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 = 1 BILLION TERABYTES

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

2010: 1.2 ZETTABYTES

2018: 33 ZETTABYTES

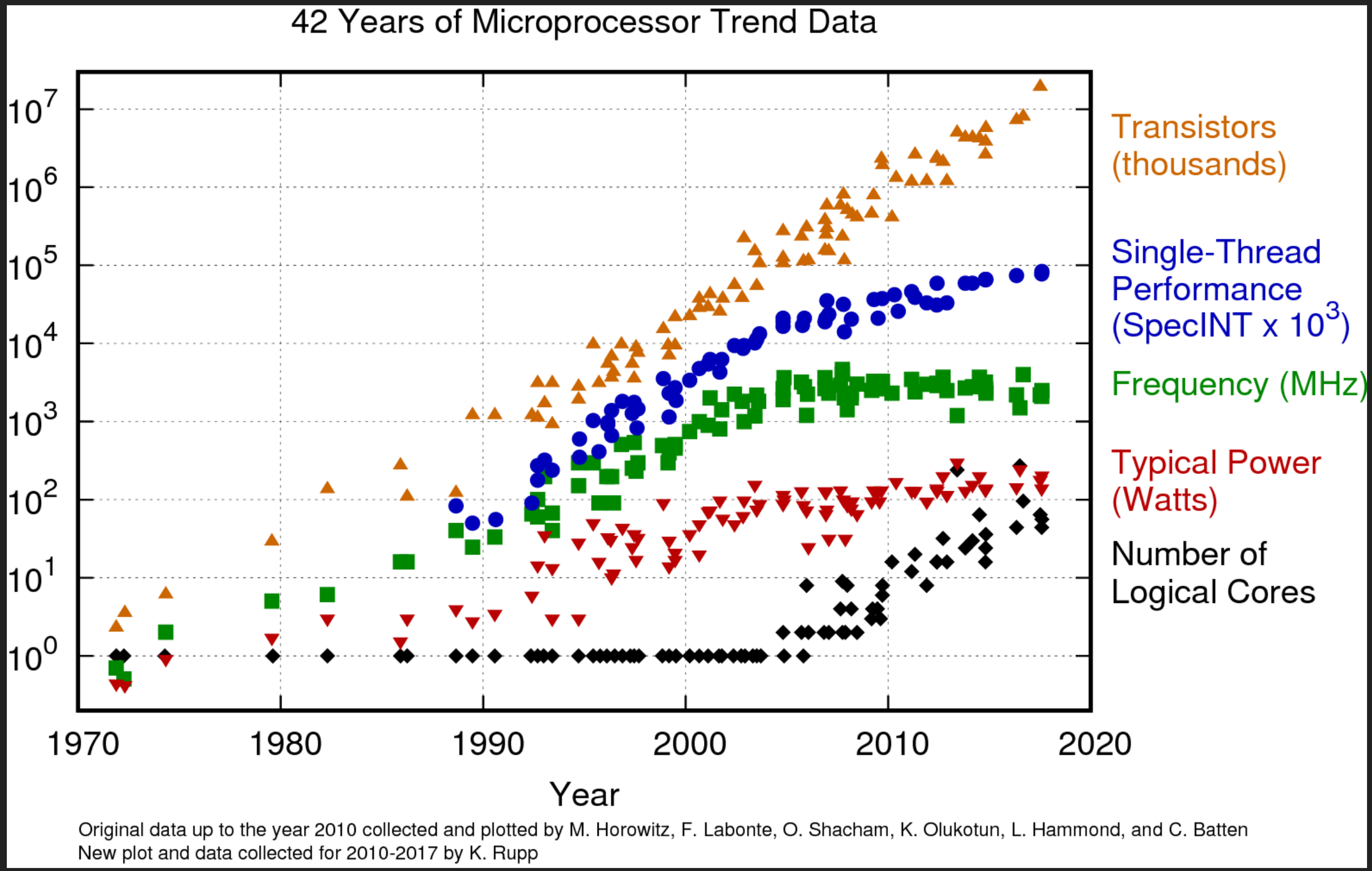
2025: 178 ZETTABYTES

A 61% annual increase!

*1 ZETTABYTE = 1BILLION TERABYTES

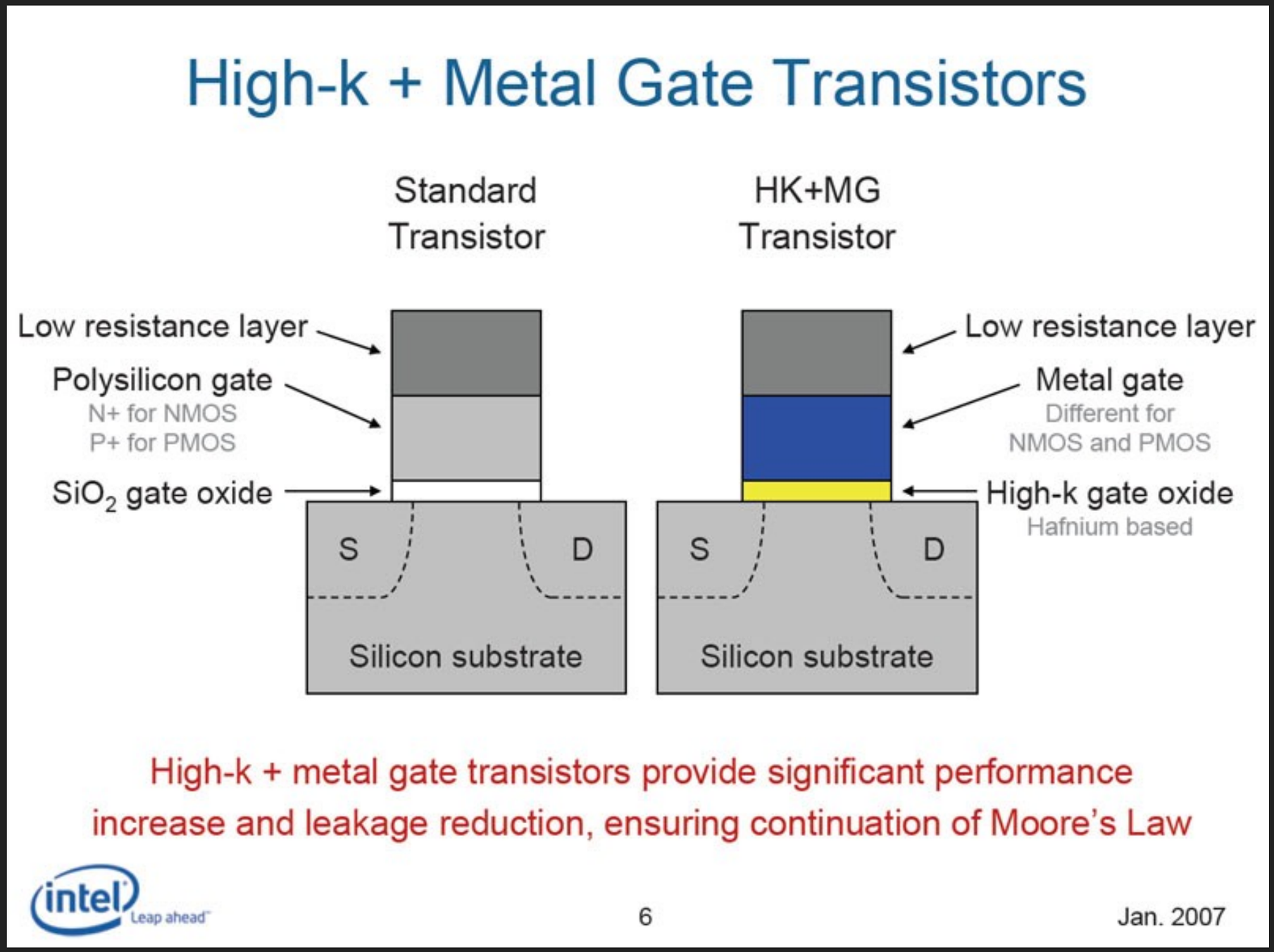
https://www.seagate.com/our-story/data-age-2025/

INDIVIDUAL THREAD PERFORMANCE IS ~FLATTENING



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

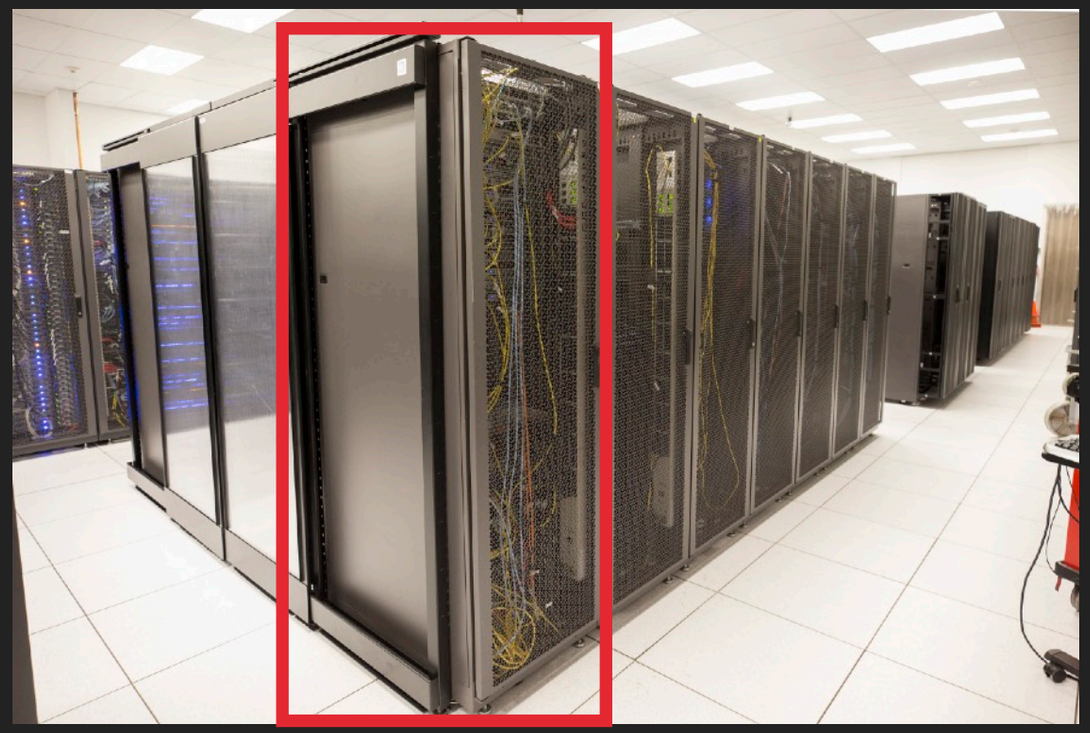
CURRENT LEAKAGE



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!

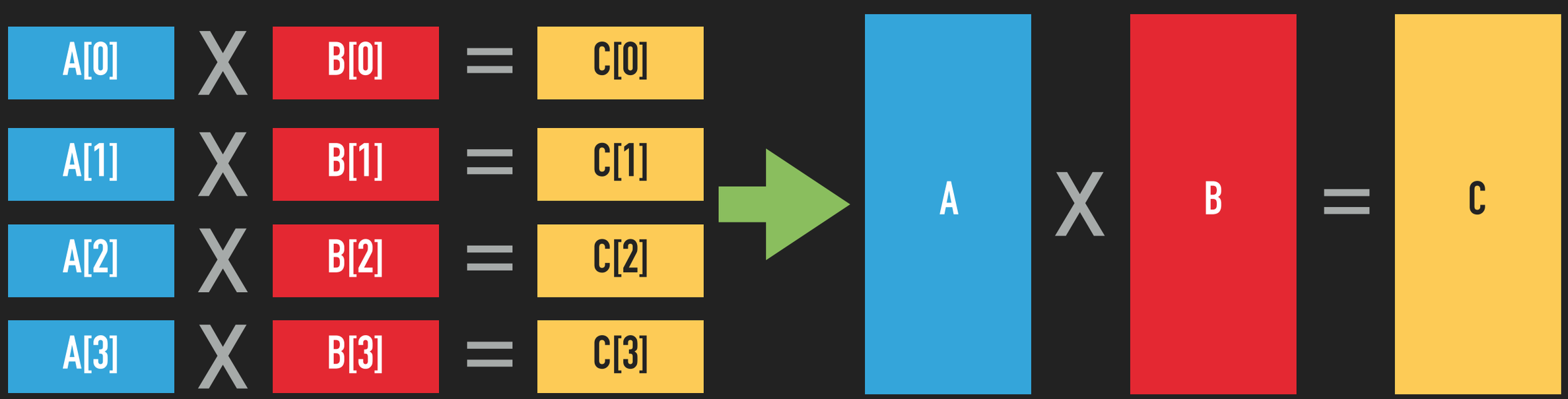
SINGLE-THREAD PERFORMANCE

- 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



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

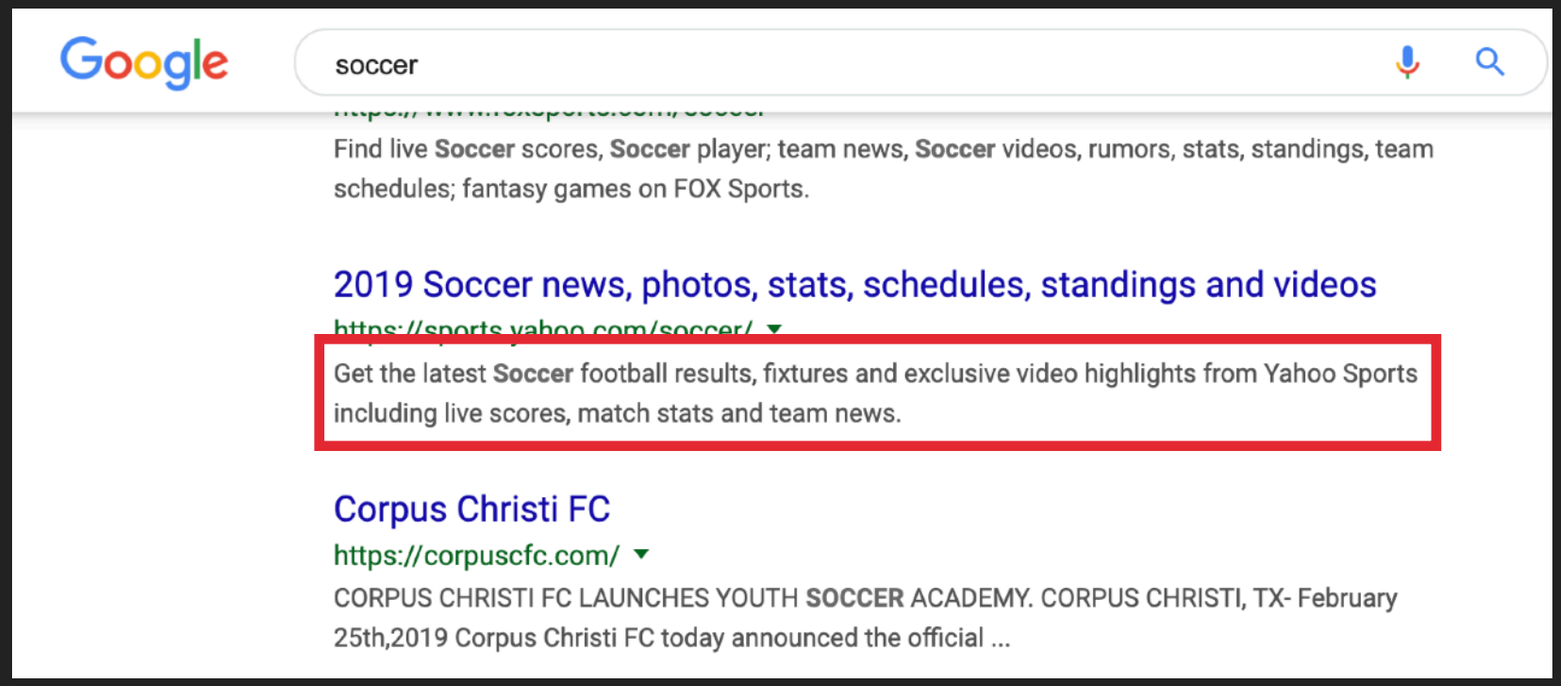
A PERFECT STORM FOR A FUNDAMENTAL CHANGE

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



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

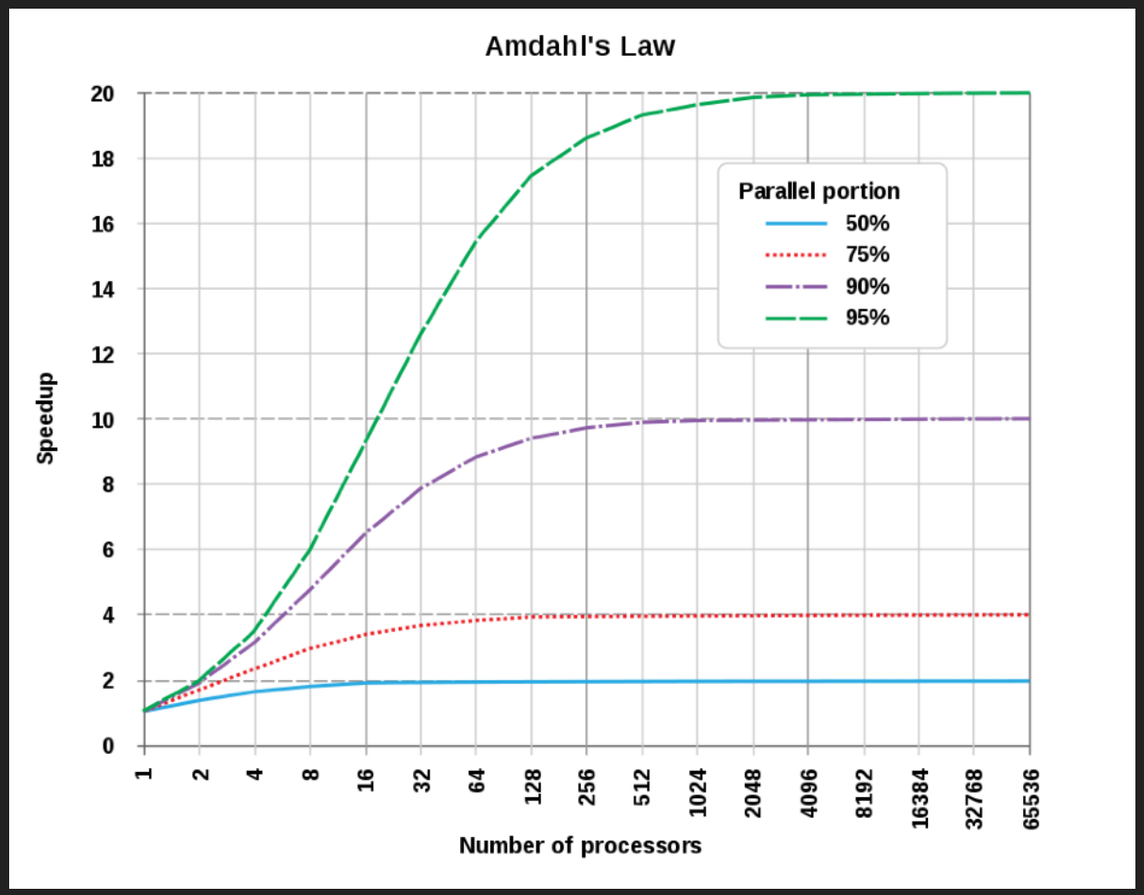
PROCESSING THESE DATA VOLUMES

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

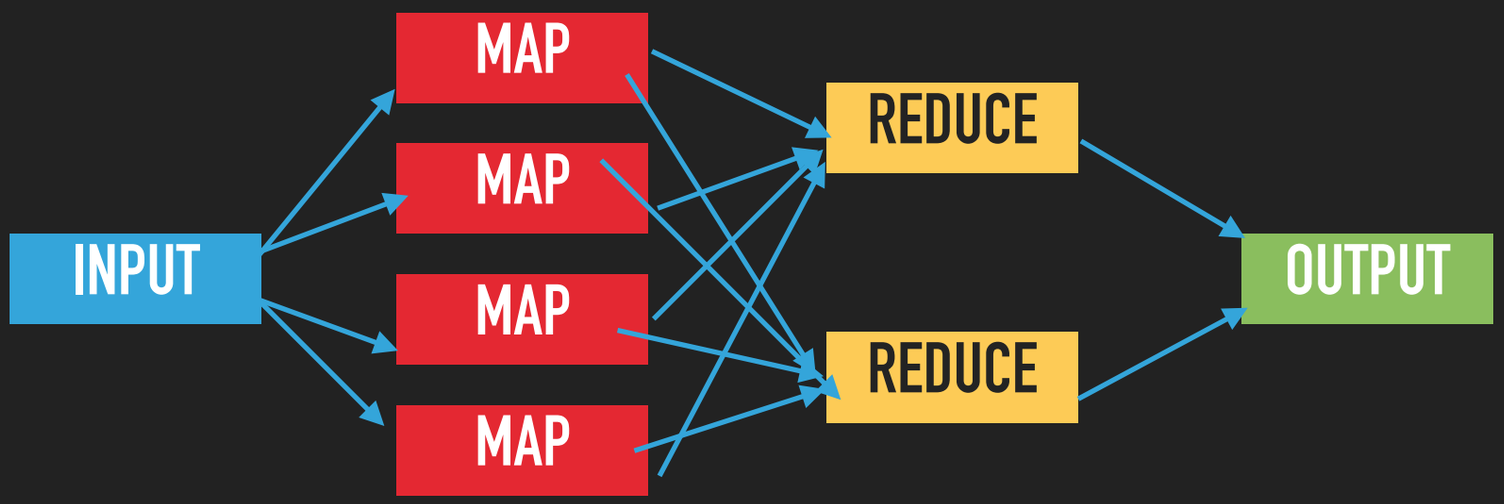
- Performance scales sub-linearly with resources
  - 2x the CPUs necessarily produces <2x the performance



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

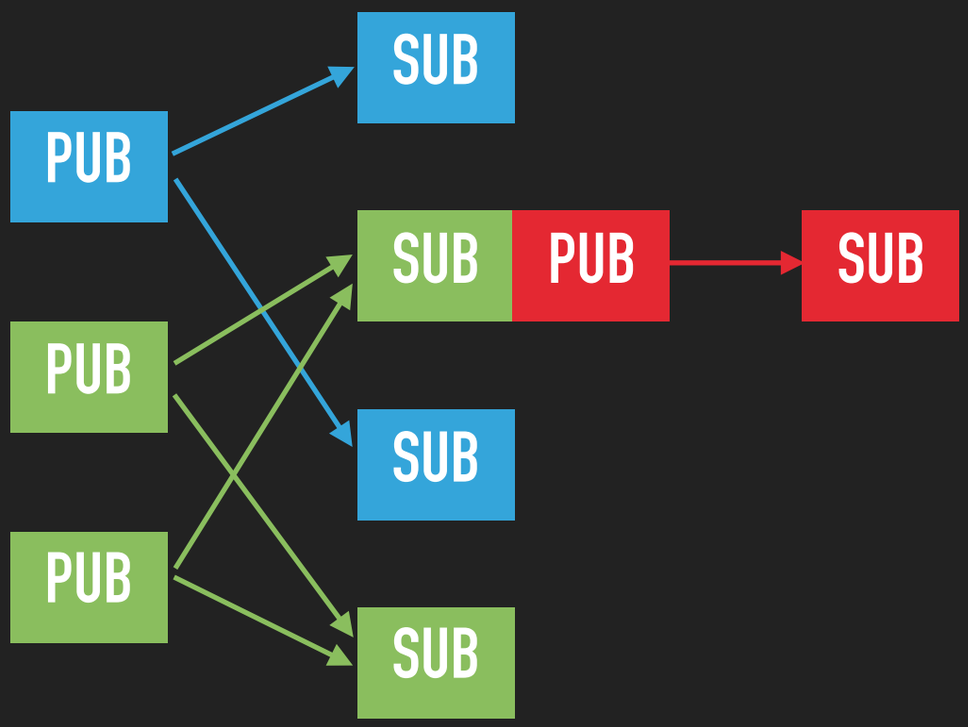
DIVIDE AND CONCUR

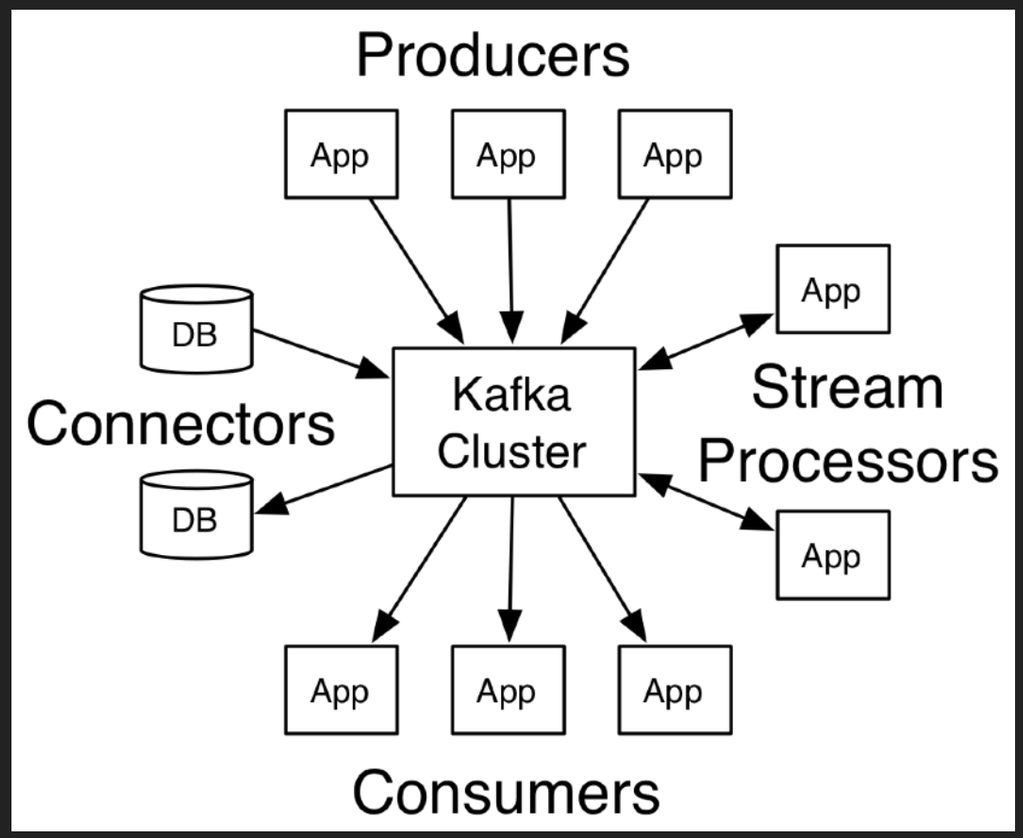
- Split input into N pieces
- Do "something" over each piece
- Shuffle/sort/combine intermediate outputs
- Generate final output



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;
}
```

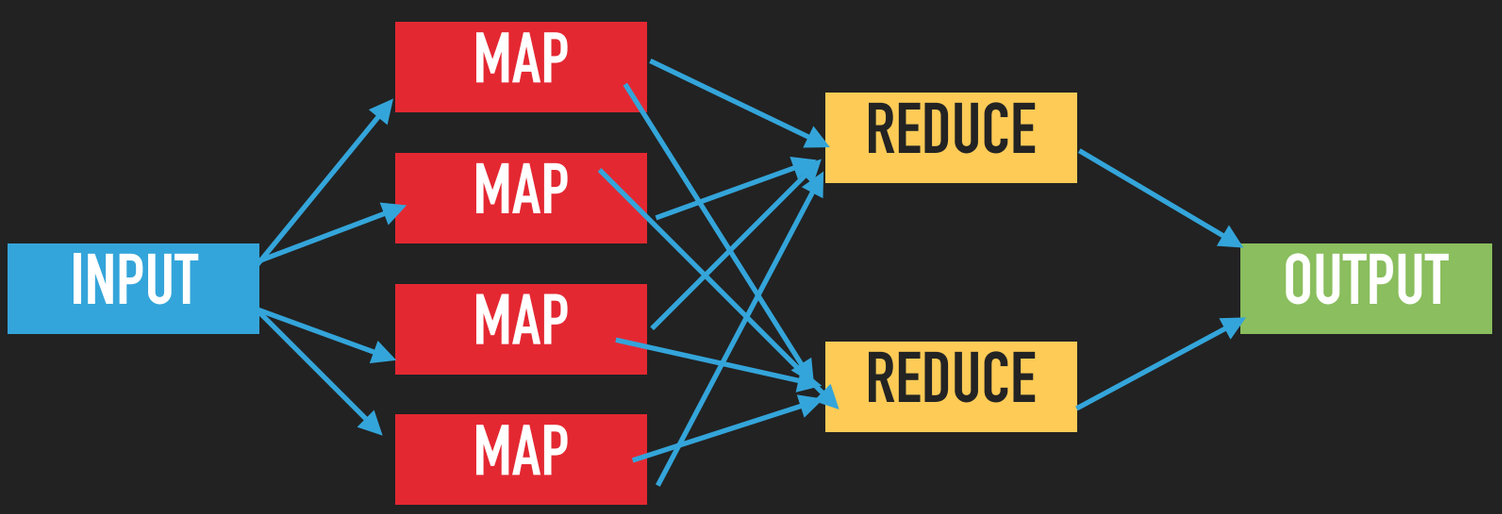
Imperative

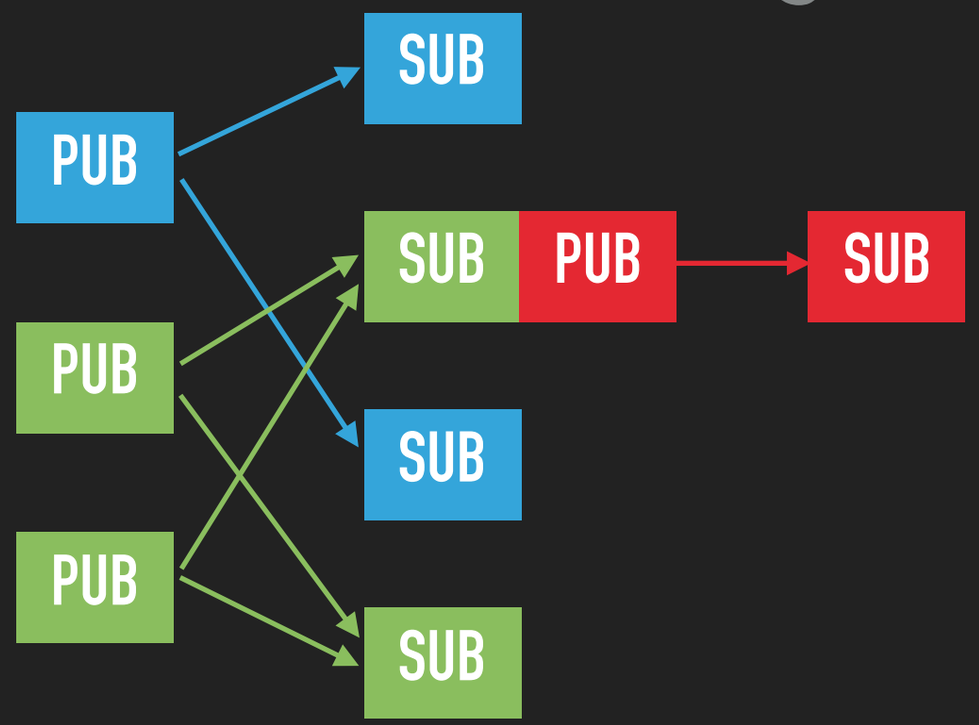
The for loop is "baked in", and adapting this code to run in parallel would involve significant changes/boiler plate

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





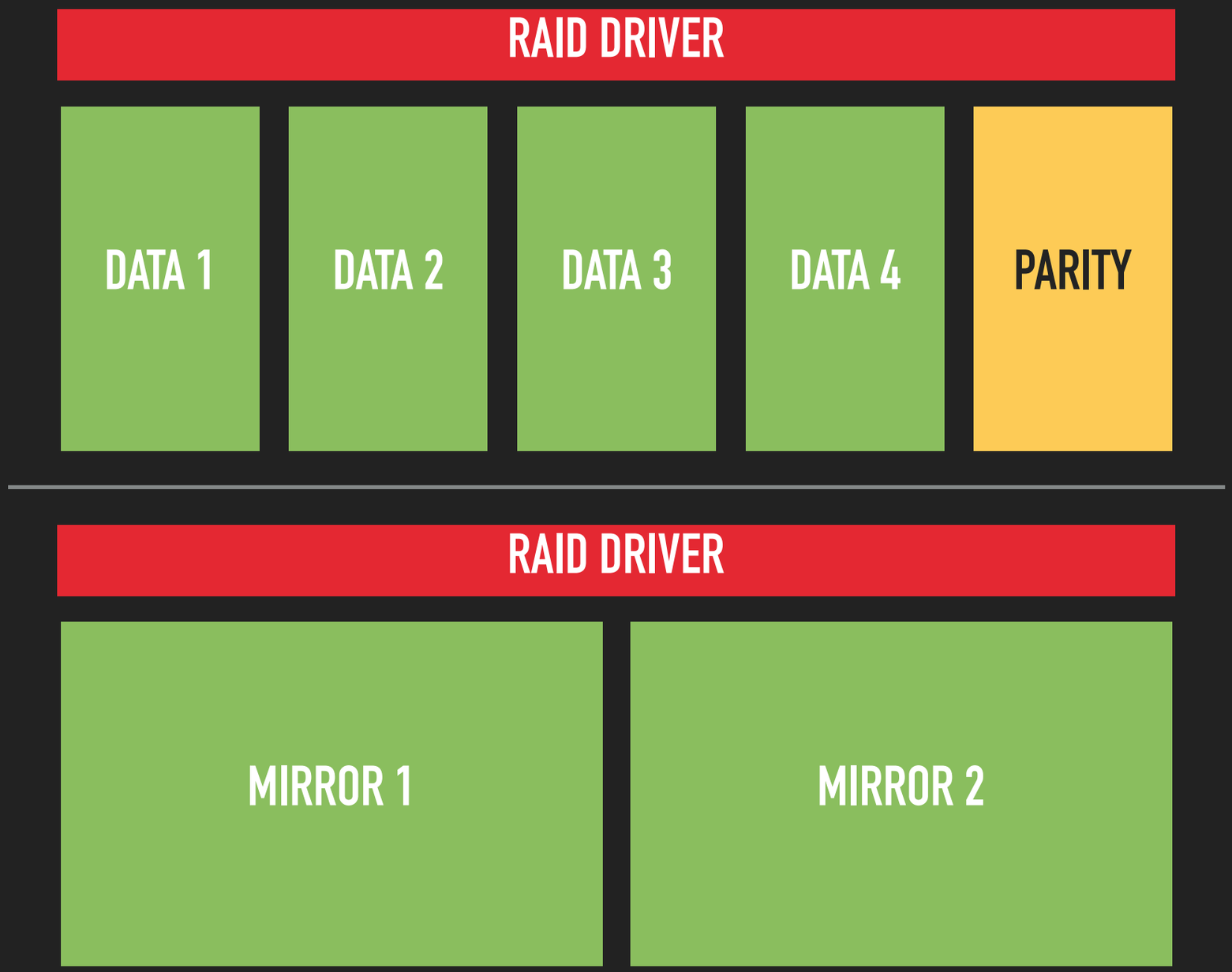
DATA MANAGEMENT

DATA MANAGEMENT

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

RAID-1/5

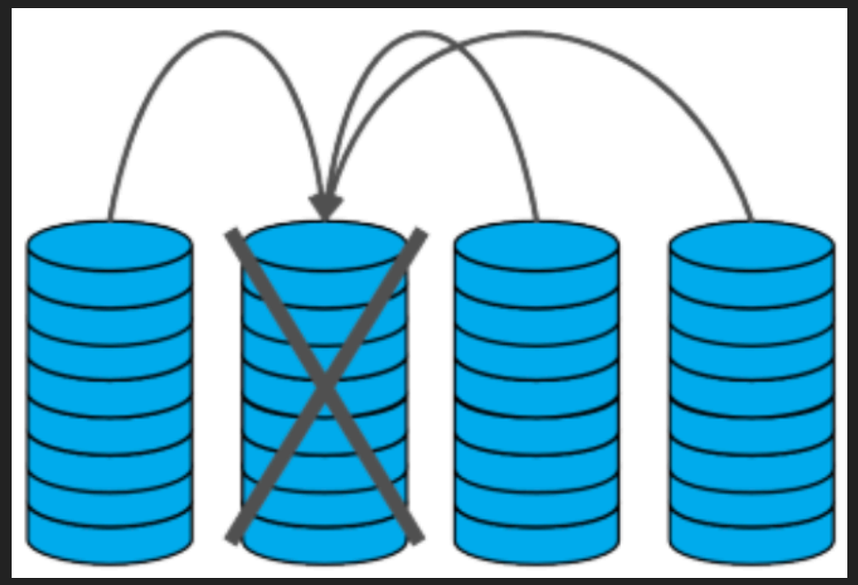
- Common on workstation-class machines

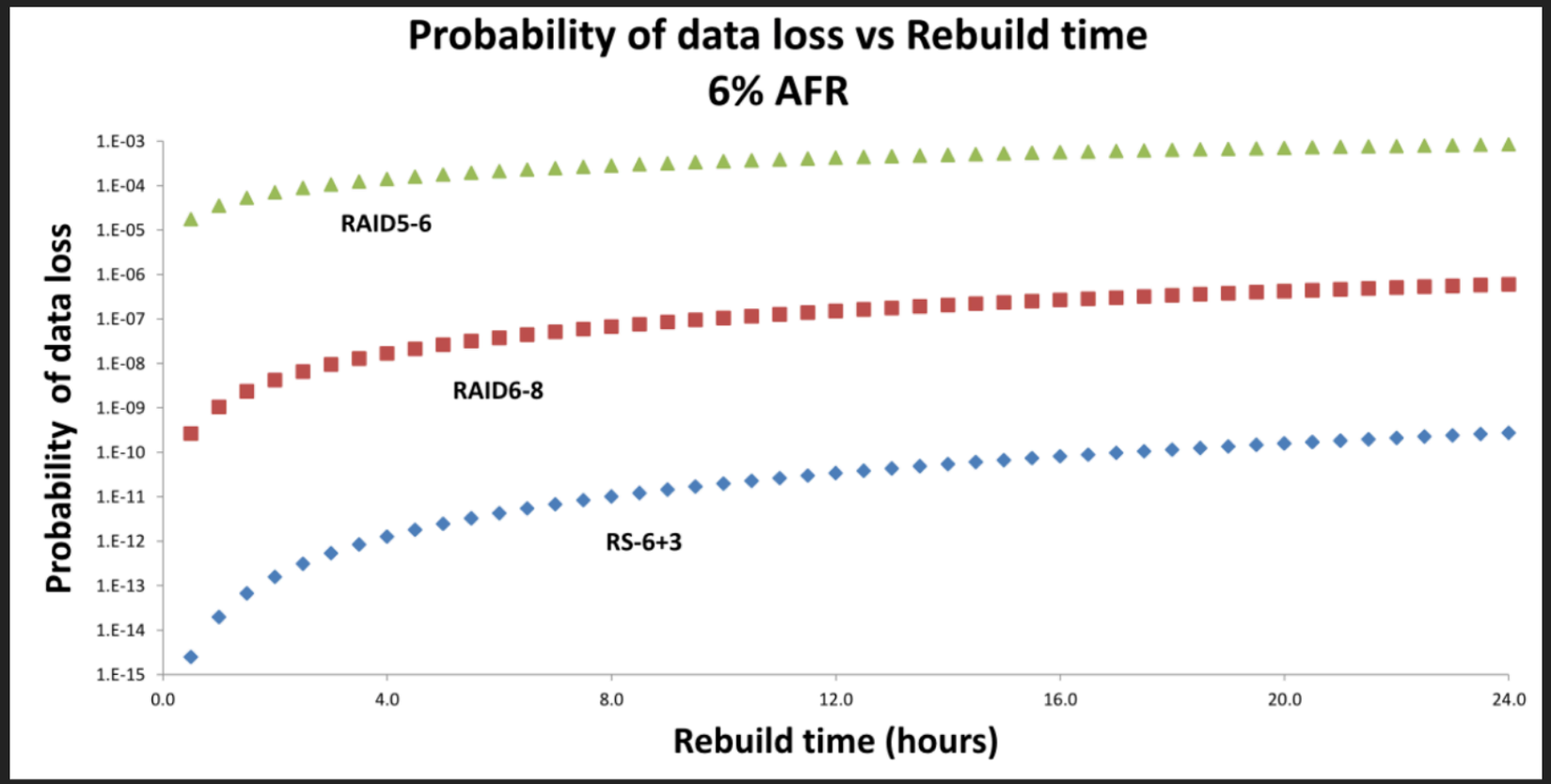


- Limited to single machine
- Bad performance during rebuilds
- Bad probability of double-fault errors

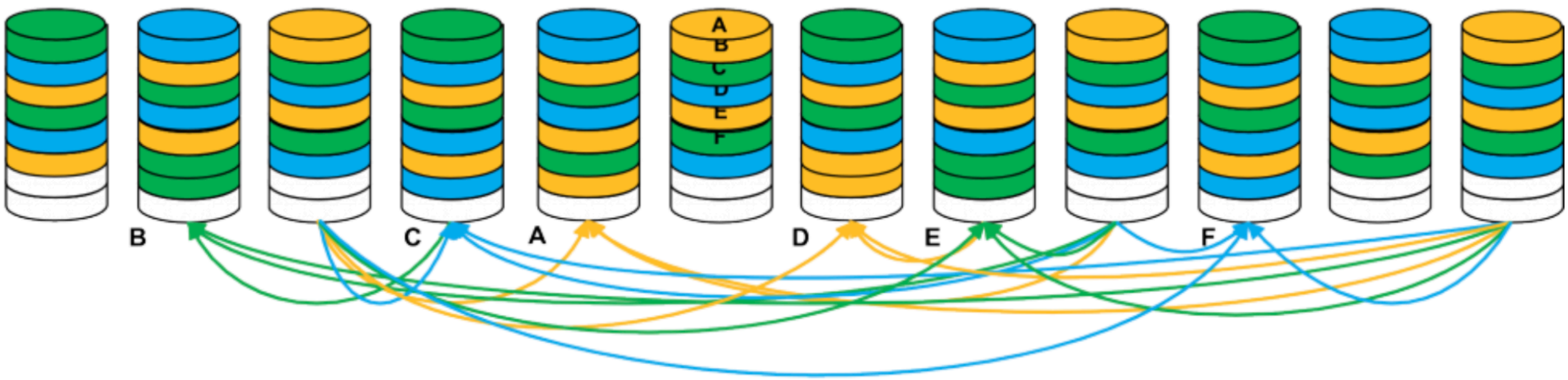
RAID-REPAIR IS DANGEROUS

- 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.

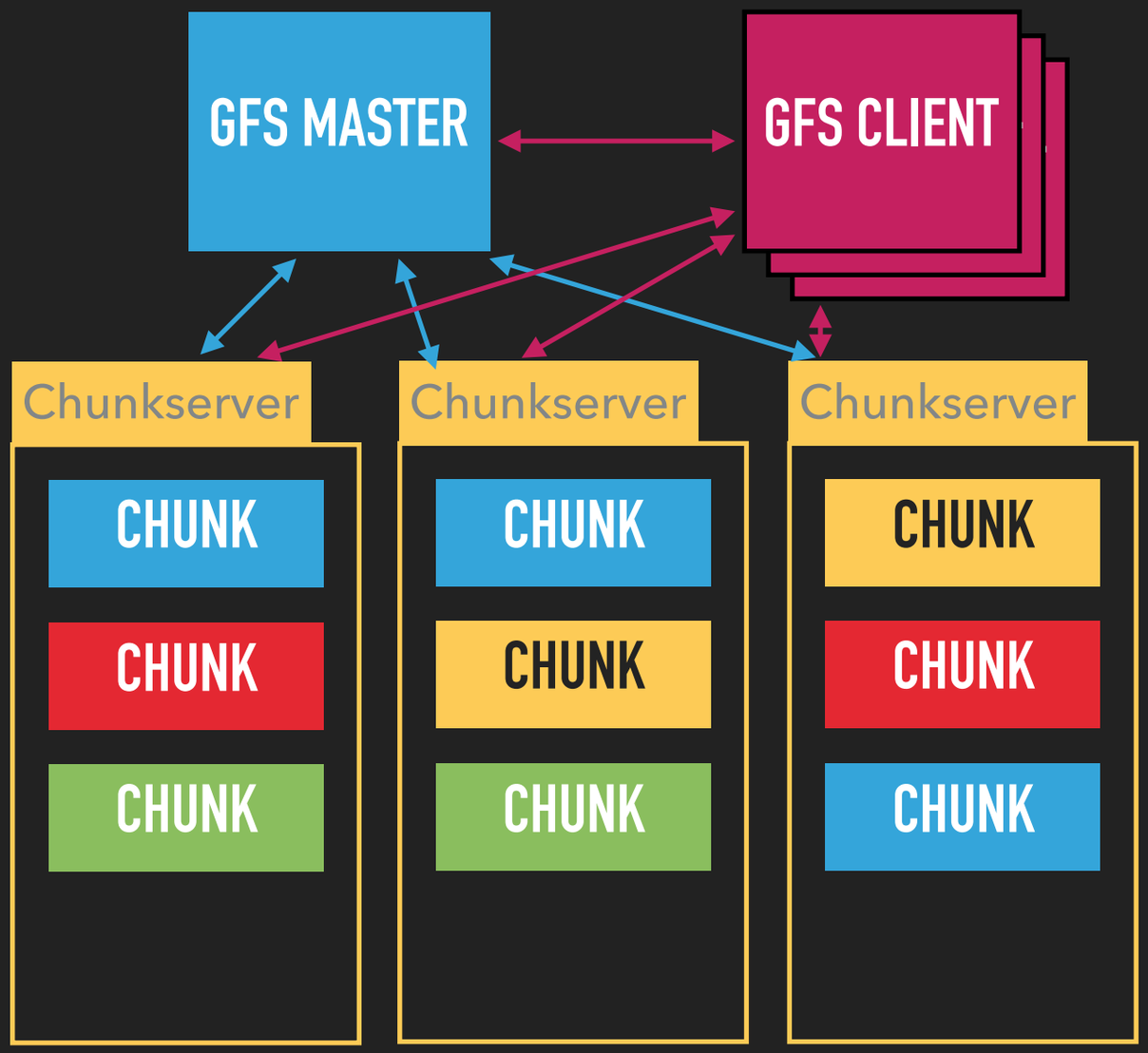
https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

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

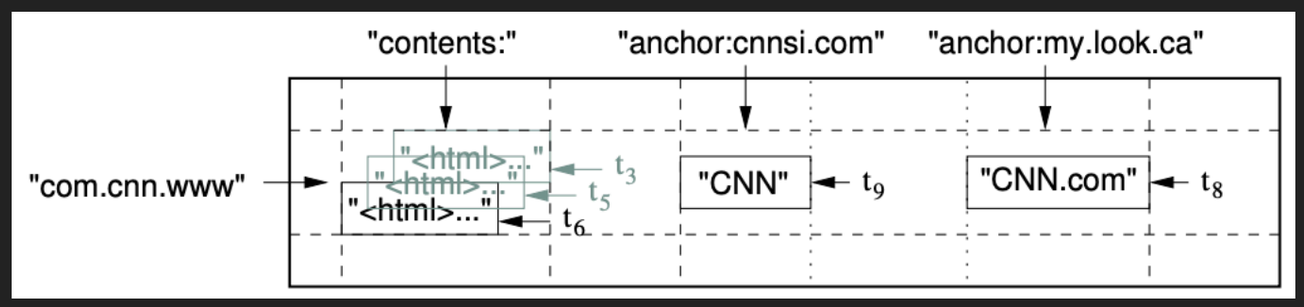
- GFS is arguably the "secret sauce" that helped Google grow as it did
- Many initial services were built over GFS

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.

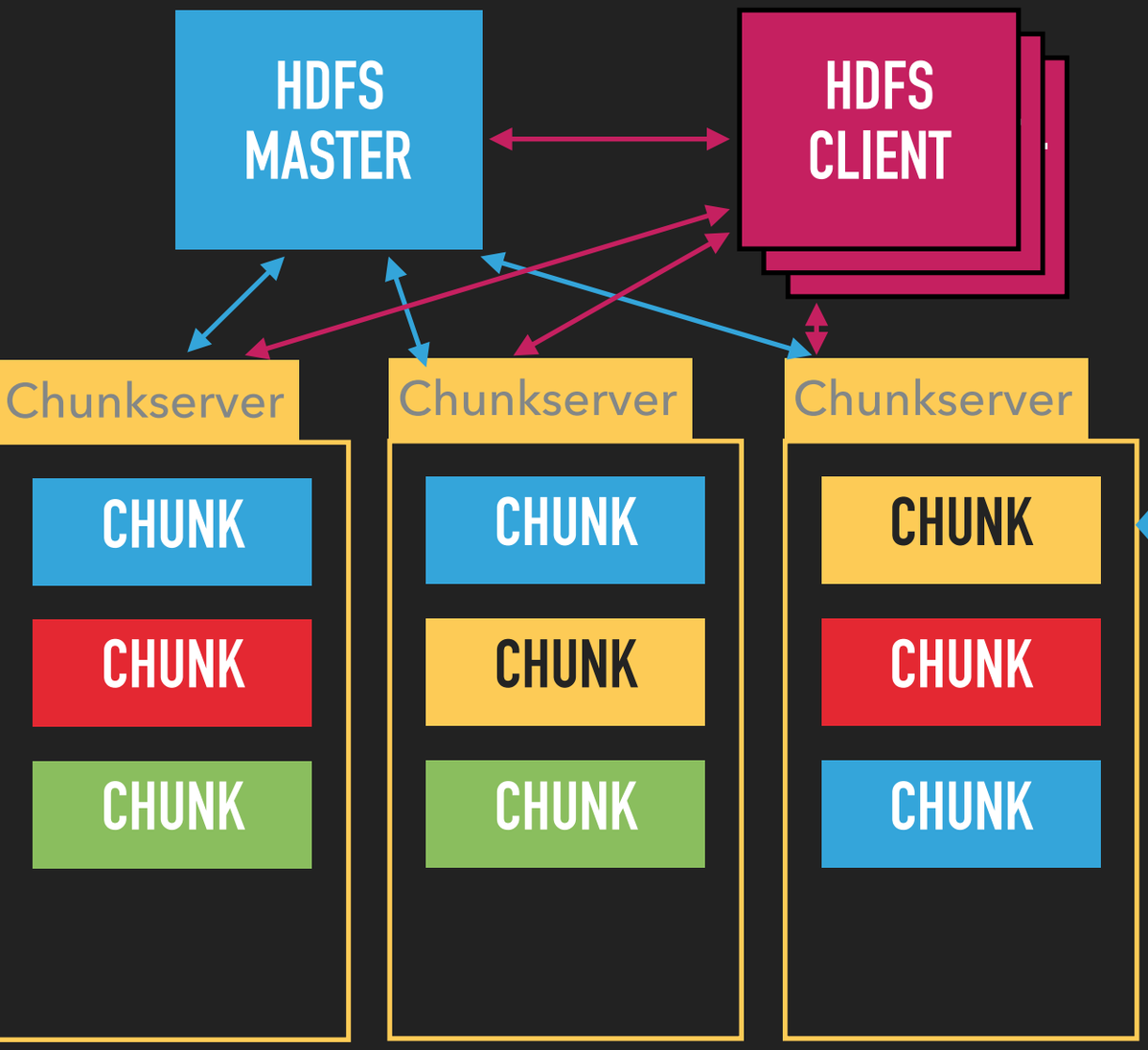
Bigtable paper, 2006

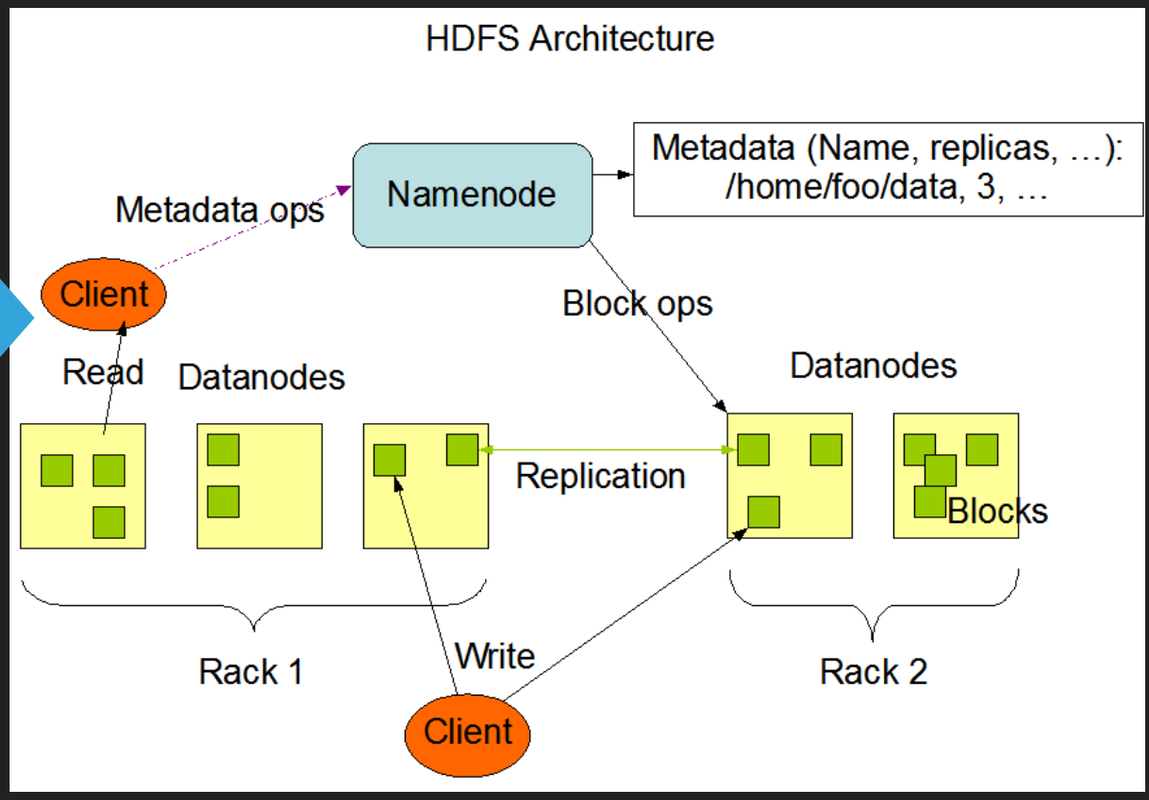
LOOK FAMILIAR?



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

APACHE HDFS ARCHITECTURE





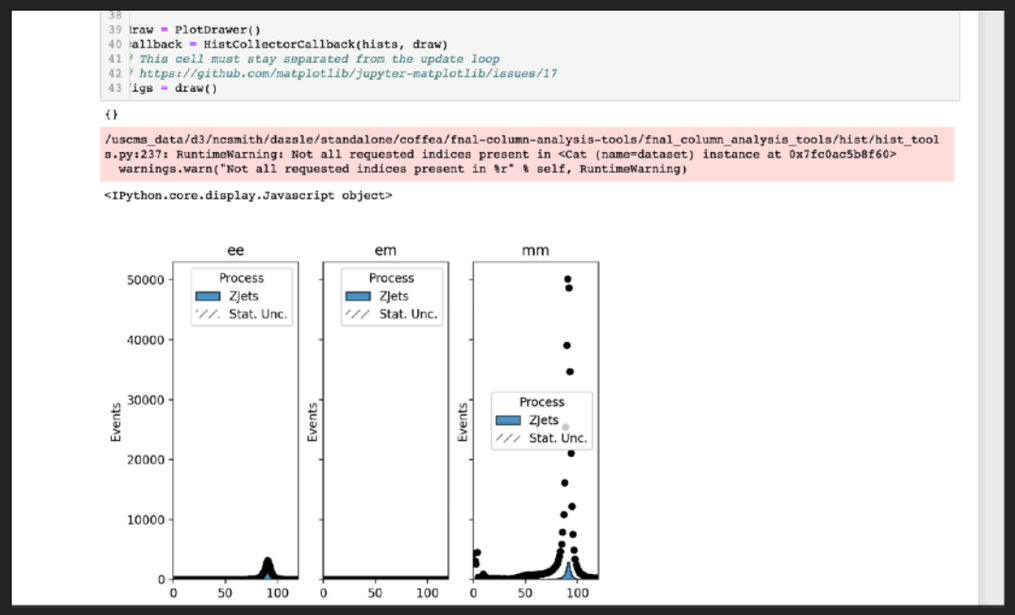
LOOK FAMILIAR?

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

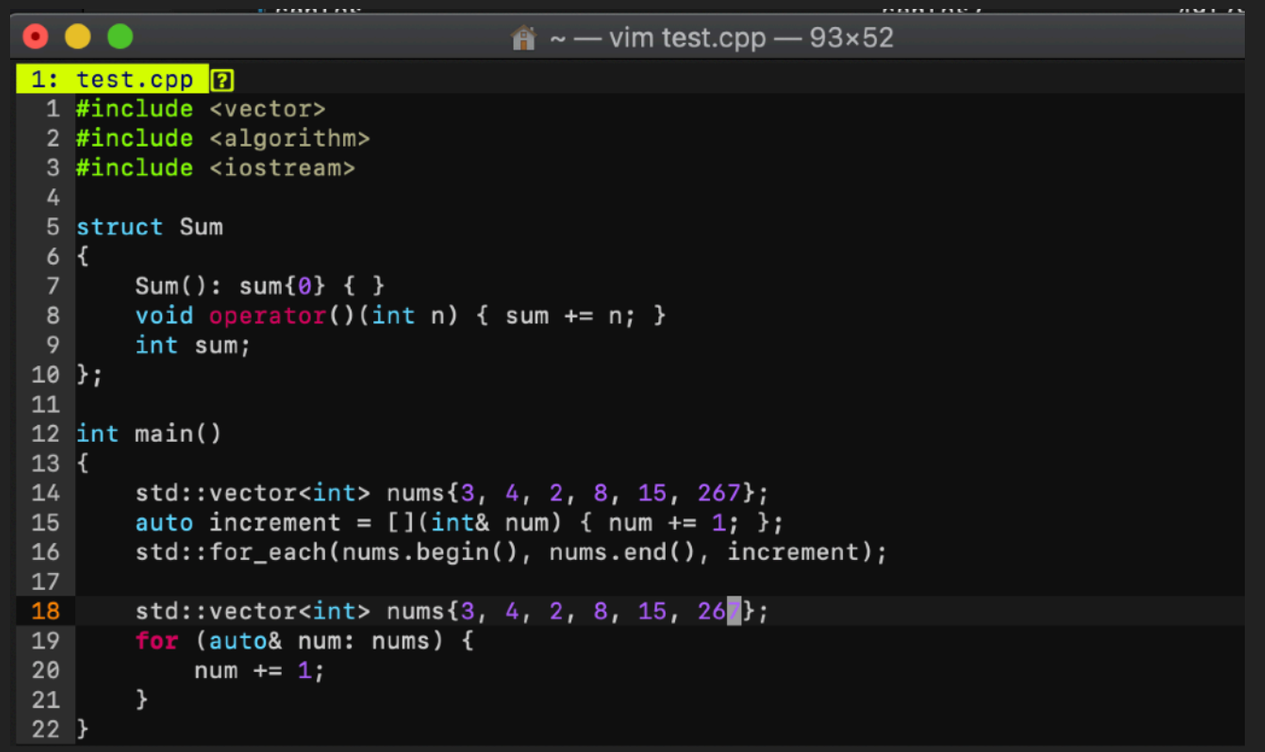
MISCELLANEOUS

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



or

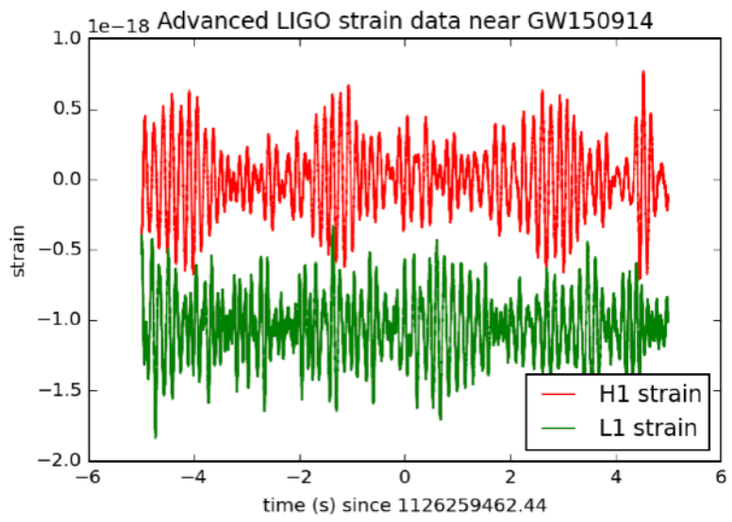


DATA SCIENCE AND NOTEBOOKS 2

- 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

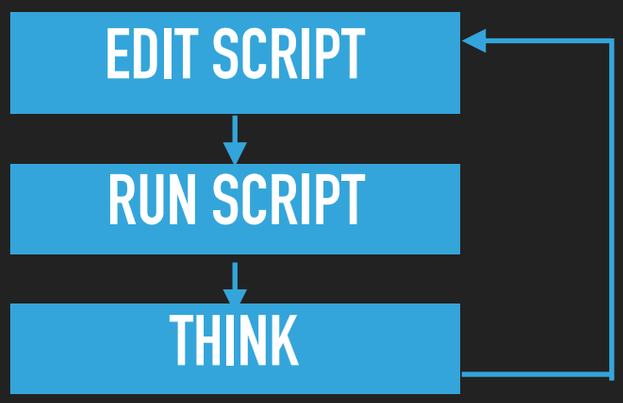
Dynamic Version!

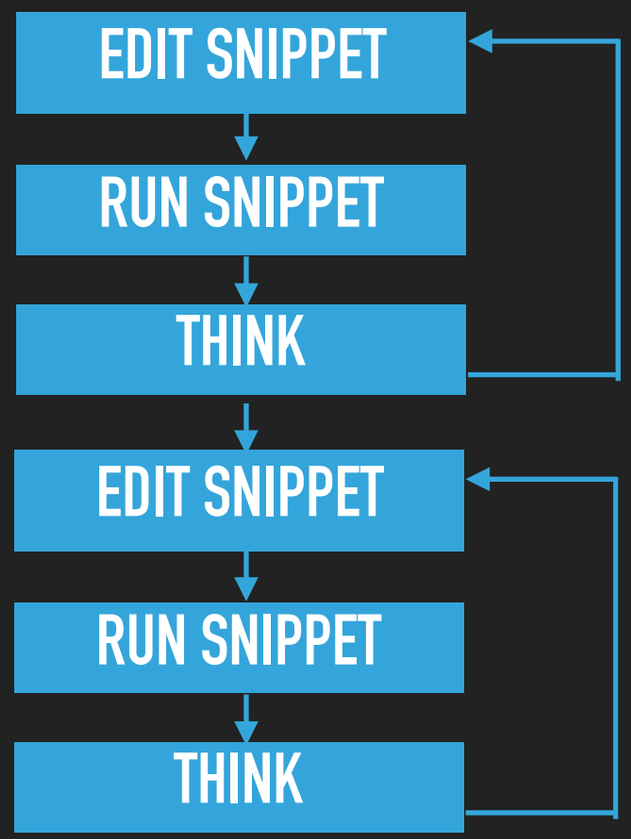
http://beta.mybinder.org/repo/losc-tutorial/LOSC_Event_tutorial

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

DATA SCIENCE AND NOTEBOOKS 3

- Different Paradigm



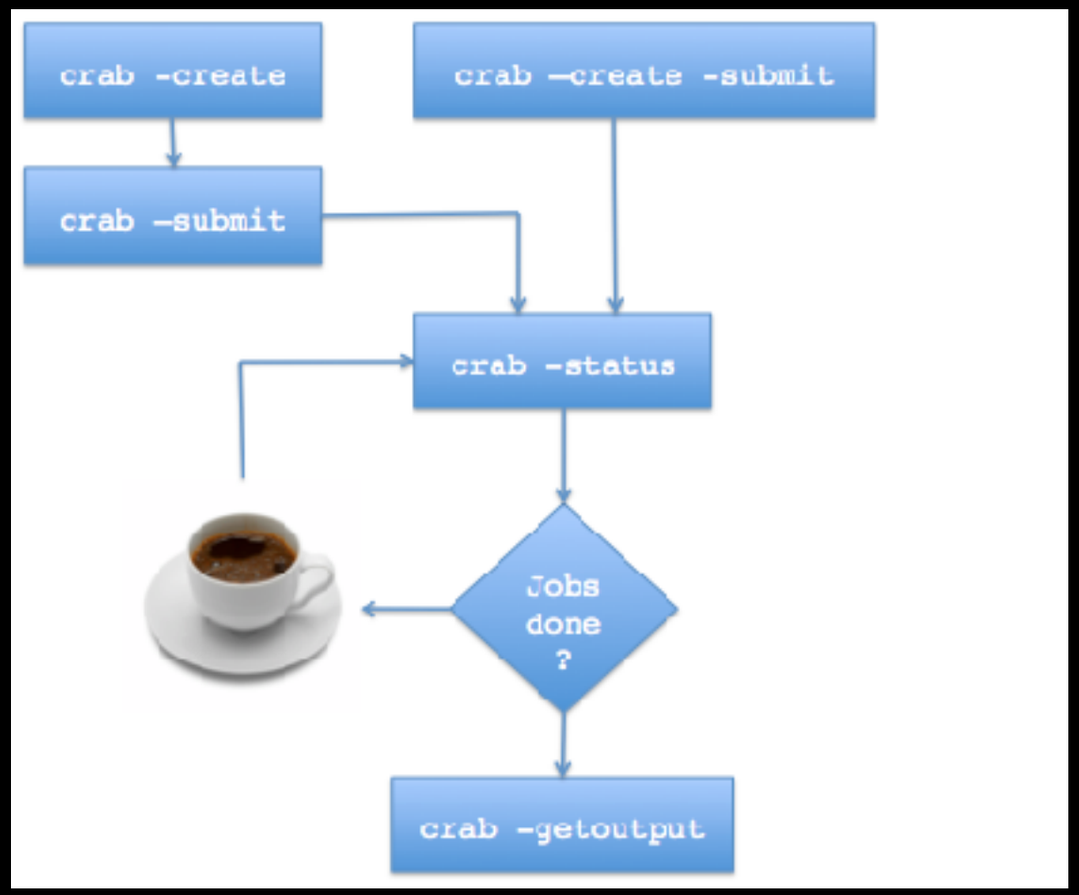


Instead of running a script from the beginning each time, run only the modified snippets

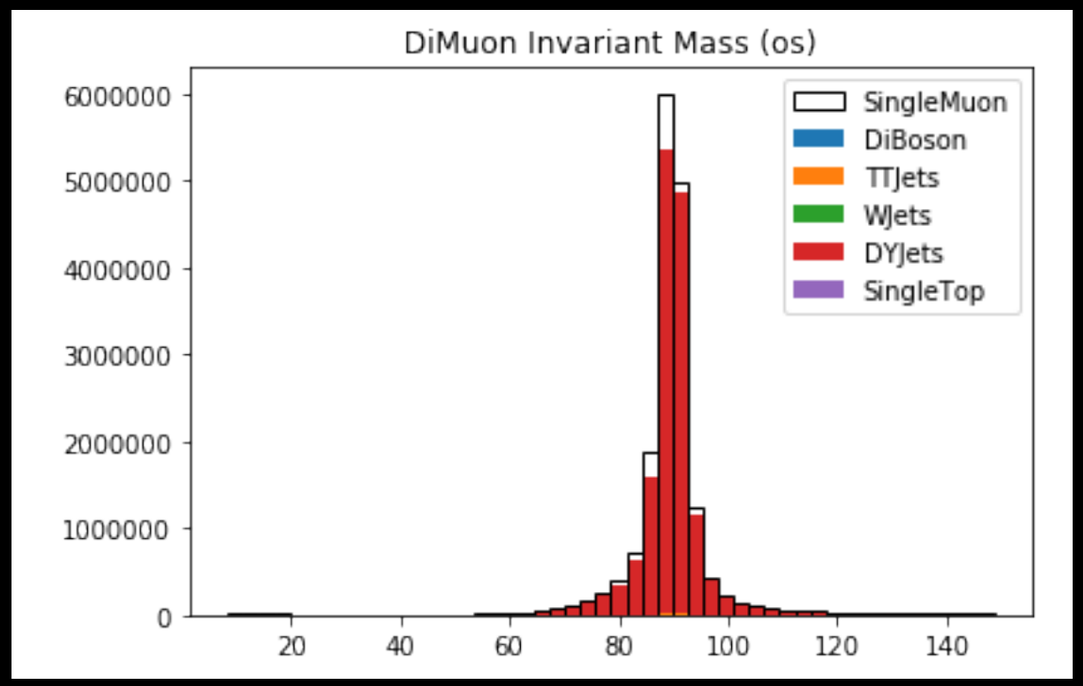
Can greatly accelerate analysis

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

LECTURE 2

CONTEMPORARY BIG DATA IDEAS

CONTENT

- MapReduce
- DataFrames
- Spark
- Dask
- Scientific Python

MAPREDUCE

- Another seminal paper by Google - 2004
- Open-Source implementation by Yahoo called Hadoop

MapReduce: Simplified Data Processing on Large Clusters

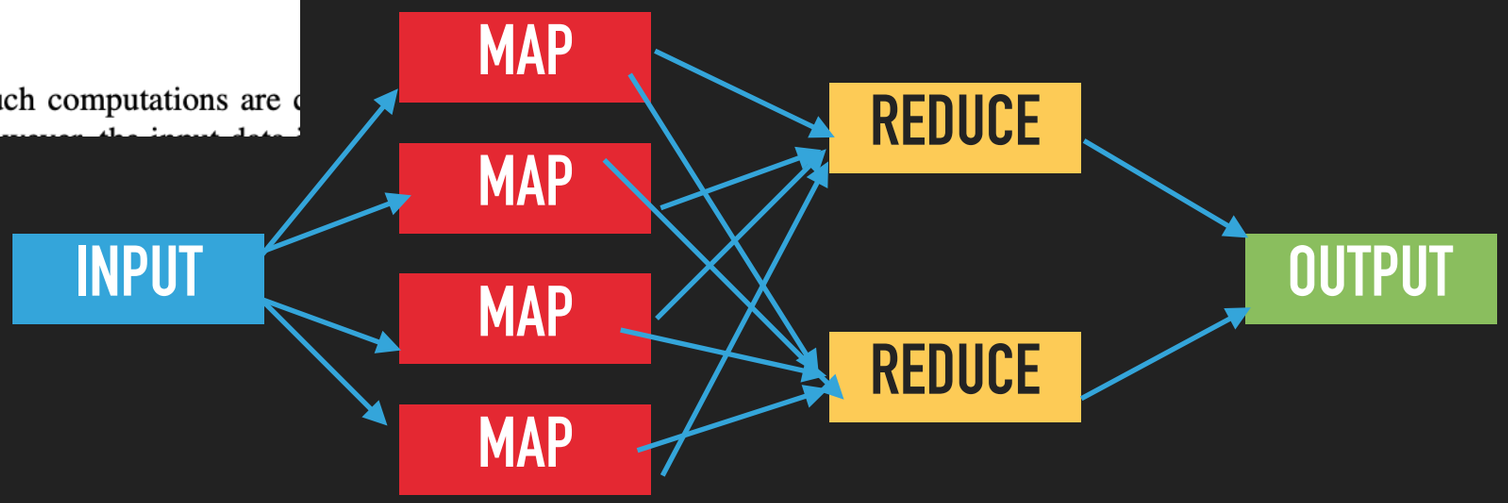
Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

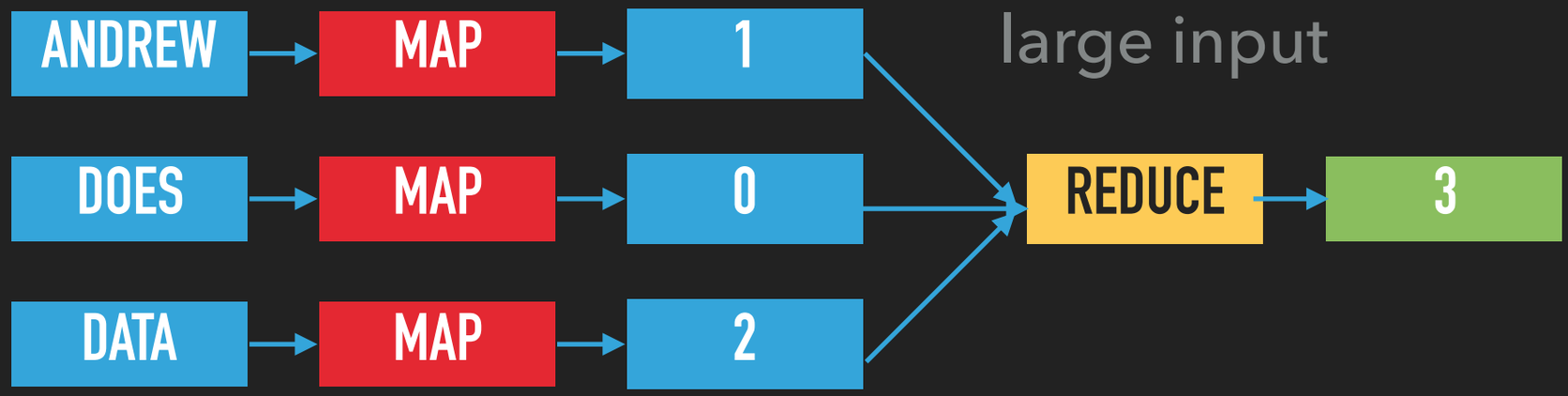
given day, etc. Most such computations are



https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MAPREDUCE

- User provides a "map" function and a "reduce" function
  - "map" accepts an input datum and outputs a new datum
  - "reduce" accepts several input data and outputs a single datum



Count the number of "A"s in an arbitrarily large input

```
def map(d):
    return d.count('A')

def reduce(d):
    return sum(d)
```

COMPLEXITY IS HIDDEN BY FRAMEWORK, USER CODE IS TRIVIAL

MAPREDUCE

- Map takes (large) dataset and produces a new one
- Reduce takes a dataset and makes it smaller
- Many algorithms can be expressed as MR
  - Batch image conversion (skip the reduce step!)
  - Text operations - sorting, searching, NLP
  - "You have 300 friends in common with X..."
  - PageRank!

PAGERANK

- A page is "better" if it gets more citations (links)
  - Citations from "better" pages count more
- Need to find links to, not from, any given page
  - Reverse the graph!
  - Eventually converges

The PageRank Citation Ranking: Bringing Order to the Web

January 29, 1998

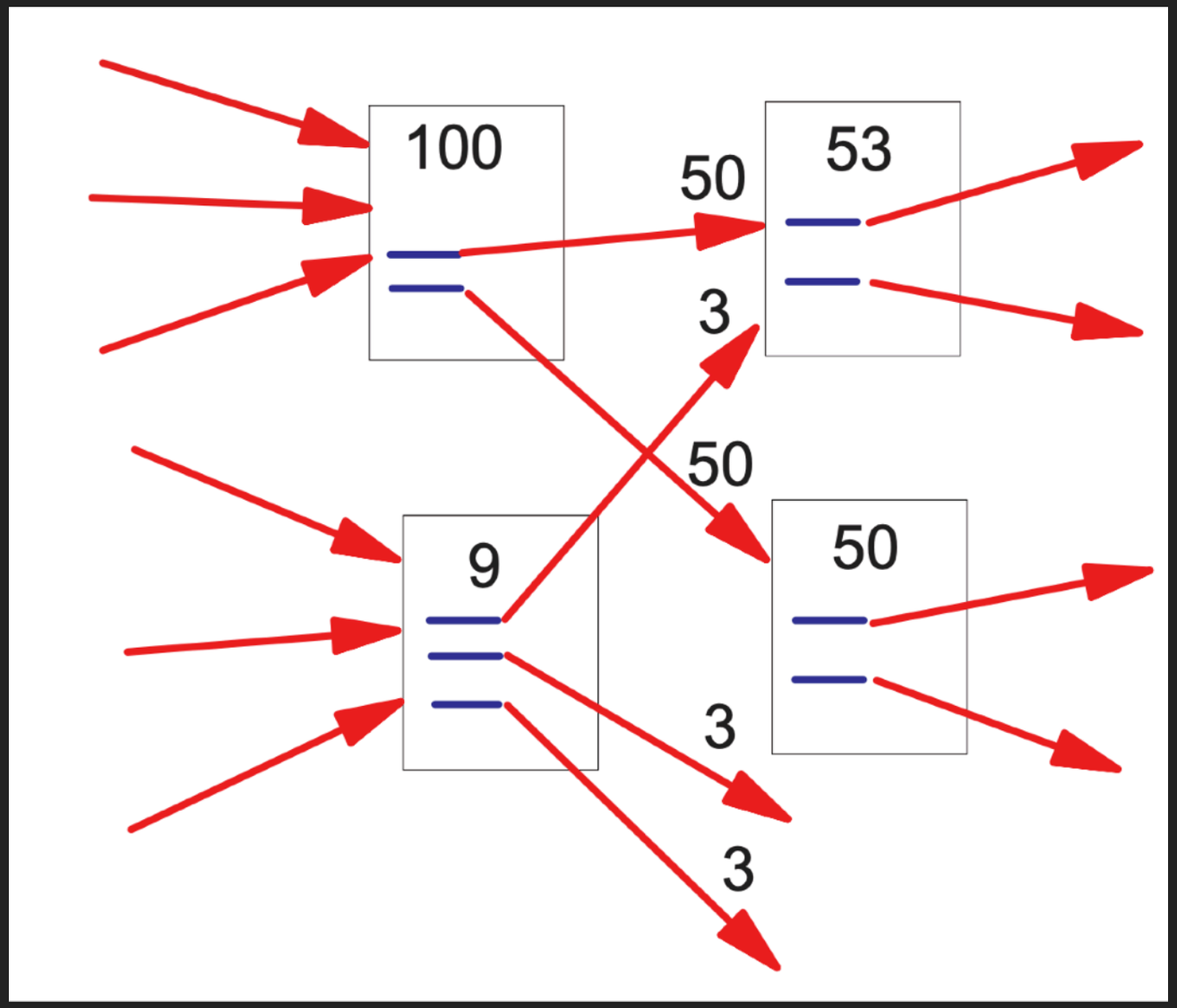
Abstract

The importance of a Web page is an inherently subjective matter, which depends on the readers interests, knowledge and attitudes. But there is still much that can be said objectively about the relative importance of Web pages. This paper describes PageRank, a method for rating Web pages objectively and mechanically, effectively measuring the human interest and attention devoted to them.

We compare PageRank to an idealized random Web surfer. We show how to efficiently compute PageRank for large numbers of pages. And, we show how to apply PageRank to search and to user navigation. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

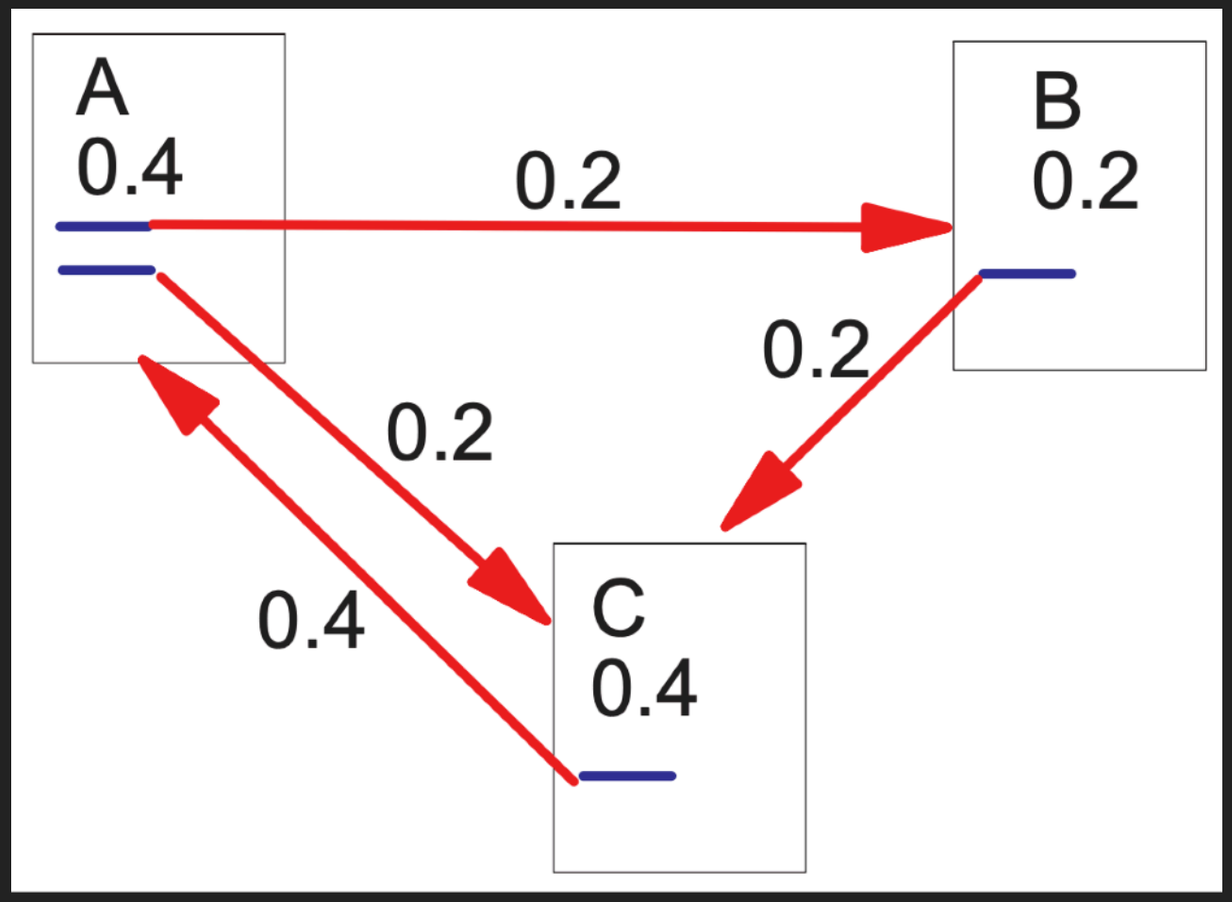
PAGERANK

- Sum(Out)=Sum(In)
- Initial Parameters?
- Initial Pages?

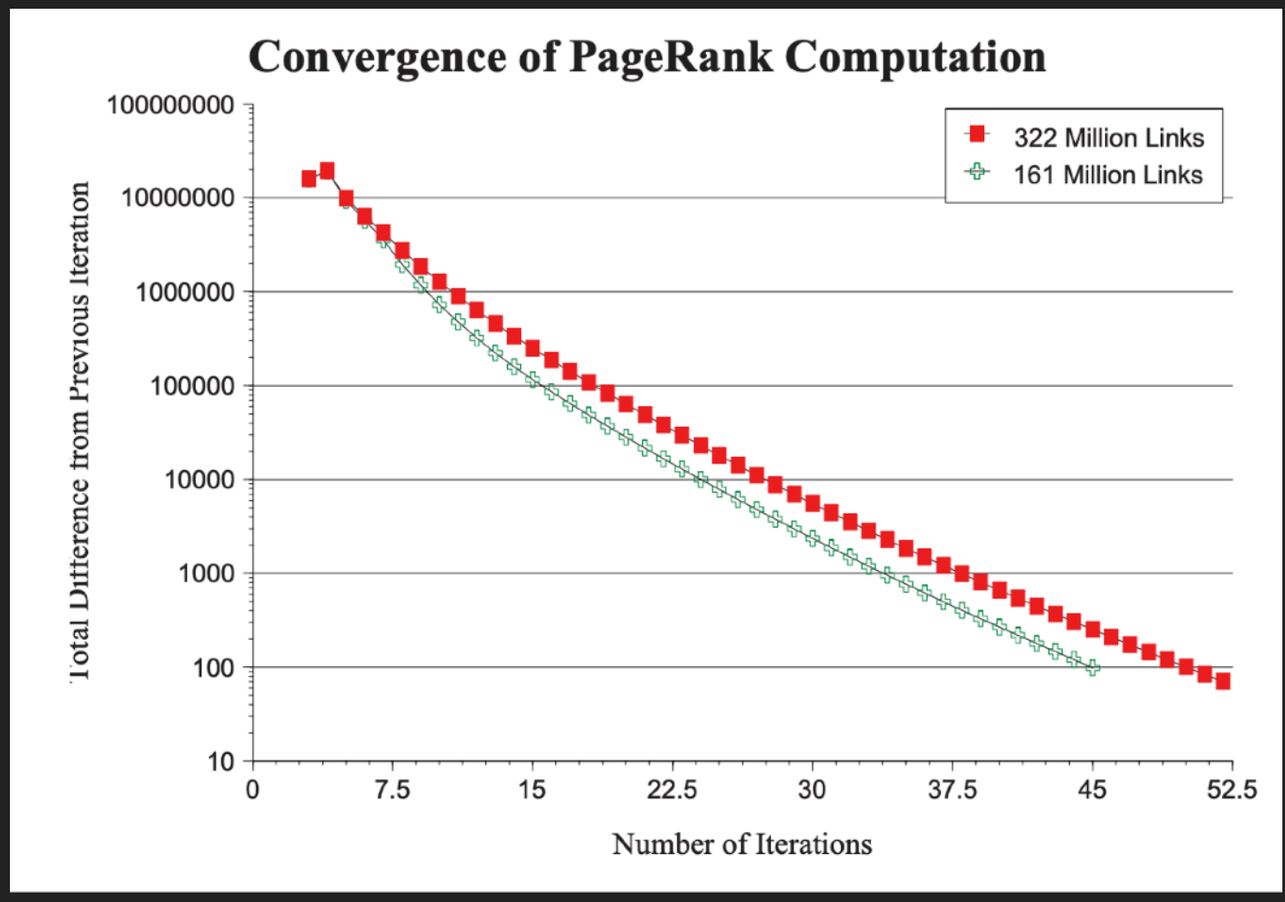


http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

PAGERANK



LINKS ARE (BASICALLY) CYCLICAL



CAN ITERATE AND EVENTUALLY CONVERGE

DATAFRAMES/NUMPY – "LINGUA FRANCA" FOR PYTHON ANALYSIS?

- One interesting development is the convergence of multiple frameworks to use similar DataFrame/NumPy array APIs
  - DataFrame - a "tabular" abstraction
  - NumPy - an array/matrix abstraction
- Even common in-memory layouts between multiple languages
  - Zero-Copy to/from Python/Java/C++, etc...

ENABLES INTERESTING MIXING/MATCHING BETWEEN TOOLKITS

PROBLEMS WITH MAPREDUCE

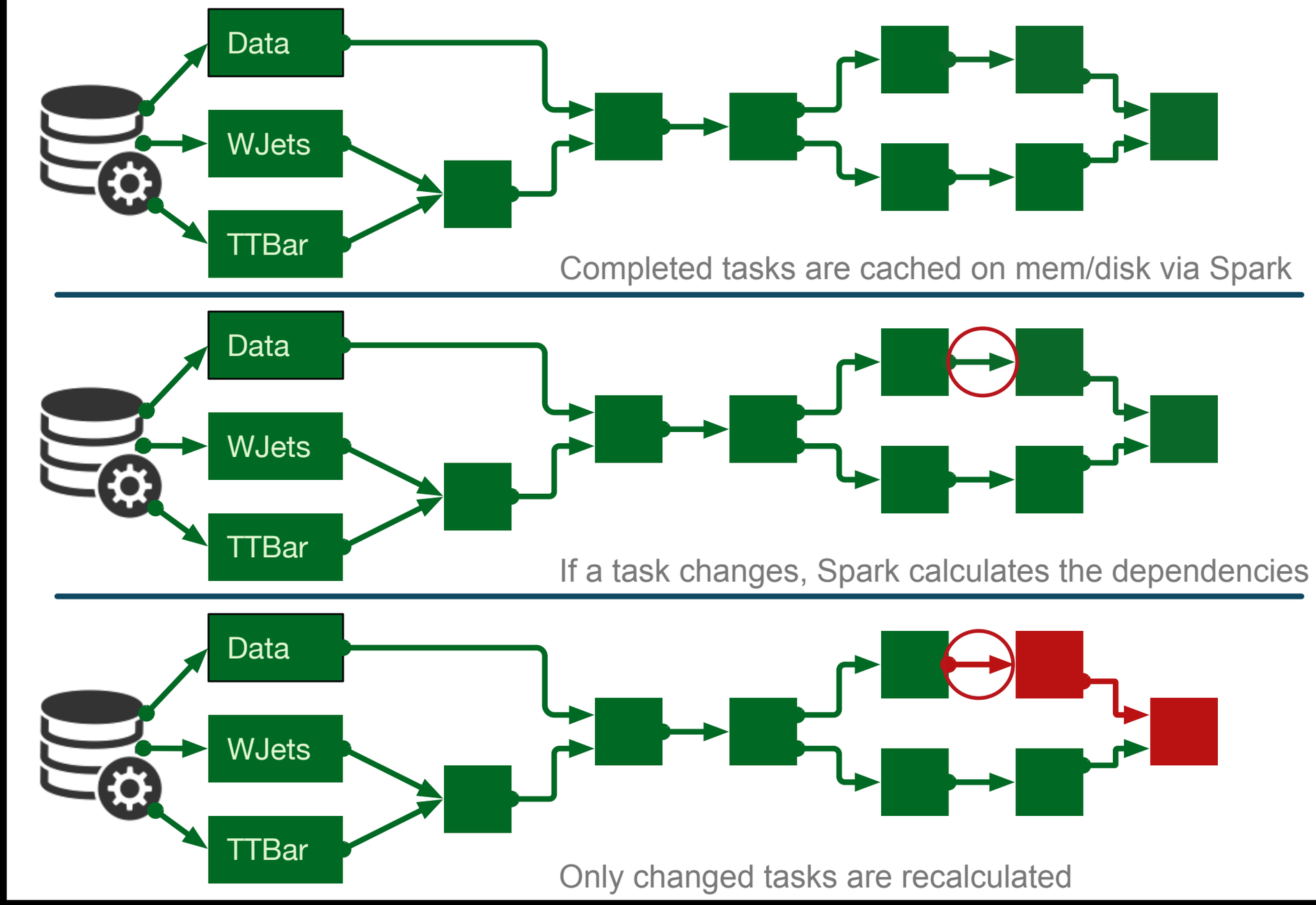
- MapReduce is a good, but somewhat rigid framework
  - Single map stage, then a single reduce stage
  - No arbitrary computation graphs
- Batch-like, not iterative
  - Can emulate iteration (e.g. PageRank), but not quite
  - Load data -> perform computation -> store data

Apache Spark

- "...a fast and general engine for large-scale data processing"
- Replacement of the "Hadoop" component of the Hadoop ecosystem
- Spark schedules an optimized DAG of operations
  - Intermediate results are elided or cached
- Code is same running locally or on a cluster
  - Spark handles splitting tasks, propagating results, retries, etc..

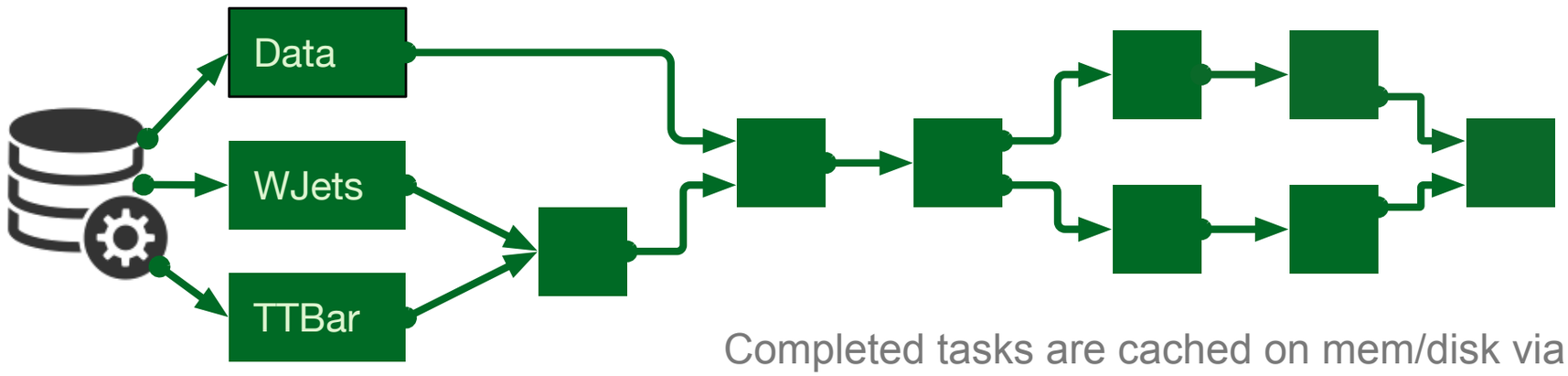
"Functional" Development

- Processing of DataFrames is delayed until results are needed
  - Until then, a DF is a *promise* to perform computations later
- Each DF knows the chain of operations that must be done to produce themselves from scratch
- Each step is idempotent. VERY IMPORTANT

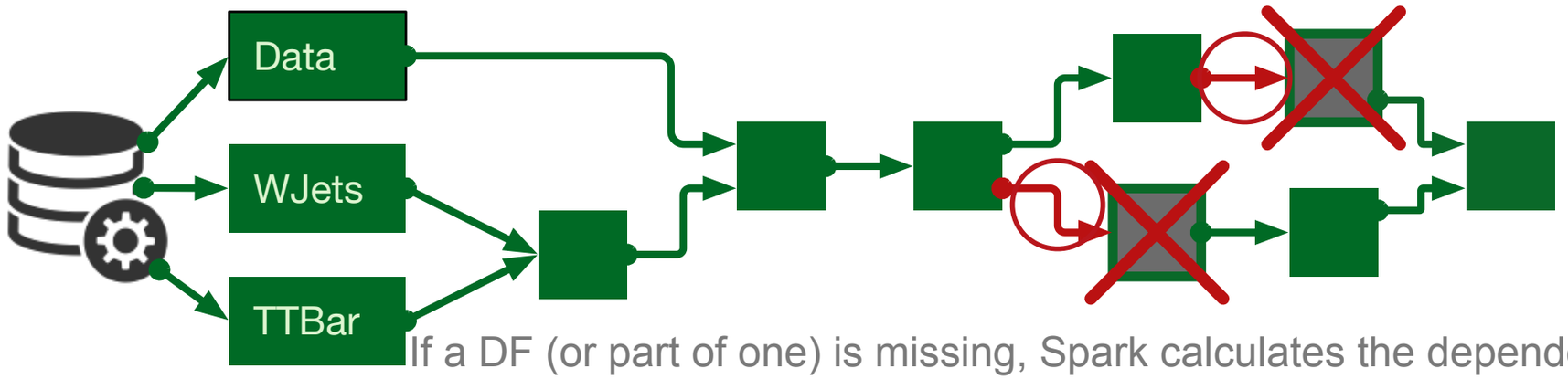


Iterative Analysis

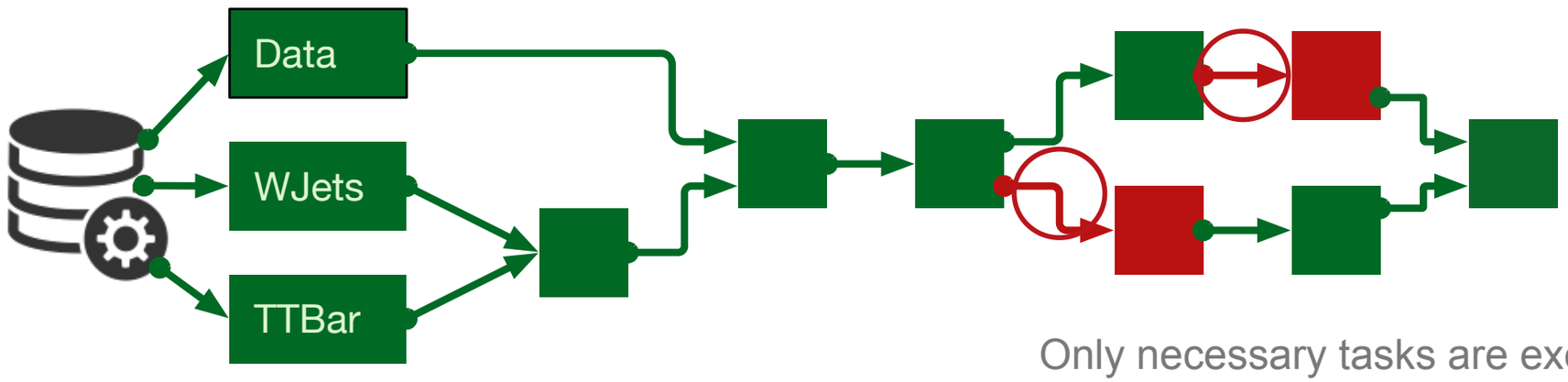
Spark only recalculates necessary parts of the DAG.
Once the inputs are cached, changes go quickly



Completed tasks are cached on mem/disk via Spark



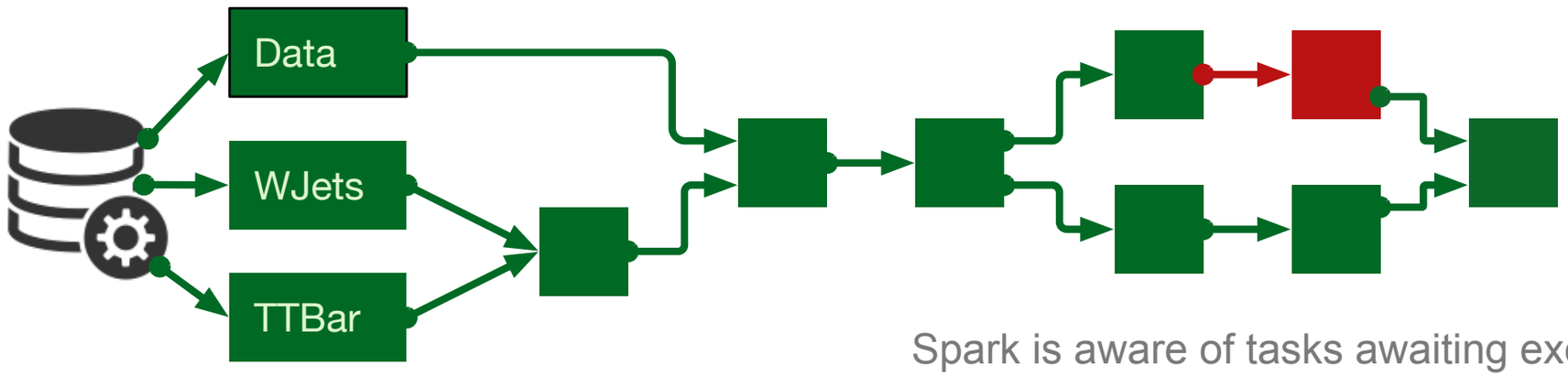
If a DF (or part of one) is missing, Spark calculates the dependencies



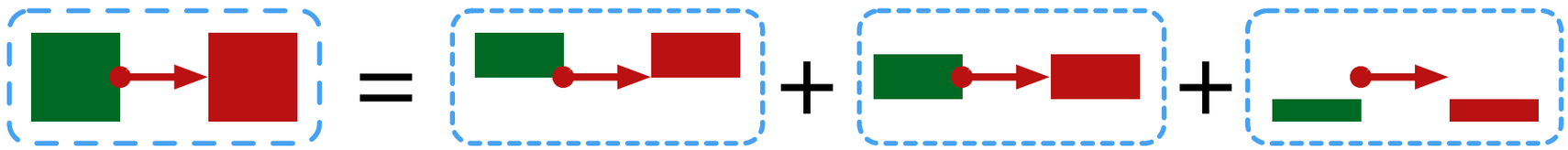
Only necessary tasks are executed

Fault Tolerance

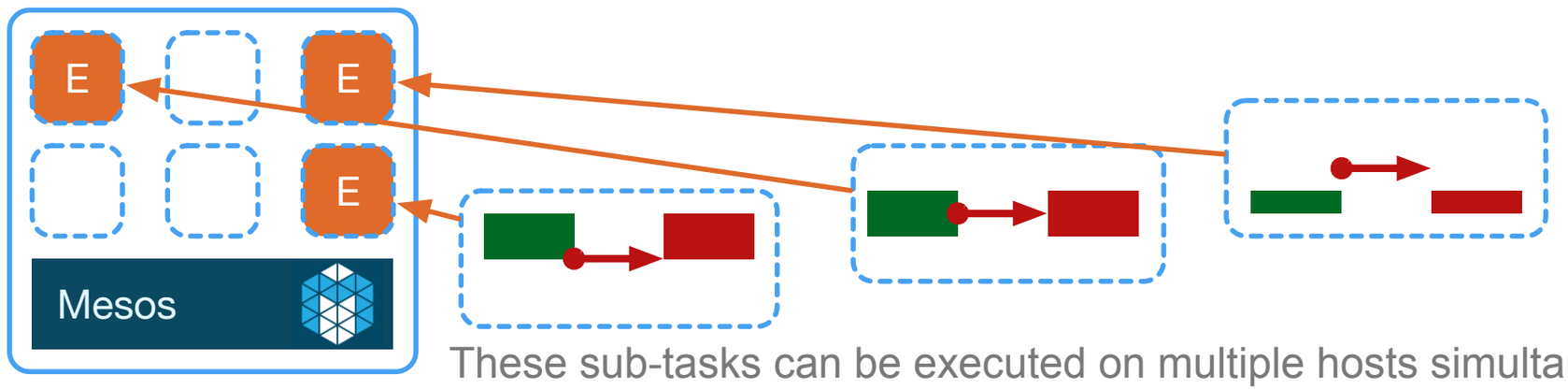
Resilient to hardware, etc.. failures - any missing data can be regenerated from scratch



Spark is aware of tasks awaiting execution



A task over a DF can be decomposed into multiple sub-tasks



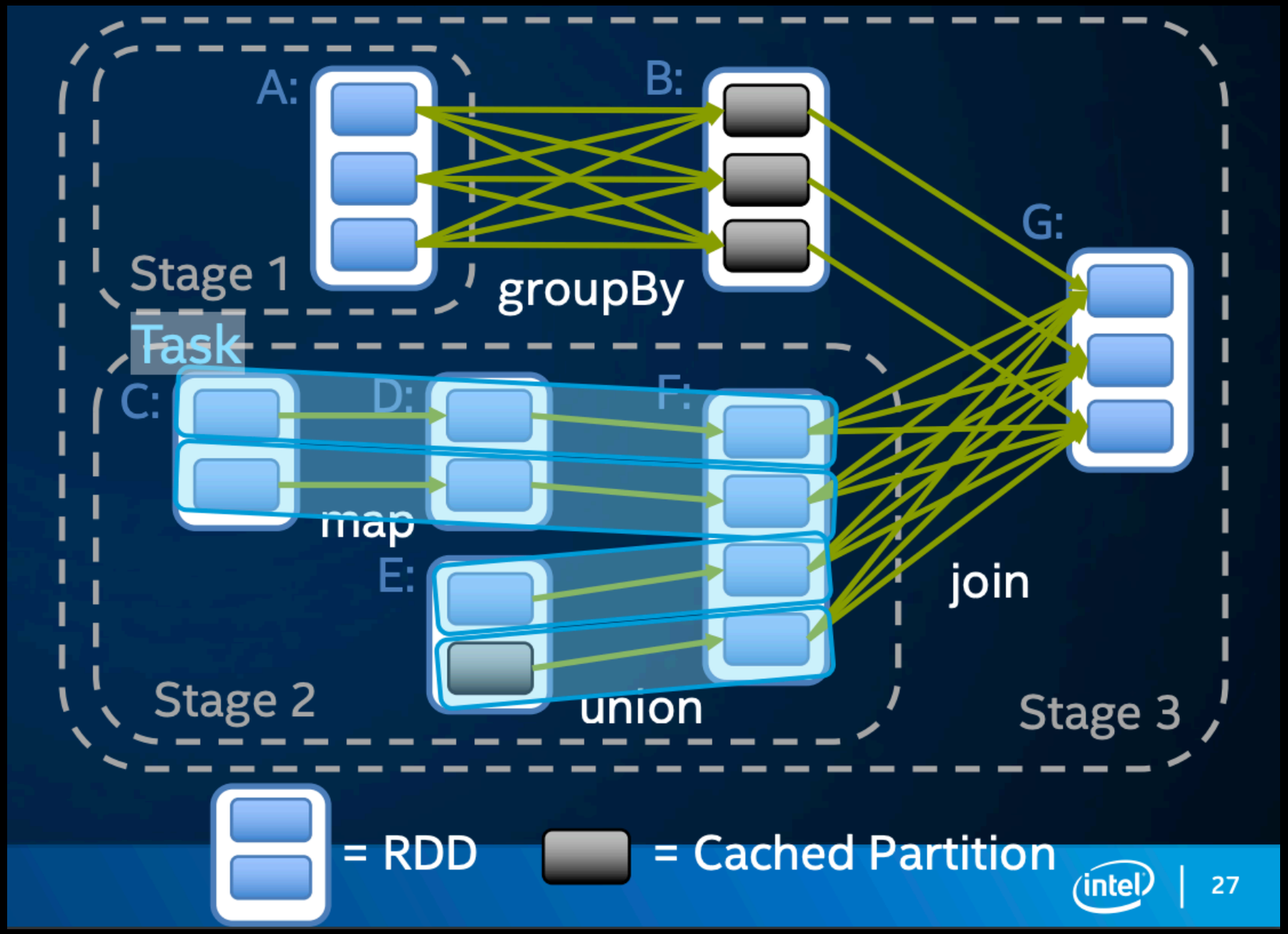
These sub-tasks can be executed on multiple hosts simultaneously

Trivially Parallelized

A DF is split into multiple *partitions* which can be scattered to multiple nodes.

https://indico.cern.ch/event/771843/contributions/3207165/attachments/1772028/2879893/BigDL__Analytics_Zoo_-_Part_1.pdf

Source: “Parallel programming with Spark”, Matei Zaharia, AMPCamp 3



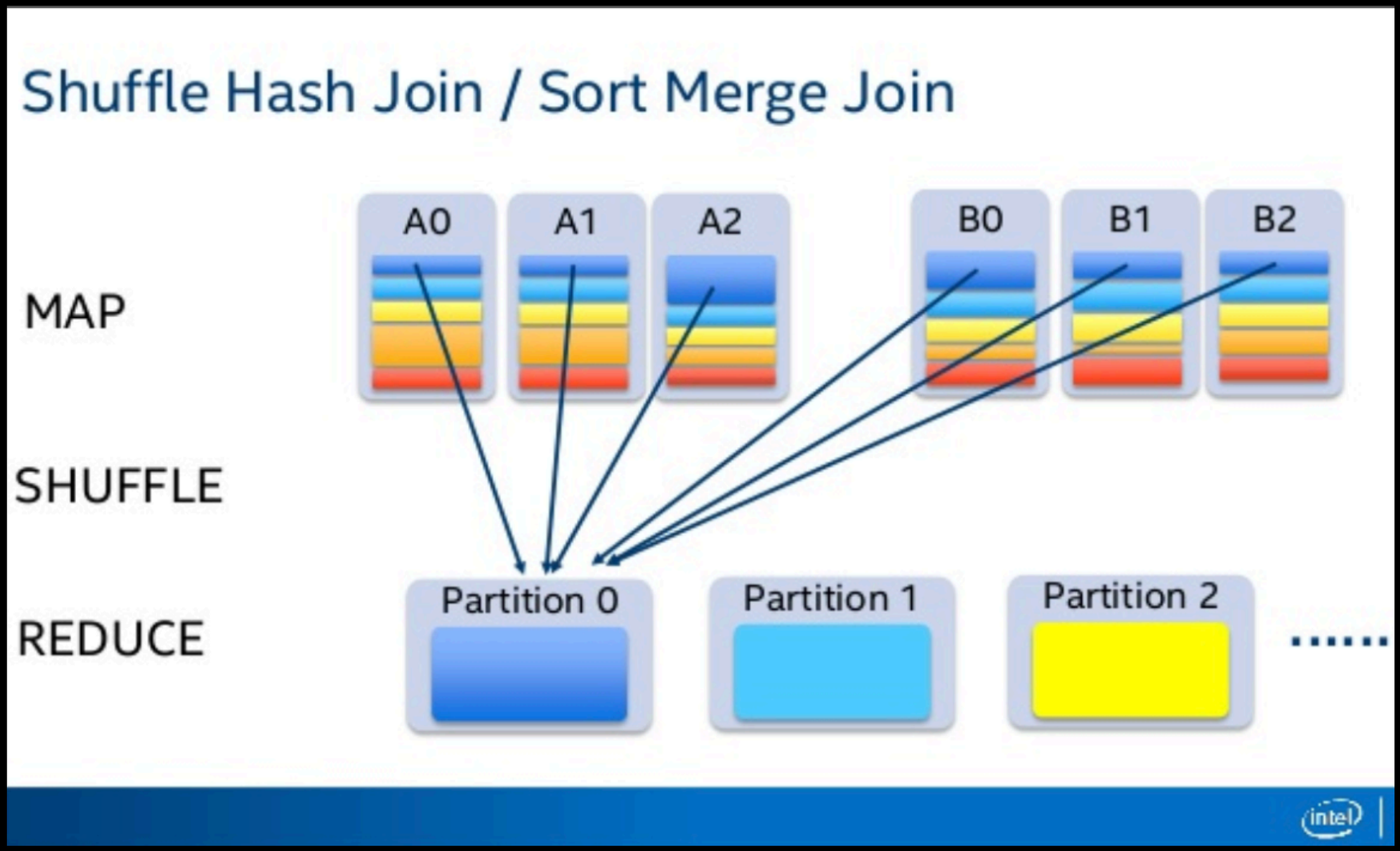
Very Complex Execution Graphs

Spark careful to avoid shuffles

Pipeline operations

High-Level Optimization

https://www.slideshare.net/databricks/an-adaptive-execution-engine-for-apache-spark-with-carson-wang

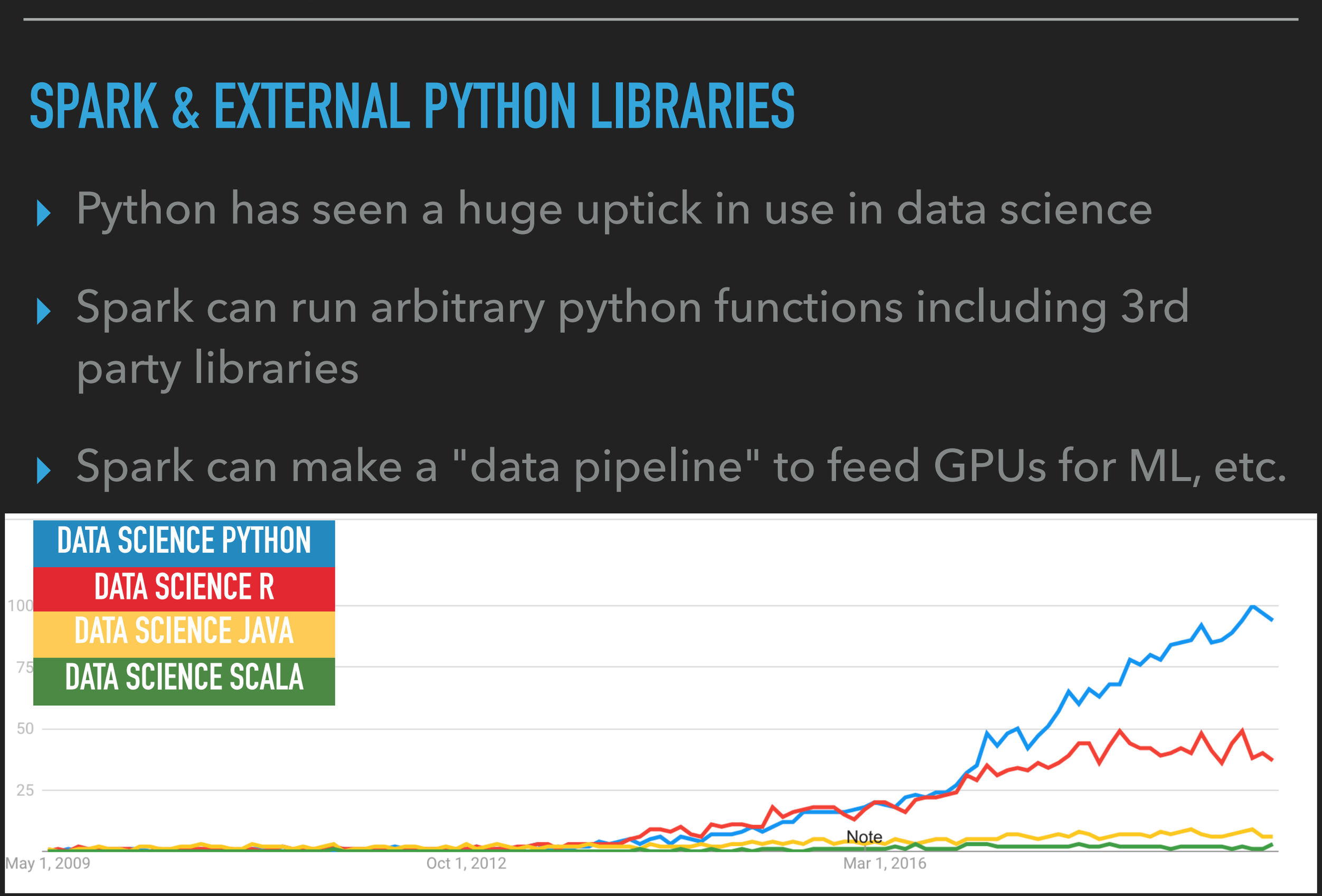


Spark Shuffle

Not usually elide-able
Skew detection
Many tunables

SPARK & EXTERNAL PYTHON LIBRARIES

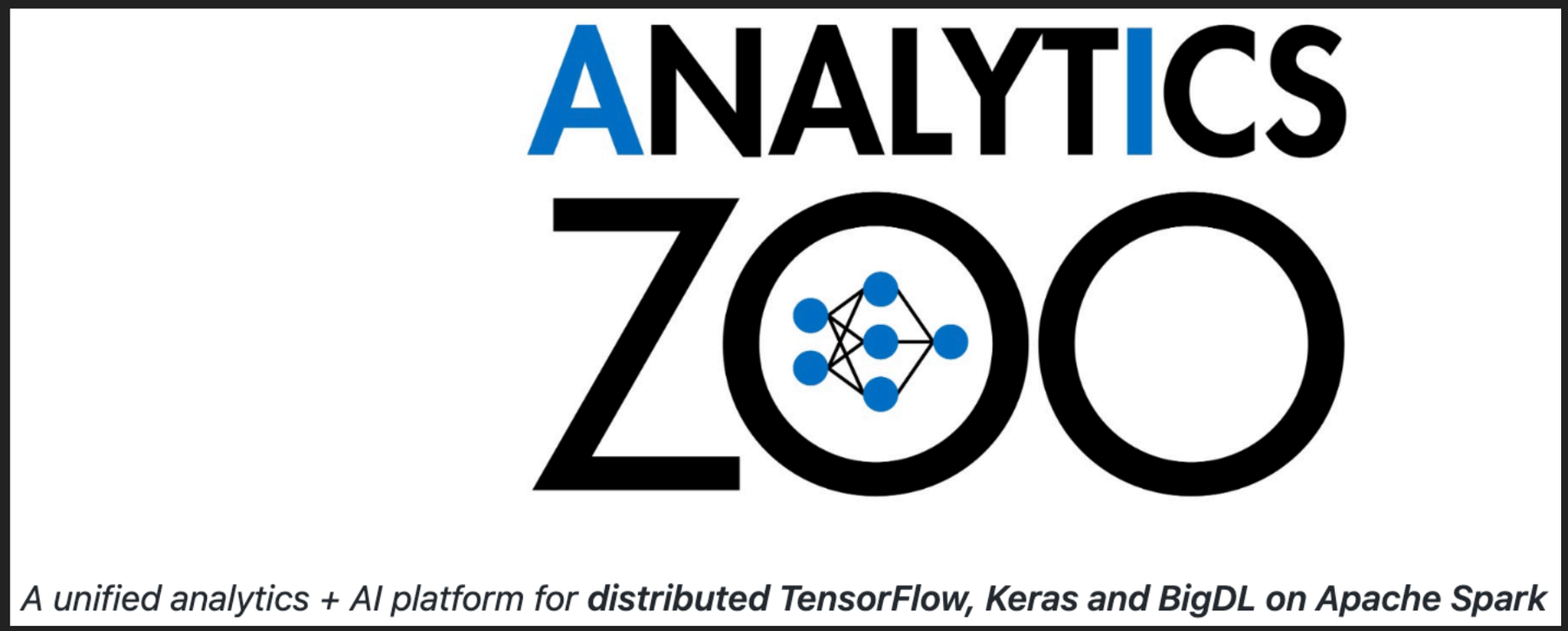
- Python has seen a huge uptick in use in data science
- Spark can run arbitrary python functions including 3rd party libraries
- Spark can make a "data pipeline" to feed GPUs for ML, etc.



(data from google, obviously not scientific)

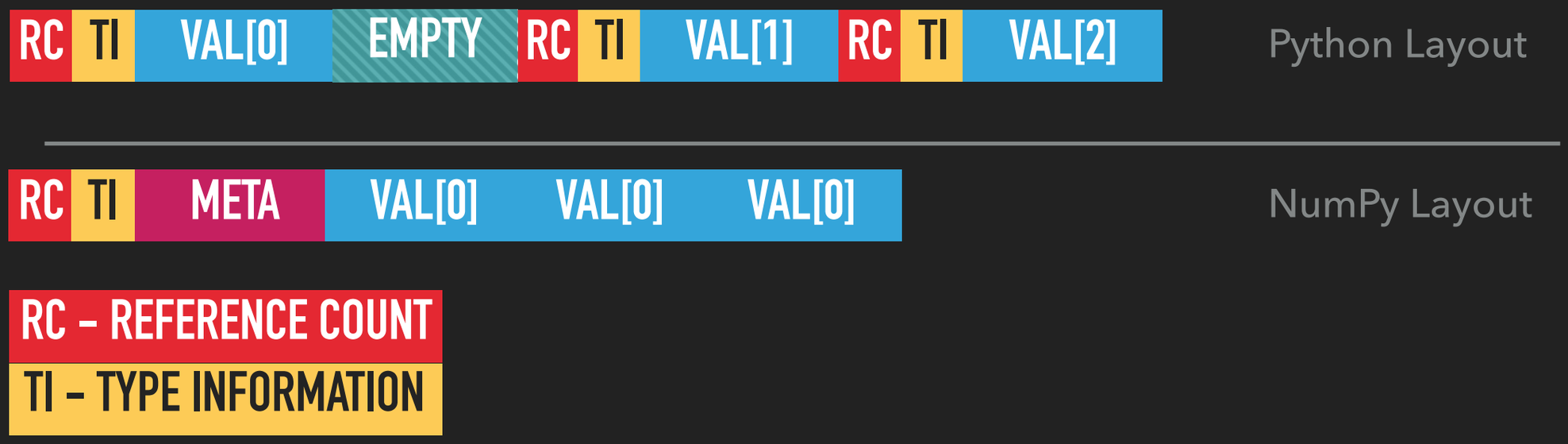
BIGML ANALYTICS ZOO

- Intel-supported "glue" between Spark and ML libraries
- Take a Tensorflow DAG and run it "at scale" via Spark
- DAG of DAGs, etc..



NUMPY

- Highly optimized Python array math library
- Represents arrays of #s as raw byte arrays
- Skip all python memory-management overhead
- Can use CPU SIMD intrinsics via builtin C++ code



NUMPY

- Operations happen on higher level array/matrix objects
  - Don't depend on a compiler "rediscovering" structure
- Many (all?) operations have C++ implementations for speed
  - Can also run on GPU with CuPy
- Some common combinations have fused implementations
  - Can fuse more ops/add custom ops with Numba

```
c = []
b = [1,2,3]
a = [4,5,6]
for i in range(size):
    c[i] = a[i] * b[i]
```

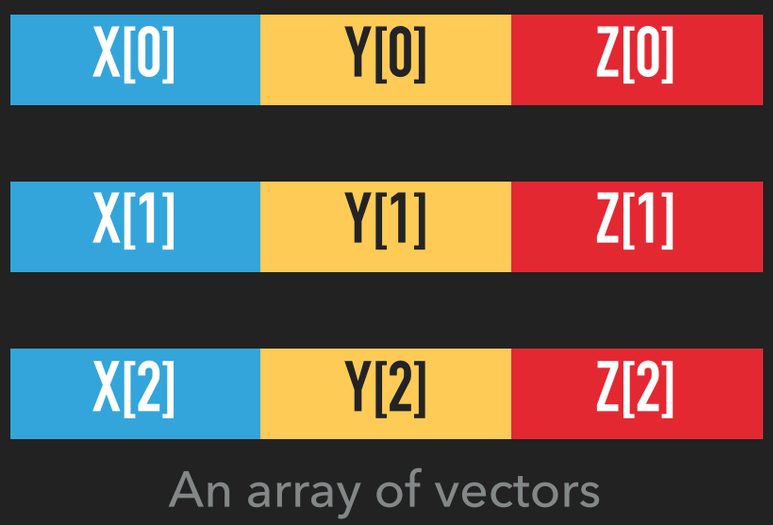
Python array multiplication

```
c = numpy.array([])
b = numpy.array([1,2,3])
a = numpy.array([4,5,6])
# NumPy will auto-vectorize this loop
c = a * b
```

NumPy array multiplication

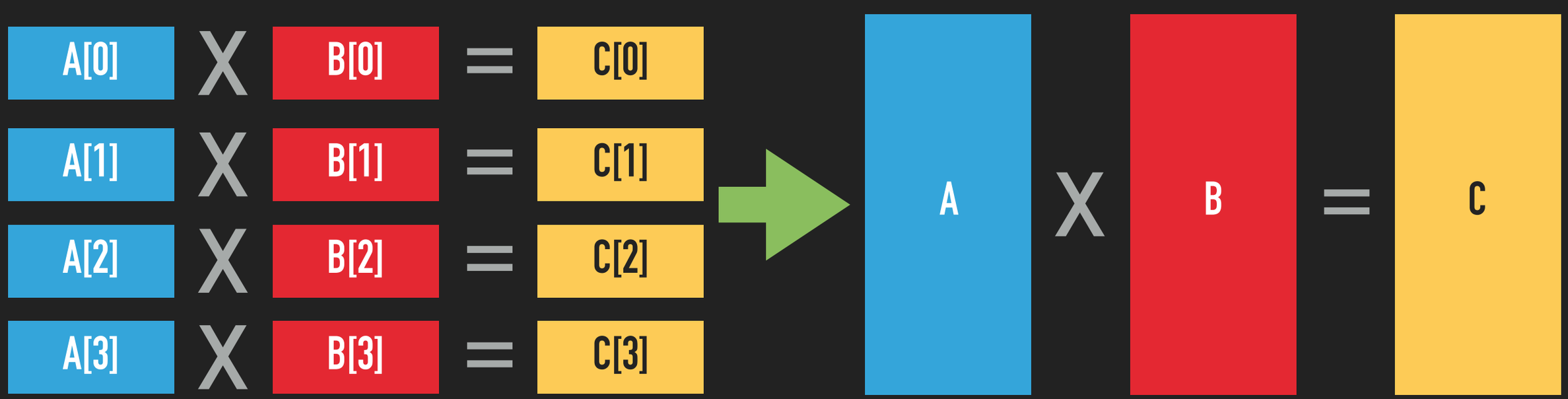
MEMORY LAYOUT

- Data can be stored either row- or column-wise
  - Decision important for SIMD!



An array of vectors

SIMD – SINGLE INSTRUCTION MULTIPLE DATA (REDUX)



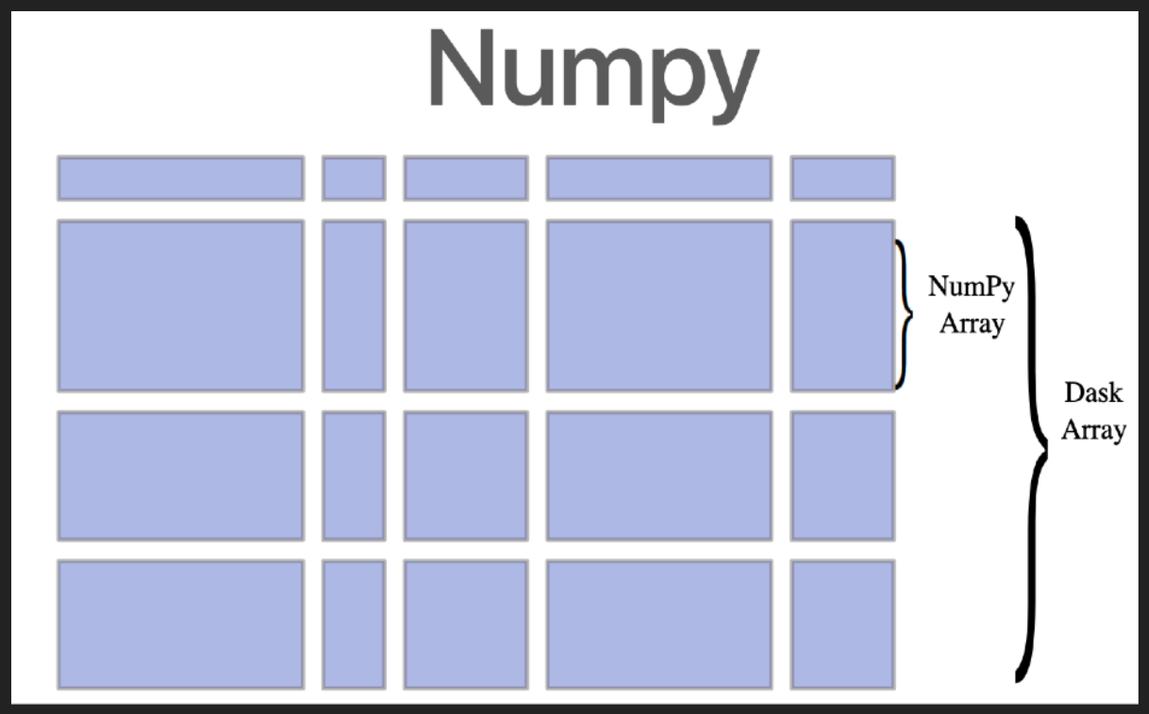
NUMPY IS INHERENTLY COLUMN-WISE AND CAN EXPLOIT SIMD

DASK

- A Python distributed computing framework
- Executes Python functions on many workers
- Integrated with e.g. NumPy

```
c = numpy.array([])
b = numpy.array([1,2,3])
a = numpy.array([4,5,6])
# NumPy will auto-vectorize this loop
c = a * b
```

```
c = dask.array([])
b = dask.array([1,2,3])
a = dask.array([4,5,6])
# Dask will divide these arrays
# NumPy will auto-vectorize this loop
c = a * b
```



Dask handles the details "under the hood"

See for more info

https://docs.dask.org/en/latest/presentations.html

DASK

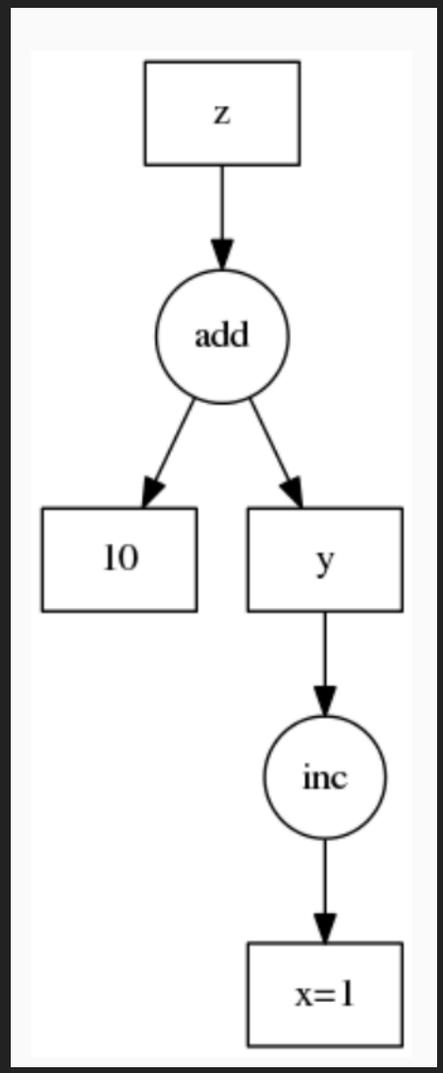
- A graph execution engine, executing arbitrary python functions in a low-level DAG representation

Consider the following simple program:

```
def inc(i):
    return i + 1

def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

We encode this as a dictionary in the following way:

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```



(arrow directions are opposite of execution flow)

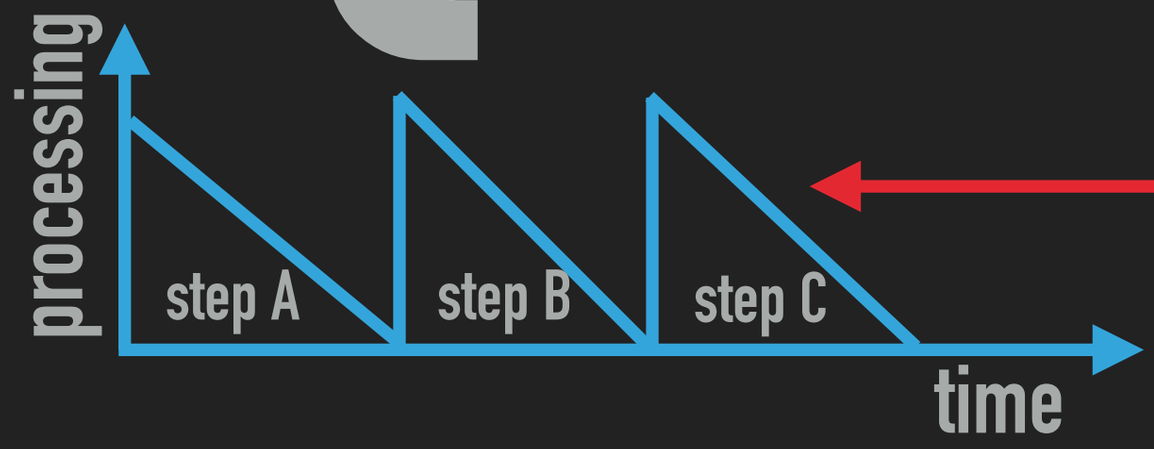
https://docs.dask.org/en/latest/graphs.html

"TRADITIONAL" ANALYSIS BATCH SUBMISSION IN HEP

assumes fixed execution

manually produce submit script for specific Tier 2/ Tier 3

```
for iList in listOfSamples:
    condorJobFile = open("dilBatch.submit", "w")

    condorJobFile.write( "universe = vanilla\n"+"executable = runPlotsCondor.csh\n")
    condorJobFile.write( "Zmask = %s\n" % iZmask)
    condorJobFile.write( "Label = %s\n" % jobLabel)
    condorJobFile.write( "List = %s\n" % iList)
    condorJobFile.write( "JES = %s\n" % jesChoice)
    condorJobFile.write( "JER = %s\n" % jerChoice)
    condorJobFile.write( "PV = %s\n" % iPV)
    condorJobFile.write( "Charge = %s\n" % iCharge)
    condorJobFile.write( "arguments = $(List) $(Zmask) $(Label) $(PV) $(Charge)\n")
    condorJobFile.write( "output = batchBEAN/condorLogs/condor_$(List)_$(Process).stdout\n")
    condorJobFile.write( "error = batchBEAN/condorLogs/condor_$(List)_$(Process).stderr\n")
    condorJobFile.write( "queue 1\n")
```



dependencies resolved by manually running steps

Parsl slides courtesy Anna Woodard

PARSL BASICS

- Pure python; easy installation
- Rather than define code/input/output mapping externally, the user annotates functions to make Parsl apps
  - Bash apps call external applications
  - Python apps call Python functions
- Apps return “futures”: a proxy for a result that may not yet be available

pip install

future

```python
@python_app
def hello():
    return 'Hello world!'

print(hello().result())
```

Python app

```
Hello world!
```

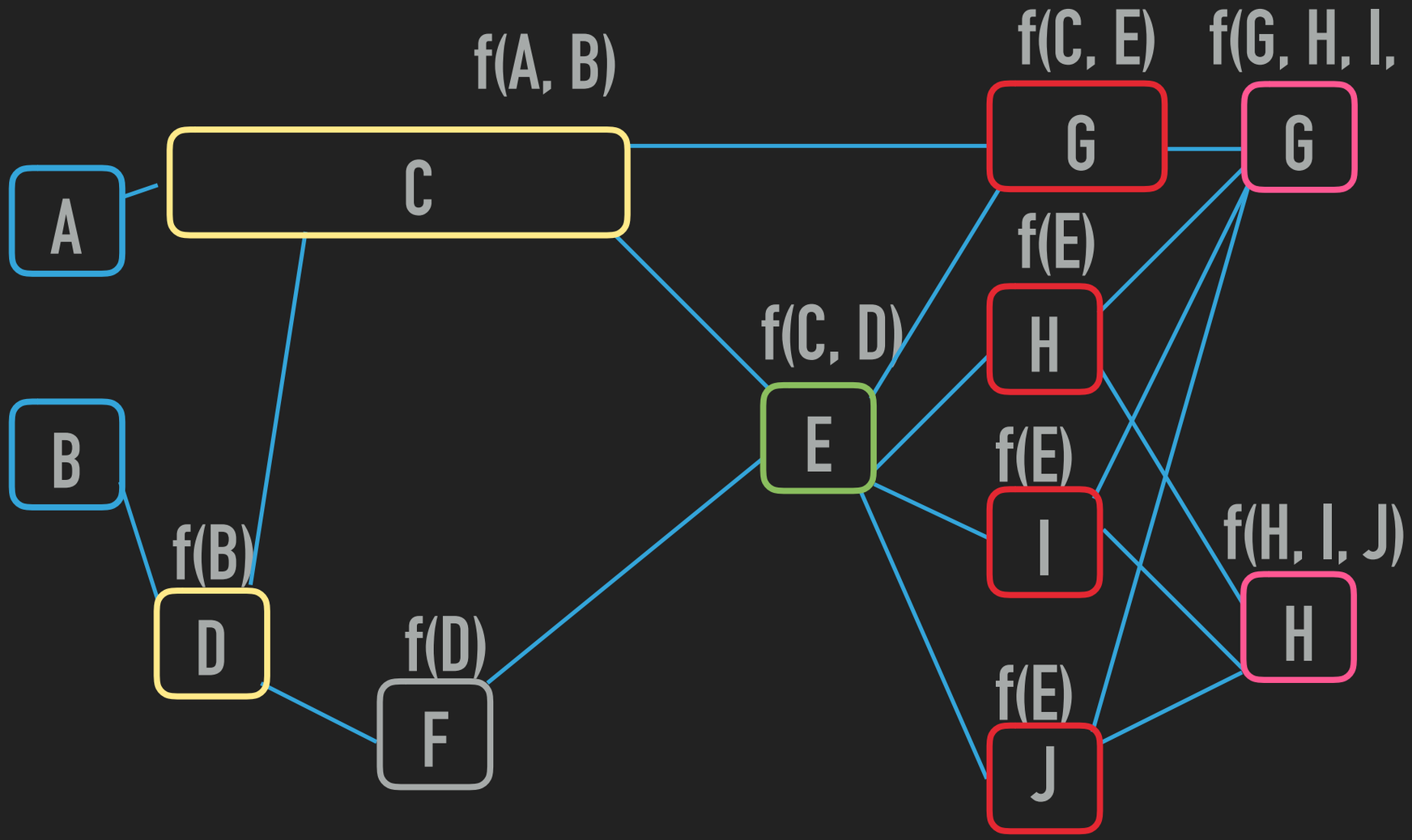
```python
@bash_app
def echo_hello(stdout='hello.stdout'):
    return 'echo "Hello world!"'

echo_hello().result()
with open('hello.stdout') as f:
    print(f.read())
```

Bash app

```
Hello world!
```

Parsl slides courtesy Anna Woodard

RICH EXPRESSION OF DEPENDENCIES

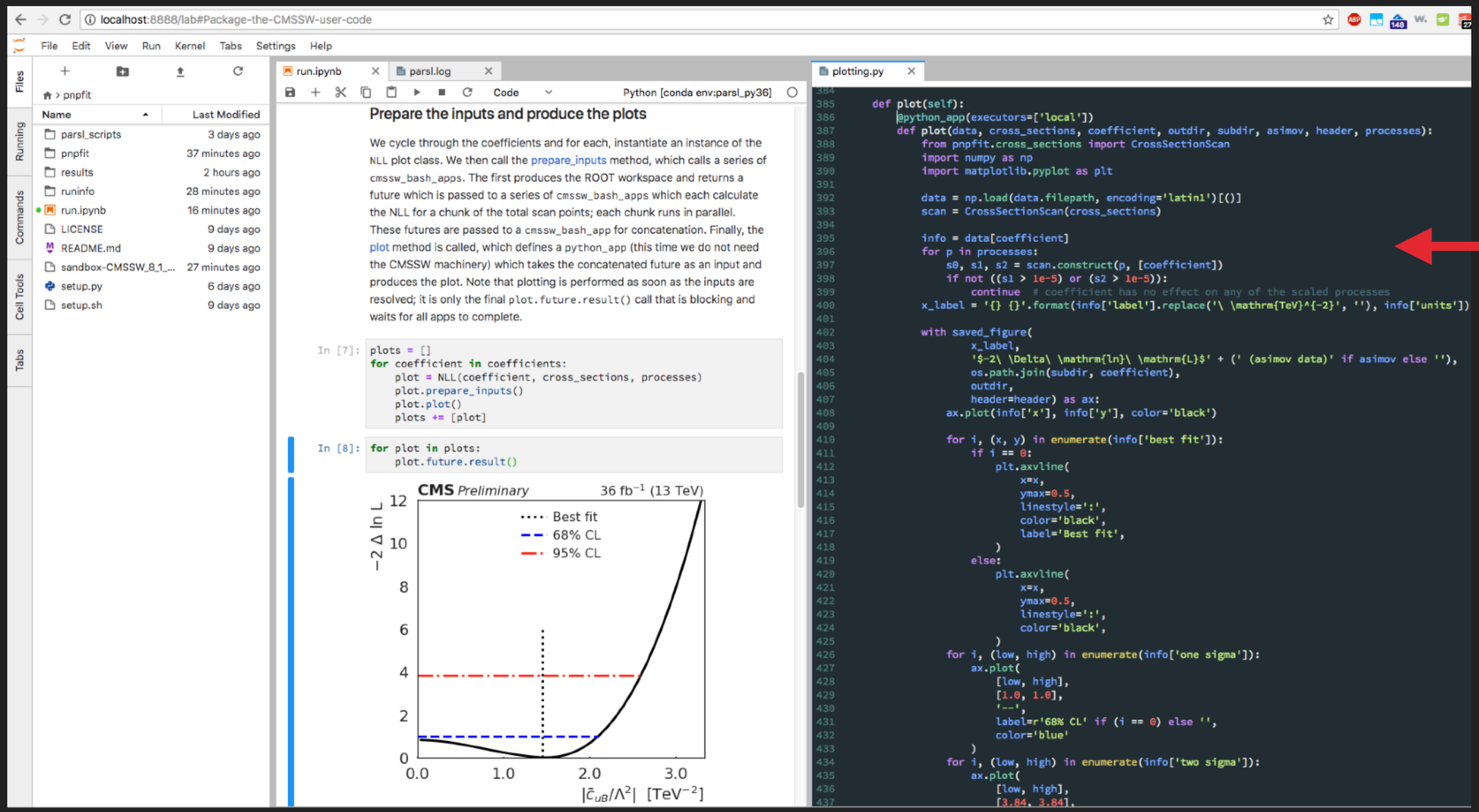


Apps run concurrently, respecting data dependencies via futures. Implicit parallel

Dynamic: apps can create apps!

Parsl slides courtesy Anna Woodard

PARSL + NOTEBOOKS FOR HEP



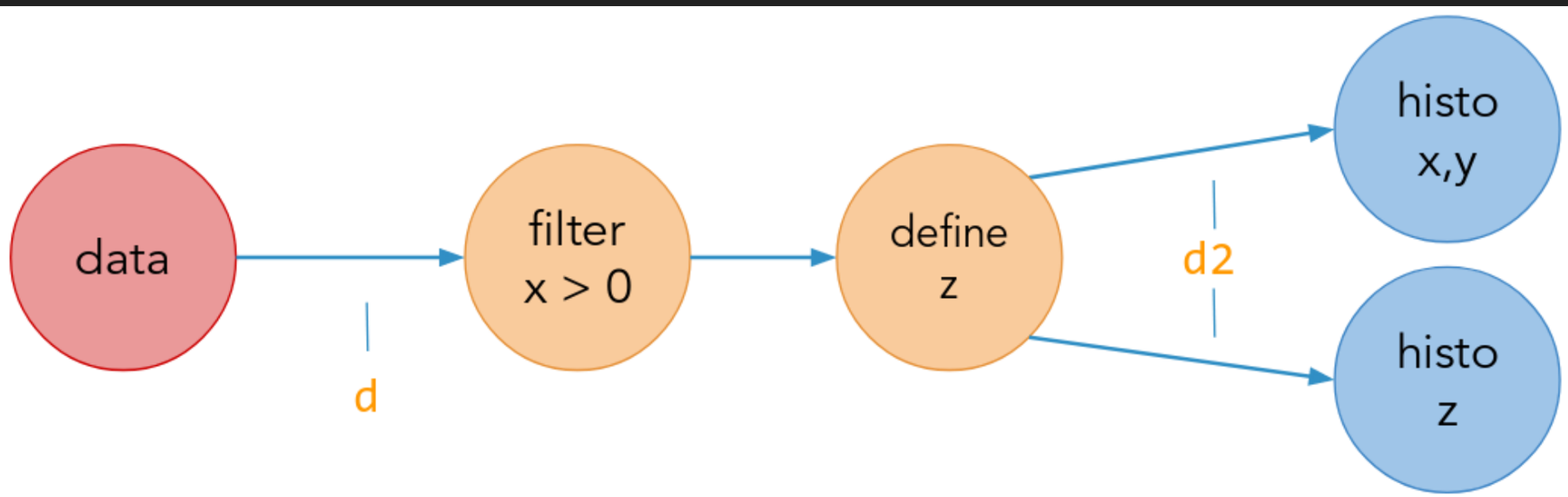
Real-world example [1] implementing part of my dissertation workflow in Parsl

[1] https://github.com/annawoodard/pnpfit/blob/master/run.ipynb

Parsl slides courtesy Anna Woodard

RDATAFRAME

- ROOT's implementation of a DataFrame-like interface




```
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z", "x*x + y*y");
// make multiple histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D("x","y");
```

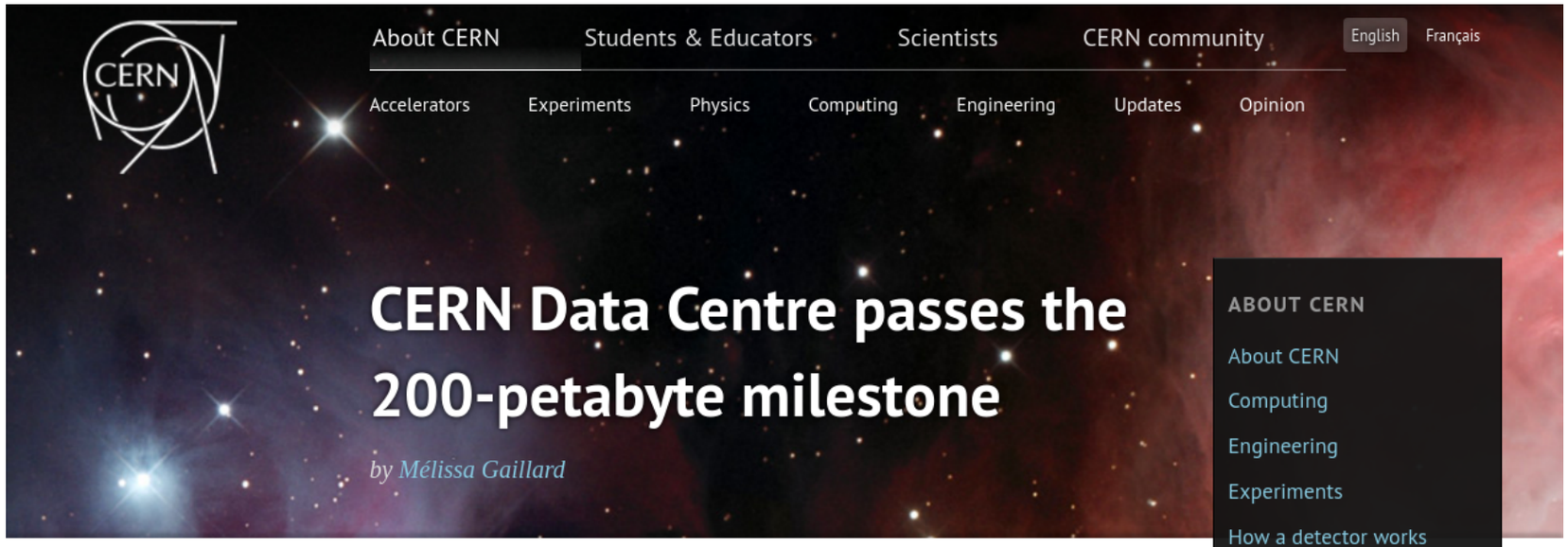
TAKES ADVANTAGE OF EXISTING ROOT CODE

SCIENTIFIC PYTHON OR PYTHON IN SCIENCE?

- Many "ancillary" python packages were produced and popularized in the last ~decade
  - Plotting
  - Numeric computation
  - Machine Learning
  - Statistics

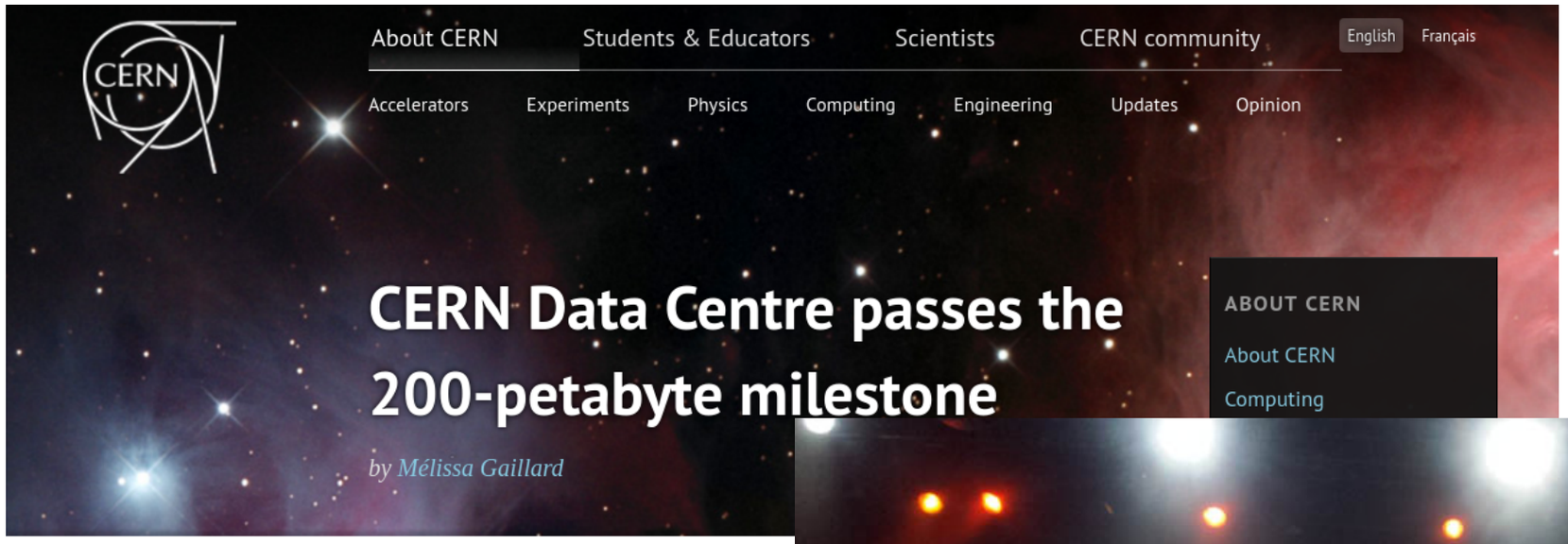
ARE WE (PHYSICS) IN A COMPUTATIONAL RENAISSANCE?
SHOULD WE EMBRACE THESE TOOLS? IF SO, HOW?

We measure globally distributed data in hundreds of PB



Courtesy Jim Pivarski - Princeton University/DIANA-HEP

But for "web scale" companies, 100 PB = 1 truck



Posted by Stefania Pandolfi on 6 Jul 2017. Last updated 7 Jul 2017, 11.18.

Voir en français

This content is archived on the CERN Document Server

CERN's Data Centre (Image: Robert Hradil, Mo

Courtesy Jim Pivarski - Princeton University/DIANA-HEP

Option #1

All of our needs are specialized.

Continue developing our own everything.

Option #2

Modern big data software has some good ideas; integrate those *ideas* into our stack.

Option #3

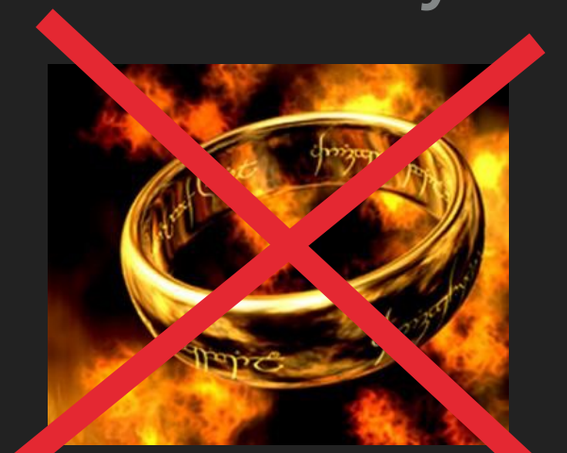
Narrow our scope to HEP-specific tools, what no one else is developing, and make them interoperate with non-HEP tools for the common parts.

Option #4

Convince the world to start using HEP analysis techniques so that they will develop solutions for these, too.

RECAP

- An explosion in the data science ecosystem
  - Most packages "born" in the last 10 or even 5 years
- Many different variations on a theme
  - But that's OK!
- Lots of different ideas about airplanes, but we eventually probed the phase space and found the "best" way



LECTURE 3

HEP GLUE

HEP GLUE

- Given the advances in the data science field, how should we (HEP) proceed
- I argued for "Option 3"
- Spend our effort on HEP-specific parts, using other tools as much as possible

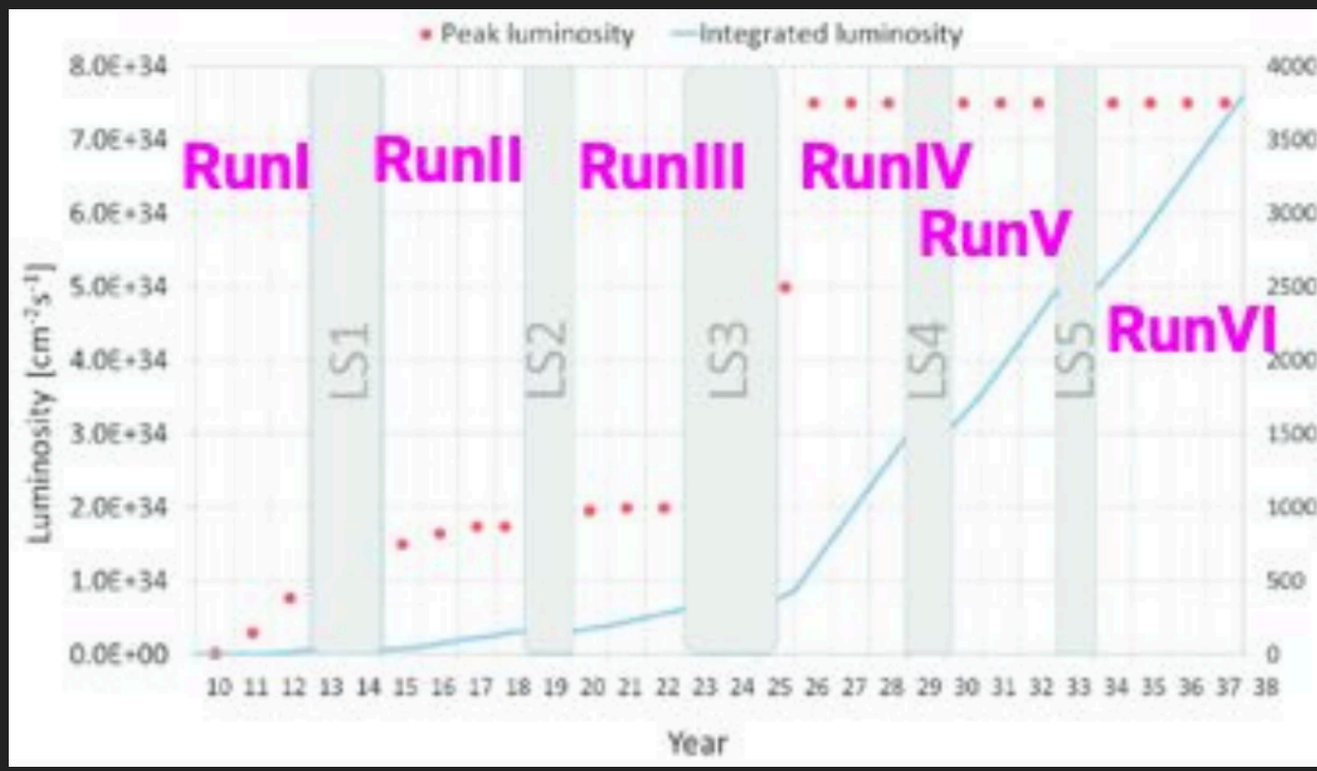
WHAT IS MISSING?

- Histogramming (in the Physics sense)
- Variable-length/nested NTuples
- "Scale-out"

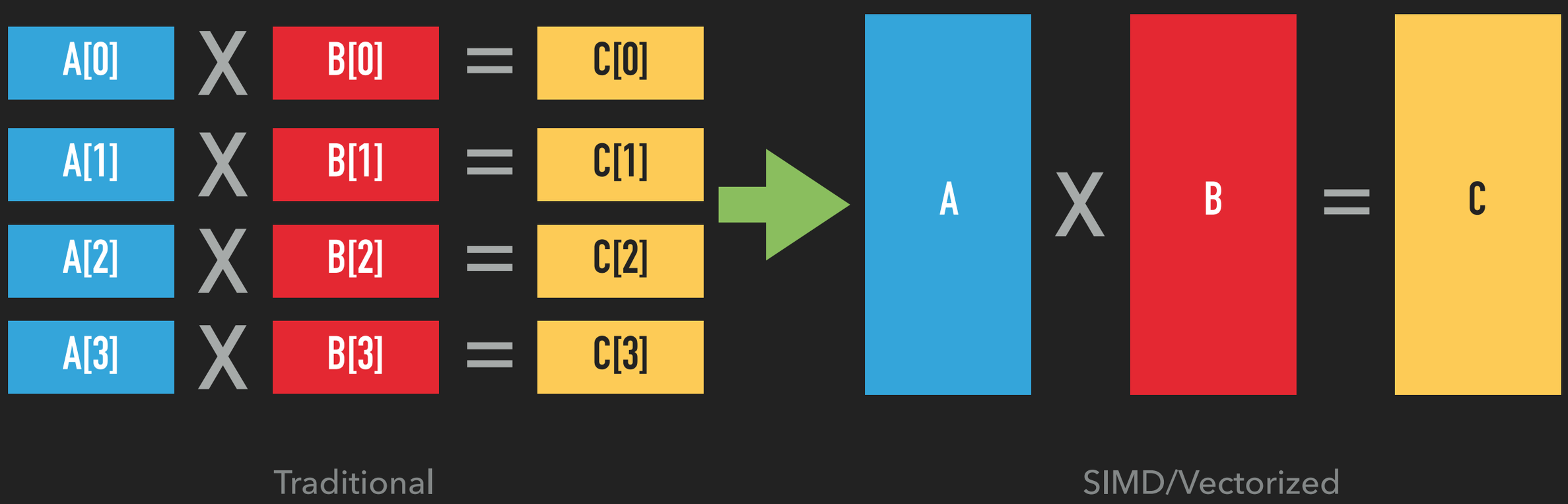
Visit github.com/scikit-hep for these packages and more!

WHAT ARE THE CONSTRAINTS?

- User experience
- HL-LHC is coming
- 10x increase in computing needs
- Not even worth asking for a 10x increase in computing funding

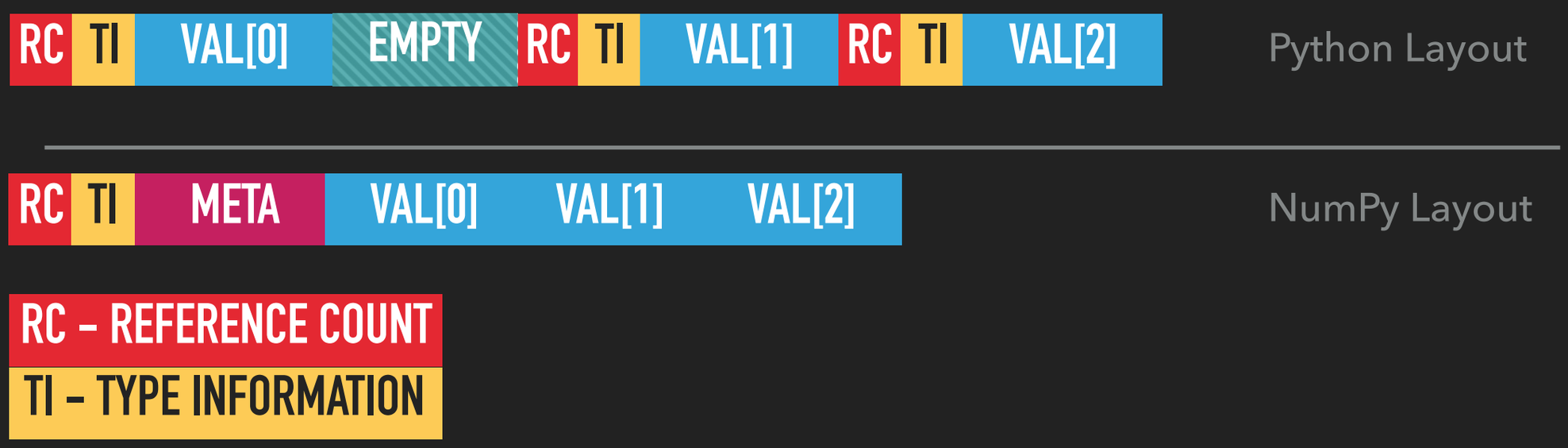


SIMD – SINGLE INSTRUCTION MULTIPLE DATA (REDUX AGAIN)



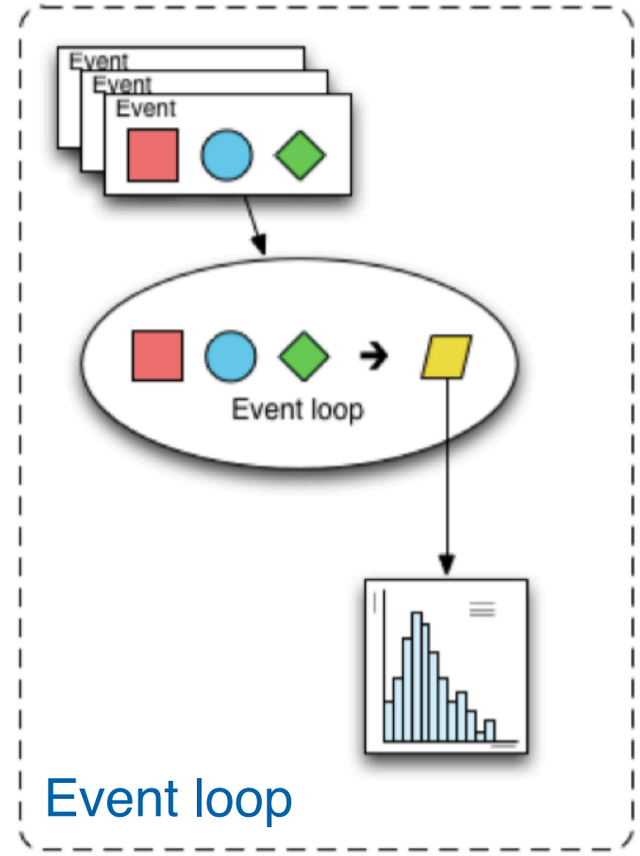
NUMPY (REDUX)

- Highly optimized Python array math library
- Represents arrays of #s as raw byte arrays
- Skip all python memory-management overhead
- Can use CPU SIMD intrinsics via builtin C++ code

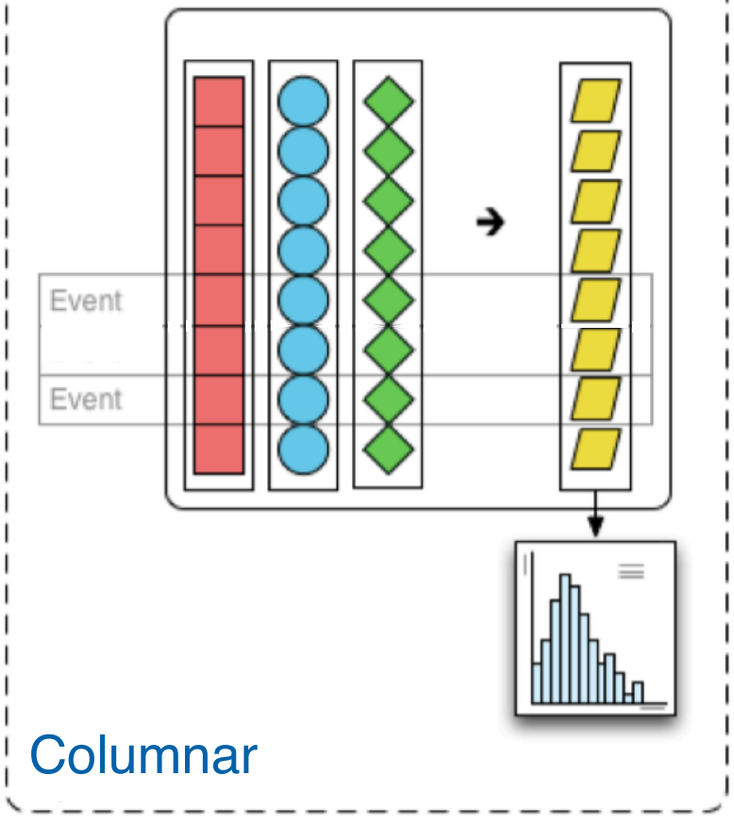


Prologue: terminology

- Event loop analysis:
  - Load relevant values for a specific event into local variables
  - Evaluate several expressions
  - Store derived values
  - Repeat (explicit outer loop)



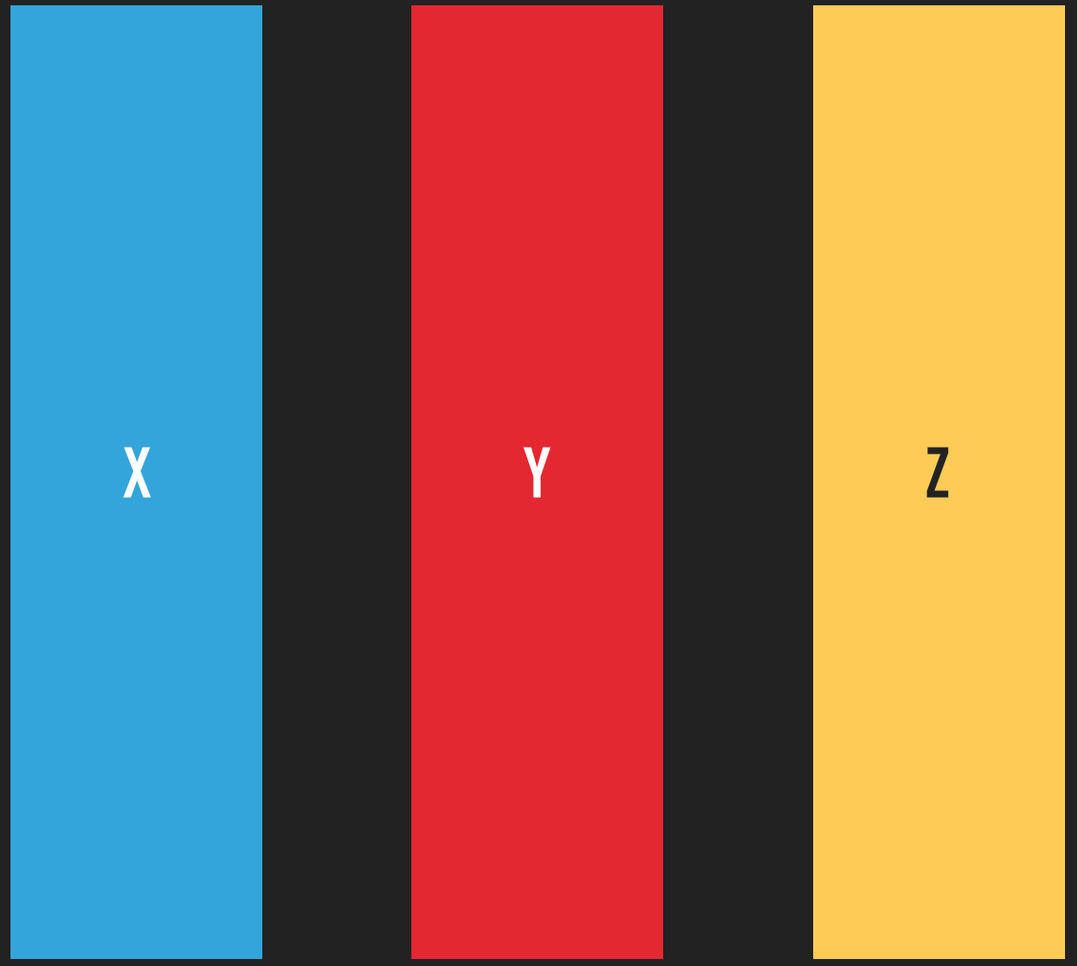
- Columnar analysis:
  - Load relevant values for many events into contiguous arrays
    - Nested structure (array of arrays) → flat content + offsets
      - This is how TTree works!
  - Evaluate several **array programming** expressions
    - Implicit *inner* loop**s**
  - Store derived values



https://indico.cern.ch/event/759388/contributions/3356303/attachments/1816023/2968101/ncsmith-how2019-columnar.pdf

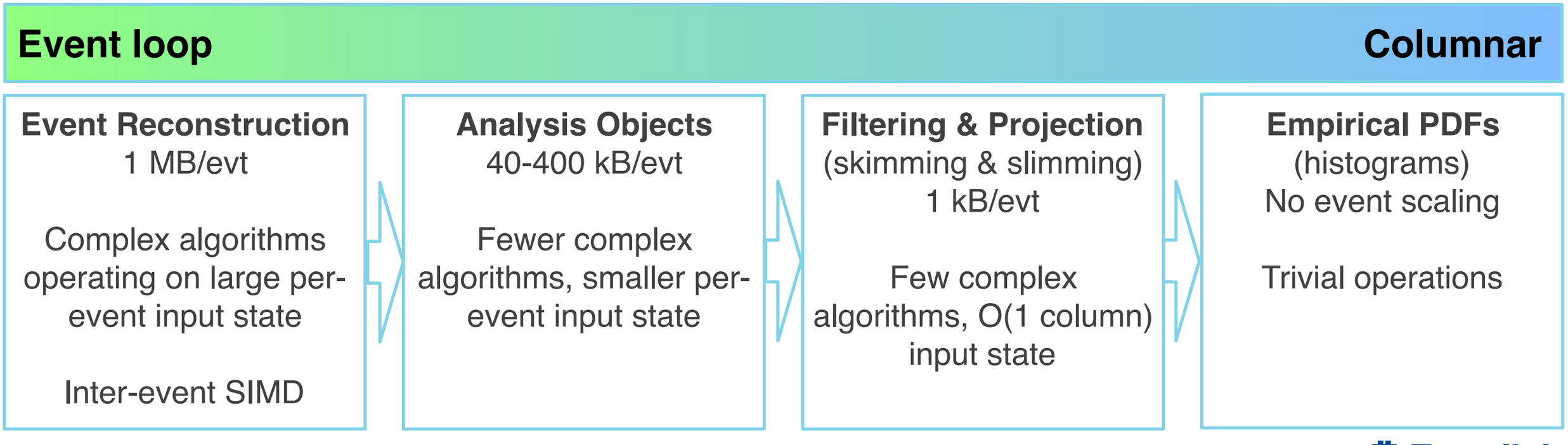
HEP MAPS NICELY INTO COLUMNAR/ARRAY PROGRAMMING

- We (usually) want to do the same computation on every event in our sample
- Some operations (filtering) are implemented non-intuitively
- Doesn't always make sense!



Domain of applicability

- Domain of applicability depends on:
  - Complexity of algorithms
  - Size of per-event input state
- Examples:
  - JEC (binned parametric function): use binary search, masked evaluation: **columnar ok**
  - Object gen-matching, cross-cleaning: min(metric(pairs of offsets)): **columnar ok**
  - Deterministic annealing PV reconstruction: large input state, iterative: **probably not**
- How far back can columnar go?
  - *Missing array programming primitives not a barrier, can always implement our own*



NO INNER LOOPS

- Hiding the loop enables a number of (necessary) optimizations

```
c = []
b = [1,2,3]
a = [4,5,6]
for i in range(size):
    c[i] = a[i] * b[i]
```

Python array multiplication

```
c = numpy.array([])
b = numpy.array([1,2,3])
a = numpy.array([4,5,6])
# NumPy will auto-vectorize this loop
c = a * b
```

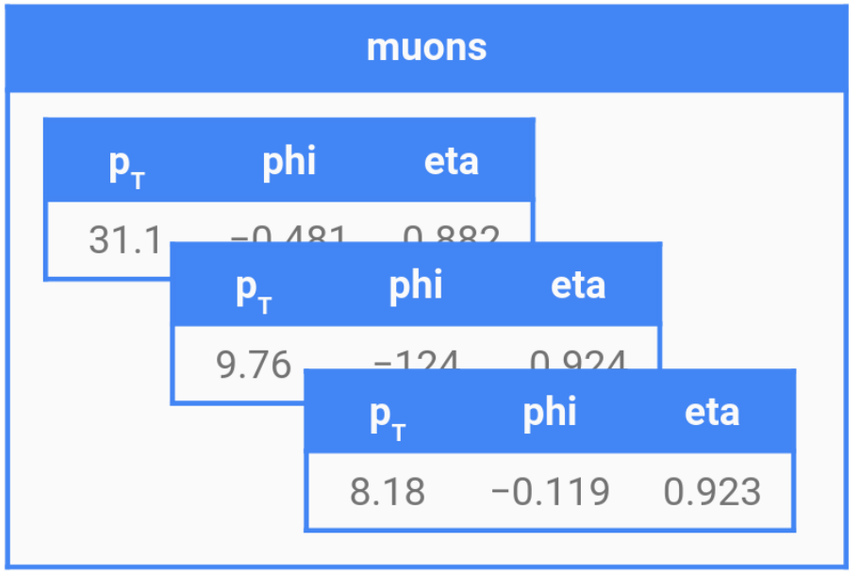
NumPy array multiplication

Calculating the 4-momentum of each muon in every event

is a double loop - how can we efficiently represent that?

A BIT TRICKER THAN EXPECTED

Nested data structures



mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

Objects are essential in HEP analysis.

Many physicists consider TTrees with `std::vector<float>` branches to be "minimal" or "flat."

Most data analysis tools have an SQL mindset, with rectangular data tables.

Objects → rectangular tables is lossy!

Performance claims often start the stopwatch after this "data cleaning."

Nested data structures

```
>>> import uproot
>>> t = uproot.open("tests/samples/HZZ.root")["events"]
>>> t.pandas.df(["MET_px", "Muon_Px", "Electron_Px"], entrystart=-20, flatten=True)
```

entry	subentry	MET_px	Muon_Px	Electron_Px
2401	0	2.998099	-1.492689	NaN
2402	0	27.944883	-4.560287	NaN
2403	0	3.787466	-9.715589	NaN
2404	0	9.378232	-31.072098	NaN
2405	0	-17.310106	47.484627	NaN
	1	NaN	4.695312	NaN
2406	0	-81.965927	74.756172	NaN
	1	NaN	-20.911081	NaN
2407	0	-9.059591	25.786427	NaN
	1	NaN	-29.265024	NaN
2408	0	25.649775	NaN	NaN
2409	0	29.691553	-24.736799	NaN
2410	0	-25.754967	53.005814	-37.681973
	1	NaN	-30.208649	18.453588
2411	0	-2.426847	55.720299	NaN
	1	NaN	-26.914448	NaN
2412	0	-15.611773	14.896802	NaN
2413	0	18.921183	-24.158083	NaN
2414	0	-11.730723	-9.204197	NaN
2415	0	-10.648725	34.506527	NaN
	1	NaN	-31.567780	NaN
2416	0	-14.607650	-39.285824	NaN
2417	0	22.208313	35.067146	NaN
2418	0	18.101646	-29.756786	NaN
2419	0	79.875191	1.141870	NaN
2420	0	19.713749	23.913206	NaN

AWKWARD ARRAYS

https://github.com/scikit-hep/awkward-array

Awkward Array

- Underlying library mapping complex, structured datatypes to NumPy arrays, allowing accelerated mathematical computation

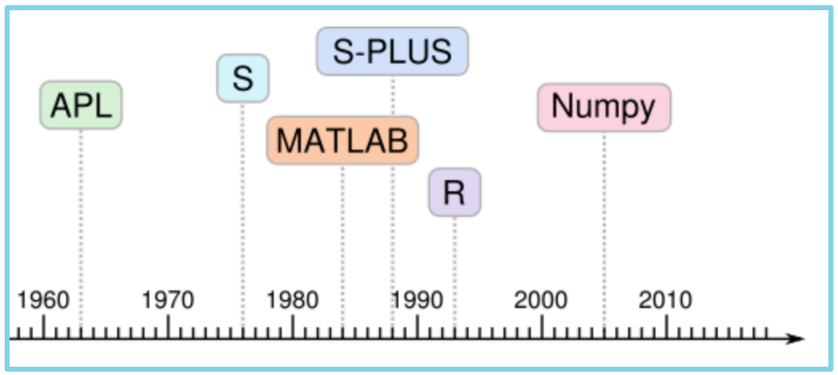
```
import awkward
array = awkward.fromiter([[1.1, 2.2, None, 3.3, None],
                          [4.4, [5.5]],
                          [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]
                         ])
```

```
numpy.sin(array)
# returns <JaggedArray [[0.8912073600614354 0.8084964038195901 None -0.1577456941432482 None]
#                       [-0.951602073889516 [-0.70554033]]
#                       [<Row 0> None <Row 1>]] at 70a40c3a61d0>
```

THIS CALCULATION INVOLVES ZERO PYTHON FOR LOOPS!

Prologue: technology

- Array programming:
  - Simple, composable operations
  - Extensions to manipulate offsets
  - *Not declarative* but towards goal



- Awkward array programming:
  - Extension of numpy syntax
  - Variable-length dimensions: “jagged arrays”
  - View SoA as AoS, familiar object syntax, e.g. p4.pt()
  - References, masks, other useful extensions
  - See awkward, talk by J. Pivarski at ACAT2019

Awkward Array

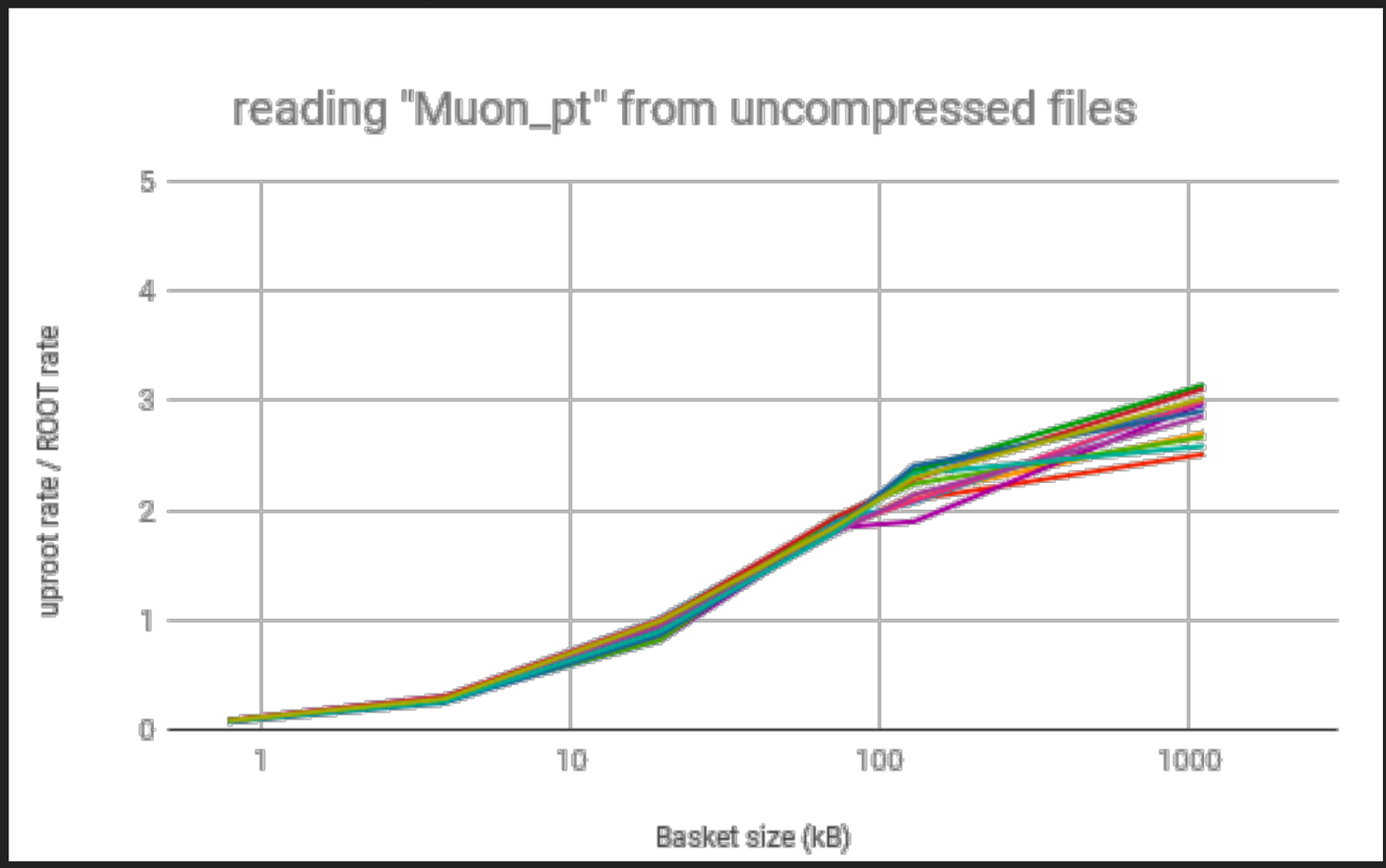
- Coffea framework:
  - Prototype analysis framework utilizing columnar approach
  - Provide lookup tools, histogramming, other ‘missing pieces’ usually f
  - See fnal-column-analysis-tools
    - Functionality will be factorized as it matures

https://indico.cern.ch/event/759388/contributions/3356303/attachments/1816023/2968101/ncsmith-how2019-columnar.pdf

UPROOT

https://github.com/scikit-hep/uproot

- A mature ROOT I/O substitute written in Python on top of NumPy & Awkward Arrays
- Can read TTree directly* into NumPy buffers



COFFEA https://github.com/CoffeaTeam/fnal-column-analysis-tools

- HEP-Specific glue
  - Scale factor lookups
  - 4-Vector math
  - Candidate (e.g. good muon) tools
- Different bits factorized apart as they mature

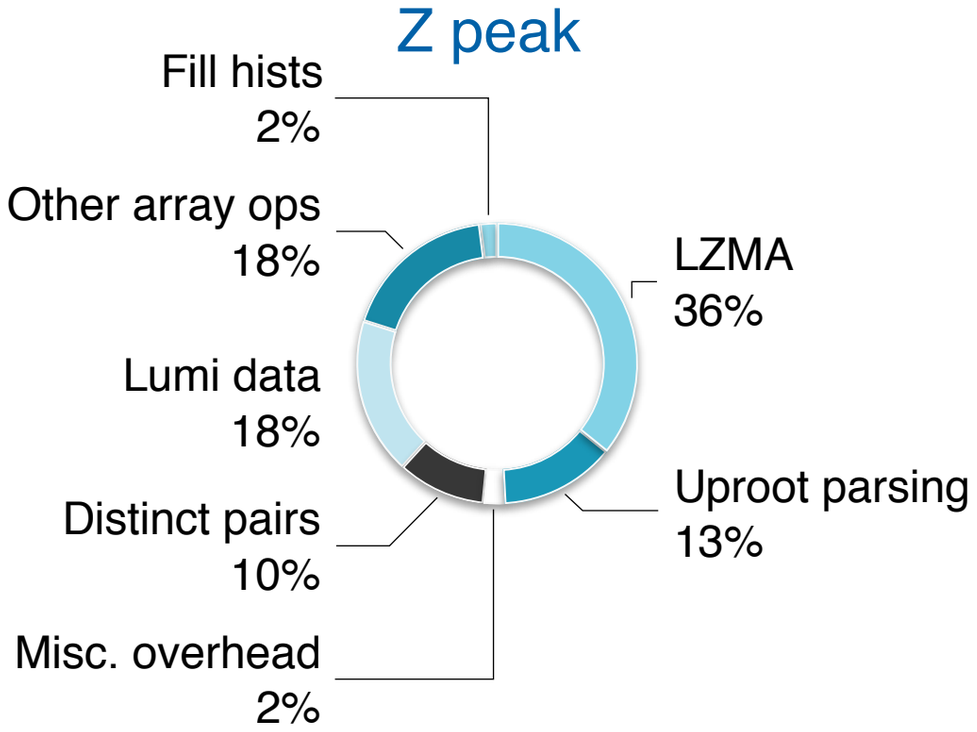
LIVE DEMO HERE (LOOK AT "PLOTTING-DEMO" FIRST)

https://bit.ly/2Yrrd1g

Performance

- Z peak benchmark
  - Includes many typical corrections: lumimask, PU correction, ID scale factors, flavor-categorized
  - 350 lines jupyter notebook, 25 columns accessed
  - 6 *μs*/evt/thread (125 kHz) wall time
    - ROOT C++ TBranch::GetEntry(): ~1.5x faster
- Two prototype analyses
  - “end-to-end” = NanoAOD-like nTuple to templates
  - Varies from 30-150 *μs*/evt/thread
  - Already being used to steer analysis, present results in analysis group meetings
- Many inefficiencies known
  - Can be removed with further development in awkward and helper libraries

```
    def run(self, events, job):
        times = OrderedDict()
        times['start'] = time.time()

        if self.weights_eval is None:
            self.weights_eval = cloudpickle.loads(zlib.decompress(job["weights_eval"]))
        if self.luminask is None:
            self.luminask = cloudpickle.loads(zlib.decompress(job["luminask"]))


        dataset = job['dataset']
        hists = job['hists']

        isRealData = False
        good_trigger = np.zeros(events.nevents, dtype=bool)
        if dataset in self.triggers:
            isRealData = True
            # Real data, prevent overlaps
            for trigger in self.triggers[dataset]:
                good_trigger |= getattr(events, trigger)
            # apply 'golden json' certified lumiblocks
            good_trigger &= luminask(events.run, events.luminosityBlock)
            # record lumis processed
            #  in uproot jobs, full lumi completion guaranteed by full-file processing
            #  in striped jobs, partial lumi completion is not traceable at present
            if dataset == 'DoubleEG':
                hists['lumis_ee'] += LumiList(events.run, events.luminosityBlock)
            elif dataset == 'DoubleMuon':
                hists['lumis_mm'] += LumiList(events.run, events.luminosityBlock)
        else:
            # MC, OR all
            for tlist in self.triggers.values():
                for trigger in tlist:
                    good_trigger |= getattr(events, trigger)
                times['luminask, trigger'] = time.time()
            genw = np.sign(events.LHEWeight_originalXWGTUP)


        electronCols = PhysicalColumnGroup(events, 'Electron', **{k:k for k in self.electron_cols})
        electrons = jaggedFromColumnGroup(electronCols)
        electrons['weight'] = weights_eval["eleScaleFactor_TightId_POG"](electrons.p4.eta, electrons.p4.pt)

        muonCols = PhysicalColumnGroup(events, 'Muon', **{k:k for k in self.muon_cols})
        muons = jaggedFromColumnGroup(muonCols)
        muons['weight'] = weights_eval["muScaleFactor_TightId_Iso"](np.abs(muons.p4.eta), muons.p4.pt)

        ele = electrons[(electrons.p4.pt > 20) &
                                (np.abs(electrons.p4.eta) < 2.5) &
                                (electrons.cutBased >= 4)]

        mu = muons[(muons.p4.pt > 20) &
                           (np.abs(muons.p4.eta) < 2.4) &
                           (muons.tightId > 0)]

        times['good leptons'] = time.time()

        ee = ele.distincts()
        mm = mu.distincts()
        em = ele.cross(mu)

        dileptons = {}
        dileptons['ee'] = ee[(ee.i0.pdgId*ee.i1.pdgId == -11*11) & (ee.i0.p4.pt > 25)]
        dileptons['mm'] = mm[(mm.i0.pdgId*mm.i1.pdgId == -13*13)]
        dileptons['em'] = em[(em.i0.pdgId*em.i1.pdgId == -11*13)]

        times['good pairs'] = time.time()

        channels = {}
        channels['ee'] = good_trigger & (ee.counts == 1) & (mu.counts == 0)
        channels['mm'] = good_trigger & (mm.counts == 1) & (ele.counts == 0)
        channels['em'] = good_trigger & (em.counts == 1) & (ele.counts == 1) & (mu.counts == 1)

        times['channels'] = time.time()

        dupe = np.zeros(events.nevents, dtype=bool)
        tot = 0
        for channel, cut in channels.items():
            scands = dileptons[channel][cut]
            dupe |= cut
            tot += cut.sum()
            weight = np.array(1.)
            if not isRealData:
                weight = scands.i0['weight'] * scands.i1['weight'] * genw[cut]
                hists['genw'].fill(dataset=dataset, genw=genw)
            hists['lepton_pt'].fill(dataset=dataset, channel=channel,
                                    lep0_pt=scands.i0.pt.flatten(),
                                    lep1_pt=scands.i0.pt.flatten(),
                                    weight=weight.flatten(),
                                    )
            hists['sMass'].fill(dataset=dataset, channel=channel,
                                mass=scands.p4.mass.flatten(),
                                weight=weight.flatten(),
                                )

        if dupe.sum() != tot:
            raise Exception('Double-counting events!')

        times['plots'] = time.time()

        #profiling info
        t = list(times.values())
        for i, name in enumerate(times):
            if i==0: continue
            dt = t[i] - t[i-1]
            hists['profile'].fill(op=name, dt=1e6*(dt/len(events.Muon.count)))
        hists['profile'].fill(op="total", dt=1e6*(t[-1]-t[0])/events.nevents)

        job.send(hists=hists)
```



Fermilab

BIG DATA + HEP (OR YOUR FAVORITE "P")?

- All the plumbing exists and the tools are maturing
- I'd recommend exploring the Uproot docs/tutorials
  - Solid now and getting better weekly
- Movement away from single monolith to ecosystem of smaller parts
  - Do one thing well
- Make analysis easier!

Data delivery system	*Z peak wall-time throughput*	*Subjective 'ease of use'*
uproot on laptop	*~ 100 kHz*	*5/5*
uproot + xrootd + multiprocessing	*~ 250 kHz @ 10 cores **	*5/5*
uproot + condor jobs	*Arbitrary*	*3/5*
striped system	*~ 10 MHz @ 100 cores*	*2/5*
Apache spark	*~ 1 MHz @ 100 cores ***	*4/5*

GLOBAL RECAP & THOUGHTS

- Everyone wants to look at more data in less time
  - Hardware trends make this hard
- Everyone is developing novel solutions to this problem
  - I only had time to skim the surface
- We've now got all (okay, most) of the necessary physics-specific glue to exploit common solutions
- In my opinion, these techniques will become increasingly popular

QUESTIONS?