Programming with Big Data

Tim Savage

June 22, 2017

New York City

De-Mystifying the Un-Mystical

Like a hammer and nails,

these are just tools

My Background

- I started in the pre-digital age using small, administrative data.
- NLSY79: 3,000 young men tracked annually for about 20 years.
- Complicated algorithm to examine whether early adverse labor market outcomes affected their "lives".
- The data was cheap, but the computation was very expensive.

Programming Languages

- Fortran
- SAS/Stata
- R
- Python
- Julia (in order to use a single, open-source forecasting tool from the Federal Reserve)

Core Takeaways

1. If it can be **digitized**, it can be analyzed. And now **everything** is digitized.
2. The incremental cost of store has been **effectively zero** for a decade, so everything has been stored. Does this have any **business intelligence value**?
3. The open source computing community **defeated** Gresham's "law". Without open source, there would be **no big data** analytics.
4. There is a reason **Google** won the search engines wars: MapReduce and Hadoop. **Apache Spark** has brought clustered computing to the masses.

What is Big Data?

- “Big Data” is data whose scale, distribution, diversity, and/or timeliness requires the use of new computing architectures and analytical tools that did not exist **10 years ago**.
- Organizations are beginning to derive benefit from analyzing ever larger and more complex data sets in **real time**.

Key Characteristics

- **Volume**, which has increased nearly 44 times since 2009.
  - Awash in the digital exhaust of human activity.
- **Variety** of different data structures to mine and analyze.
  - Structured, semi-structured, and unstructured.
  - Processing complexity because of changing data structures.
- **Veracity** and integrity.

Business Drivers (Good and Bad)

- **Identifying** potential risks from customer churn or fraud.
- **Predicting** new opportunities or ventures.
- **Complying** with regulation (and the battle with open source).
- **Optimizing** operations for profitability and efficiency.
  - First degree price discrimination and the elimination of consumer surplus.

Big Data = Big Challenges

- Valuable data is **hard** to reach and properly leverage.
- It sits in fragmented “**puddles**” without a proper data “**lake**”.
- Predictive analytics are the last step in the value chain.
  - Such data is often proprietary, which complicates its use.
- A large share remains data **preprocessing**.

Big Data Analytics Lifecycle

1. **Preparing**: Is there enough good data to be potentially useful.
2. **Planning**: There are many ML models to choose from. Which one(s) to use?
3. **Building**: Is the model robust? How well does it predict out of sample?
4. **Operationalizing**: Scale the model(s) for deployment using the ideas discussed in this presentation.

Common Big Data Challenges

Volume and Velocity

(Veracity)

Volume

- **Too much** data to process conventionally.
- Reduce processing time by using distributed and parallel computing.
  - Performing OCR on thousands of articles simultaneously.
- **Too big** to fit into memory when no clustered resources available.
  - Given 30GB of NYC taxi data, how would one calculate the average or median fare?

Velocity

- Massive data arrives in **real time**, as in high frequency trading or detailed transactional data.
- Cannot be stored or processed in real time without expensive computational operations.
  - What's expensive?

How Do We Deal With This?

Henry Ford’s Assembly Line

- 100 year-old invention that revolutionized manufacturing.
- Took an older concept and improved it using moving platforms on a conveyor system.
- Exactly the conceptual framework you should consider for the remainder of this talk.

Common Big Data Challenges

Volume and Velocity

The Volume Problem: Parallelization

- Utilize **parallel** computing **architectures**, such as multiple cores, multiple processors or clusters of machines.
  - **Multiple cores** for video processing.
  - **Multiple processors** for web servers and video games.
  - **Multiple clusters** for simulation and Bayesian inference. (What I do.)

Task Parallelism

- Distributing tasks to run **simultaneously**, achieving efficiency if the number of tasks is large.
  - Data cleaning.
  - Running linear regression or algorithms that can be distributed, such as decision trees/random forests.
- The **assembly line**:
  - A bunch of boxes at FedEx need both shipping labels and barcode scanning. One worker can stamp the shipping labels, while the other scans the boxes.

Pipeline Parallelism

- Explicitly allocating resources for **each phase** of the data processing pipeline, achieving efficiency when the number of phases is large.
- Processes must **communicate** throughout the pipeline, so it is a combination of both the **task** and the **streaming**.
- The **assembly line**:
  - Pass the boxes by the stamper and then by the scanner.

Data Parallelism

- Distributing data to **different** processors that run simultaneously, achieving efficiency based on the number of processor nodes.
- The **assembly line**:
  - Leave all boxes on the ground, where each worker can both stamp and scan a single box.
  - Get more workers.

The Velocity Problem: Scaling

- Scale **up** by making the process scalable on a single computer.
  - Reduce the amount of data processed or the resources needed to perform the processing.
  - Increase the computing resources using parallel processing and faster memory and/or storage.
  - Improve the efficiency and/or performance of the process by, for example, **better coding**.
- Scale **out** by adding more computing resources through the networking of multiple computers.

Too Much Data

- Scale **up** with “streaming”.
- An active area of research among “computing scientists.”
- Started in the 1970’s, but now **very popular** because of its successful application to massive data processing.
- Big data world adopts the stream processing model.
  - Process data as soon as it **arrives**.
  - **Continuous** and incremental processing.

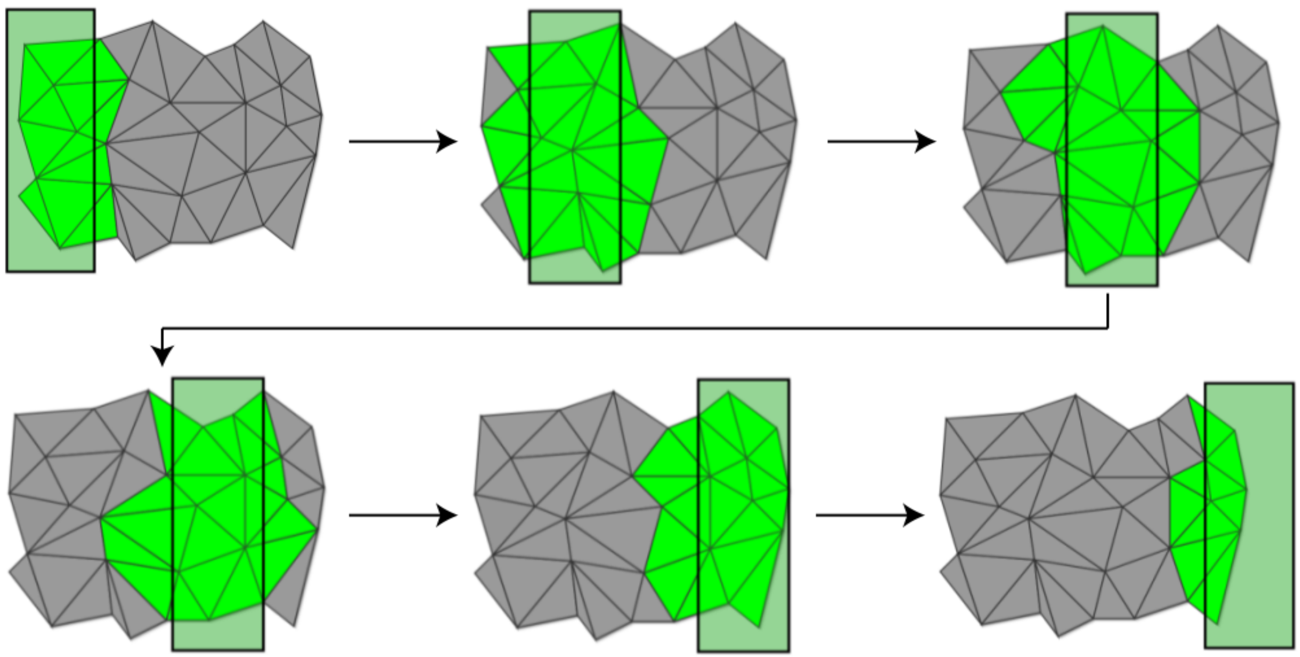
Streaming Computation

- Given a data series of n elements, [a_1, a_2, …, a_n], that can only be examined in a limited number of passes, **typically one**.
- Compute a function of the stream, such as an average, a median, or a histogram.
- Primary constraints:
  - Limited working memory of size m (m << n).
  - Elements are accessed sequentially.

Handling Data Streams

- There are many flavors of the streaming model:
  - **Time series**: price data in high frequency trading.
  - **Cash register**: storing incremental counts and totals.
  - **Turnstile**: arrival-departure summarization.
  - **Sliding window**: keeping a continuous but fixed subset of the input.
- There are many classes of techniques to process data elements:
  - **Sampling**: data input reduction.
  - **Sketching**: data aggregation.
  - **Counting**: data compression.

Picturing the Sliding Window



Example: Detecting Omission

- There are 11 football players, numbered 1 to 11, walking from the locker room to the field.
- Only 10 arrive: 8, 2, 6, 1, 10, 3, 5, 11, 9, and 7.
- How to determine the missing number in the "stream" of players?
- Given the constraints:
  - We can only look at one number at a time.
  - We can only store one number in our head.

What Is Omitted?

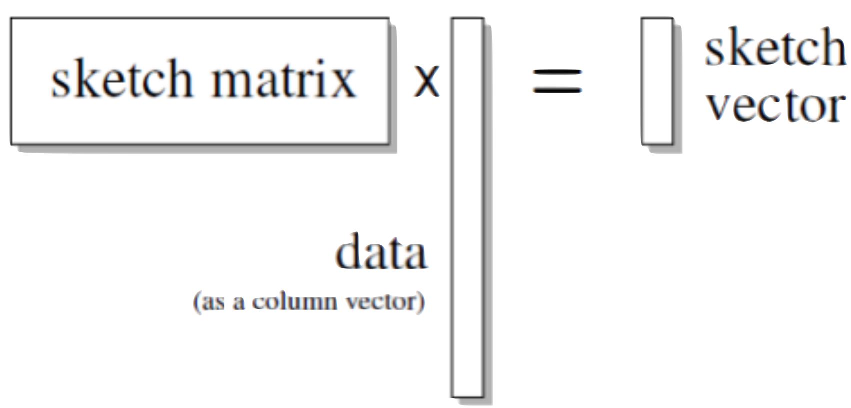
- It is 4. How?
- The sum of all numbers is fixed.
  - (1 + 11) * 11 / 2 = 66.
- Record only the sum of the numbers as we scan through the stream of players.
  - 8, 10, 16, 17, 27, 30, 35, 46, 55, and 62.
- Our missing number is 66 - 62 = 4.
- Simple example that highlights many important ideas about processing constraints and exploiting “lazy” knowledge.

Sampling

- Motivation: small random sample of the data can be a **good representation**. (Hal Varian)
- Action: sample the data based on a probability model, which is a challenge for data streams of unknown size.
- Useful for showing patterns but **not at detection** deviations from central tendencies.

Sketching

- Motivation: only certain pieces of the data are needed for computation.
- Action: project data into a "sketch" space, progressively building the function of stream.
- Useful for aggregation, but the sketch vector could be very large.



Sketching Examples

- Computing **a streaming average** by sketching the (sum, count) pair.
  - For each element, add the incremental value to the sum and increase the count.
  - [2] $\longrightarrow$ (sum + 2, count + 1)
- Computing **a streaming histogram** by sketching the count per category.
  - For each element, add to the count of its category.
  - Lions, tigers, and bears.

For Streaming, Use Approximation

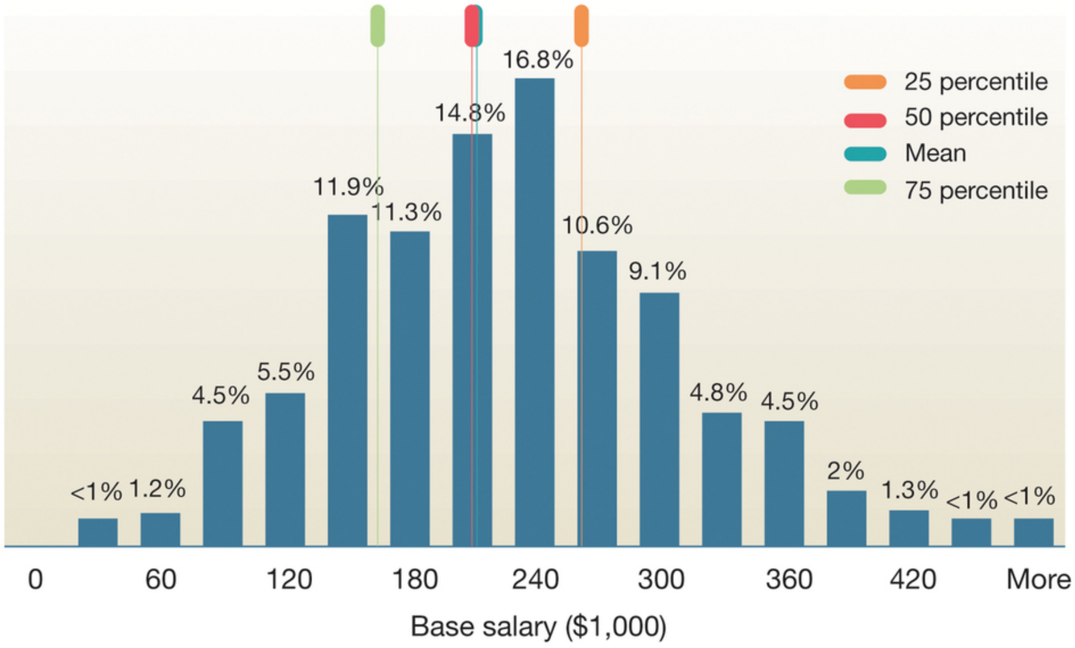
- We need to **speed up** the computation.
  - Brute force approach often takes a too much processing time.
  - As a result, we **sacrifice accuracy** or granularity for **efficiency**.
- Common approaches:
  - Memoization for repeated queries or computation.
  - Indexing for quickly retrieving a small subset of data.
  - Data cubes for computing aggregation.

Example: Calculating a Median

- A median is the **middle mark** and can differ wildly from an average, but tends to be stable. Example: median income of your customer.
  - Well-studied problem in streaming algorithms.
  - Lots of methods to approximate the median value within **tolerable error bounds**.
  - But computationally expensive, needing multiple passes through the data.
- Most of these methods rely on assuming that data are **continuous** and of large range (such as income).

Example: Calculating a Median

- Build a histogram of values based on bucketed ranges.
- Select the 50th percentile bucket.
- Choose the first bucket that passes the that mark.
- In this case, $210,000.



Streaming and Big Data Queries

- Exact answers are **not required** for real-time decision making, and results can be pre-computed.
- This can be done in a **streaming fashion** (as the data arrives).
  - Yields more accurate answers than sampling.
  - Storage is not expensive.
- This approach is used at Google, Twitter, and Facebook.

MapReduce
Hadoop

Translation: Divide and Conquer

Why the MapReduce Paradigm?

- Big data is **too large** to handle using conventional means.
- Sooner or later, there are **energy limits** on **scaling up** a given machine.
- But we can add more machines by **scaling out**.
- MapReduce paradigm allows us to **scale out**.
  - Issue is: what's going to **herd** all these cats?
  - Google's paradigm won them the search-engine wars (and big bucks).

What is MapReduce?

- A **programming paradigm** for big data processing.
  - Data is split into distributed chunks.
  - Transformations are performed on the chunks, running in parallel.
- MapReduce is **scalable** by adding more machines to process distributed chunks.
- It is the foundation for "Hadoop", which is a specific implementation of MapReduce.

Map(and)Reduce

- A programming paradigm that processes data in two phases/operations: **map()** and then **reduce()**.
- In a nutshell, that is all there is:
  - User provides a data collection of **separable records**.
  - User applies a **map function** to each data record, such as a count.
  - User **reduces** the mapped output with another user-defined function, if needed.

Python Example

Simulation Example: A Circle Within a Square

Area of a circle = πr^2

Area of a square = $height * length$

A circle with radius one fits inside a square whose height and length are two. This implies the ratio of the areas is $\pi/4$. Therefore, we could simulate the value of π, which is an irrational number.

I ran this simulation on CUSP's cluster. Let me show you.

In [ ]:

```
# The ratio of the unit circle to the unit square is pi/4.
# MC simulation of the value of pi using two independent draws from uniform.
from __future__ import print_function

import os
os.environ['MPLCONFIGDIR'] = '/tmp'

from pyspark import SparkContext, SparkConf
from numpy import random, pi

sc = SparkContext(appName="pipy", environment={'MPLCONFIGDIR' : '/tmp'})

nSamples = 1000000000

def sample(n):
    x, y = random.uniform(), random.uniform()
    return 1 if x * x + y * y < 1 else 0

# This parallelizes an RDD of size, 0 to NUM_SAMPLES.
# Passes this RDD through the sample function using the map transformation.
# Which creates an RDD of 0's and 1's.
# Which is aggregated using the reduce action.

count = sc.parallelize(xrange(0, nSamples)).map(sample).reduce(lambda a, b: a + b)

print("Size is %i" % (nSamples))
print("Pi is roughly %f" % (4.0 * count / nSamples))
print("Pi is exactly %f" % (pi))
```

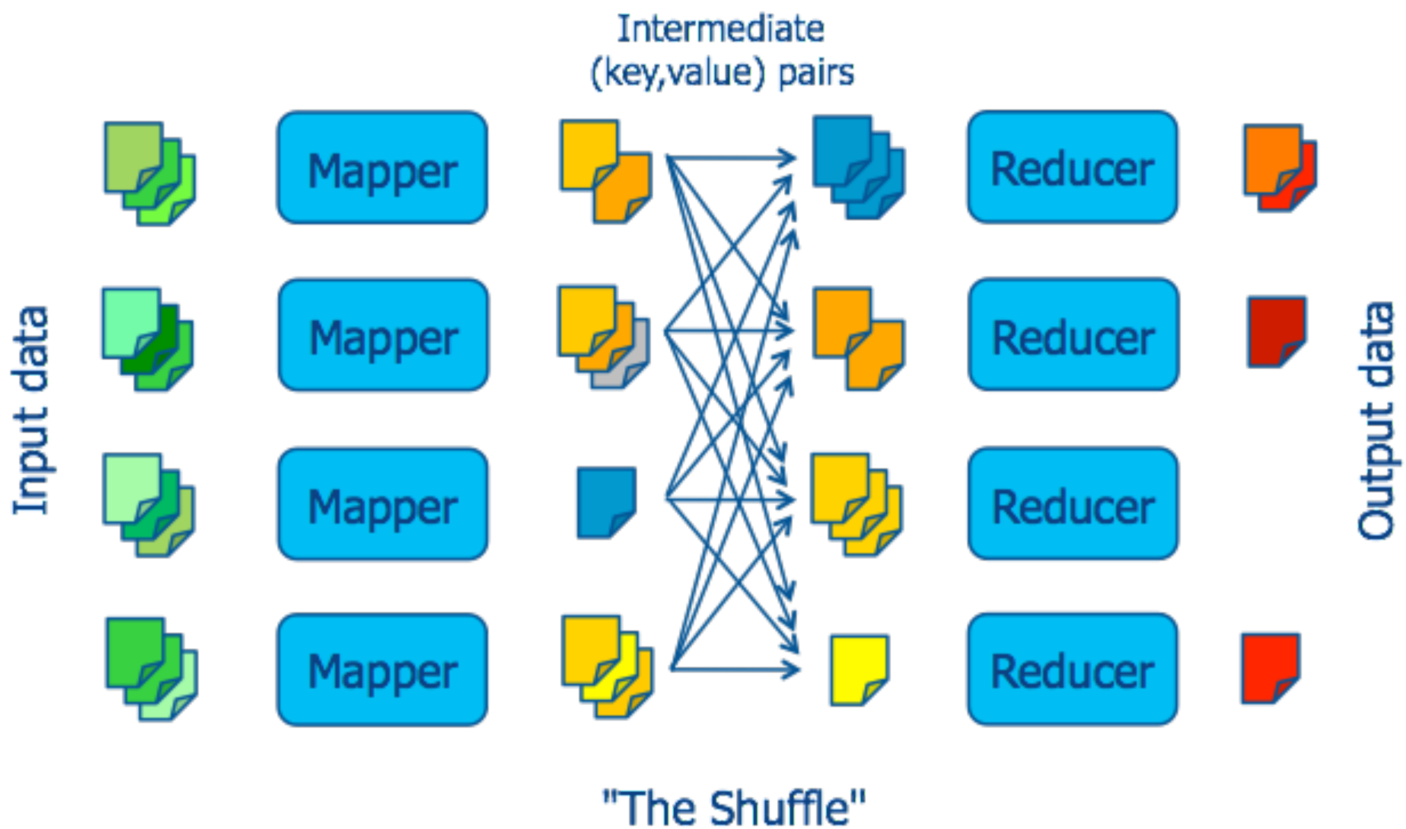
Input

- Data must be **separable** into records.
  - Lines of text or rows of a spreadsheet.
  - CSV works easily, but not true for other data types, such as JSON and XML.
- Key/value **pairs**: (key, value).
  - Key = line number or record index.
  - Value = text string or row data.

Phases

- Map phase: transform each input record with a user-defined function.
- Shuffle (and sort) phase: complicated but it amounts to ensuring stuff lines up properly. (This is herding cats).
- Reduce phase: Transform the output of the shuffle phase with another user-defined function. (May or may not be necessary.)

MapReduce Dataflow



When to use MapReduce?

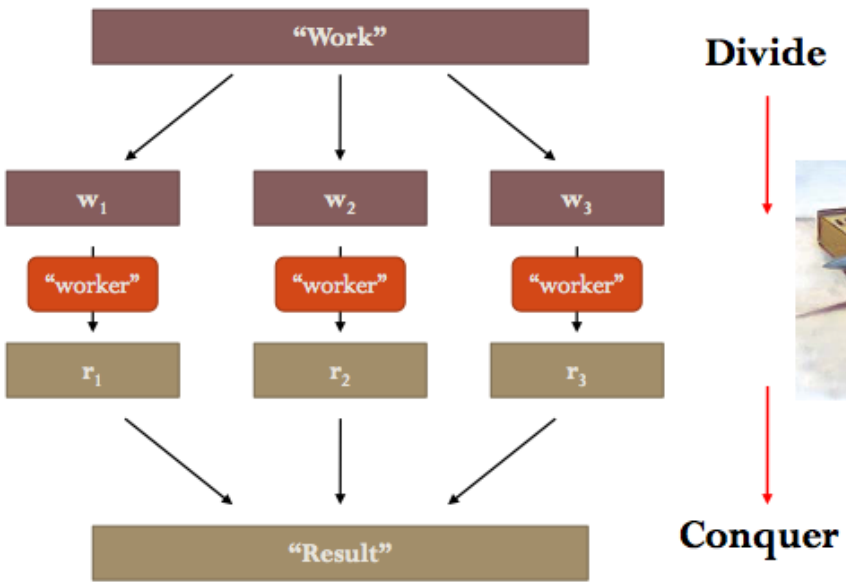
- When there are **efficiencies** to be gained from the types of parallelization we discussed earlier.
- In other words, whenever you have **big data**.
- Data **must be** “split-able” into chunks and records.

Designing MapReduce Algorithms

- User must decide what is to be done by map and **separately** by reduce.
  - Map can act on **individual** key-value pairs, but it cannot look at other key-value pairs.
  - Reduce can **aggregate** data by looking at multiple values, as long as map has properly mapped them.
- **Never easy to herd cats**.

MapReduce for Big Data

- Data parallelism by scaling out:
  - Divide and Conquer
- Distributed computing:
  - Data on different machines



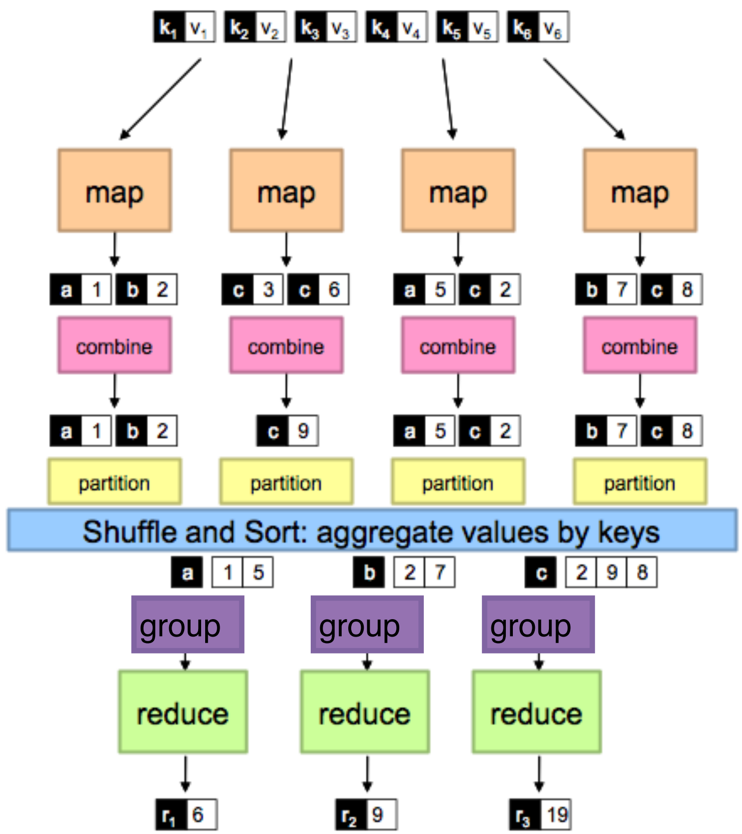
What is Hadoop?

- Hadoop is used in **common parlance** to describe:
  1. The **MapReduce** Paradigm.
  2. Massive unstructured data **storage**.
  3. HDFS: the Hadoop distributed file system.
- **In other words**:
  – Hadoop = HDFS + MapReduce.
  – Hadoop = Big Data + Analytics.

(H)DFS

- Files are divided into **chunks**.
- Chunks are **replicated** at different compute nodes.
- Chunk size and the degree of replication are chosen **by the user**.
- A special file (**the master node**) stores, for each file, the positions of its chunks.
  - So-called “master/slave” architecture.
- This is an important element of properly **herding the cats**.

Hadoop and MapReduce



Hadoop Tasks at Runtime

- Handles **scheduling**.
  - Assigns workers to map and reduce tasks.
- Handles data **distribution**.
  - Gets data to the workers.
- Handles **synchronization**.
  - Gathers, sorts, and shuffles intermediate data.
- Handles **errors and faults**.
  - Detects worker failures and restarts.
- Everything happens on top of a **distributed** file sharing system.

Hadoop Operation Modes

- Java MapReduce Mode.
  - Write Mapper, Combiner, Reducer functions in Java using Hadoop Java APIs.
  - Read records one at a time.
- Streaming Mode.
  - Any statistical computing language, such as Python.
  - Input can be a line at a time or a stream at a time.

How I Use It: Simulation

- One of the most powerful simulation tools is the **Markov chain Monte Carlo (MCMC)**, a technique that can be massively scaled.
  - Like a calendar: I know tomorrow is Friday because today is Thursday and it doesn't matter what yesterday was.
- Re-ignited **Bayesian** approaches to analysis and inference, rapidly displacing frequentist approaches based on the non-existent idea of "in repeated samples."

Where I Started: Apache Spark

- Brings clustered computing **to the masses**.
- Native APIs for R, Python and SQL.
- Massively **extended** the MapReduce paradigm so that folks **like us** can use it in everyday practice.
- The distributed computing environment is ideal for simulation, including MCMC.

Small Data v. Big Data

- In the old days, data were **relatively** cheap, but the computation was expensive.
- The reverse is now true. Good data are **relatively very expensive**, but computation is pocket change.
- Building data lakes is difficult not because of the physical architecture but because of the **organizational structure**.

Rapidly Developing Environment

- Hard for mere mortals **to keep up**.
- Spark 0.1 to 2.0 in **less than** three years.
- Closed source **buys** its way into open source.
  - Microsoft acquires Revolution Analytics and sticks it in its cloud. Legal protection for developers.
  - IBM acquires Anaconda. Abandons its license-fee approach of SPSS.
- SaaS becomes MLaaS.

A Golden Age of Empiricism

- Despite it all, we live in golden age of empiricism.
- We need to think deeply about empirical causation in a world of digital exhaust.
- We can look to past conversations: Judea Pearl.
- Don’t count out the statisticians: Efron and Hastie.

Thank You