



Data-Intensive Distributed Computing

CS 431/631 451/651 (Winter 2019)

Part 1: MapReduce Algorithm Design (1/4)
January 8, 2019

Adam Roegiest
Kira Systems

These slides are available at <http://roegiest.com/bigdata-2019w/>



This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details

Agenda for Today

Who am I?

What is big data?

Why big data?

What is this course about?

Administrivia

Who am I?

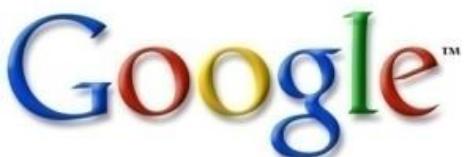
PhD from Waterloo (2017)

TA for this course in its first UW offering

Research Scientist at Kira Systems (now)



Big Data



Processes 20 PB a day (2008)
Crawls 20B web pages a day (2012)
Search index is 100+ PB (5/2014)
Bigtable serves 2+ EB, 600M QPS (5/2014)



400B pages,
10+ PB (2/2014)



19 Hadoop clusters: 600
PB, 40k servers (9/2015)



Hadoop: 10K nodes, 150K
cores, 150 PB (4/2014)

300 PB data in Hive +
600 TB/day (4/2014)



S3: 2T objects, 1.1M
request/second (4/2013)

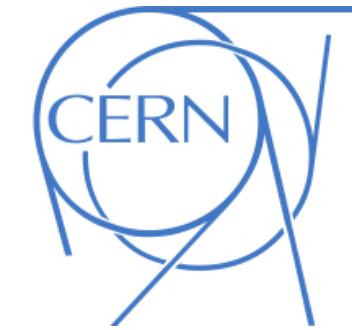
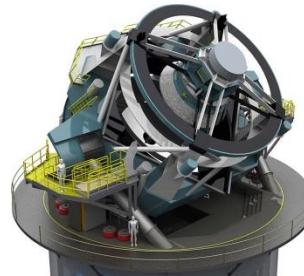


640K ought to be
enough for
anybody.



150 PB on 50k+ servers
running 15k apps (6/2011)

LHC: ~15 PB a year



LSST: 6-10 PB a year
(~2020)



SKA: 0.3 – 1.5 EB
per year (~2020)

How much data?



Why big data? Science
Business
Society



Science

Emergence of the 4th Paradigm

Data-intensive e-Science

Business

Data-driven decisions

Data-driven products





Society

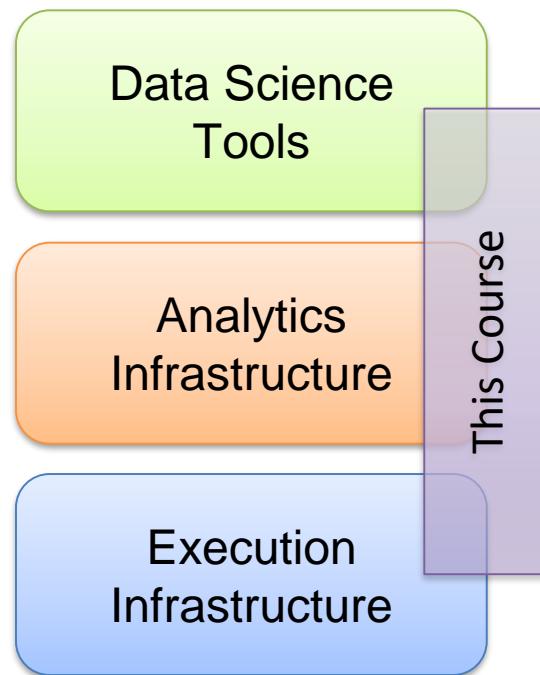
Humans as social sensors

Computational social science



Source: Popular Internet Meme

What is this course about?

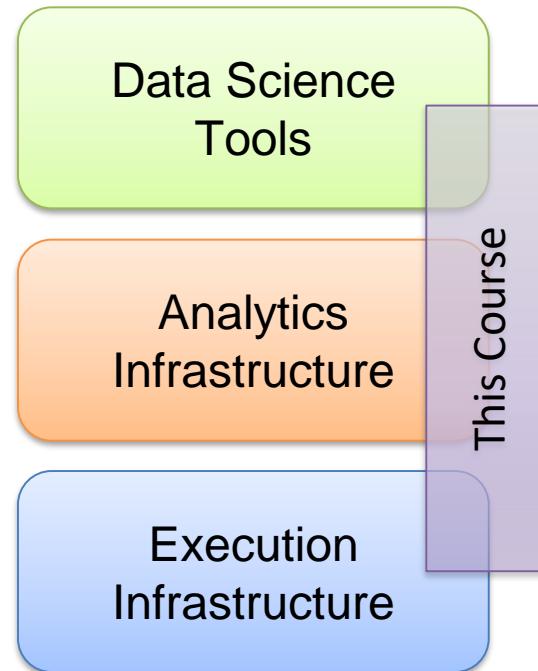


“big data stack”

Buzzwords

data science, data analytics,
business intelligence, data
warehouses and data lakes

MapReduce, Spark, Flink,
Pig, Dryad, Hive, Dryad,
noSQL, Pregel, Giraph,
Storm/Heron



“big data stack”

Text: frequency estimation,
language models, inverted
indexes

Graphs: graph traversals,
random walks (PageRank)

Relational data: SQL, joins,
column stores

Data mining: hashing,
clustering (k -means),
classification,
recommendations

Streams: probabilistic data
structures (Bloom filters,
CMS, HLL counters)

This course focuses on algorithm design and “thinking at scale”

Structure of the Course

Analyzing Text

Analyzing Graphs

Analyzing
Relational Data

Data Mining and
Machine Learning

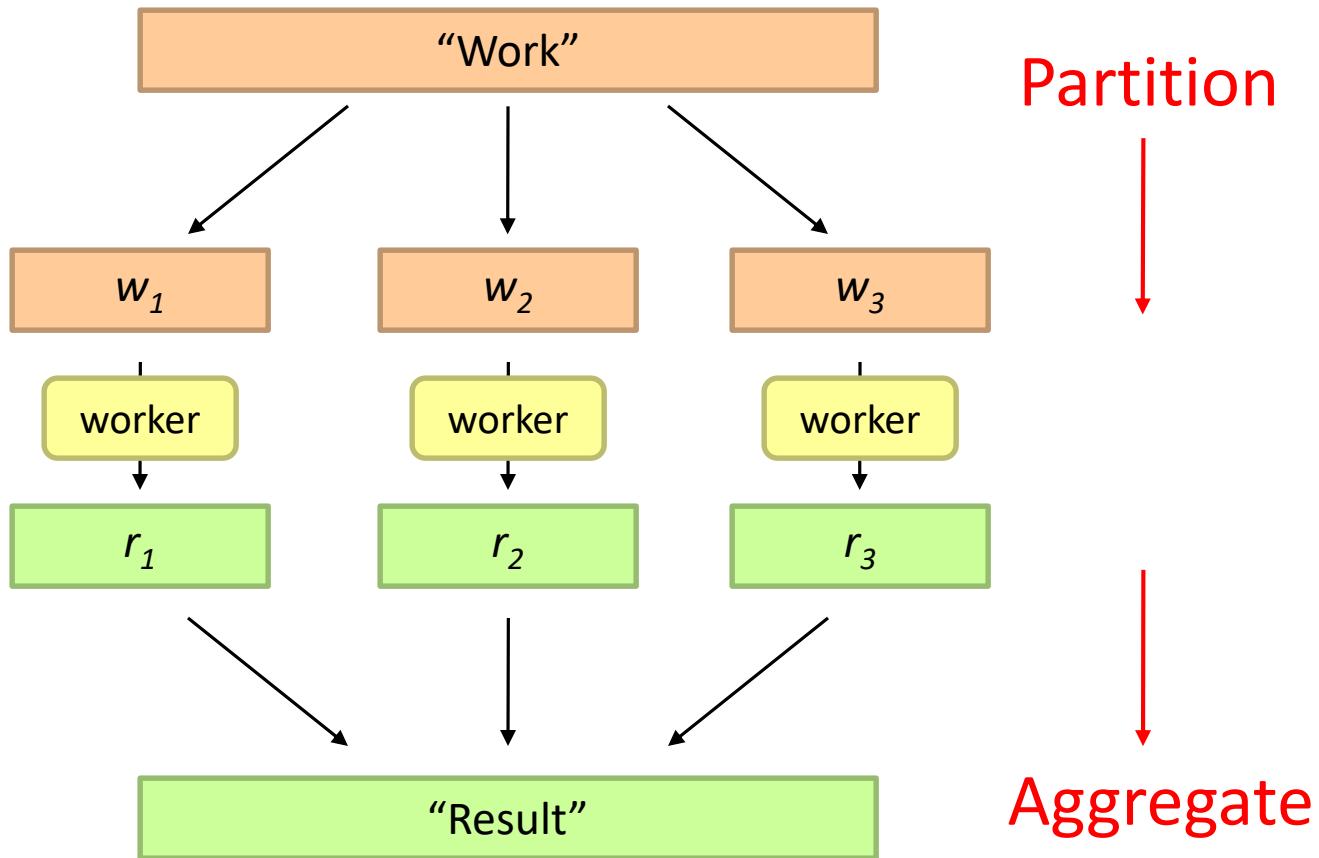
What's beyond batch processing?

“Core” framework features and
algorithm design for batch processing

A wide-angle photograph of a massive server room. The space is filled with floor-to-ceiling server racks, their front panels glowing with various colors like blue, green, and yellow. A complex network of overhead steel trusses and beams supports a grid of fluorescent light fixtures. The floor is a polished concrete surface. In the background, a large wooden wall and more server racks are visible under a dark, industrial-style ceiling.

Tackling Big Data

Divide and Conquer



Parallelization Challenges

How do we assign work units to workers?

What if we have more work units than workers?

What if workers need to communicate partial results?

What if workers need to access shared resources?

How do we know when a worker has finished? (Or is simply waiting?)

What if workers die?

Difficult because:

We don't know the order in which workers run...

We don't know when workers interrupt each other...

We don't know when workers need to communicate partial results...

We don't know the order in which workers access shared resources...

What's the common theme of all of these challenges?

Common Theme?

Parallelization challenges arise from:

- Need to communicate partial results
- Need to access shared resources

(In other words, sharing state)

How do we tackle these challenges?

“Current” Tools

Basic primitives

Semaphores (lock, unlock)

Conditional variables (wait, notify, broadcast)

Barriers

Awareness of Common Problems

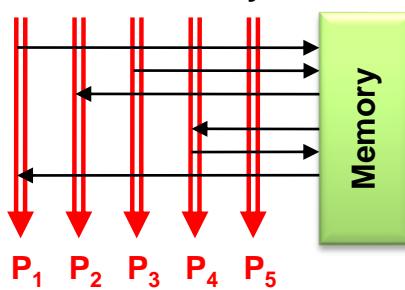
Deadlock, livelock, race conditions...

Dining philosophers, sleeping barbers, cigarette smokers...

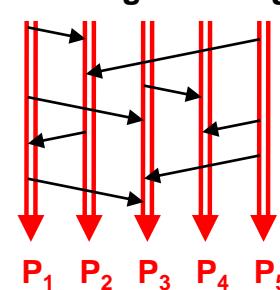
“Current” Tools

Programming Models

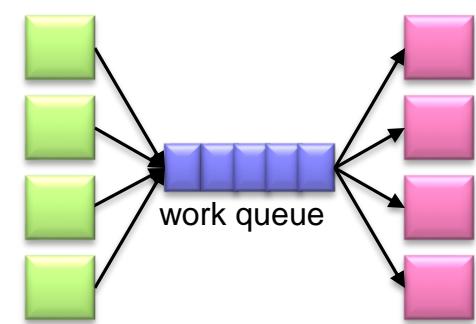
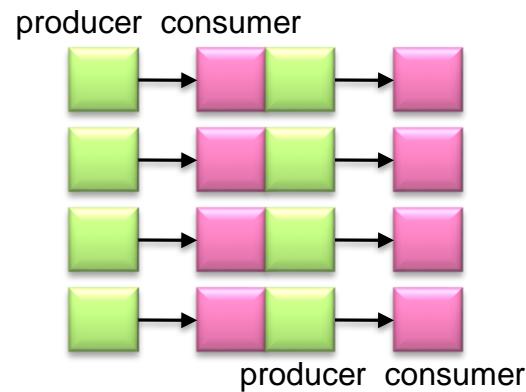
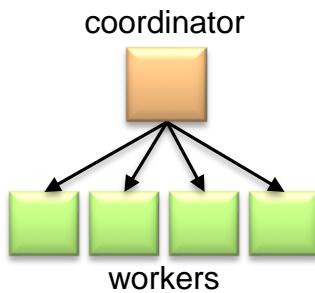
Shared Memory



Message Passing



Design Patterns



When Theory Meets Practices

Concurrency is already difficult to reason about...

Now throw in:

The scale of clusters and (multiple) datacenters

The presence of hardware failures and software bugs

The presence of multiple interacting services

The reality:

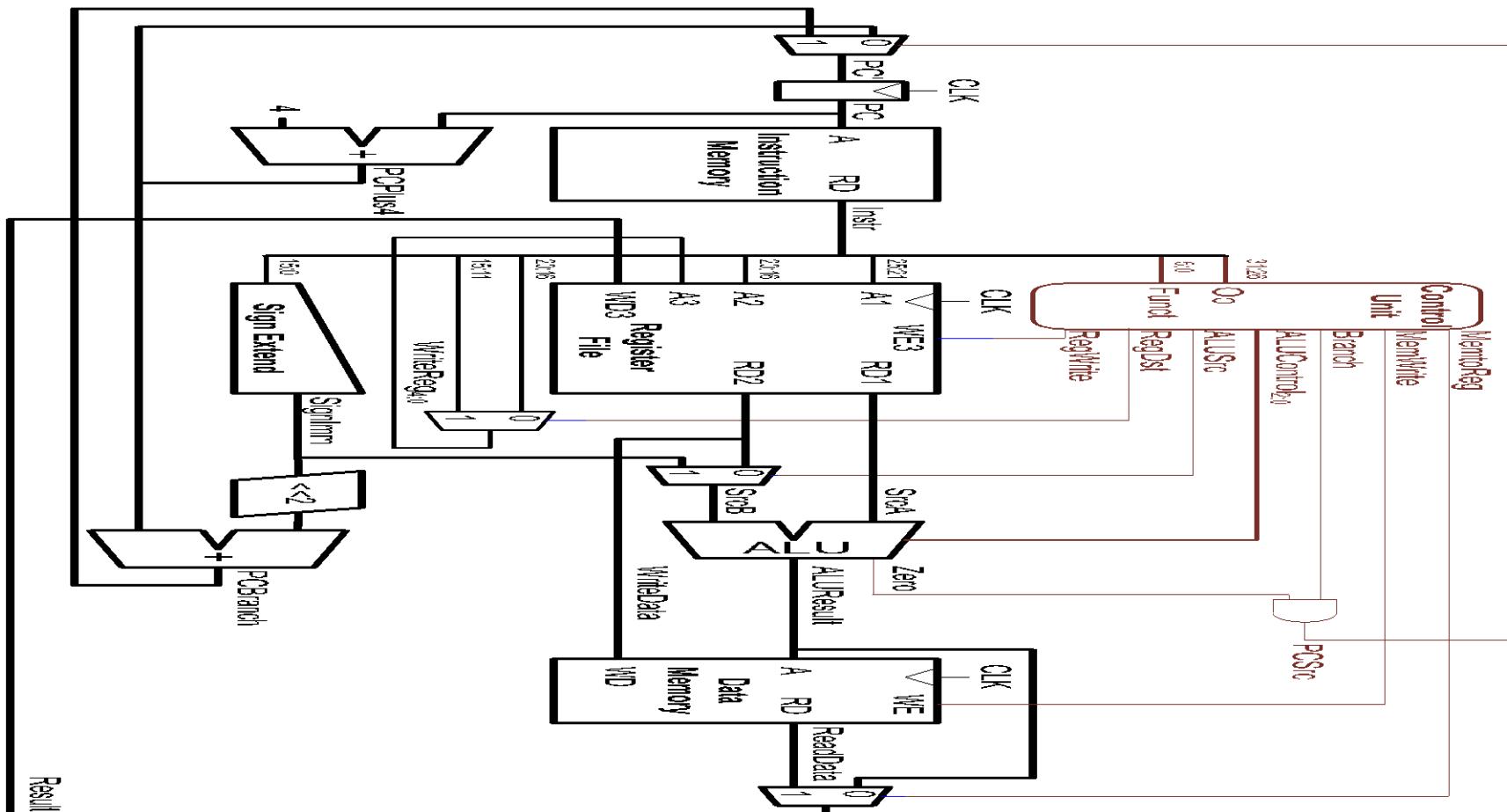
Lots of one-off solutions, custom code

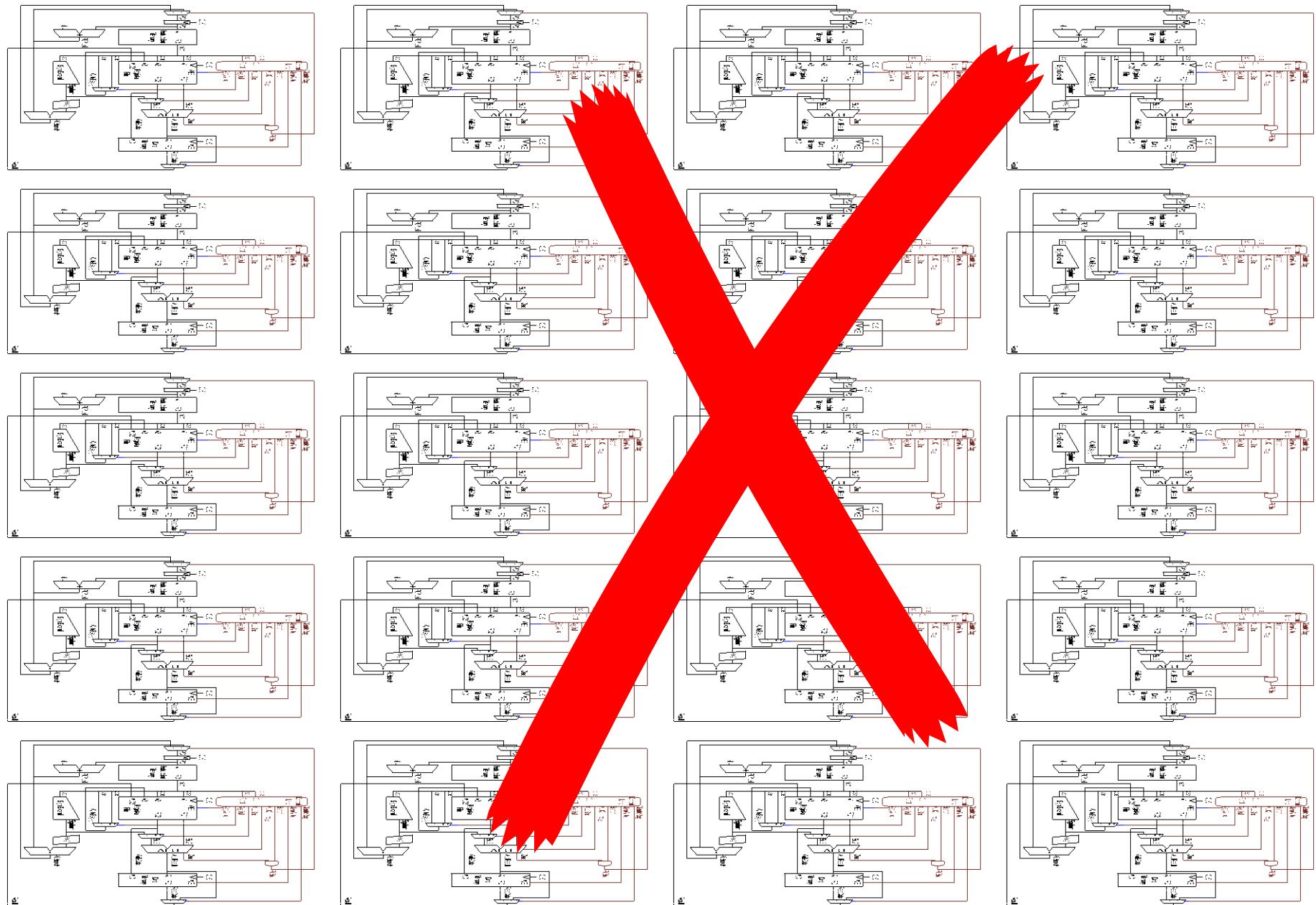
Write your own dedicated library, then program with it

Burden on the programmer to explicitly manage everything

Bottom line: it's hard!







An aerial photograph of a large datacenter complex during sunset. The sky is a vibrant orange and yellow. In the foreground, there are several large industrial buildings, parking lots, and rows of white shipping containers. A major highway runs through the middle ground, with some traffic visible. The background shows a vast, flat landscape with green fields and distant hills under the setting sun.

The datacenter *is* the computer!

The datacenter *is* the computer!

It's all about the right level of abstraction

Moving beyond the von Neumann architecture

What's the “instruction set” of the datacenter computer?

Hide system-level details from the developers

No more race conditions, lock contention, etc.

No need to explicitly worry about reliability, fault tolerance, etc.

Separating the *what* from the *how*

Developer specifies the computation that needs to be performed

Execution framework (“runtime”) handles actual execution

MapReduce is the first instantiation of this idea... but not the last!



MapReduce

What's different?

Data-intensive vs. Compute-intensive
Focus on *data-parallel* abstractions

Coarse-grained vs. Fine-grained parallelism
Focus on *coarse-grained data-parallel* abstractions

Logical vs. Physical

Different levels of design:

“Logical” deals with abstract organizations of computing
“Physical” deals with how those abstractions are realized

Examples:

Scheduling
Operators
Data models
Network topology

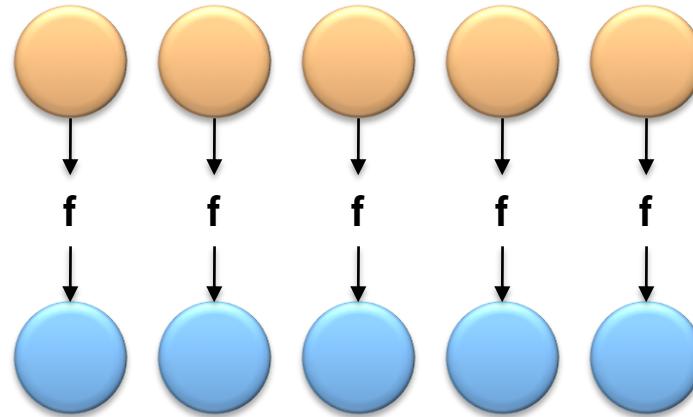
Why is this important?

Roots in Functional Programming

Simplest data-parallel abstraction

Process a large number of records: “do” something to each

Map



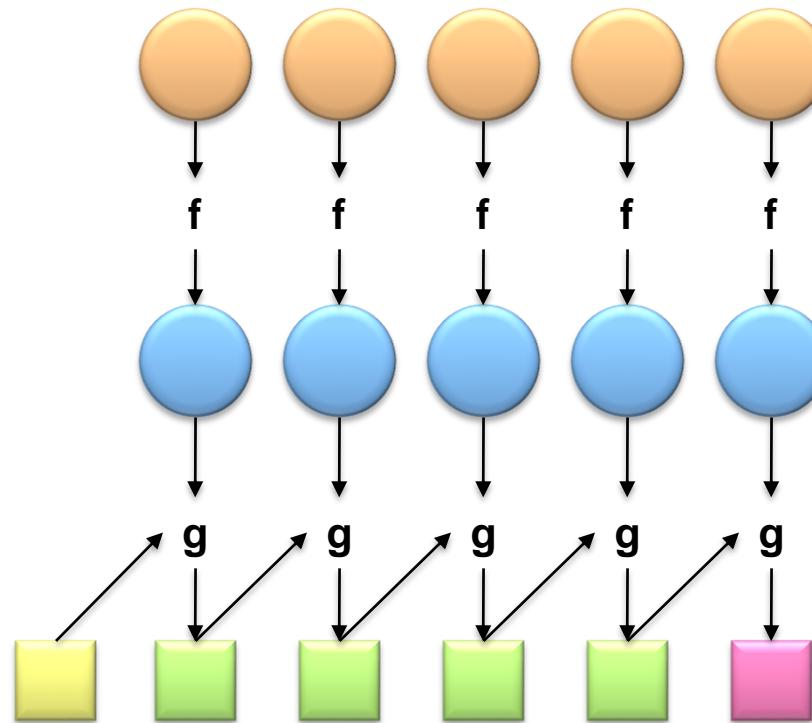
We need something more for sharing partial results across records!

Roots in Functional Programming

Let's add in aggregation!

Map

Fold



MapReduce = Functional programming + distributed computing!

Functional Programming in Scala

```
scala> val t = Array(1, 2, 3, 4, 5)
t: Array[Int] = Array(1, 2, 3, 4, 5)
```

```
scala> t.map(n => n*n)
res0: Array[Int] = Array(1, 4, 9, 16, 25)
```

```
scala> t.map(n => n*n).foldLeft(0)((m, n) => m + n)
res1: Int = 55
```

Imagine parallelizing the map and fold across a cluster...

A Data-Parallel Abstraction

Process a large number of records

Map “Do something” to each

Group intermediate results

“Aggregate” intermediate results
Reduce

Write final results

Key idea: provide a functional abstraction for these two operations

MapReduce

Programmer specifies two functions:

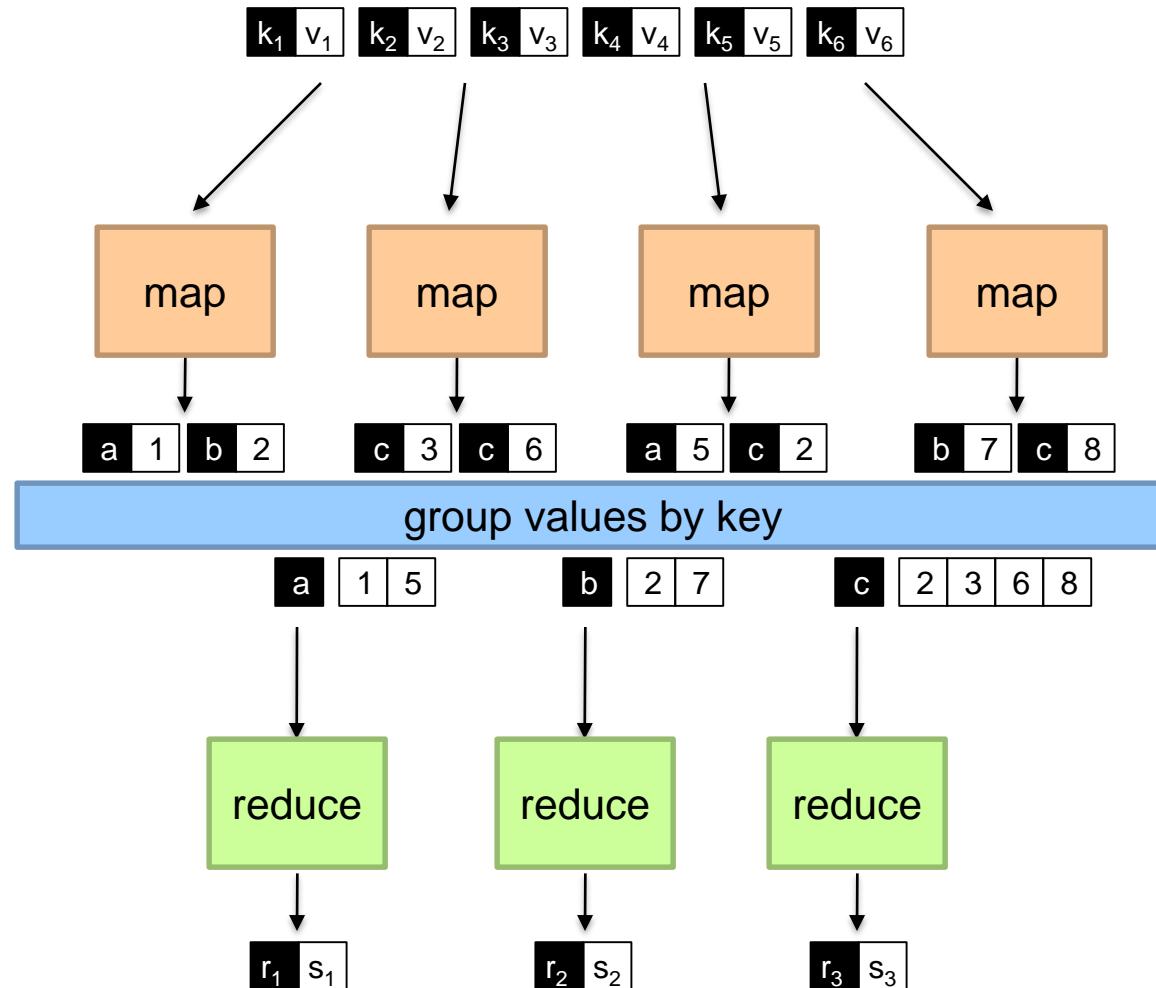
$\text{map } (k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]$

$\text{reduce } (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]$

All values with the same key are sent to the same reducer

What does this actually mean?

The execution framework handles everything else...



MapReduce

Programmer specifies two functions:

map (k_1, v_1) \rightarrow List[(k_2, v_2)]

reduce (k_2 , List[v_2]) \rightarrow List[(k_3, v_3)]

All values with the same key are sent to the same reducer

The execution framework handles everything else...

What's “everything else”?

MapReduce “Runtime”

Handles scheduling

Assigns workers to map and reduce tasks

Handles “data distribution”

Moves processes to data

Handles synchronization

Groups intermediate data

Handles errors and faults

Detects worker failures and restarts

Everything happens on top of a distributed FS (later)

MapReduce

Programmer specifies two functions:

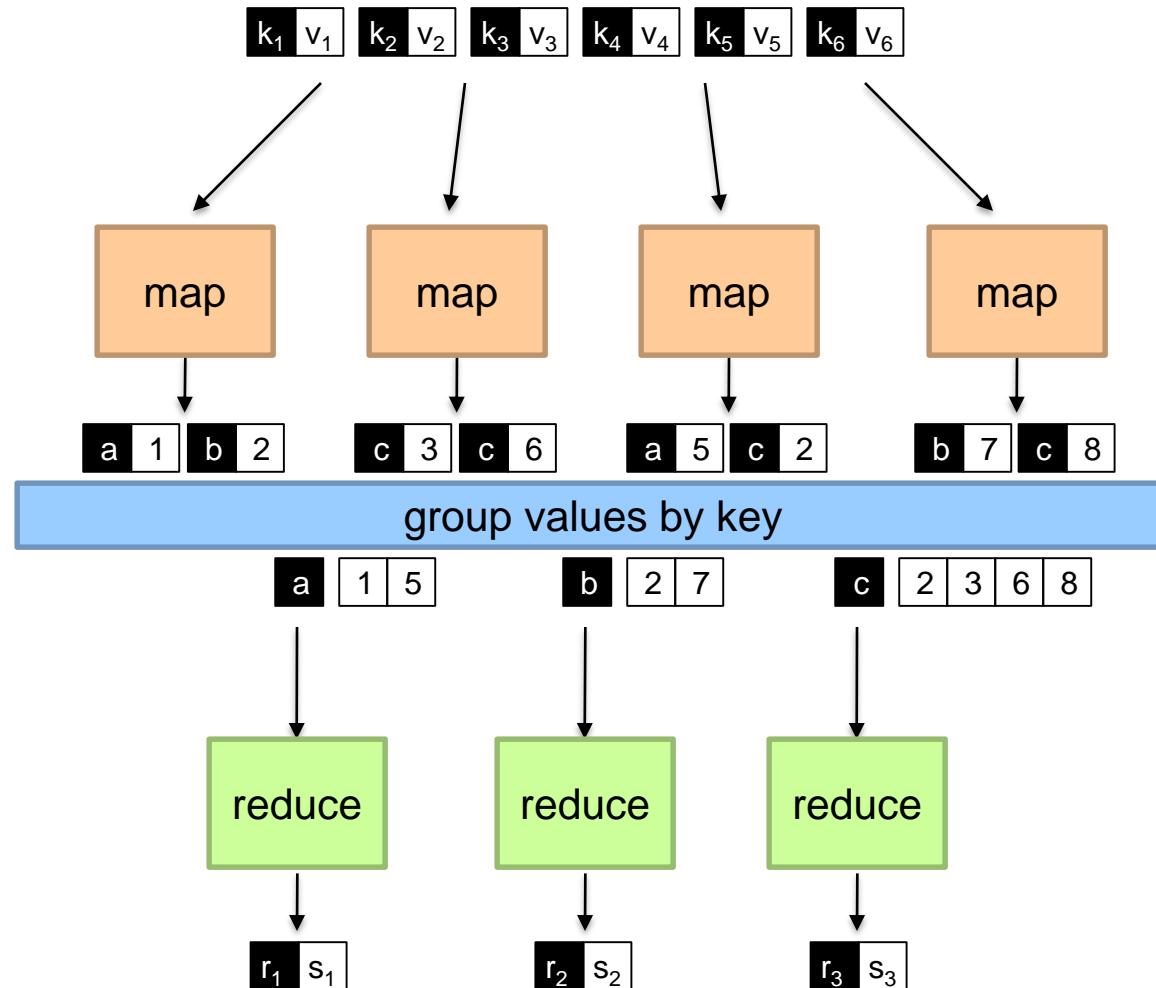
map (k_1, v_1) \rightarrow List[(k_2, v_2)]

reduce (k_2 , List[v_2]) \rightarrow List[(k_3, v_3)]

All values with the same key are sent to the same reducer

The execution framework handles everything else...

Not quite...



What's the most complex and slowest operation here?

MapReduce

Programmer specifies two functions:

map (k_1, v_1) \rightarrow List[(k_2, v_2)]

reduce (k_2 , List[v_2]) \rightarrow List[(k_3, v_3)]

All values with the same key are sent to the same reducer

partition (k' , p) \rightarrow 0 ... p-1

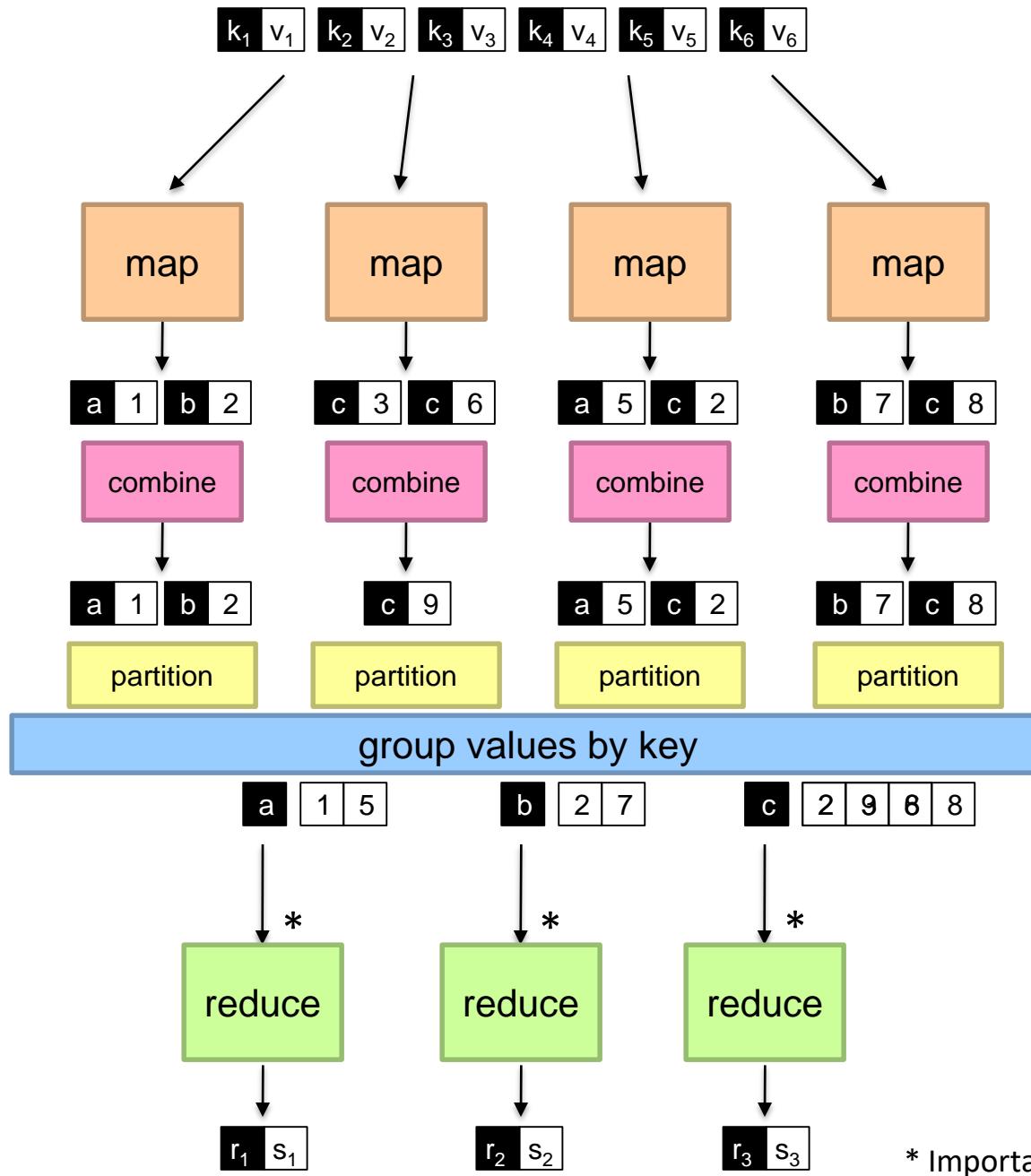
Often a simple hash of the key, e.g., $\text{hash}(k') \bmod n$

Divides up key space for parallel reduce operations

combine (k_2 , List[v_2]) \rightarrow List[(k_2, v_2)]

Mini-reducers that run in memory after the map phase

Used as an optimization to reduce network traffic



“Hello World” MapReduce: Word Count

```
def map(key: Long, value: String) = {
    for (word <- tokenize(value)) {
        emit(word, 1)
    }
}
```

```
def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
        sum += value
    }
    emit(key, sum)
}
```

MapReduce can refer to...

The programming model

The execution framework (aka “runtime”)

The specific implementation

Usage is usually clear from context!

MapReduce Implementations

Google has a proprietary implementation in C++
Bindings in Java, Python

Hadoop provides an open-source implementation in Java
Development begun by Yahoo, later an Apache project
Used in production at Facebook, Twitter, LinkedIn, Netflix, ...
Large and expanding software ecosystem
Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations



A grid of approximately 100 small wooden stick figures arranged in 10 rows and 10 columns. Each figure has a small wooden head, a thin body, and thin legs. They are wearing simple, triangular wooden dresses in various colors: yellow, orange, red, maroon, purple, blue, and green. The figures are positioned with their arms raised and legs spread, resembling a group of dancing or celebrating figures.

Course Administrivia

Four in One!

CS 451/651 431/631 all meet together

CS 451: version for CS ugrads (most students)

CS 651: version for CS grads

CS 431: version for non-CS ugrads

CS 631: version for non-CS grads

Course instructors

Adam Roegiest: The guy talking right now

ISAs: Alex Weatherhead, Matt Guiol

TAs: Ryan Clancy, Peng Shi, Yao Lu, Wei (Victor) Yang

Important Coordinates

Course website:

<http://roegiest.com/bigdata-2019w/>

Lots of info there, read it!

("I didn't see it" will not be accepted as an excuse)

Communicating with us:

Piazza for general questions ([link on course homepage](#))

uwaterloo-bigdata-2019w-staff@googlegroups.com

(Mailing list reaches all course staff – use Piazza unless it's personal)

Bespin

<http://bespin.io/>

Course Design

This course focuses on algorithm design and “thinking at scale”

Not the “mechanics” (API, command-line invocations, et.)

You’re expected to pick up MapReduce/Spark with minimal help

Components of the final grade:

6 (CS 431) or 8 (CS 451) individual assignments

Final exam

Additional group final project (CS 631/651)

Expectations (CS 451)

Your background:

Pre-reqs: CS 341, CS 348, CS 350

Comfortable in Java and Scala (or be ready to pick it up quickly)

Know how to use Git

Reasonable “command-line”-fu skills

Experience in compiling, patching, and installing open source software

Good debugging skills

You are:

Genuinely interested in the topic

Be prepared to put in the time

Comfortable with rapidly-evolving software

MapReduce/Spark Environments (CS 451)

See “Software” page in course homepage for instructions

Single-Node Hadoop: Linux Student CS Environment

Everything is set up for you, just follow instructions

We'll make sure everything works

Single-Node Hadoop: Local installations

Install all software components on your own machine

Requires at least 4GB RAM and plenty of disk space

Works fine on Mac and Linux, YMMV on Windows

Important: For your convenience only!

We'll provide basic instructions, but not technical support

Distributed Hadoop: Datasci Cluster

New feature this offering!

Assignment Mechanics (CS 451)

We'll be using private GitHub repos for assignments

Complete your assignments, push to GitHub

We'll pull your repos at the deadline and grade

Note late policy (details on course homepage)

Late by up to 24 hours: 25% reduction in grade

Late 24-48 hours: 50% reduction in grade

Late by more than 48 hours: not accepted

By assumption, we'll pull and mark at deadline:

If you want us to hold off, you must let us know!

Important: Register for (free) GitHub educational account!

https://education.github.com/discount_requests/new

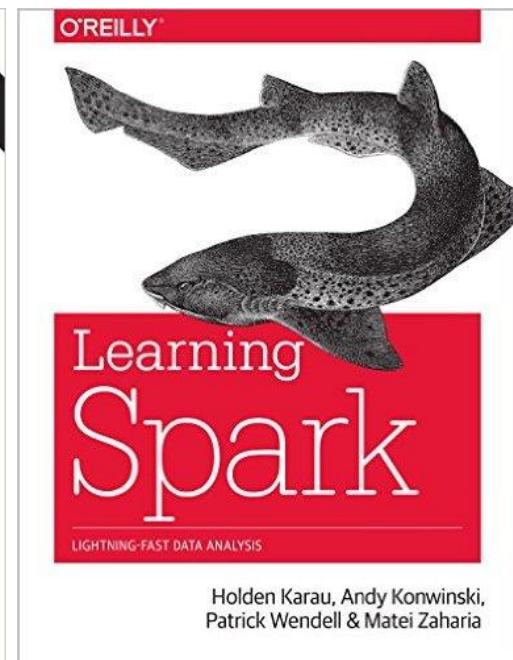
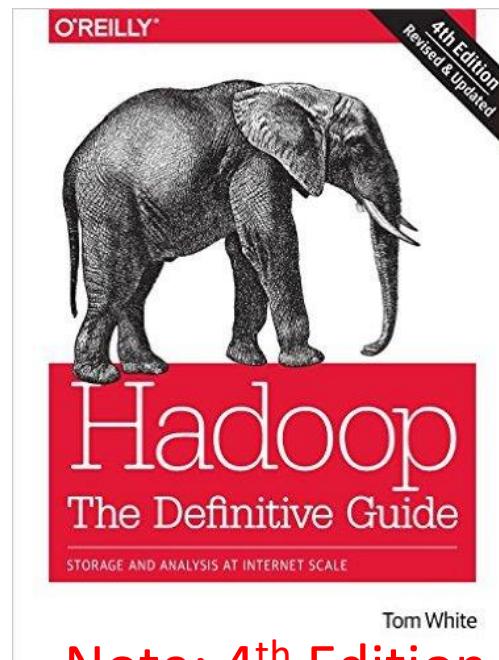
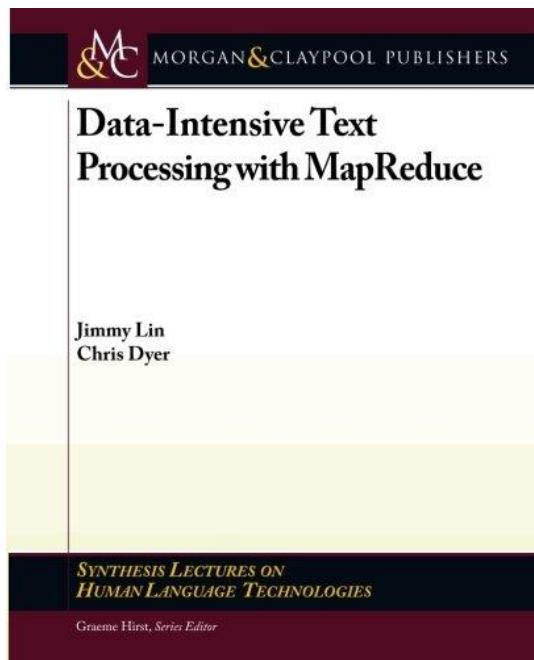
Assignment Mechanics (CS 431)

Assignments will use Python and Jupyter
Everything you need to know is in the assignment itself

Assignments will generally be submitted using Marmoset
Details are on the course website for the appropriate assignment

Course Materials

One (required) textbook +
Two (optional but recommended) books +
Additional readings from other sources as appropriate



Note: 4th Edition
(optional but
recommended)

If you're not (yet) registered:

Register for the wait list at:

By sending Adam an email at aroegies@uwaterloo.ca

Priority for unregistered students

CS students

Have all the pre-reqs

Final opportunity to take the course (e.g., 4B students)

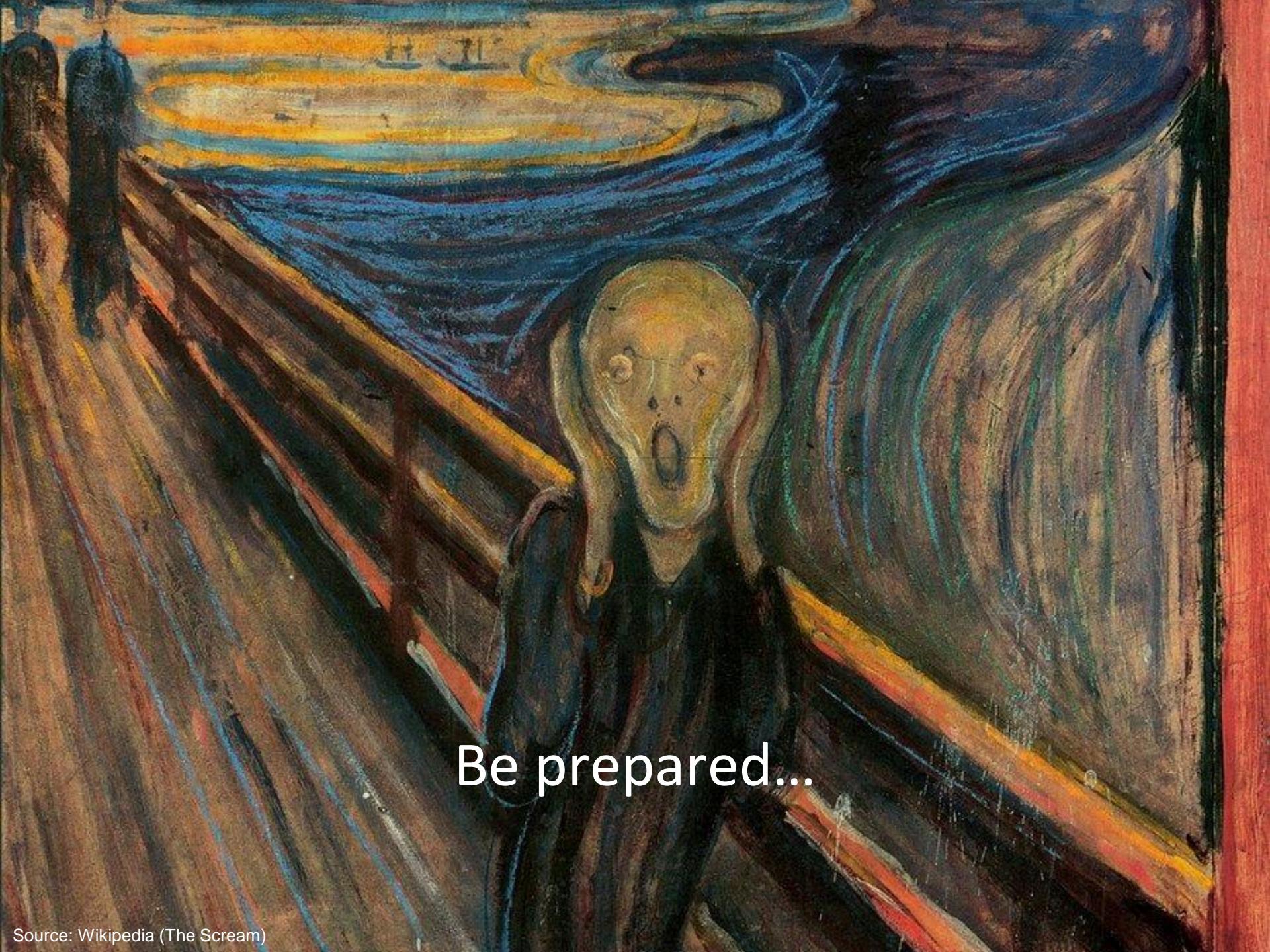
Continue to attend class until final decision

Once the course is full, it is *full*

Note: late registration is not an excuse for late assignments



Luke: I won't fail you. I'm not afraid.
Yoda: You will be. You... will... be.



Be prepared...

“Hadoop Zen”

Parts of the ecosystem are *still* immature

We've come a long way since 2007, but still far to go...

Bugs, undocumented “features”, inexplicable behavior, etc.

Different versions = major pain

Don't get frustrated (take a deep breath)...

Those W\$*#T@F! moments

Be patient...

We will inevitably encounter “situations” along the way

Be flexible...

We will have to be creative in workarounds

Be constructive...

Tell me how I can make everyone's experience better

A photograph of a traditional Japanese rock garden. In the foreground, a gravel path is raked into fine, parallel lines. Several large, dark, irregular stones are scattered across the garden. A small, shallow pond with a few rocks is located in the center-right. The background features a variety of trees and shrubs, some with autumn-colored leaves, and a building with a tiled roof. The overall atmosphere is serene and minimalist.

“Hadoop Zen”

A photograph of a traditional Japanese rock garden. In the foreground, there is a gravel path with several large, light-colored stones. Beyond the path is a small, shallow pond containing several large, dark, irregular stones. The background features a well-maintained lawn with various shrubs and trees, some with autumn-colored leaves. In the far background, there are traditional Japanese buildings with dark, tiled roofs.

Questions?

To Do:

1. Bookmark course homepage
2. Get on Piazza
3. Register for GitHub educational account