CSE 519: Data Science

Steven Skiena
Stony Brook University

Lecture 23: Achieving Scale

Data Science and Big Data

The buzzword “Big Data” presumes analysis of truly massive data sets:

- all of Twitter or Facebook
- Web logs for major websites
- genome sequences of thousands of people
- all images on Flickr

Working with data generally gets harder with size

How Big Is… (2016)

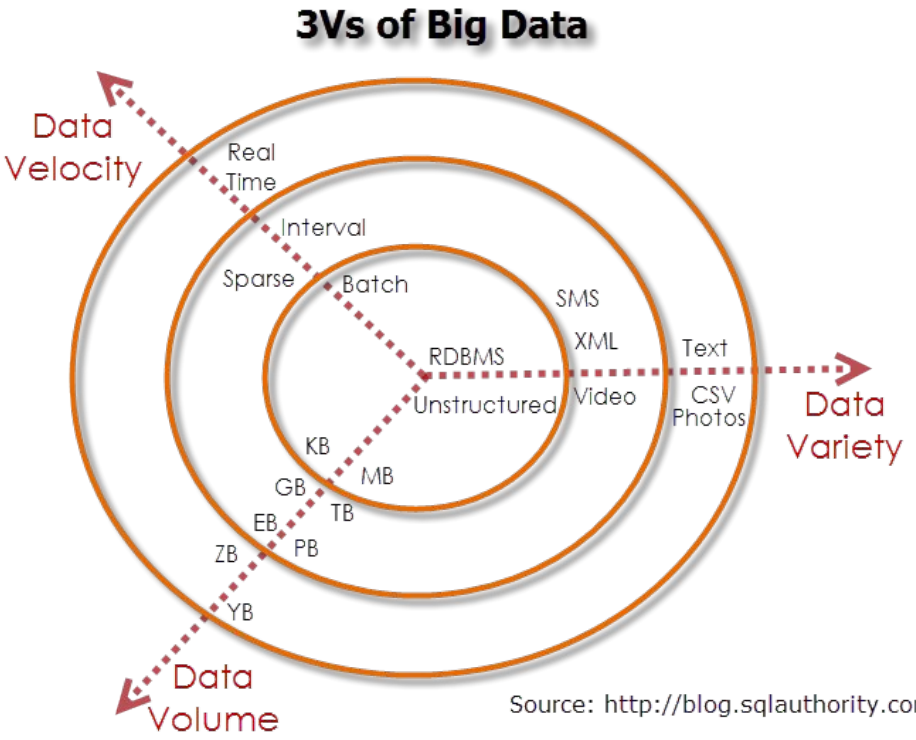
- Twitter (600 million tweets/day)
- Facebook (>600 TB incoming data per day)
- Google (3.5 billion search queries/day)
- Instagram (52 million photos per day)
- Apple (130 billion app downloads)
- Email (205 billion message/day)

http://www.internetlivestats.com

The Three V's of Big Data

Student projects are typically batch problems on MB scale CSV-type data.

Veracity is another V to worry about.



Could You Go Big? (Projects)

- Miss Universe?
- Movie gross?
- Baby weight?
- Art auction price?
- Snow on Christmas?
- Super Bowl / College Champion?
- Ghoul Pool?
- Future Gold / Oil Price?

Big Data as Bad Data

Massive data sets are the result of opportunity instead of design, with problems of:

- Unrepresentative participation (bias)
- Spam and machine-generated content
- Power-laws mean too much redundancy
- Susceptibility to temporal bias (e.g Google Flu Trends)

Large-Scale Machine Learning

The learning algorithms we have studied generally do not scale well to huge data sets.

- Models with few parameters cannot really benefit from large numbers of examples.
- Algorithmic complexity must be near linear to run on large data sets.
- Big matrices better be sparse for big data.

Customization and Specialization

Big data on all your customers translates to modest data on each of many individuals.

Customization means training large numbers of small models, only possible with big data.

Filtering Data

An important benefit of Big Data is that you can discard much of it to make analysis cleaner.

English accounts for only 34% of all tweets on Twitter, but you can exclude the rest and leave enough for meaningful analysis.

Filtering away irrelevant or hard-to-interpret data requires application-specific knowledge.

Subsampling Data

It can pay to subsample good, relevant data:

- Cleanly separate training, testing, and evaluation data.
- Simple, robust models generally have few parameters, making Big Data is overkill.
- Spreadsheet-sized data sets are fast and easy to explore.

Subsampling by Truncation

Taking the first *n* records is reproducible and simple but record order often has meaning:

- temporal biases: only analyze old data.
- lexicographic biases: only analyze the A's, e.g. more Arabic names, fewer Chinese.
- numerical biases: ID numbers can encode meaning, e.g. social security numbers.

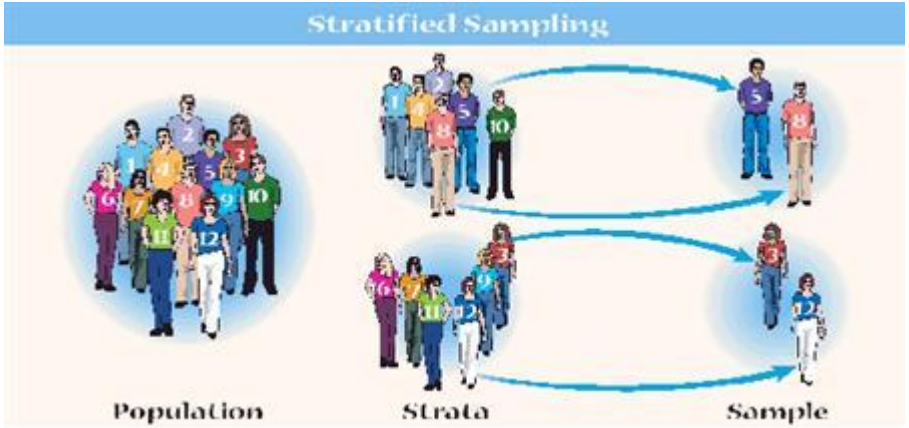
Random Sampling

Randomly sampling records with probability *p* ensures no explicit biases, but:

- Statistical discrepancies ensure some regions will be oversampled.
- Random sampling is not reproducible without the seed and random generator.
- Multiple random samples will not be disjoint.

Stratified Random Sampling

To ensure no group is statistically over- or under-sampled, we can explicitly sample proportionally from each group.



Uniform Sampling

Sampling records which are congruent to *i mod m* provide a way to balance many concerns:

- Obtain an exact number of records.
- Quick and reproducible.
- Ensures disjoint samples

Twitter uses this method to govern API services (spritzer vs. garden hose vs. fire hose)

Stream Sampling

Often we seek a uniform sample of size *k* from a stream, where we don't know *n* in advance.

Solution: keep an array of *k* active elements so far, then replace one of them with the nth element with probability *k/n*.

Select the position of the new element at random if it makes the cut.

Distributed vs. Parallel Processing

The distinction here is how tightly coupled the machines are, roughly:

- Parallel processing happens on one machine, through threads and OS processes
- Distributed processing happens on many machines, using network communication.

Easy parallel jobs do not communicate much.

Data Parallelism

The easiest way to exploit parallelism partitions big data among multiple machines and trains independent models.

Natural partitions are established by time, clustering algorithms, or given categories.

It is typically hard to combine the results of these runs together later

Grid Search

The easiest way to exploit parallelism involves independent runs on independent data.

Grid search is the quest for the right meta-parameters for training, like deciding the right *k* for k-means clustering.

Multiple independent fits can run in parallel, where in the end we take the best one.

One, Two, Many...

The complexity of distributed processing grows rapidly with the number of machines:

- One: keep the cores of your box busy.
- Two: manually run programs on a few boxes
- Many: employ a system like MapReduce for efficiently managing multiple machines.

Complexities of Scale: Social Gatherings

- 1 person: easy to arrange.
- >2 persons: coordination.
- >10 persons: requires leader in charge.
- >100 persons: requires fixed menu.
- >1000 persons: no one knows many people.
- >10,000 persons: too few hotels for most cities.
- >100,000 persons: someone **will** die that day

MapReduce / Hadoop

Google's MapReduce paradigm for distributed computing has spread widely through the open-source implementation Hadoop, offering:

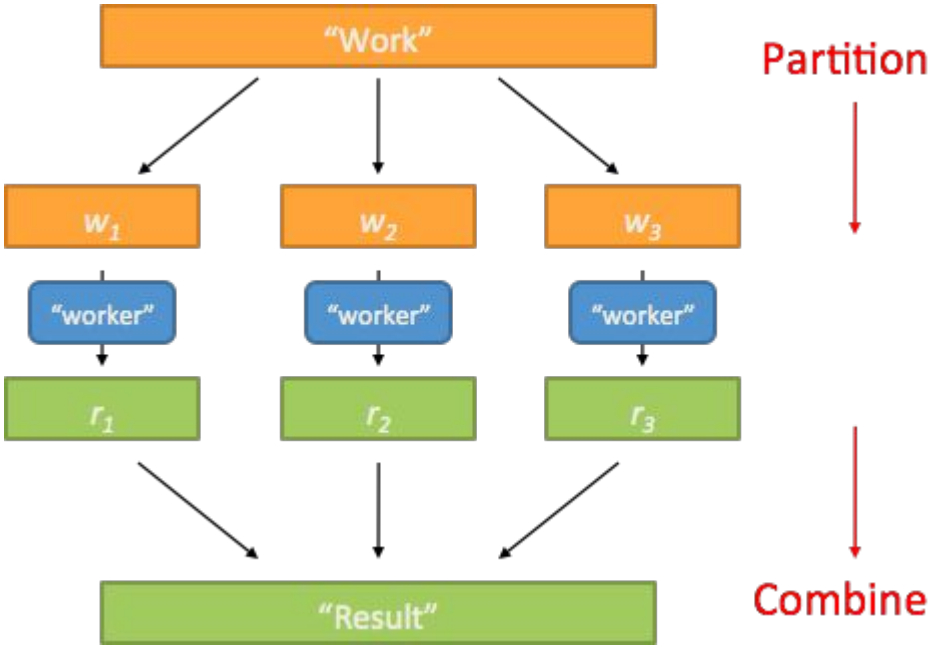
- Simple parallel programming model
- Straightforward scaling to hundreds/thousands of machines.
- Fault tolerance through redundancy

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Think word counting and k-means clustering

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 share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

Ideas Behind MapReduce

- Scale “out”, not “up”: recognize limits of large shared-memory machines
- Move processing to the data: clusters have limited bandwidth
- Process data sequentially, avoid random access: seeks are expensive, disk throughput is reasonable
- Seamless scalability: from the mythical man-month to the tradable machine-hour

Components of Hadoop

- Core Hadoop has two main systems:
  - **Hadoop/MapReduce**: distributed big data processing infrastructure (abstract/paradigm, fault-tolerant, schedule, execution)
  - **HDFS (Hadoop Distributed File System)**: fault-tolerant, high-bandwidth, high availability distributed storage

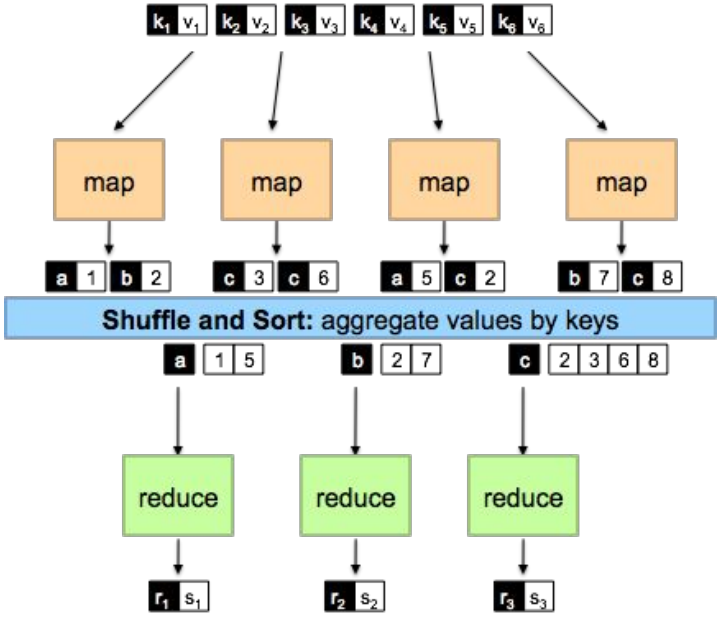
Map and Reduce

- Programmers specify two functions:

map (k, v) → [(k', v')]

reduce (k', [v']) → [(k', v')]

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



MapReduce Word Count

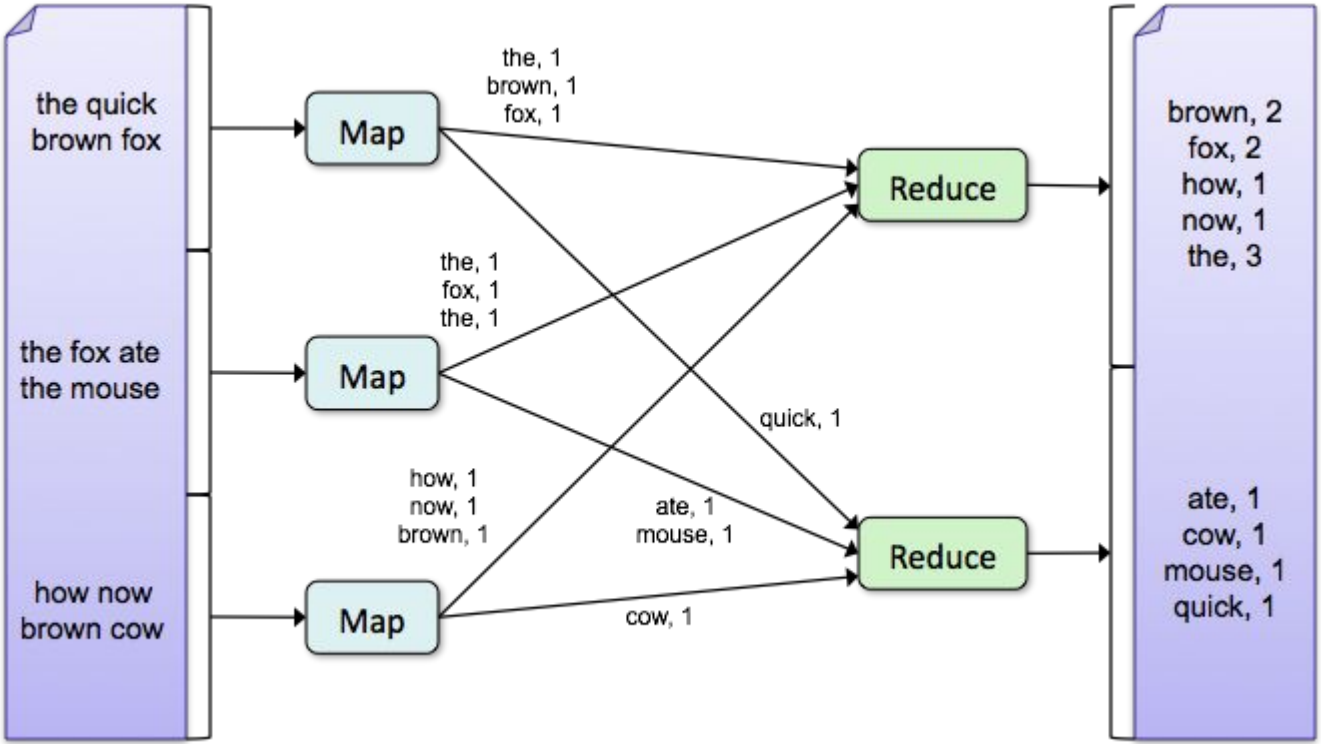
```
Map(String docid, String text):
    for each word w in text:
      Emit(w, 1);
Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
      sum += v;
    Emit(term, sum);
```

Word Count Example

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

```
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Word Count Execution



Other Programming Primitives

partition (k', number of partitions) → partition for k'

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

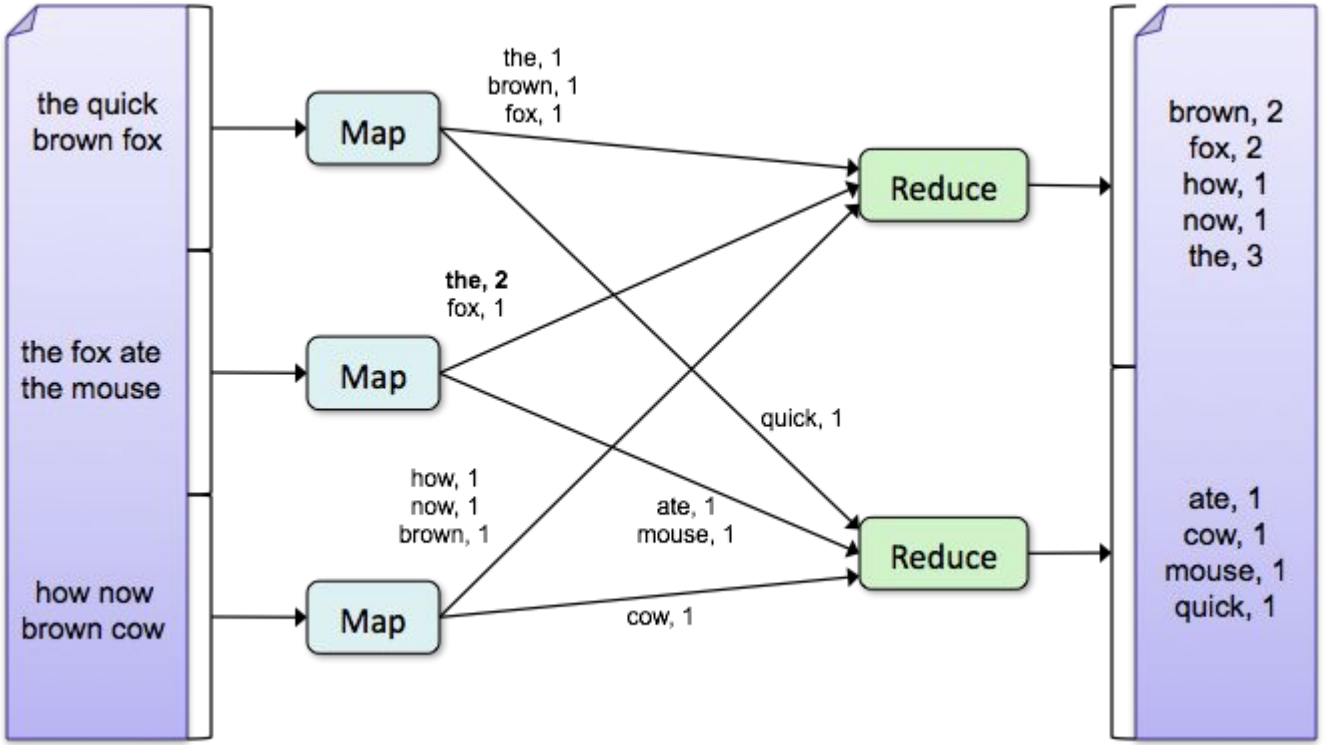
–Divides up key space for parallel reduce operations

combine (k', [v']) → [(k', v'')]

–Mini-reducers that run in memory after the map phase

–Used as an optimization to reduce network traffic

Word Count with Combiner



```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

MapReduce “RunTime”

- Handles scheduling: assigning workers to map and reduce tasks
- Handles “data distribution”: moves processes to data
- Handles synchronization: Gathers, sorts, and shuffles intermediate data
- Handles errors and faults: Detects worker failures and restarts

Hadoop Distributed File System (HDFS)

–Store data on the local disks of nodes in the cluster, because not enough RAM to hold all the data in memory

–Disk access is slow, but disk throughput is reasonable, so linear scans through files are fine.

--Replicate everything 3 times for reliability on commodity hardware.

Cloud Computing Services

Platforms like Amazon make it easy to rent large numbers of machines for short-term jobs.

There are charges on bandwidth, processors, memory, long-term storage: making it non-trivial to price exactly.

Spot pricing and reserved instances lower costs for special usage patterns.

Feel Free to Experiment

Micro instances are only 1GB, single processor virtual machines.

Reasonable machines rent for 10 to 30 cents/hr.

Free Tier*

As part of AWS's Free Usage Tier, new AWS customers can get started with Amazon EC2 for free. Upon sign-up, new AWS customers receive the following EC2 services each month for one year:

- 750 hours of EC2 running Linux, RHEL, or SLES t2.micro instance usage
- 750 hours of EC2 running Microsoft Windows Server t2.micro instance usage
- 750 hours of Elastic Load Balancing plus 15 GB data processing
- 30 GB of Amazon Elastic Block Storage in any combination of General Purpose (SSD) or Magnetic, plus 2 million I/Os (with Magnetic) and 1 GB of snapshot storage
- 15 GB of bandwidth out aggregated across all AWS services
- 1 GB of Regional Data Transfer