UNIVERSITY OF WATERLOO

Data-Intensive Distributed Computing

CS 451/651 (Fall 2018)

Part 1: MapReduce Algorithm Design (1/4)

September 6, 2018

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2018f/

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

Source: Wikipedia (All Souls College, Oxford)

Source: Wikipedia (Factory)

Source: Wikipedia (All Souls College, Oxford)

Big Data

Source: Wikipedia (Hard disk drive)

Google

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)

INTERNET ARCHIVE

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

YAHOO!

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

JPMorganChase

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

ebay

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

facebook

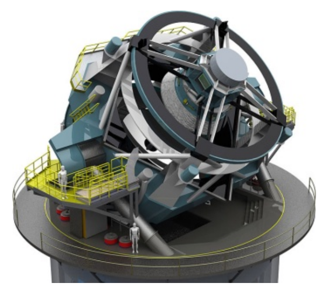
300 PB data in Hive + 600 TB/day (4/2014)

LHC: ~15 PB a year

CERN

amazon web services

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



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

640K ought to be enough for anybody.

SKA: 0.3 – 1.5 EB per year (~2020)

How much data?

Why big data?

Science
Business
Society

Source: Wikipedia (Everest)

Science

Emergence of the 4th Paradigm

Data-intensive e-Science

Maximilien Brice, © CERN

Business

Data-driven decisions

Data-driven products

Source: Wikiedia (Shinjuku, Tokyo)

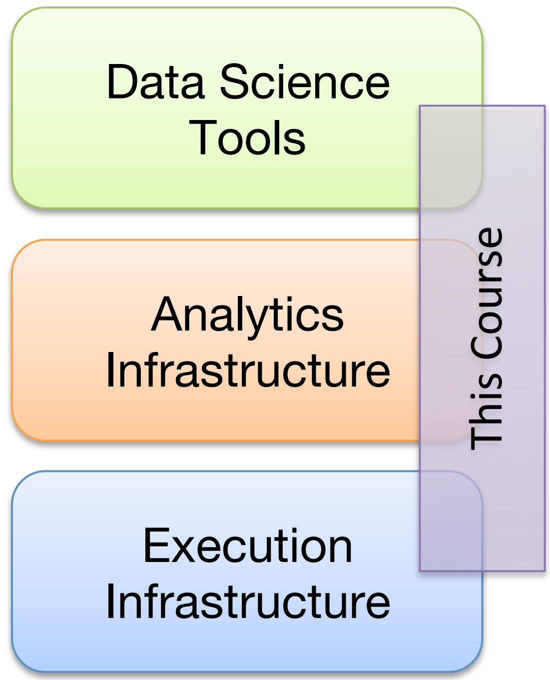
Society

- Humans as social sensors
- Computational social science

Source: Guardian

Source: Popular Internet Meme

What is this course about?

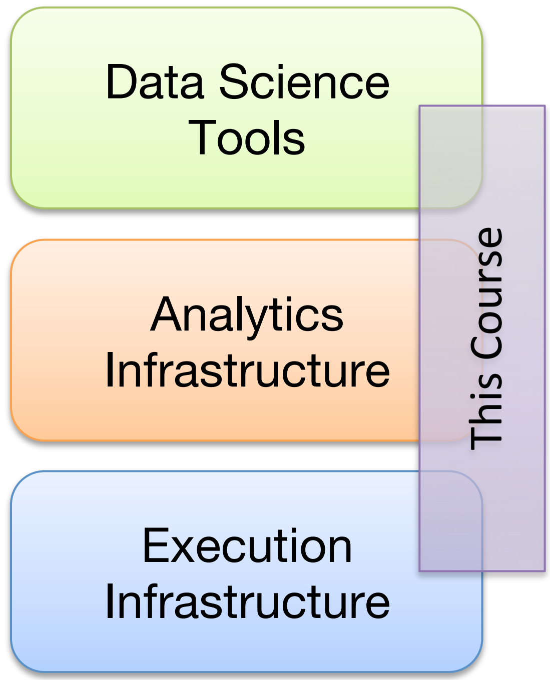


"big data stack"

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)

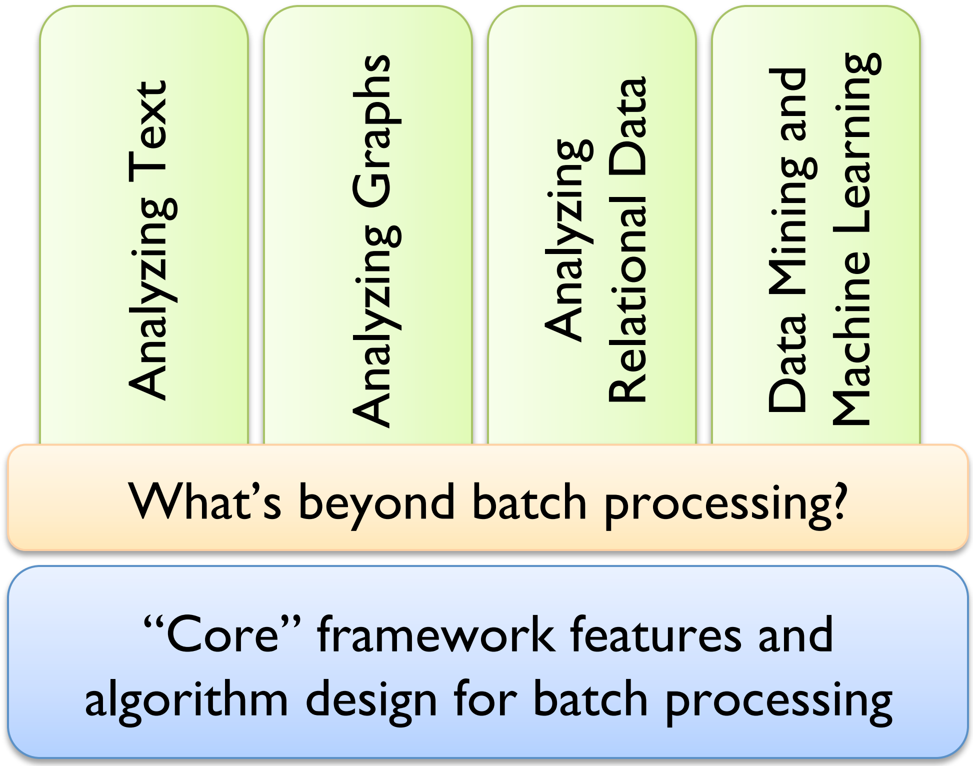
Relational data: SQL, joins, column stores

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"

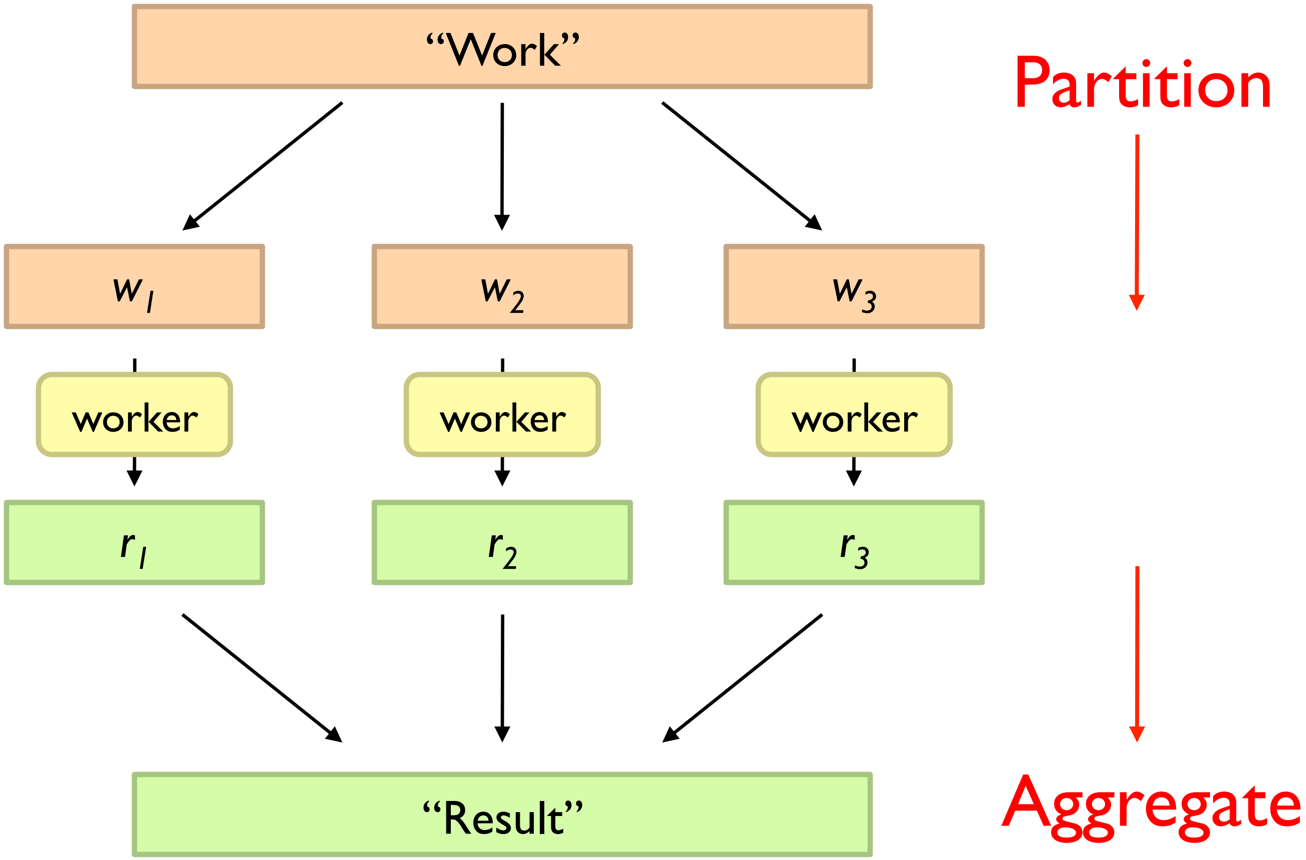
Structure of the Course



Tackling Big Data

Source: Google

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

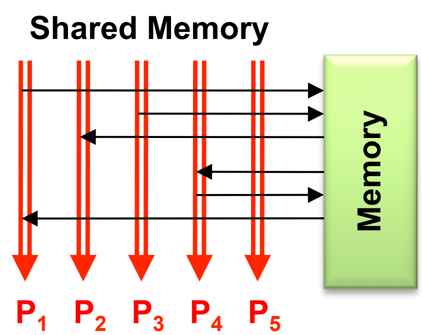
Semaphores (lock, unlock)
Conditional variables (wait, notify, broadcast)
Barriers

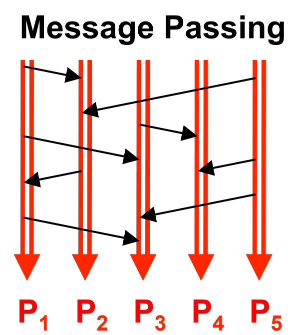
Awareness of Common Problems

Deadlock, livelock, race conditions...
Dining philosophers, sleeping barbers, cigarette smokers...

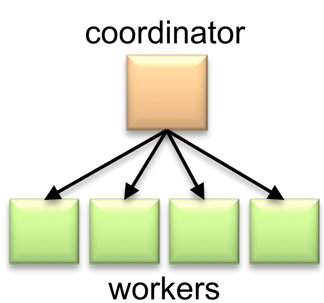
“Current” Tools

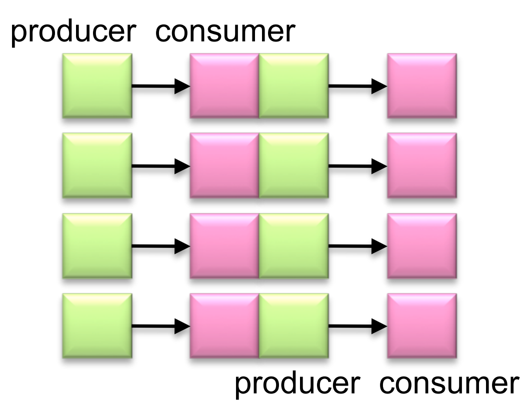
Programming Models

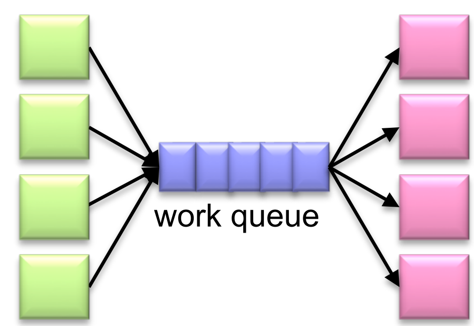




Design Patterns







When Theory Meets Practices

Concurrency is already difficult to reason about…

Now throw in:

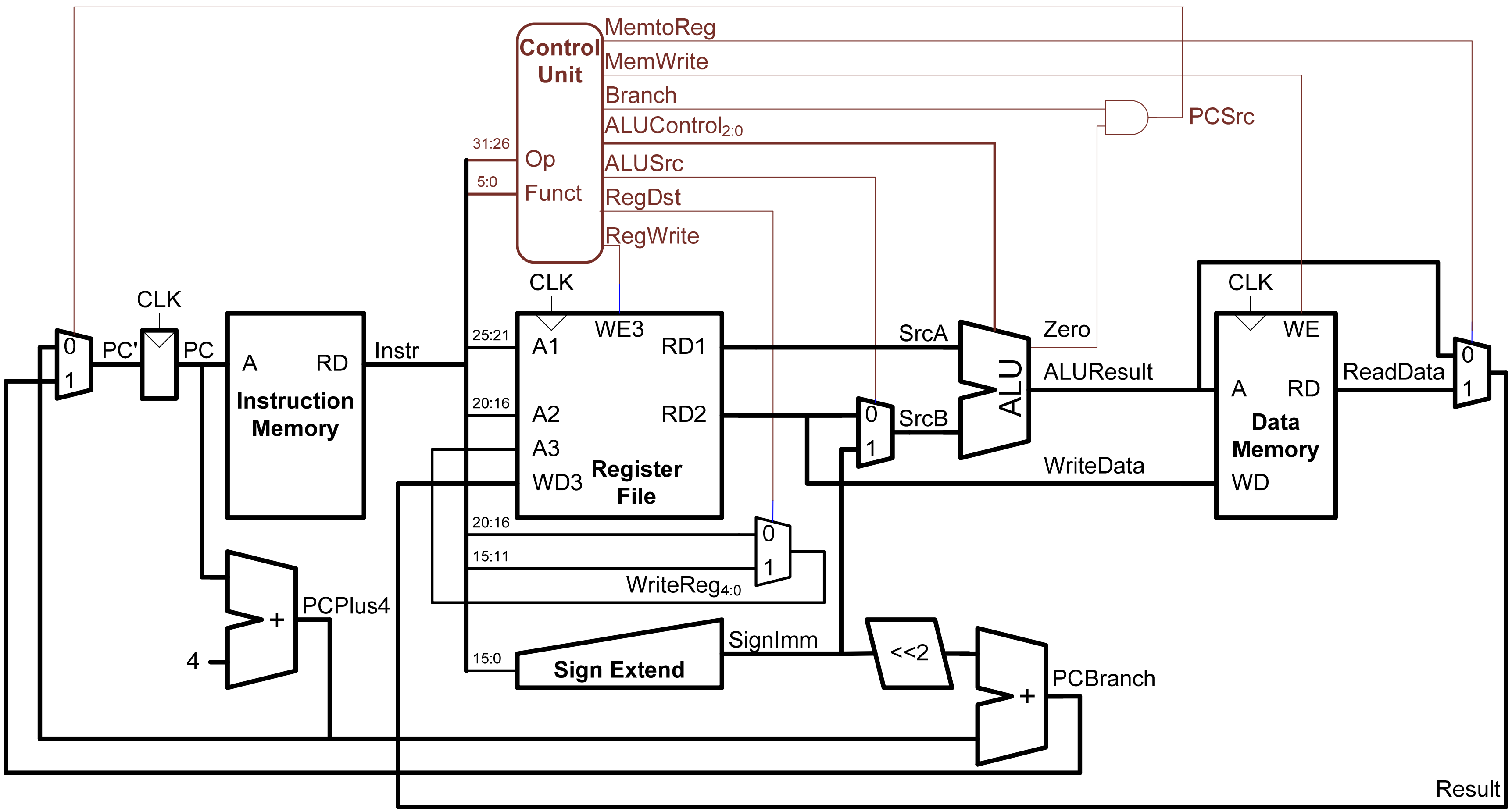
The scale of clusters and (multiple) datacenters
The presence of hardware failures and software bugs
The presence of multiple interacting services

The reality:

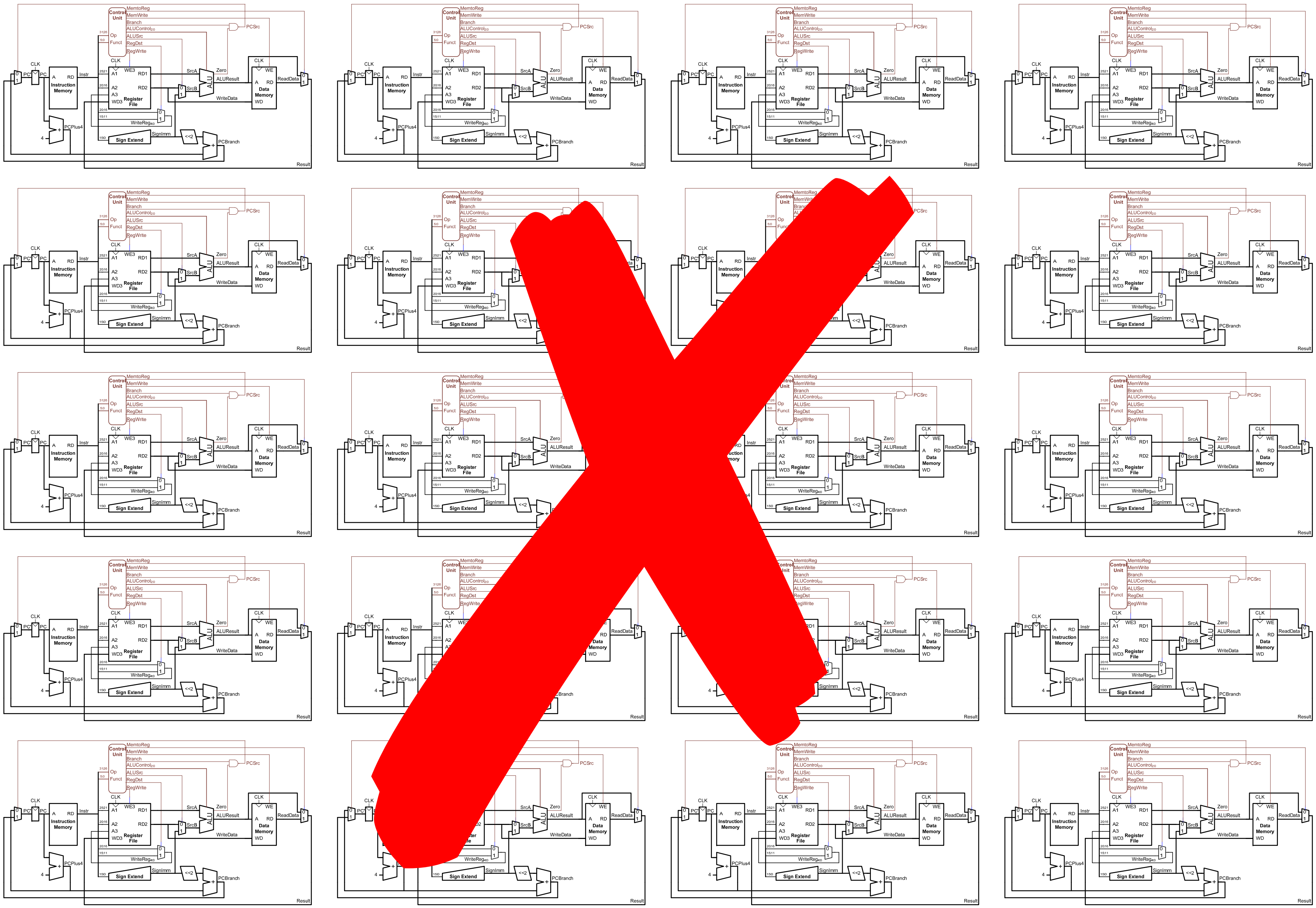
Lots of one-off solutions, custom code
Write you own dedicated library, then program with it
Burden on the programmer to explicitly manage everything

Bottom line: it’s hard!

Source: Ricardo Guimarães Herrmann



Source: CS 251



Source: CS 251

The datacenter *is* the computer!

Source: Google

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

Source: Google

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

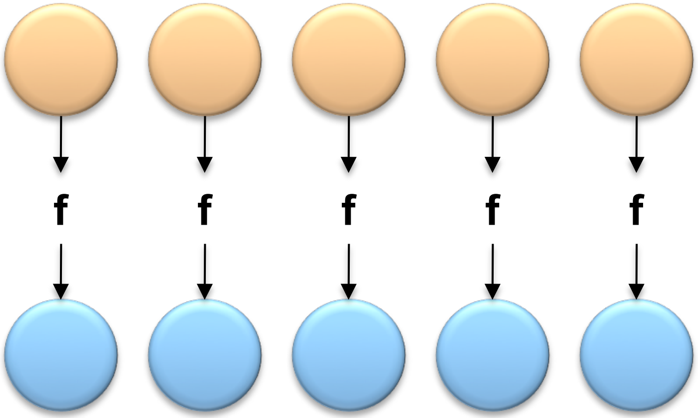
Why is this important?

Roots in Functional Programming

Simplest data-parallel abstraction

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

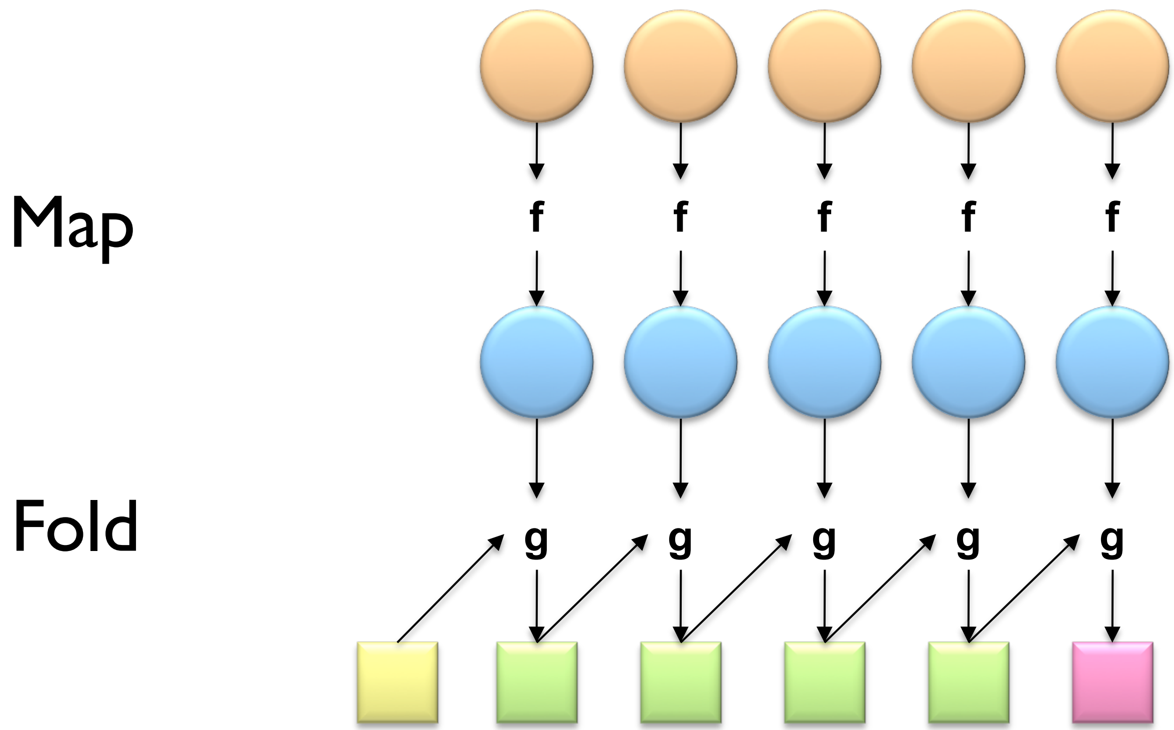
Map



We need something more for sharing partial results across records!

Roots in Functional Programming

Let's add in aggregation!



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:

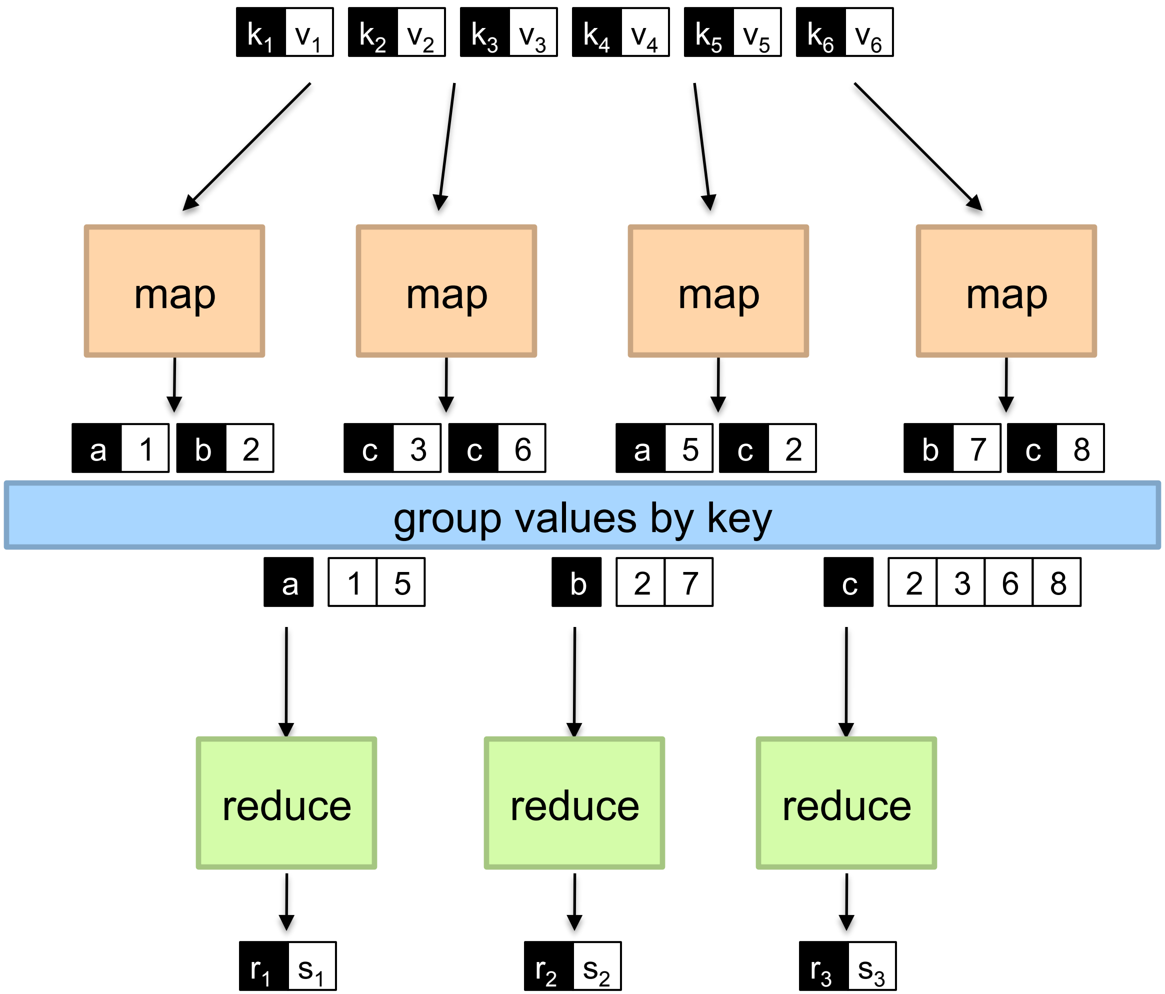
`map` $(k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]$

`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…


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
group values by key
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

Programmer specifies two functions:

```
map (k₁, v₁) → List[(k₂, v₂)]
reduce (k₂, List[v₂]) → List[(k₃, v₃)]
```

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:

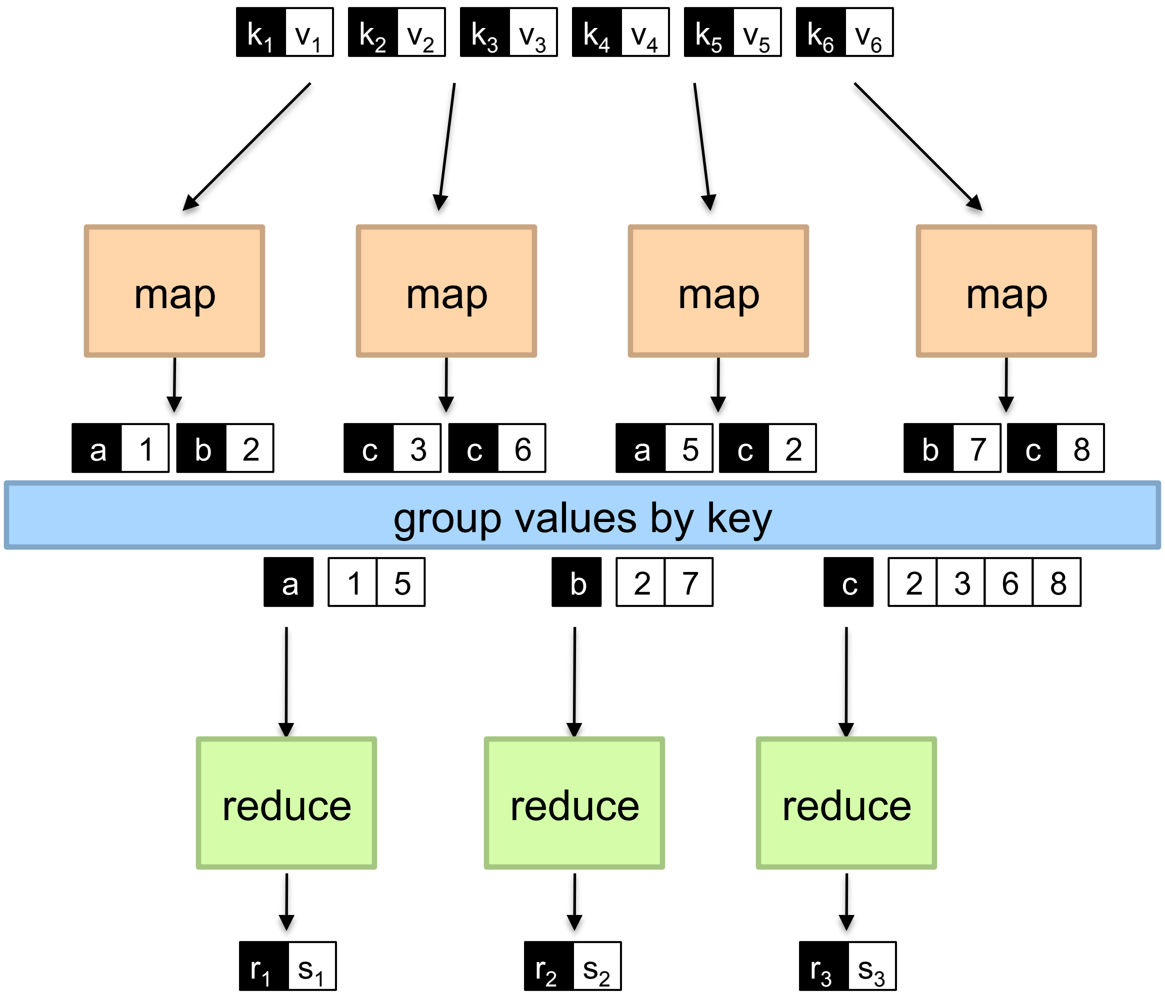
map $(k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]$

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

The execution framework handles everything else…

Not quite…



What's the most complex and slowest operation here?

MapReduce

Programmer specifies ~~two~~ four functions:

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

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

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

$\texttt{partition}\ (k', p) \rightarrow 0\ ...\ p-1$

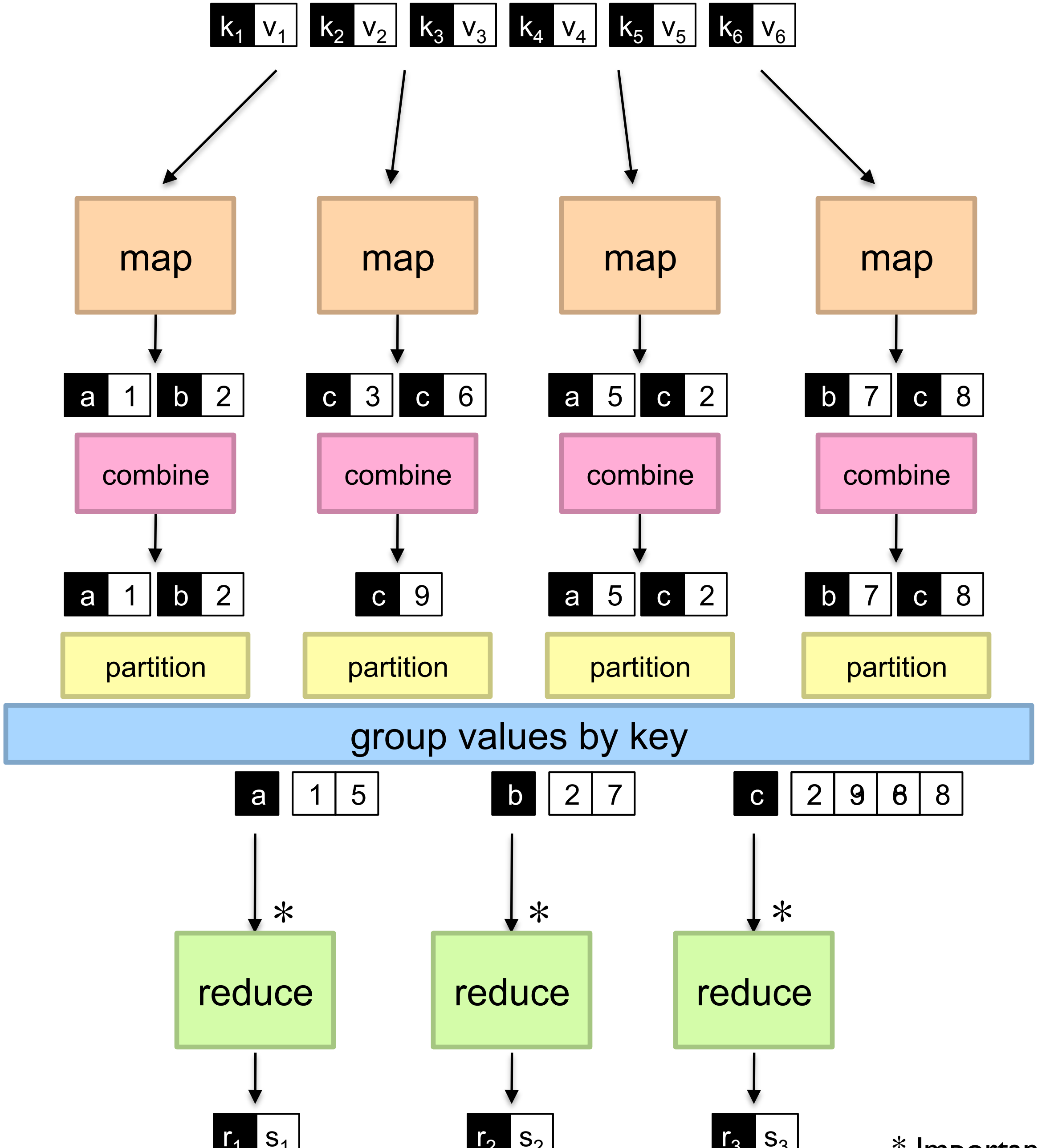
Often a simple hash of the key, e.g., `hash(k') mod n`

Divides up key space for parallel reduce operations

$\texttt{combine}\ (k_2, \texttt{List}[v_2]) \rightarrow \texttt{List}[(k_2, v_2)]$

Mini-reducers that run in memory after the map phase

Used as an optimization to reduce network traffic



* Important detail: reducers process keys in sorted order

“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

Course Administrivia

Source: http://www.flickr.com/photos/artmind_etcetera/6336693594/

Important Coordinates

Course website:

`http://lintool.github.io/bigdata-2018f/`

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-2018f-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:

8 *individual* assignments

Final exam

Additional *group* final project (CS 651)

Expectations

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

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

We'll be using <u>private</u> 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)

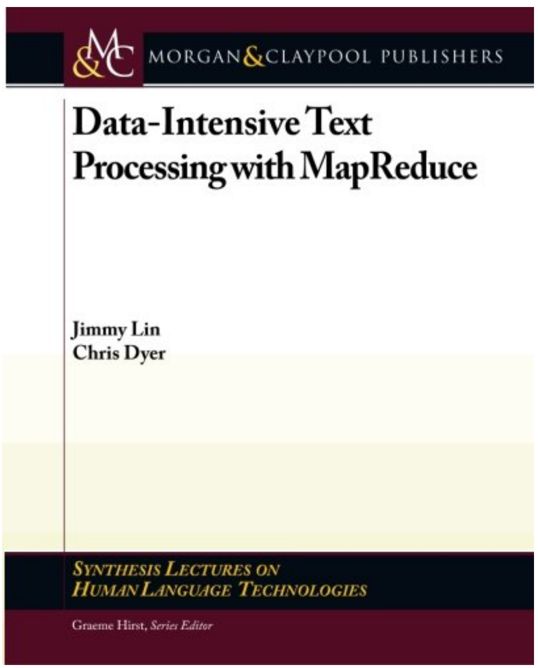
Late by up to 24 hours: 25% reduction in grade
Late 24-48 hours: 50% reduction in grade
Late by more the 48 hours: not accepted

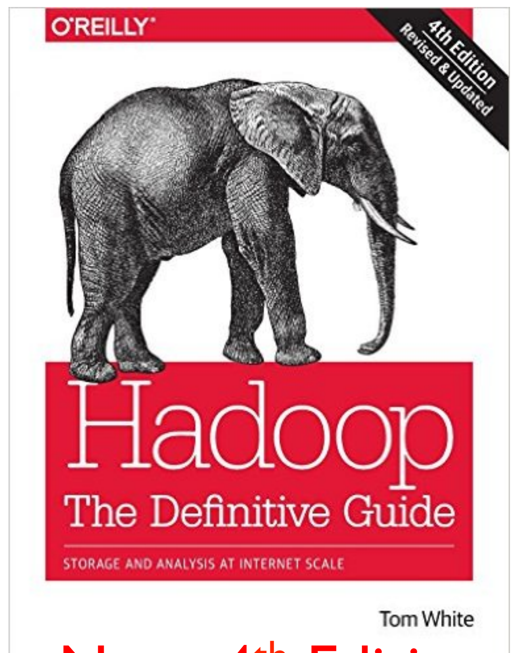
By assumption, we'll pull and mark at deadline:
If you want us to hold off, you <u>must</u> let us know!

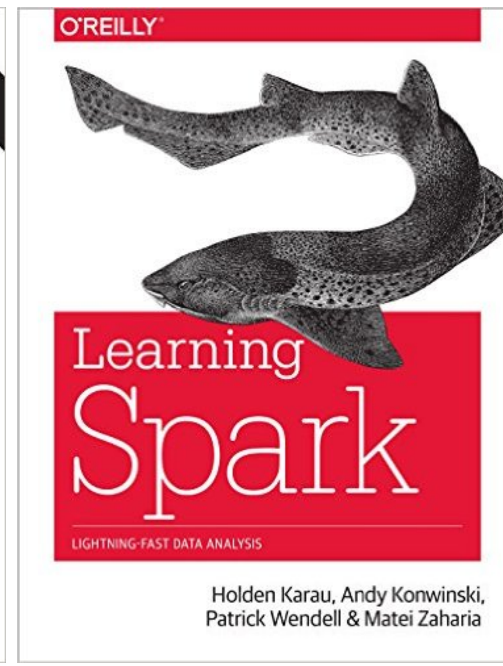
Important: Register for (free) GitHub educational account!
`https://education.github.com/discount_requests/new`

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:
`https://goo.gl/forms/7LA2QxBVXhESw8O43`

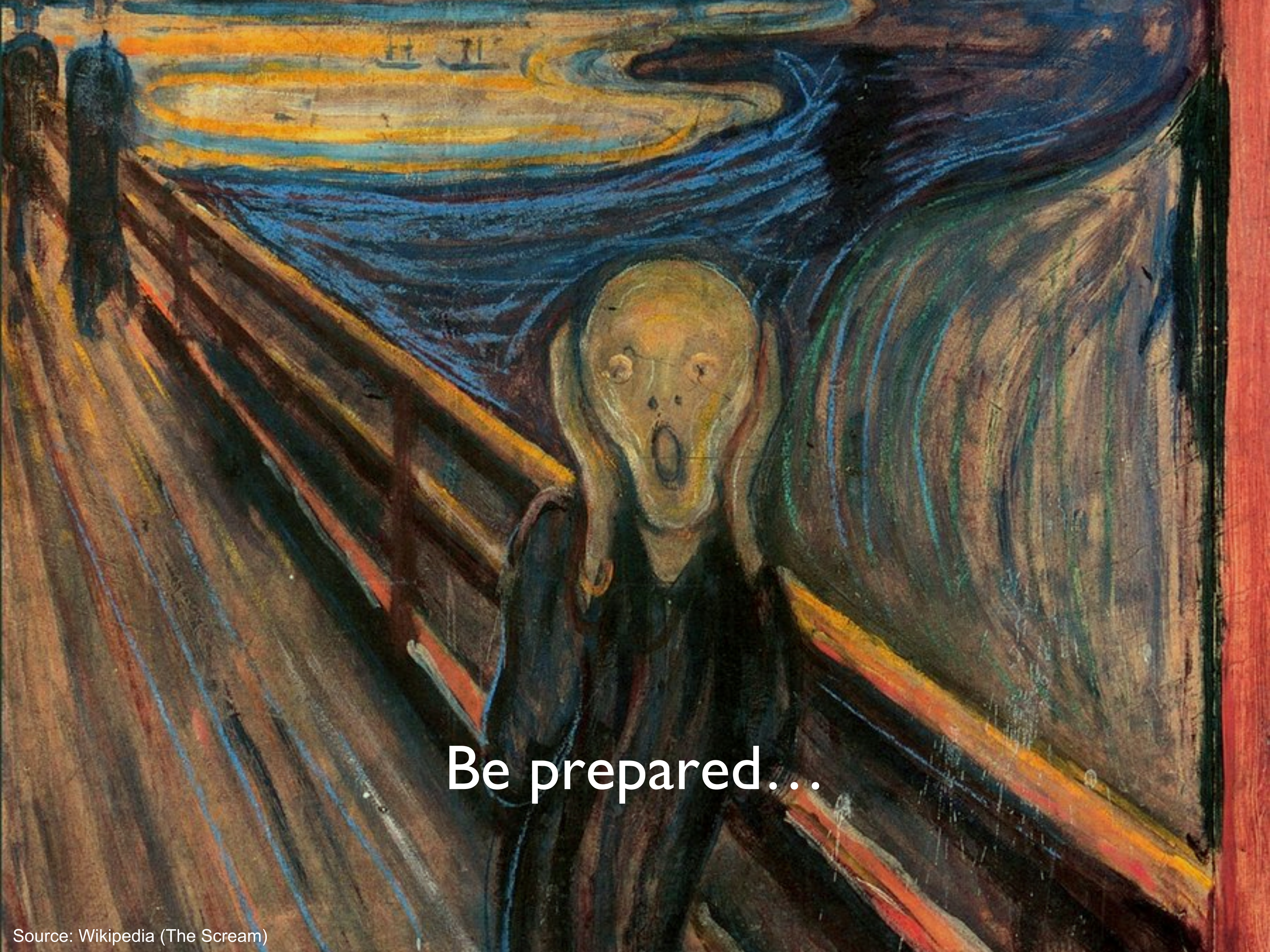
Registration begins at 8pm Thursday September 6th

Priority for unregistered students

CS students
Have all the pre-reqs
Final opportunity to take the course (e.g., 4B students)
| form submission time – 8pm 9/6/2018 |
Continue to attend class until final decision

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.



Source: Wikipedia (The Scream)

"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

“Hadoop Zen”

Source: Wikipedia (Japanese rock garden)

Questions?

To Do:

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

Source: Wikipedia (Japanese rock garden)