

# INTRODUCTION TO DATA SCIENCE

**JOHN P DICKERSON**

Lecture #24 – 11/14/2019

**CMSC320**

**Tuesdays & Thursdays**

**5:00pm – 6:15pm**



**COMPUTER SCIENCE**  
UNIVERSITY OF MARYLAND

# ANNOUNCEMENTS

**Project 3 has been extended to next Monday, November 25!**

**Project 4 will be released by November 21:**

- The shortest project by far!
- A bit of fun – visualizing and analysis geospatial data

**Please make sure you're making progress on the final tutorial!**

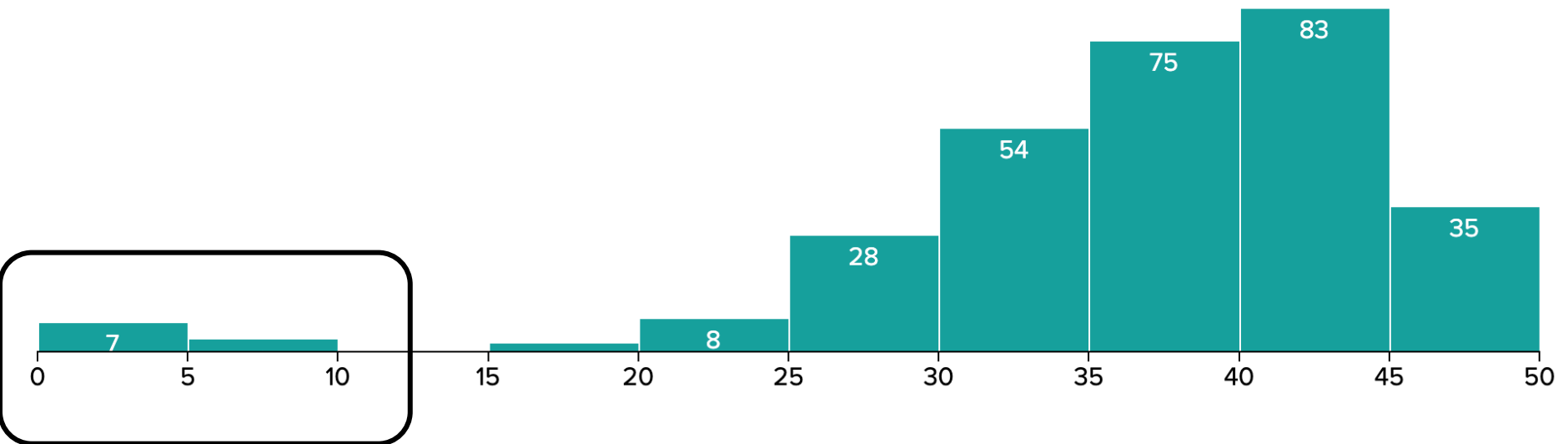
- Rubric is posted on the course webpage & on ELMS
- Rubric has linked to 4-5 previous projects I liked a lot!
- Please talk to me and/or the TAs whenever (many have)

# MIDTERM

Review Grades for **Midterm Old**

● REGRADE REQUESTS DISABLED

● GRADES PUBLISHED



MINIMUM

**0.0**

MEDIAN

**38.0**

MAXIMUM

**49.0**

MEAN

**36.48**

STD DEV

**8.87**

## Q 2B

- (b) Assume a population with an odd number of elements, and a sample with an odd number of elements. Which of the following is true?
- If the sample mean is equal to the sample median, then the *sample* distribution is symmetric.
  - If the sample mean is equal to the sample median, then the *true* distribution is symmetric.
  - If the *sample* distribution is symmetric, then the sample mean will be equal to the sample median.
  - If the *true* distribution is symmetric, then the sample mean will be equal to the sample median.

**Counterexample to (i) ??????**

**(-2, -1, 0, 0, 3)**

**(Mean = 0, Median = 0)**

## Q 2D & Q4

- (d) Regression estimates the relationship between two (or more) measurements. Which of the following is not an example of a *linear* relationship that could be measured well by a basic regression?
- The number of shoes purchased, given that I buy 1 pair of shoes for every 500 miles I run.
  - Total dollars spent buying tacos as a function of  $n$ , the number of tacos purchased, and  $p = \$1.50$ , the price of a taco.
  - Total expected calories consumed as a function of a  $n$ , the number of tacos, and  $c = 120$ , the average calories in a taco.
  - The relationship between happiness and number of miles run. If I run less than three miles, I am happy. If I run between three and six miles, I am sad. If I run more than six miles, I am happy again.
5. **[3 pts]** Recall term frequency-inverse document frequency (tf-idf). Your table above for Question 4 lists the term frequencies for each term in each document. Translate that table to its tf-idf representation. *You may use the idf function from class, or define your own, so long as it makes sense.*

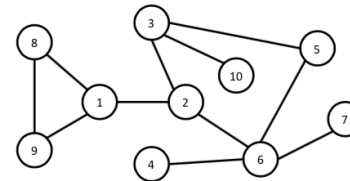
# Q6 - CENTRALITY

Weighted centrality metrics?

Negative weights?

Directed graphs?

6. [4 pts] This is a four-part question, and builds on Question 2c. Indeed, please feel free to reuse the graph from that question, reproduced below, when describing your answers to the questions below.



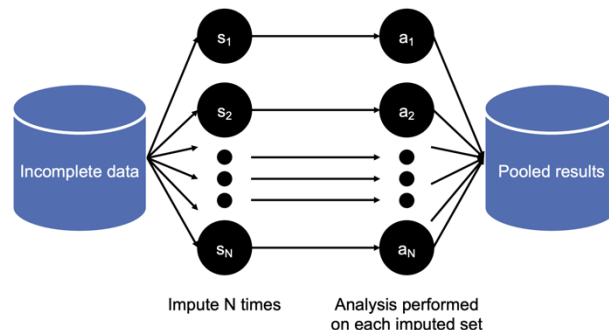
- We defined centrality measures for undirected, unweighted graphs. A *weighted* graph associates real-valued *weights* with edges. Formally extend *normalized degree centrality* to weighted, undirected graphs.
- Next, formally extend either closeness or betweenness centrality to the weighted, undirected setting.
- Do these centrality measures still make sense if weights on edges can be negative? What about zero-weight edges? Describe possible problems introduced by negative-/zero-weight edges, or describe why the measures still work as intended.
- How would centrality metrics change if the graph were *directed* instead of undirected?

# Q8 – TIDY DATA & Q10 – MULTIPLE IMPUTATION

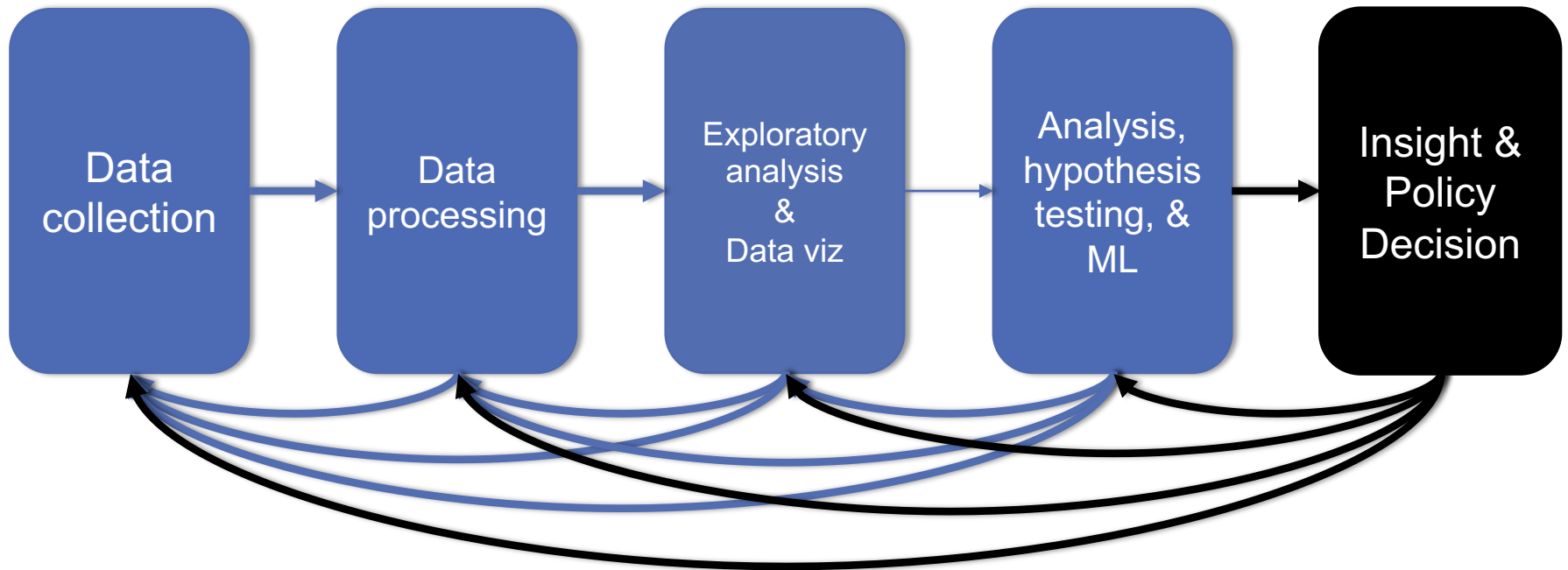
8. **[3 pts]** You are in charge of the first autonomous car race, to take place on the mean streets of College Park. The race has 500 total laps, and you record the lap times for each car, along with other information like the car's owner, make, model, et cetera. Not all cars finish the race—for example, a BMW i8 crashed into a wall after its first lap—so you also record how many laps were completed. Translate the data below into “tidy data.”

Make	Model	Team Name	Laps Completed	Lap 1	Lap 2	Lap 3	...	Lap 500
Tesla	S P100D	Ol' Musky	500	1:00	1:01	0:58	...	1:01
Nissan	Leaf	Zen Electronica	500	2:02	1:50	1:55	...	1:59
BMW	i8	Germautonomy	1	0:55	–	–	...	–
Tesla	S P100D	Autobucks	500	1:01	1:01	0:55	...	1:00
BMW	i8	Employee #1	500	0:56	1:01	0:58	...	0:56

10. **[3 pts]** Briefly explain the key steps in using multiple imputation to handle missing data.

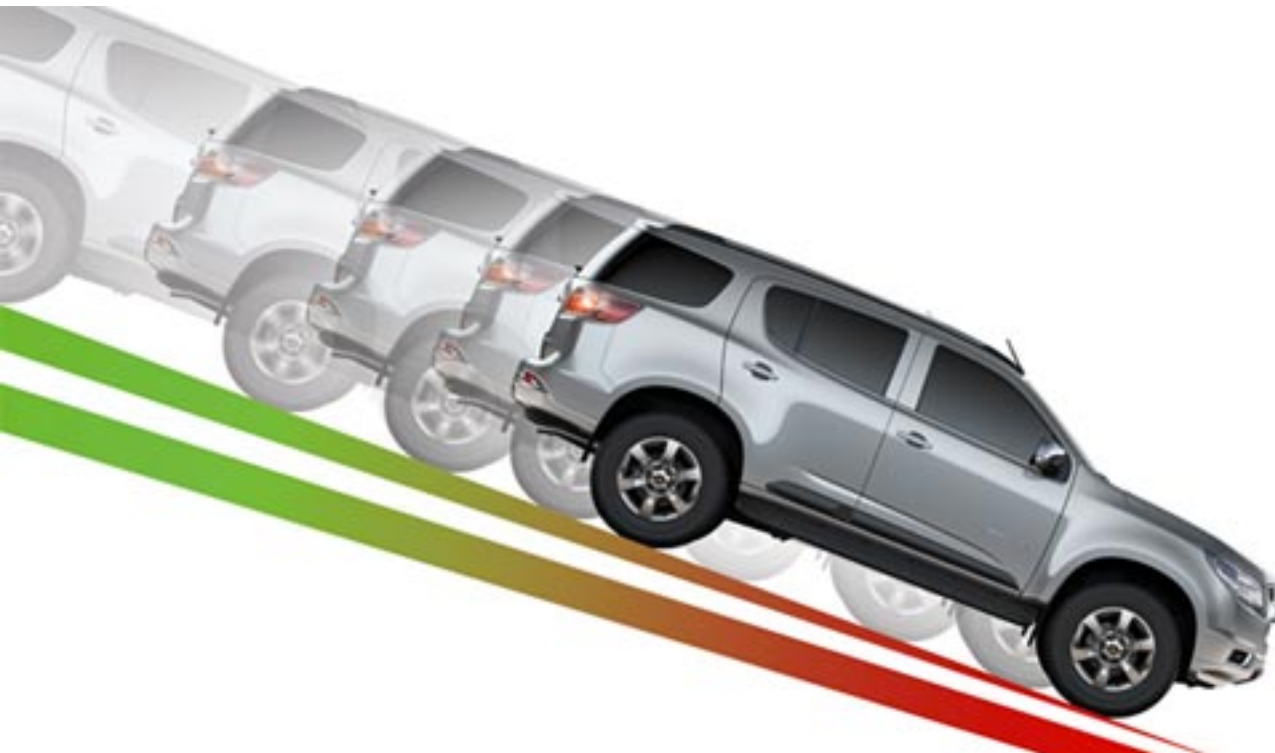


# TODAY'S LECTURE





# SCALING IT UP: STOCHASTIC GRADIENT DESCENT (SGD)



# RECAP: GRADIENT DESCENT

Algorithm for any\* hypothesis function  $h_\theta: \mathbb{R}^n \rightarrow \mathcal{Y}$ , loss function  $\ell: \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ , step size  $\alpha$ :

Initialize the parameter vector:

- $\theta \leftarrow 0$

Repeat until satisfied (e.g., exact or approximate convergence):

- Compute gradient:  $g \leftarrow \sum_{i=1}^m \nabla_{\theta} \ell(h_{\theta}(x^{(i)}), y^{(i)})$
- Update parameters:  $\theta \leftarrow \theta - \alpha \cdot g$

What if  $m$  is big?  
What if  $n$  is big?

\*must be reasonably well behaved

# STOCHASTIC GRADIENT DESCENT

Algorithm for any\* hypothesis function  $h_\theta: \mathbb{R}^n \rightarrow \mathcal{Y}$ , loss function  $\ell: \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ , step size  $\alpha$ :

Initialize the parameter vector:

- $\theta \leftarrow 0$

Repeat until satisfied (e.g., exact or approximate convergence):

- Randomly shuffle the input set  $x$
- For  $i$  in  $\{1, 2, \dots, m\}$ , shuffled:
  - Compute gradient:  $g \leftarrow \nabla_{\theta} \ell(h_{\theta}(x^{(i)}), y^{(i)})$
  - Update parameters:  $\theta \leftarrow \theta - \alpha \cdot g$

\*must be reasonably well behaved

# SGD CONTINUED

Can also make use of “mini-batch” stochastic gradient descent:

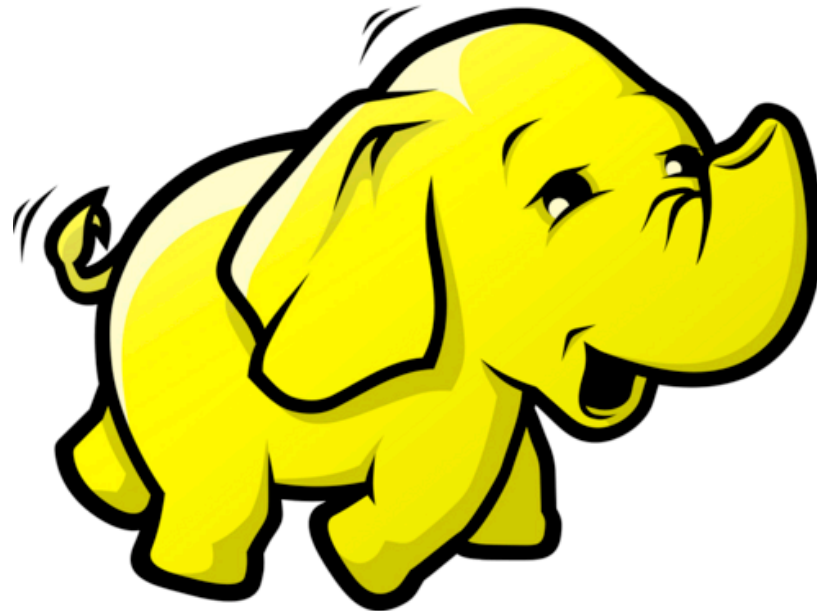
- High-level idea: at every outer iteration, shuffle the input data, then partition into  $k$  mini-batches of size  $m/k$
- Perform inner loop of SGD on these mini-batches

Batches **reduce variance** in update, make use of vectorization

Issues with SGD      ????????????????????

- Convergence: “almost surely” converges to the global optimum, assuming convexity and reasonable learning rate
- Batch size: hyperparameter? Not really – figure out how many examples fit in RAM/GPU memory, then choose the nearest power of 2 and go with that
- Sensitive to feature scaling (if batching over features)

# SCALING IT UP: BIG DATA & MAPREDUCE



*Thanks to: Jeff Dean, Sanjay Ghemawa, Zico Kolter*

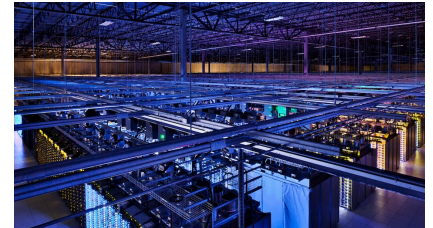
# “Big data”



My laptop  
8GB RAM  
500GB Disk

**Big data?**

No



Google Data Center  
??? RAM/Disk  
( $\gg$  PBs)

**Big data?**

Yes



# Some notable inflection points

1. Your data fits in RAM on a single machine
2. Your data fits on disk on a single machine
3. Your data fits in RAM/disk on a “small” cluster of machines (you don’t need to worry about machines dying)
4. Your data fits in RAM/disk on a “large” cluster of machine (you need to worry about machines dying)

It’s probably reasonable to refer to 3+ as “big data”, but many would only consider 4

# Do you have big data?

If your data fits on a single machine (even on disk), then it's almost always better to think about how you can design an efficient single-machine solution, unless you have extremely good reasons for doing otherwise

scalable system	cores	twitter	uk-2007-05
GraphChi [10]	2	3160s	6972s
Stratosphere [6]	16	2250s	-
X-Stream [17]	16	1488s	-
Spark [8]	128	857s	1759s
Giraph [8]	128	596s	1235s
GraphLab [8]	128	249s	833s
GraphX [8]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

**Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.**

scalable system	cores	twitter	uk-2007-05
Stratosphere [6]	16	950s	-
X-Stream [17]	16	1159s	-
Spark [8]	128	1784s	$\geq 8000s$
Giraph [8]	128	200s	$\geq 8000s$
GraphLab [8]	128	242s	714s
GraphX [8]	128	251s	800s
Single thread (SSD)	1	153s	417s

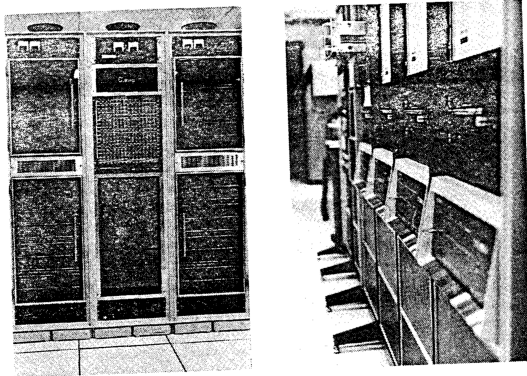
**Table 3: Reported elapsed times for label propagation, compared with measured times for single-threaded label propagation from SSD.**

Tables from [McSherry et al., 2015 “Scalability! But at what COST”]

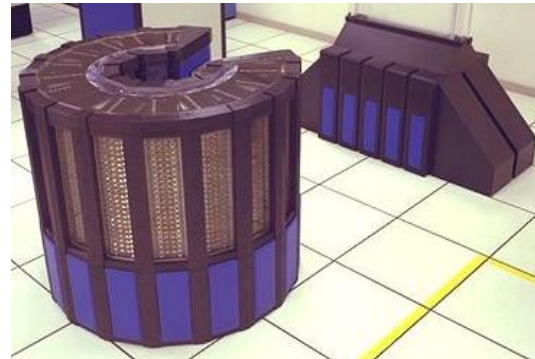


# Distributed computing

Distributed computing rose to prominence in the 70s/80s, often built around “supercomputing,” for scientific computing applications



1971 – CMU C.mmp  
(16 PDP-11 processors)



1984 – Cray-2  
(4 vector processors)

# Message passing interface



In mid-90s, researchers built a common interface for distributed computing called the message passing interface (MPI)

MPI provided a set of tools to run multiple processes (on a single machine or across many machines), that could communicate, send data between each other (all of “scattering”, “gathering”, “broadcasting”), and synchronize execution

Still common in scientific computing applications and HPC (high performance computing)

# Downsides to MPI

MPI is extremely powerful but has some notable limitations

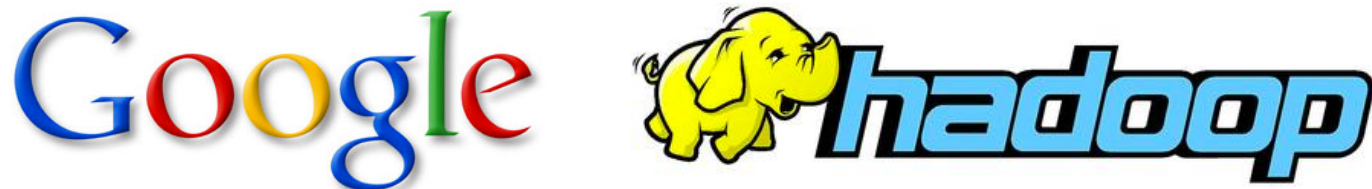
1. MPI is complicated: programs need to explicitly manage data, synchronize threads, etc
2. MPI is brittle: if machines die suddenly, can be difficult to recover (unless explicitly handled by the program, making them more complicated)

# A new paradigm for data processing

When Google was building their first data centers, they used clusters of off-the-shelf commodity hardware; machines had different speeds and failures were common given cluster sizes

Data itself was distributed (redundantly) over many machines, as much as possible wanted to do the computation on the machine where the data is stored

Led to the development of the MapReduce framework at Google [Ghemawat, 2004], later made extremely popular through the Apache Hadoop open source implementation



# AN EXAMPLE PROGRAM

Present the concepts of MapReduce using the “typical example” of MR, Word Count

- Input: a volume of raw text, of unspecified size (could be KB, MB, TB, **it doesn't matter!**)
- Output: a list of words, and their occurrence count.

**(Assume that words are split correctly; ignore capitalization and punctuation.)**

**Example:**

- **The doctor went to the store. =>**
  - The, 2
  - Doctor, 1
  - Went, 1
  - To, 1
  - Store, 1

# MAP? REDUCE?

**Mappers** read in data from the filesystem, and output (typically) modified data

**Reducers** collect all of the mappers output on the keys, and output (typically) reduced data

The outputted data is written to disk

All data is in terms of key-value pairs (“The” → 2)

# MAPREDUCE VS HADOOP

The paper is written by two researchers at Google, and describes their programming paradigm

Unless you work at Google, or use Google App Engine, you won't use it! (And even then, you might not.)

Open Source implementation is Hadoop MapReduce

- Not developed by Google
- Started by Yahoo!; now part of Apache

Google's implementation (at least the one described) is written in C++

Hadoop is written in Java

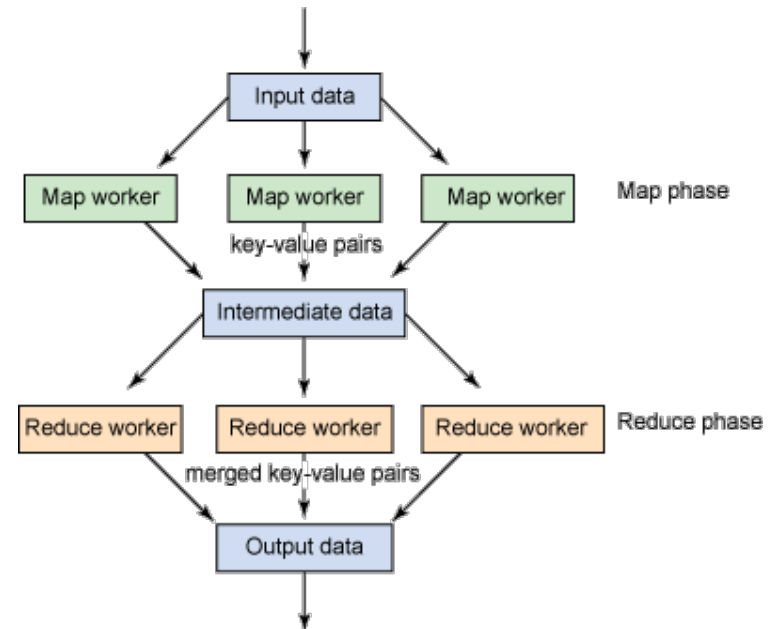
# MAJOR COMPONENTS

## User Components:

- Mapper
- Reducer
- Combiner (Optional)
- Partitioner (Optional) (Shuffle)
- Writable(s) (Optional)

## System Components:

- Master
- Input Splitter\*
- Output Committer\*
- \* You can use your own if you really want!





# KEY NOTES

**Mappers and Reducers are typically single threaded and deterministic**

- Determinism allows for **restarting of failed jobs**, or speculative execution

**Need to handle more data? Just add more Mappers/Reducers!**

- No need to handle multithreaded code
- Since they're all independent of each other, you can run (almost) arbitrary number of nodes

**Mappers/Reducers run on **arbitrary** machines. A machine typically multiple map and reduce slots available to it, typically one per processor core**

**Mappers/Reducers run entirely independent of each other**

- In Hadoop, they run in separate JVMs

# BASIC CONCEPTS

**All data is represented in key-value pairs of an arbitrary type**

**Data is read in from a file or list of files, from distributed FS**

**Data is chunked based on an input split**

- A typical chunk is 64MB (more or less can be configured depending on your use case)

**Mappers read in a chunk of data**

**Mappers emit (write out) a set of data, typically derived from its input**

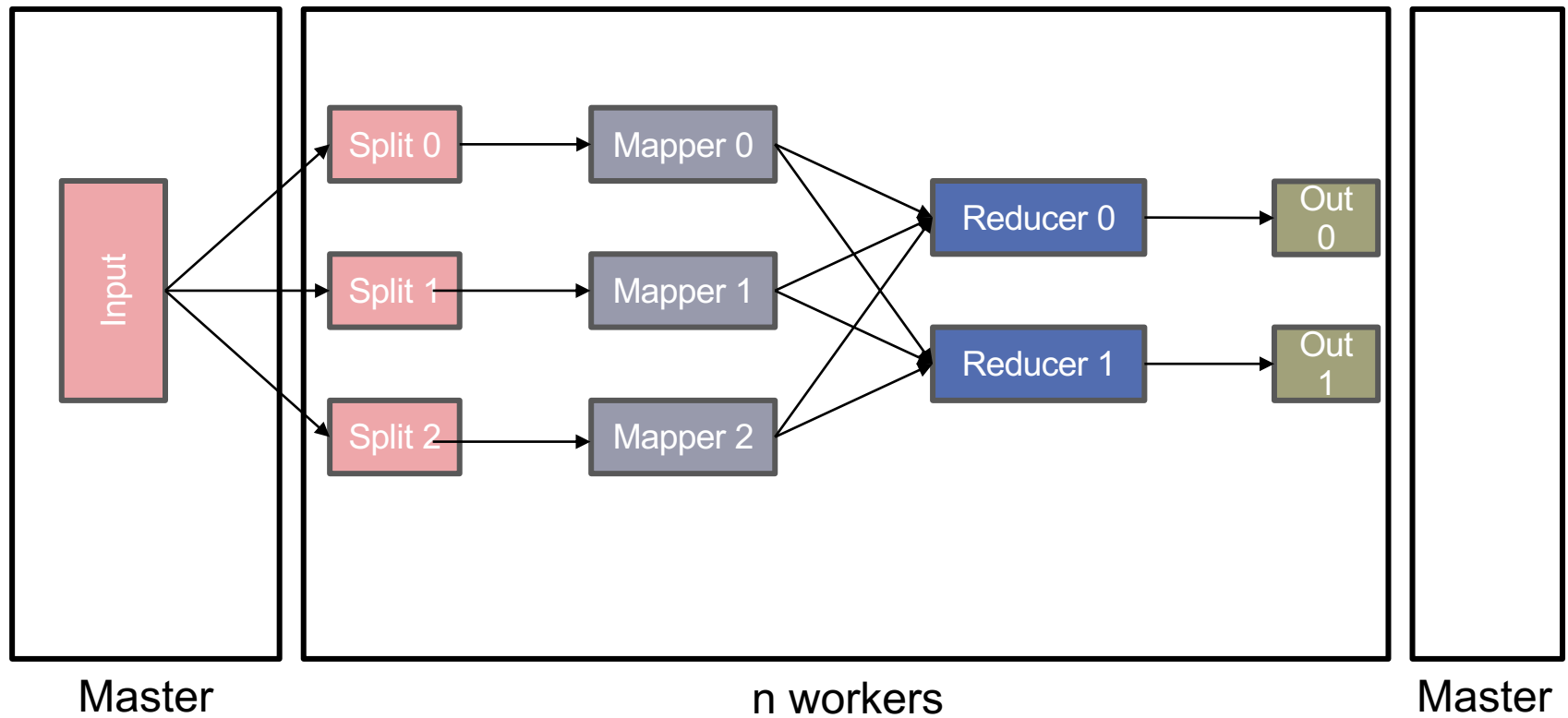
**Intermediate data (the output of the mappers) is split to a number of reducers**

**Reducers receive each key of data, along with ALL of the values associated with it (this means each key must always be sent to the same reducer)**

- Essentially, <key, set<value>>

**Reducers emit a set of data, typically reduced from its input which is written to disk**

# DATA FLOW



# INPUT SPLITTER

**Is responsible for splitting your input into multiple chunks**

**These chunks are then used as input for your mappers**

**Splits on logical boundaries. The default is 64MB per chunk**

- Depending on what you're doing, 64MB might be a LOT of data! You can change it

**Typically, you can just use one of the built in splitters, unless you are reading in a specially formatted file**

# MAPPER

Reads in input pair  $\langle K, V \rangle$  (a section as split by the input splitter)

Outputs a pair  $\langle K', V' \rangle$

Ex. For our Word Count example, with the following input: “The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”

The output would be:

- $\langle \text{The}, 1 \rangle \langle \text{teacher}, 1 \rangle \langle \text{went}, 1 \rangle \langle \text{to}, 1 \rangle \langle \text{the}, 1 \rangle \langle \text{store}, 1 \rangle$   
 $\langle \text{the}, 1 \rangle \langle \text{store}, 1 \rangle \langle \text{was}, 1 \rangle \langle \text{closed}, 1 \rangle \langle \text{the}, 1 \rangle \langle \text{store}, 1 \rangle$   
 $\langle \text{opens}, 1 \rangle \langle \text{in}, 1 \rangle \langle \text{the}, 1 \rangle \langle \text{morning}, 1 \rangle \langle \text{the}, 1 \rangle \langle \text{store}, 1 \rangle$   
 $\langle \text{opens}, 1 \rangle \langle \text{at}, 1 \rangle \langle \text{9am}, 1 \rangle$

# REDUCER

**Accepts the Mapper output, and collects values on the key**

- All inputs with the same key must go to the same reducer!

**Input is typically sorted, output is output exactly as is**

**For our example, the reducer input would be:**

- <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1>  
<the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1>  
<opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1>  
<opens, 1> <at, 1> <9am, 1>

**The output would be:**

- <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1>  
<closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

# COMBINER

**Essentially an intermediate reducer**

- Is optional

**Reduces output from each mapper, reducing bandwidth and sorting**

**Cannot change the type of its input**

- Input types must be the same as output types

# **OUTPUT COMMITTER**

**Is responsible for taking the reduce output, and committing it to a file**

**Typically, this committer needs a corresponding input splitter (so that another job can read the input)**

**Again, usually built in splitters are good enough, unless you need to output a special kind of file**



# PARTITIONER (SHUFFLER)

**Decides which pairs are sent to which reducer**

**Default is simply:**

- `Key.hashCode() % numOfReducers`

**User can override to:**

- Provide (more) uniform distribution of load between reducers
- Some values might need to be sent to the same reducer
  - Ex. To compute the relative frequency of a pair of words  $\langle W1, W2 \rangle$  you would need to make sure all of word  $W1$  are sent to the same reducer
- Binning of results

# MASTER

**Responsible for scheduling & managing jobs**

**Scheduled computation should be close to the data if possible**

- Bandwidth is expensive! (and slow)
- This relies on a Distributed File System (e.g. GFS)!

**If a task fails to report progress (such as reading input, writing output, etc), crashes, the machine goes down, etc, it is assumed to be stuck, and is killed, and the step is re-launched (with the same input)**

**The Master is handled by the framework, no user code is necessary**

# MAPREDUCE IN PYTHON

```
def mapreduce_execute(data, mapper, reducer):
    values = map(mapper, data)

    groups = {}
    for items in values:
        for k,v in items:
            if k not in groups:
                groups[k] = [v]
            else:
                groups[k].append(v)

    output = [reducer(k,v) for k,v in groups.items()]
    return output
```

# MAPREDUCE IN PYTHON

Don't do the last slide ...

Python's `mrjob` library:

- write mappers and reducers in Python
- Deploy on Hadoop systems, Amazon Elastic MR, Google Cloud

```
from mrjob.job import MRJob

class WordOccurrenceCount(MRJob):
    def mapper(self, _, line):
        for word in line.split(" "):
            yield word, 1

    def reducer(self, key, values):
        yield key, sum(values)
```

# MAPREDUCE?

## Good:

- All you need to do is write a mapper and a reducer
- Can get away with not exposing any of the internals (data splitting, locality issues, redundancy, etc) if you're using a ready-made engine

## Bad:

- Lots of reading/writing from disk (in part because this helps with redundancy)
- Sometimes communication between processes is necessary
- Talk about later: parameter servers, GraphLab aka Dato, etc

*NEXT UP:*

**REVIEW OF HYPOTHESIS TESTING  
(AND THEN A BUNCH OF STUFF LIKE  
PRIVACY, ETHICS, DEBUGGING DATA  
SCIENCE, ETC!)**