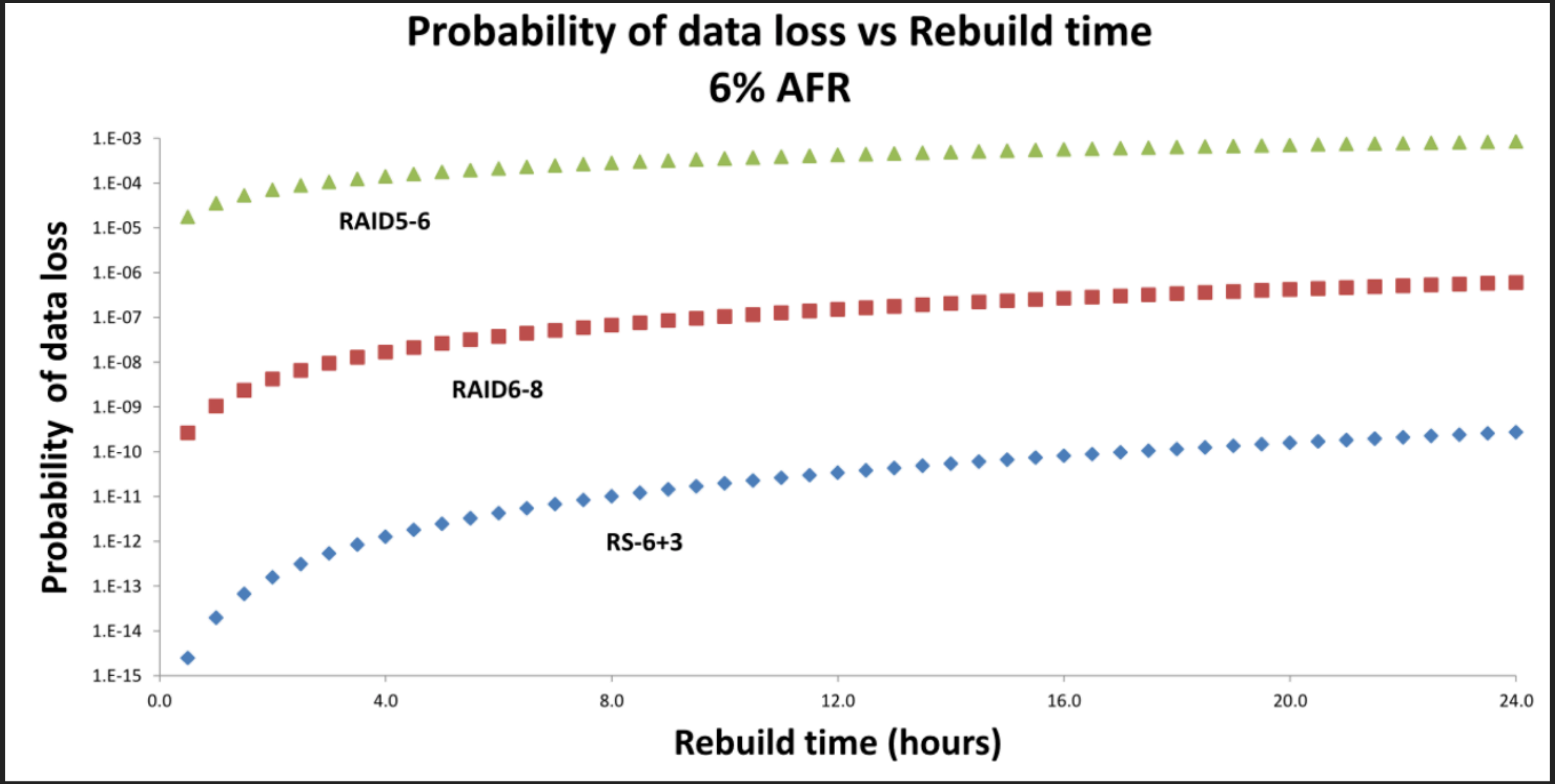
MIRROR 2

- Limited to single machine
- Bad performance during rebuilds
- Bad probability of double-fault errors

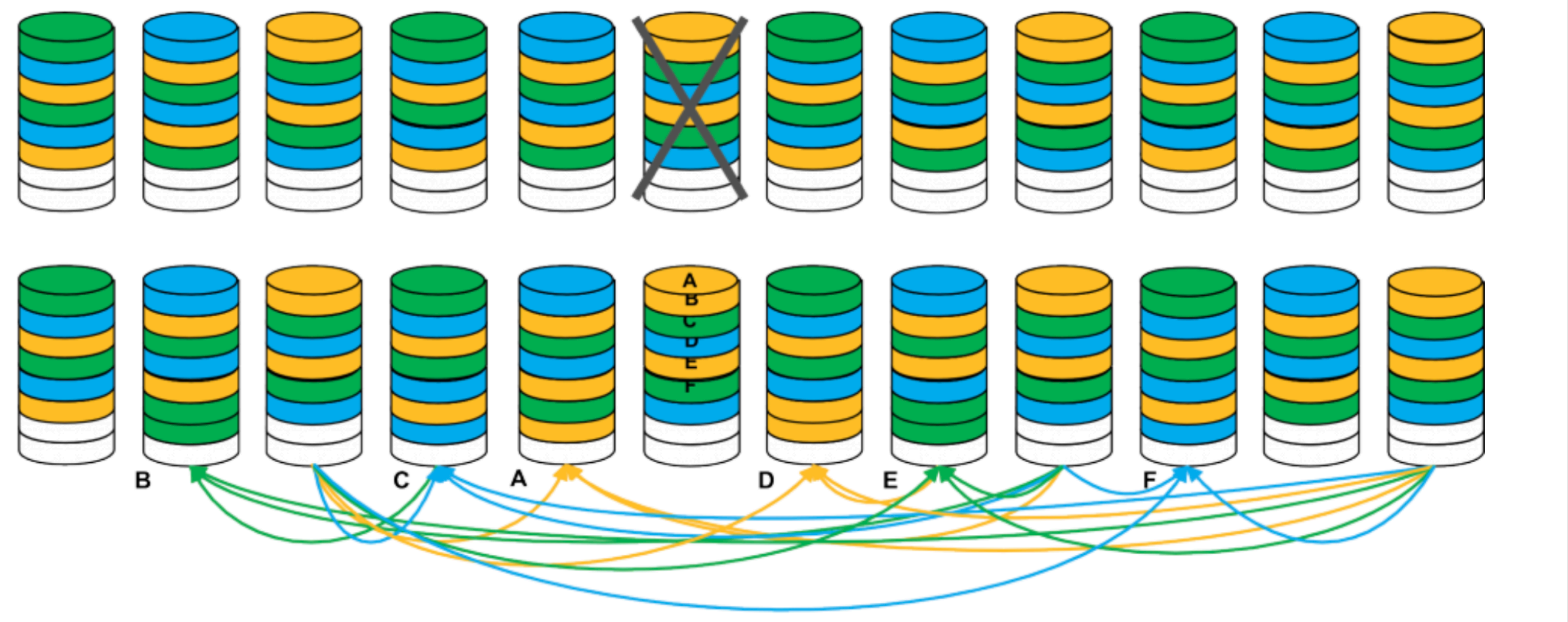
RAID-REPAIR IS DANGEROUS

- Replacing a failed drive with a new one is time consuming
- 33 hours to fill a 12TB drive @ 100MB/sec



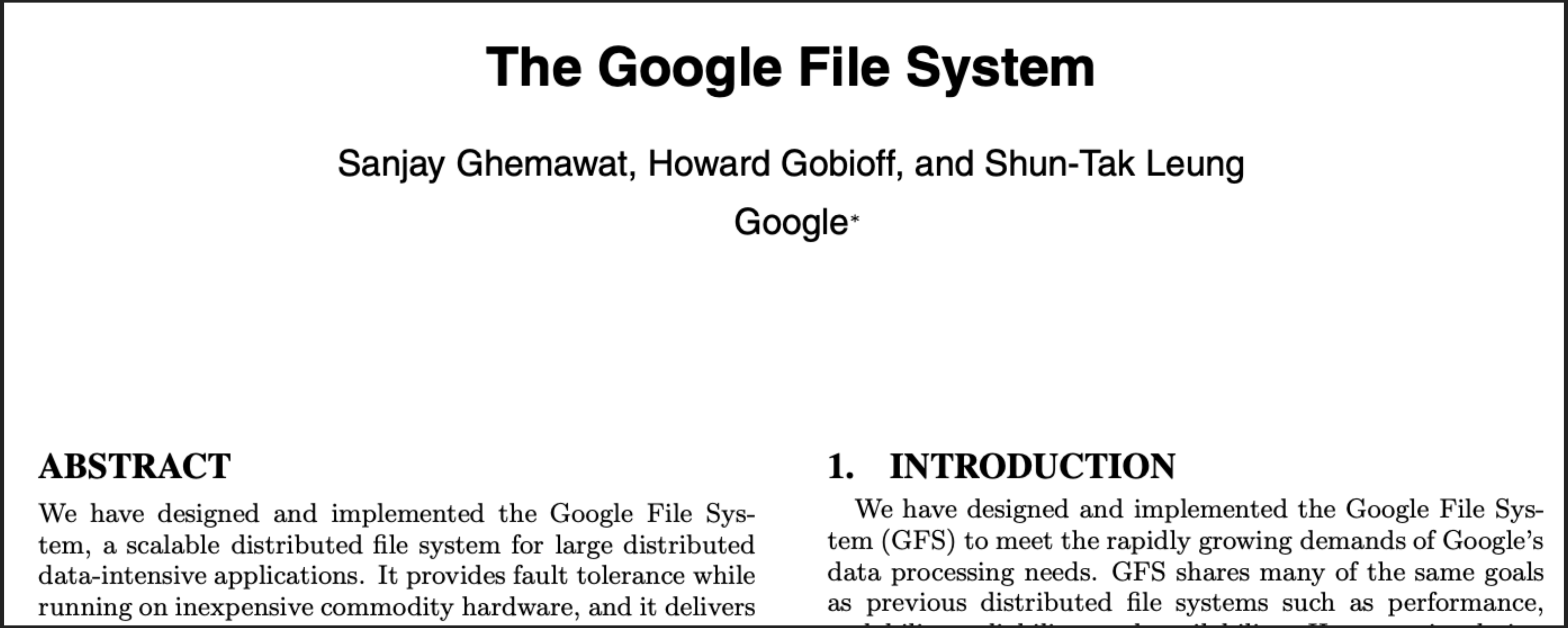


DISTRIBUTED REPAIR



INSTEAD OF A TRUE RAID ARRAY, CREATE A LOGICAL RAID ARRAY OVER MANY DISKS

GOOGLE FILE SYSTEM, 2003



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance,

> "MULTIPLE GFS CLUSTERS ARE CURRENTLY DEPLOYED FOR DIFFERENT PURPOSES. THE LARGEST ONES HAVE OVER 1000 STORAGE NODES, OVER 300 TB OF DISK STORAGE, AND ARE HEAVILY ACCESSED BY HUNDREDS OF CLIENTS ON DISTINCT MACHINES ON A CONTINUOUS BASIS.

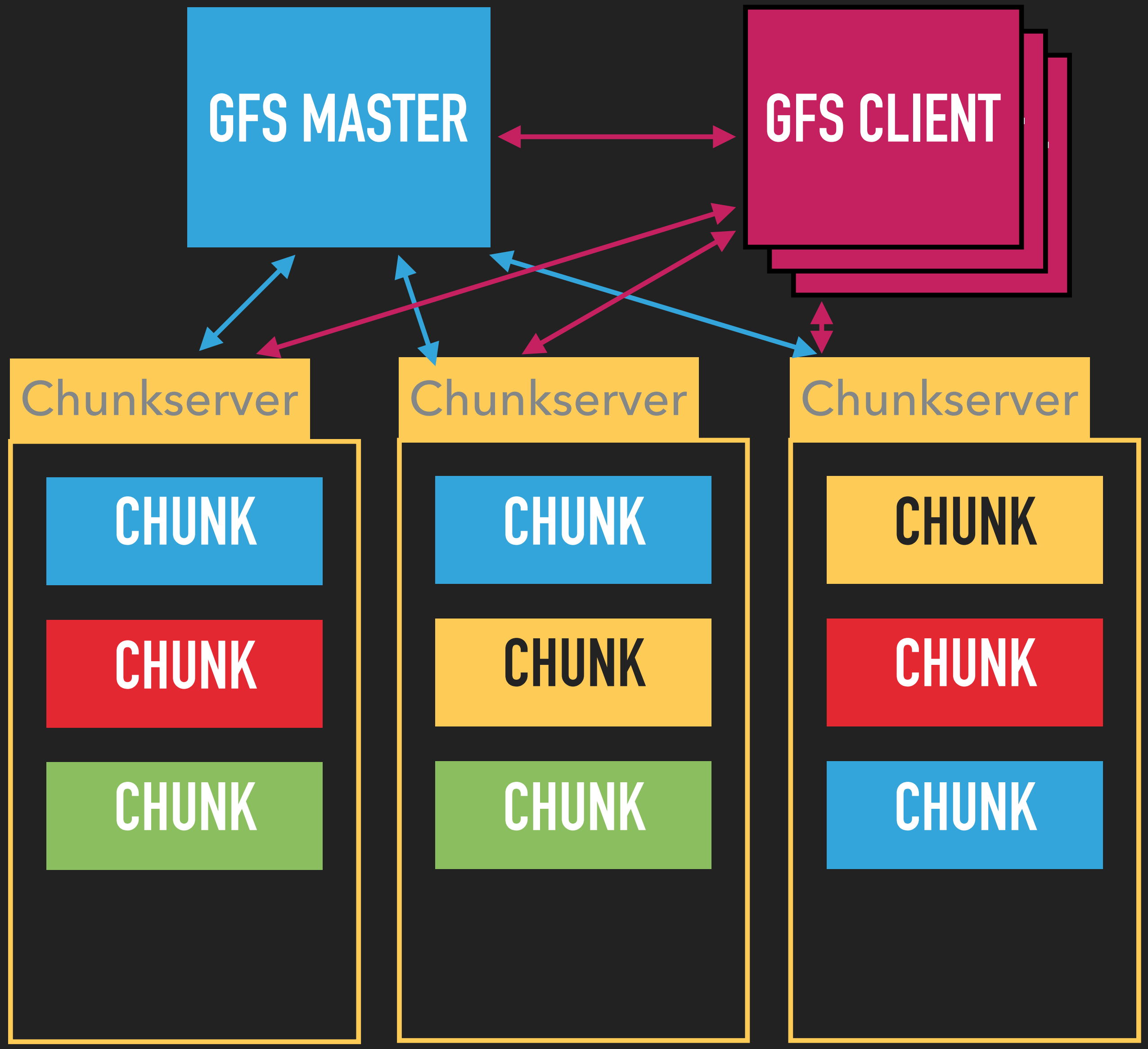
https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

GFS ASSUMPTIONS

- System built on inexpensive commodity parts
- A small number of larger files
- Primarily streaming, not random access
- All files are appended and rarely modified
- Many producers will concurrently append to the same file

HOW DOES A SYSTEM WITH OUR SCALING PRINCIPALS LOOK?

GFS ARCHITECTURE

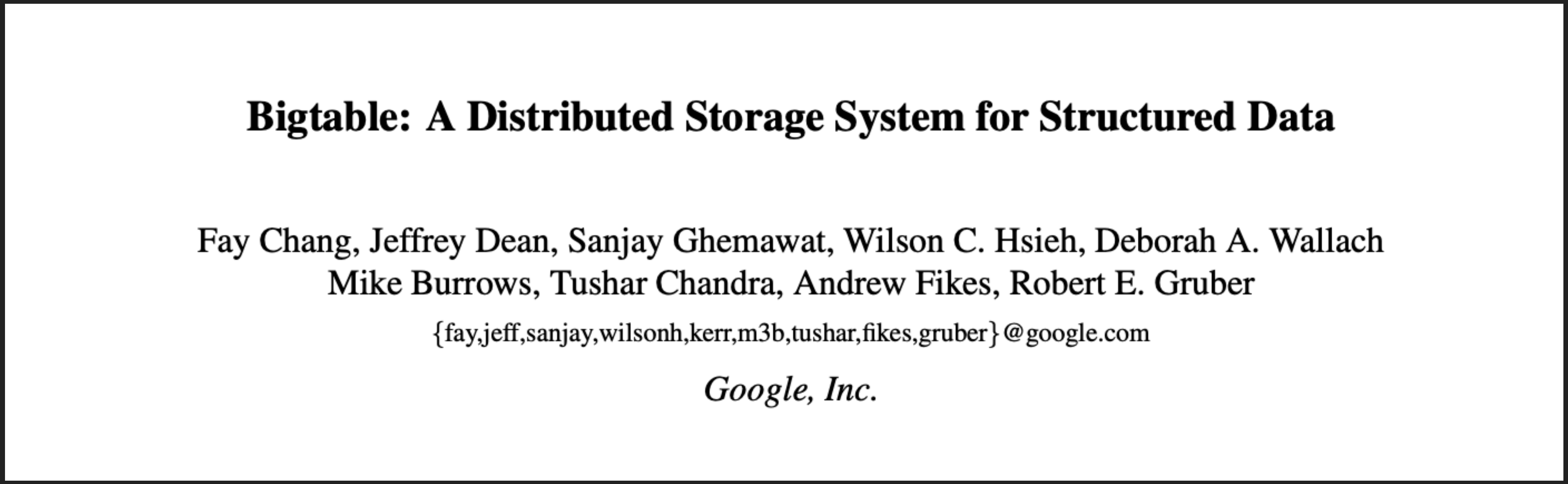


TRIVIALLY SCALABLE EXCEPT MASTER

- Files are divided into chunks
  - 64-128MByte
- Chunks are stored on ChunkServers
  - Many standard disks
  - Reports health to Master
- Master tracks metadata
  - SPOF initially
  - Clients avoid/cache Master

GFS AND GOOGLE

- GFS is arguably the "secret sauce" that helped Google grow as it did
- Many initial services were built over GFS

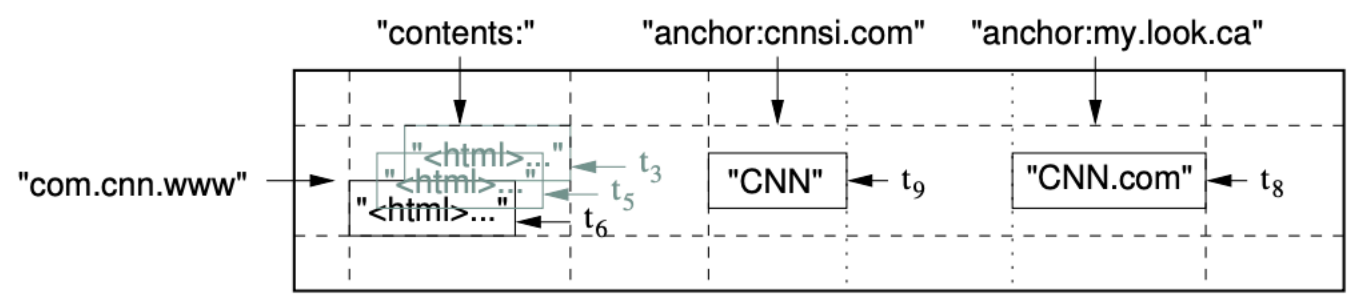


Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
Google, Inc.

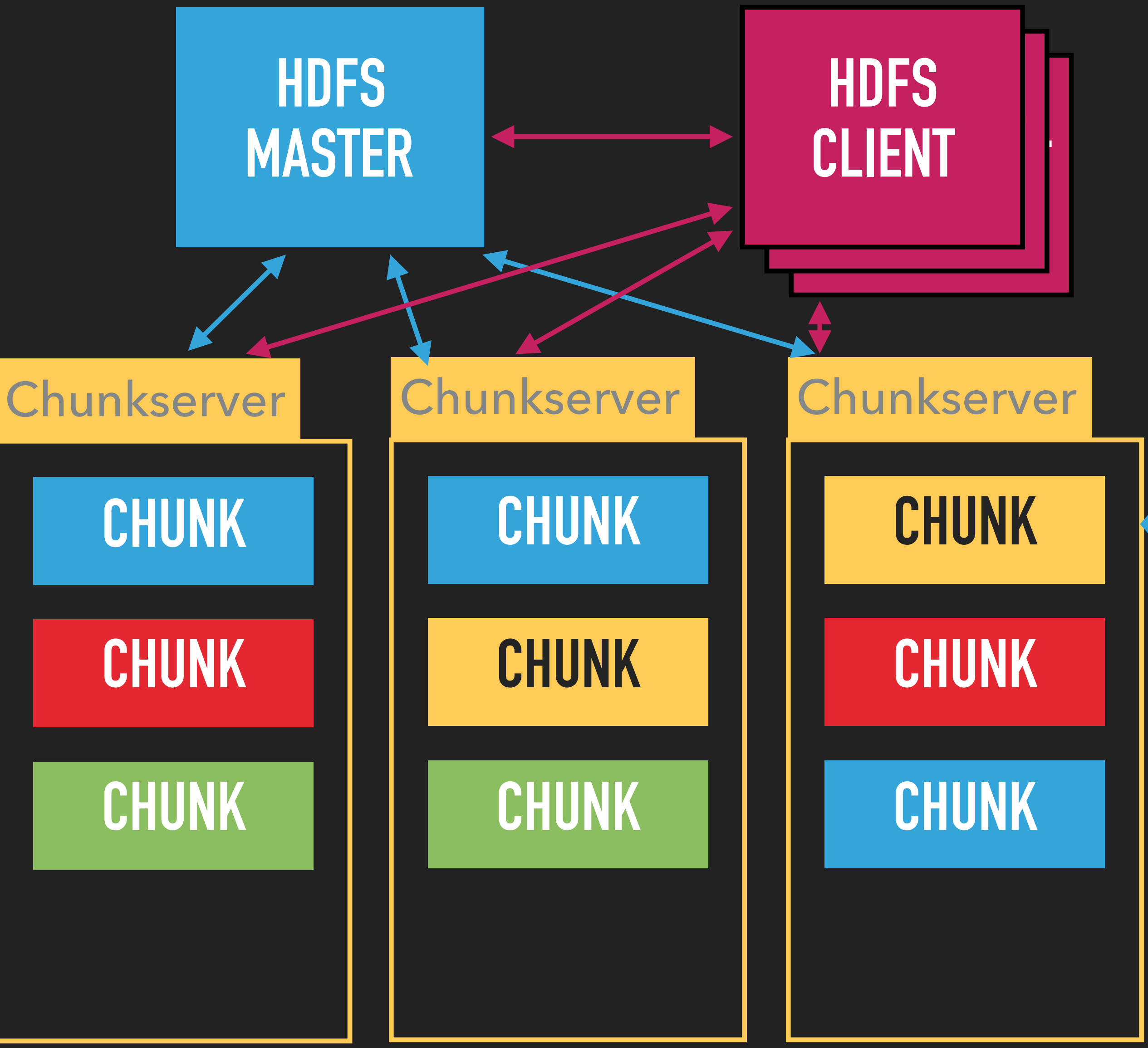
Bigtable paper, 2006

LOOK FAMILIAR?



https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

APACHE HDFS ARCHITECTURE



HDFS MASTER

HDFS CLIENT

Chunkserver

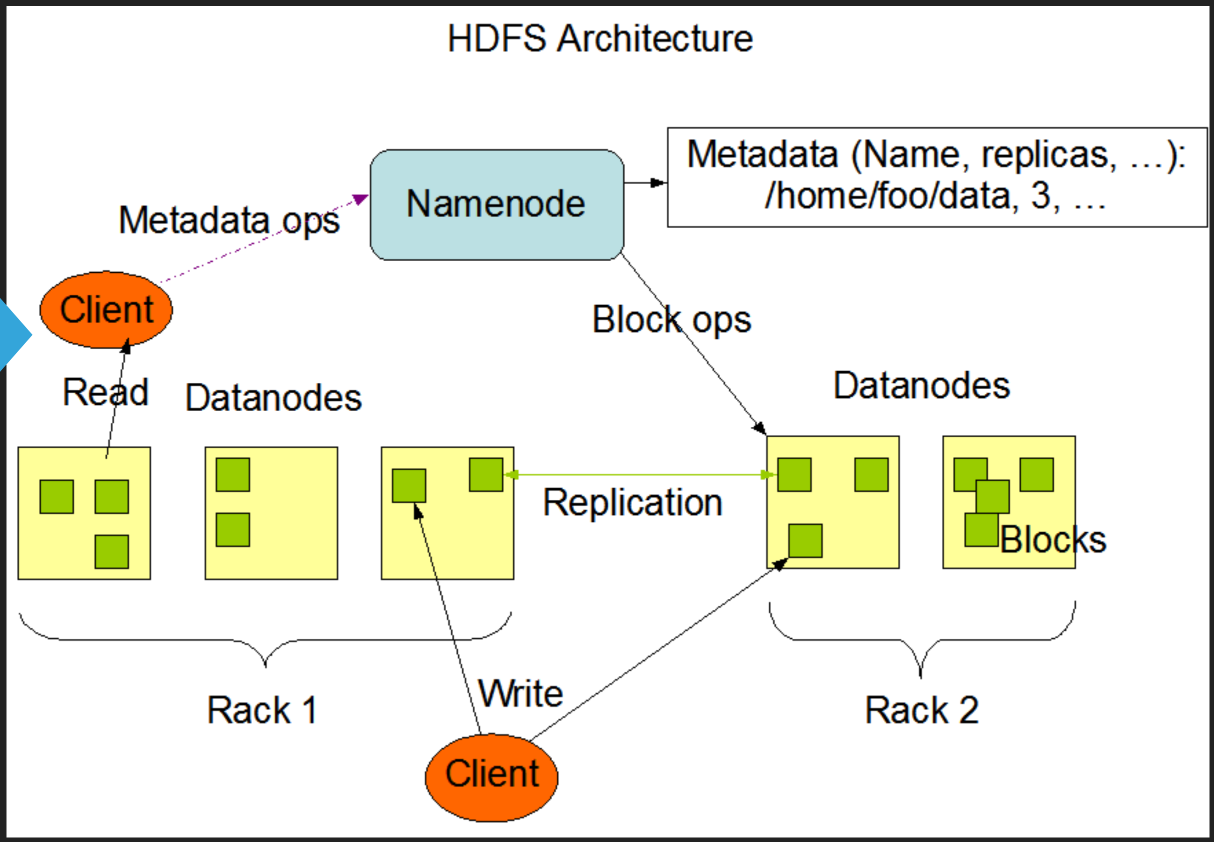
CHUNK
CHUNK
CHUNK

Chunkserver

CHUNK
CHUNK
CHUNK

Chunkserver

CHUNK
CHUNK
CHUNK



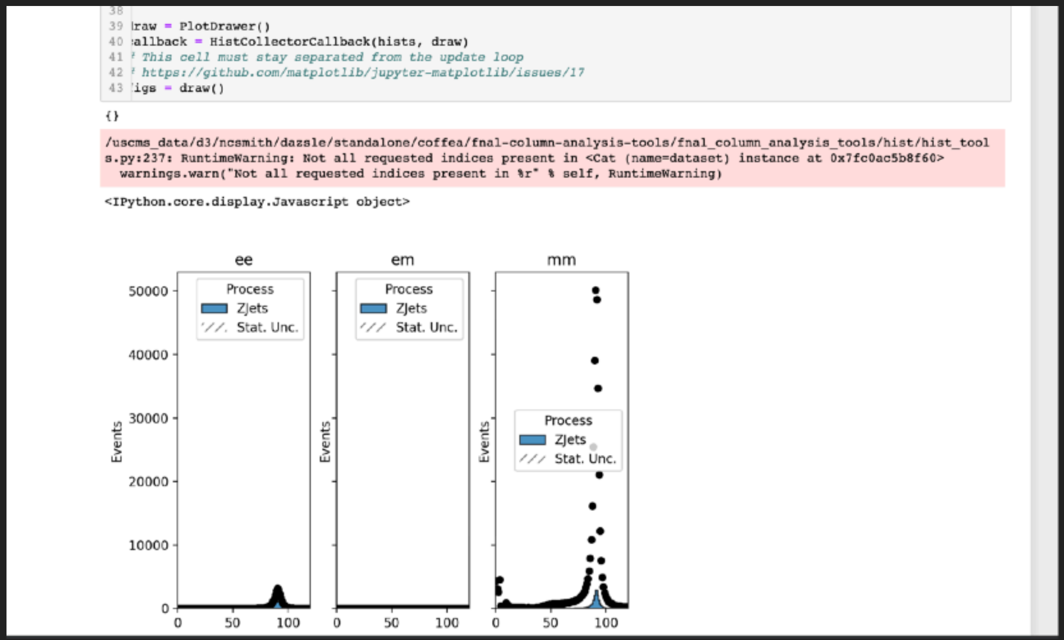
LOOK FAMILIAR?

https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

MISCELLANEOUS

DATA SCIENCE AND NOTEBOOKS

- Many people (me) want to minimize "time-to-plot"
  - AKA "time-to-insight" in industry
- Web-based, iterative data analysis/data science has become extremely popular



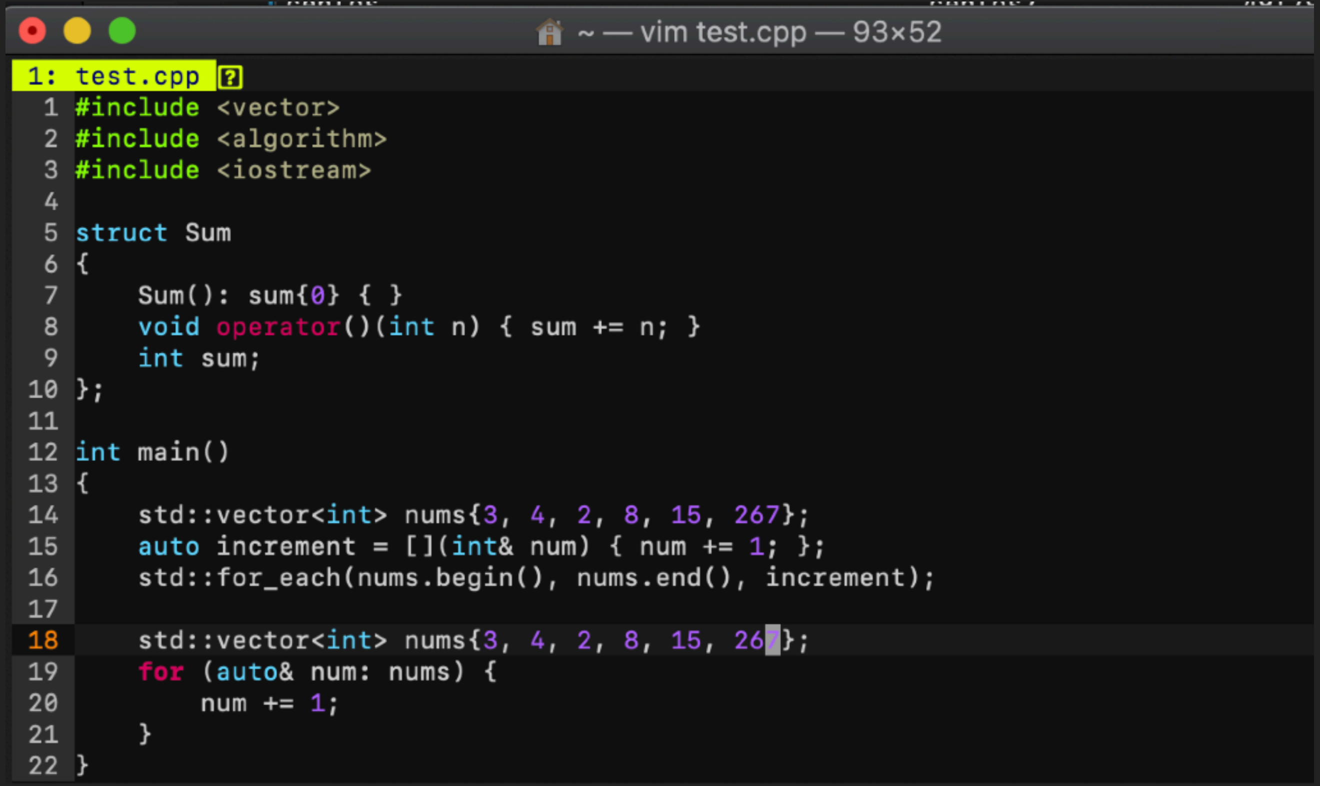
```
raw = PlotDrawer()
allback = HistCollectorCallback(hists, draw)
 This cell must stay separated from the update loop
 https://github.com/matplotlib/jupyter-matplotlib/issues/17
igs = draw()
```

```
{}
```

```
/uscms_data/d3/ncsmith/dazsle/standalone/coffea/fnal-column-analysis-tools/fnal_column_analysis_tools/hist/hist_tool
s.py:237: RuntimeWarning: Not all requested indices present in <Cat (name=dataset) instance at 0x7fc0ac5b8f60>
  warnings.warn("Not all requested indices present in %r" % self, RuntimeWarning)
```

```
<IPython.core.display.Javascript object>
```

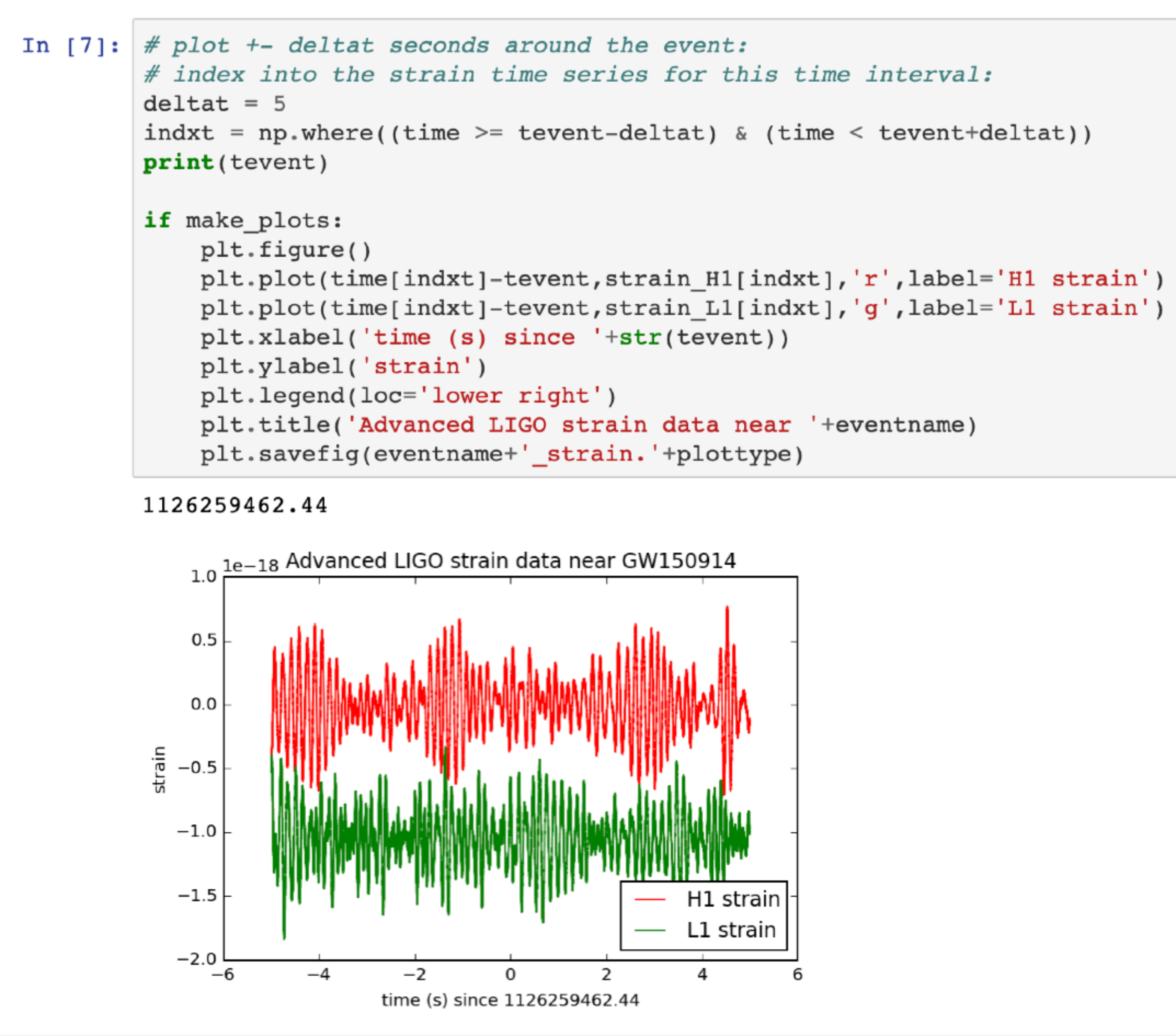
or



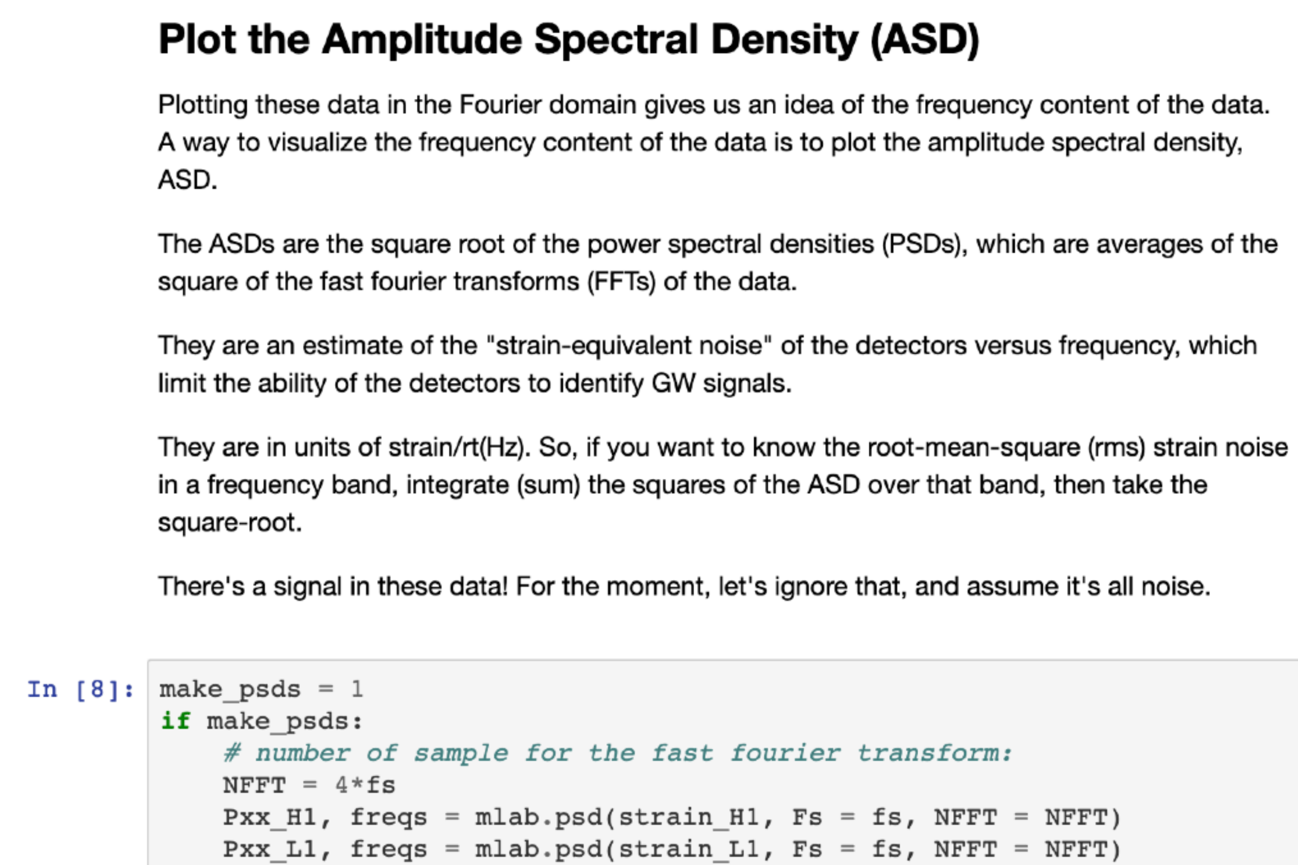
```cpp
#include <vector>
#include <algorithm>
#include <iostream>

struct Sum
{
    Sum(): sum{0} { }
    void operator()(int n) { sum += n; }
    int sum;
};

int main()
{
    std::vector<int> nums{3, 4, 2, 8, 15, 267};
    auto increment = [](int& num) { num += 1; };
    std::for_each(nums.begin(), nums.end(), increment);

    std::vector<int> nums{3, 4, 2, 8, 15, 267};
    for (auto& num: nums) {
        num += 1;
    }
}
```

DATA SCIENCE AND NOTEBOOKS 2

- Reproducibility, presentable



```
In [7]: # plot +- deltat seconds around the event:
        # index into the strain time series for this time interval:
        deltat = 5
        indxt = np.where((time >= tevent-deltat) & (time < tevent+deltat))
        print(tevent)

        if make_plots:
            plt.figure()
            plt.plot(time[indxt]-tevent,strain_H1[indxt],'r',label='H1 strain')
            plt.plot(time[indxt]-tevent,strain_L1[indxt],'g',label='L1 strain')
            plt.xlabel('time (s) since '+str(tevent))
            plt.ylabel('strain')
            plt.legend(loc='lower right')
            plt.title('Advanced LIGO strain data near '+eventname)
            plt.savefig(eventname+'_strain.'+plottype)

        1126259462.44
```



Plot the Amplitude Spectral Density (ASD)

Plotting these data in the Fourier domain gives us an idea of the frequency content of the data. A way to visualize the frequency content of the data is to plot the amplitude spectral density, ASD.

The ASDs are the square root of the power spectral densities (PSDs), which are averages of the square of the fast fourier transforms (FFTs) of the data.

They are an estimate of the "strain-equivalent noise" of the detectors versus frequency, which limit the ability of the detectors to identify GW signals.

They are in units of strain/rt(Hz). So, if you want to know the root-mean-square (rms) strain noise in a frequency band, integrate (sum) the squares of the ASD over that band, then take the square-root.

There's a signal in these data! For the moment, let's ignore that, and assume it's all noise.

```
In [8]: make_psds = 1
        if make_psds:
            # number of sample for the fast fourier transform:
            NFFT = 4*fs
            Pxx_H1, freqs = mlab.psd(strain_H1, Fs = fs, NFFT = NFFT)
            Pxx_L1, freqs = mlab.psd(strain_L1, Fs = fs, NFFT = NFFT)
```

Static Version

https://www.gw-openscience.org/GW150914data/LOSC_Event_tutorial_GW150914.html

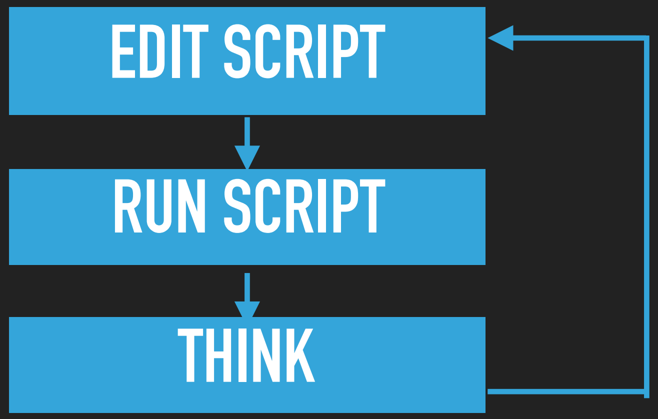
Dynamic Version!

http://beta.mybinder.org/repo/losc-tutorial/LOSC_Event_tutorial

https://nbviewer.jupyter.org/github/losc-tutorial/LOSC_Event_tutorial/blob/master/index.ipynb

DATA SCIENCE AND NOTEBOOKS 3

- Different Paradigm



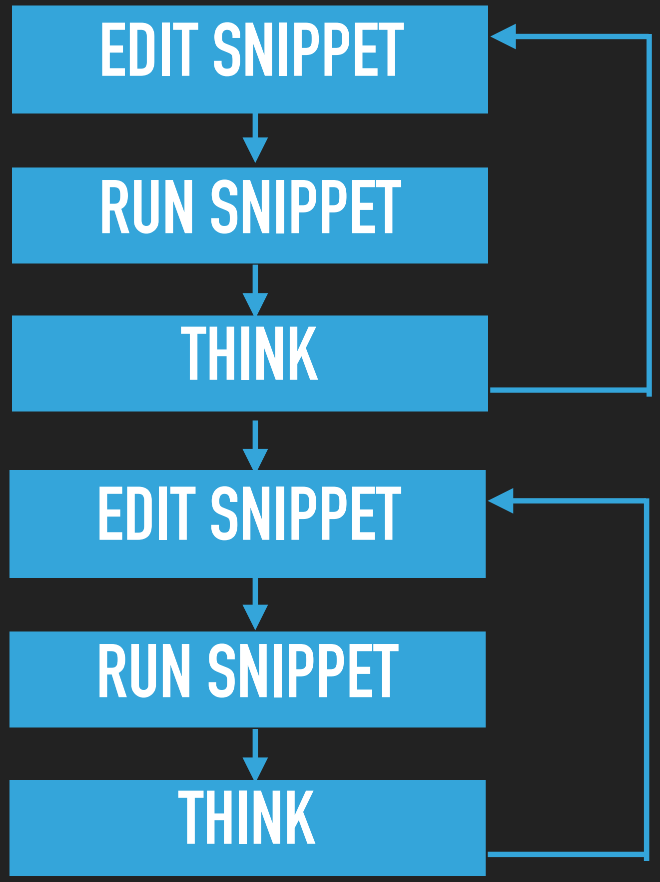
EDIT SCRIPT

RUN SCRIPT

THINK

Instead of running a script from the beginning each time, run only the modified snippets

Can greatly accelerate analysis



EDIT SNIPPET

RUN SNIPPET

THINK

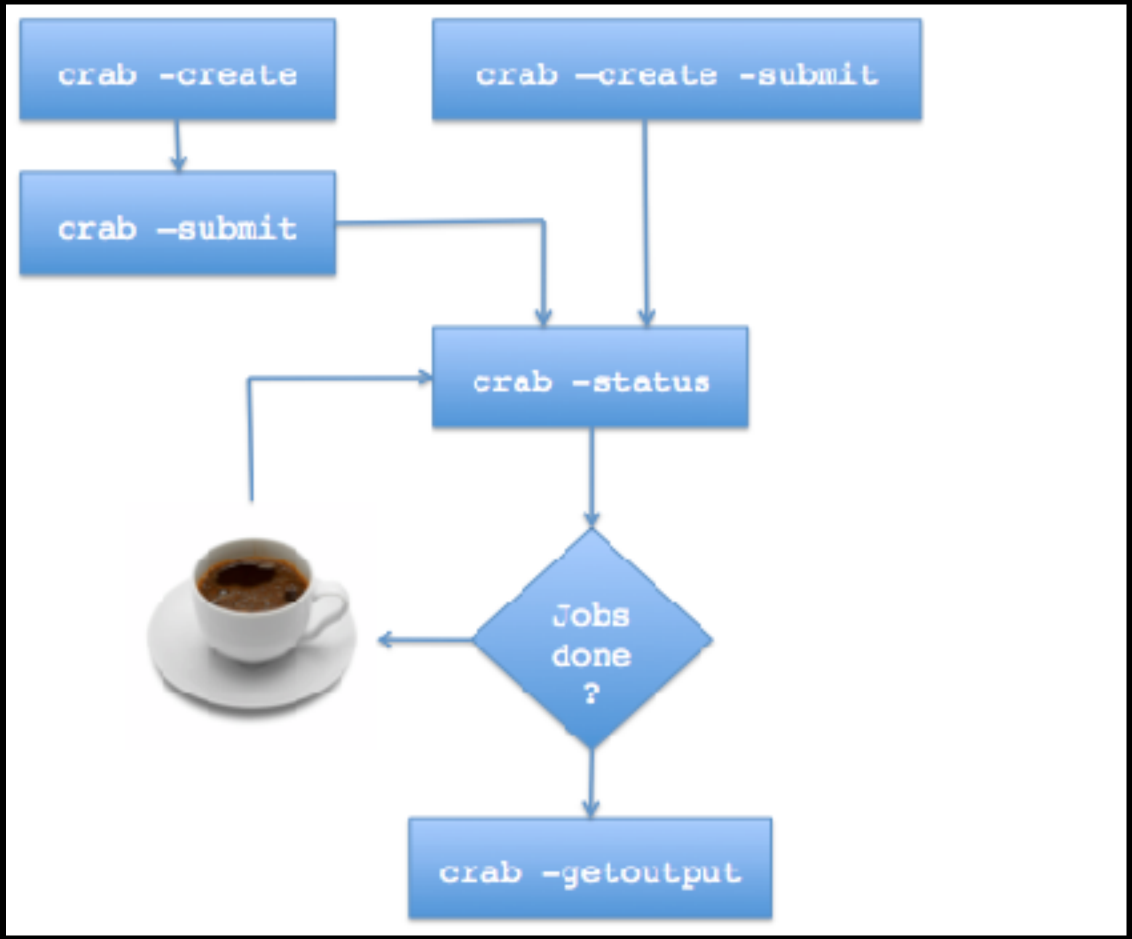
EDIT SNIPPET

RUN SNIPPET

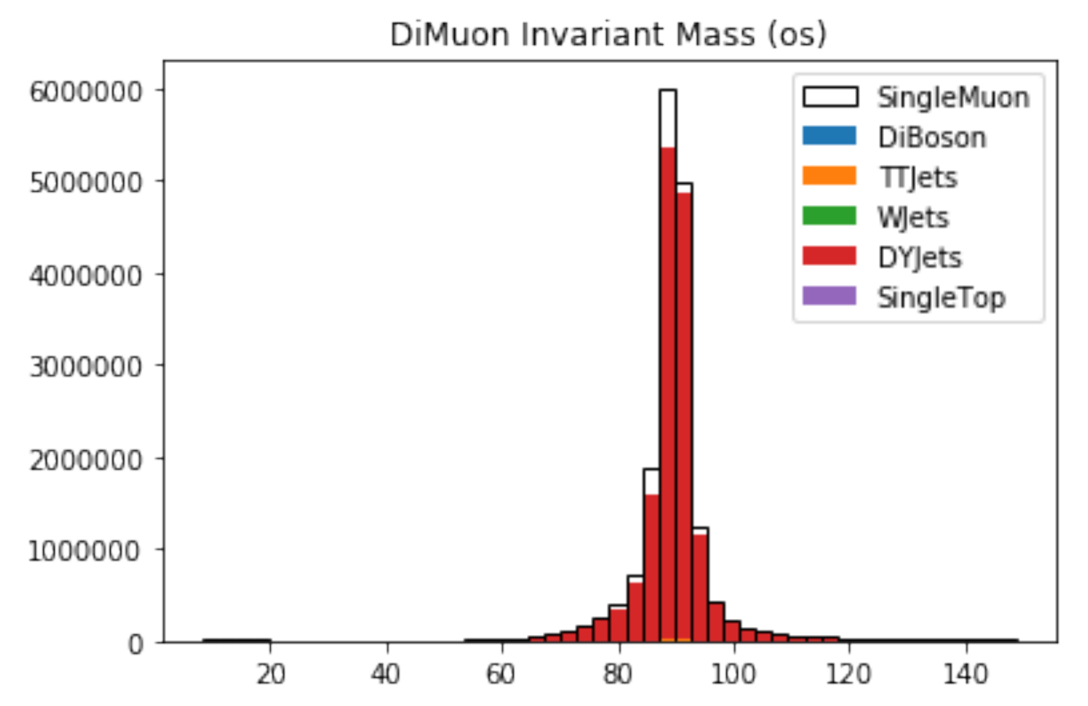
THINK

Latency Hurts

- Interactive tasks are sensitive to latency.
- Analysis is often "hit enter, come back in an hour"
- Context-switching is inefficient for humans
- Can we lower the "reducible" latency?



Analysis with Spark



931M events in ~90 secs

Change a cut

New plots in ~15 secs

- Analysis-level non-flat ntuples
  - ~20TB total size
- Time includes the full chain
  - Core aquisition, File I/O, cuts, flattening, histogramming, plotting
- 350 cores, HDFS on spinning HDDs

RECAP

- Data has exploded while machine performance stalled
    - Both CPU and storage need to be scaled-out
- Scaling out -> Distributed Computing = new problems
    - Reliability/Performance/Correctness
- New architectures
    - Shared-nothing/declarative/functional
- Increased focus on latency, not throughput