Case Study 2: Document Retrieval

Clustering Documents

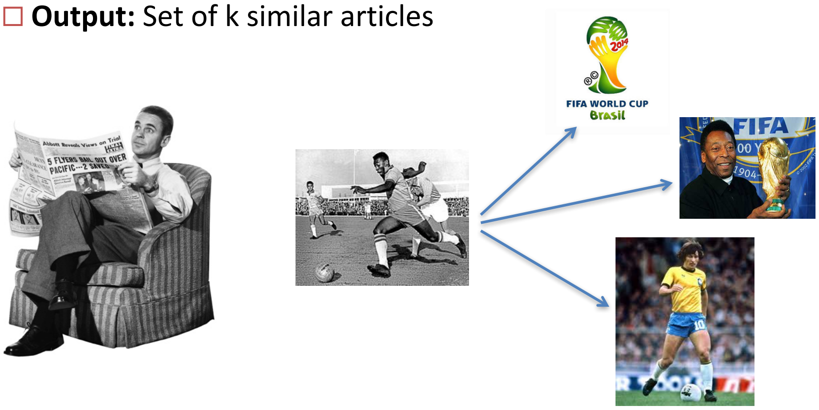
Machine Learning for Big Data
CSE547/STAT548, University of Washington
Sham Kakade
April , 2017

Document Retrieval

- **Goal:** Retrieve documents of interest
- **Challenges:**
  - Tons of articles out there
  - How should we measure similarity?

Task 1: Find Similar Documents

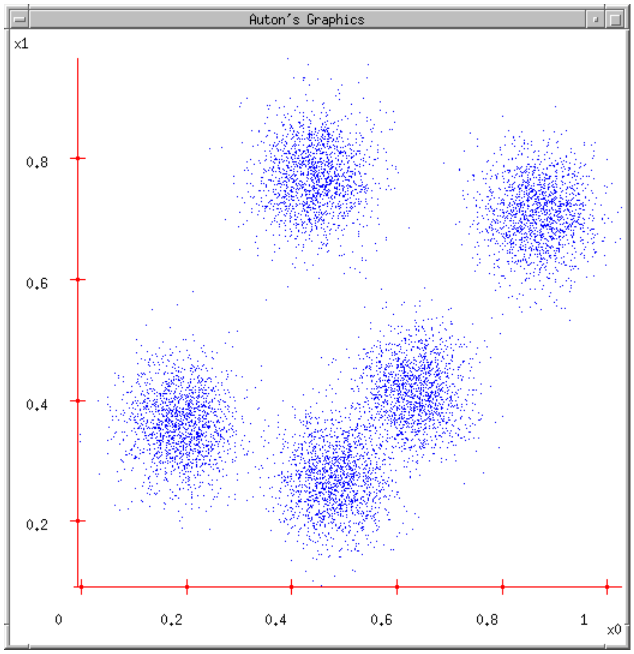
- **So far…**
  - **Input:** Query article
  - **Output:** Set of k similar articles



Task 2: Cluster Documents

- **Now:**
  - Cluster documents based on topic

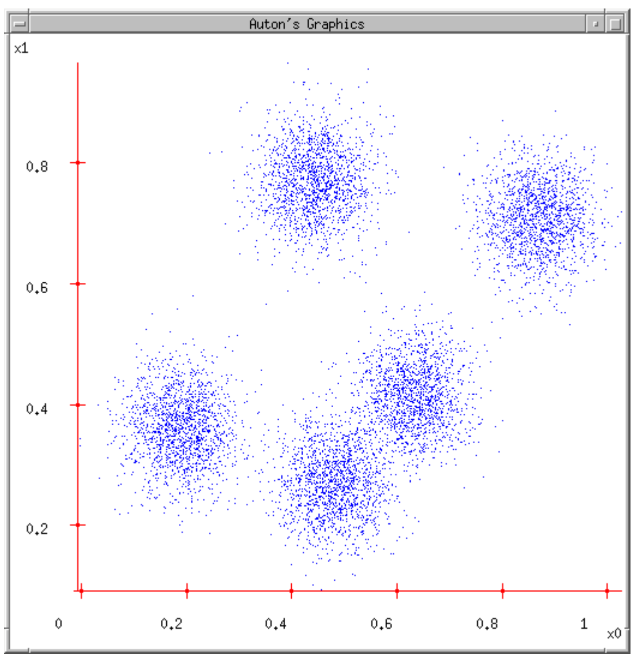
Some Data



 5

K-means

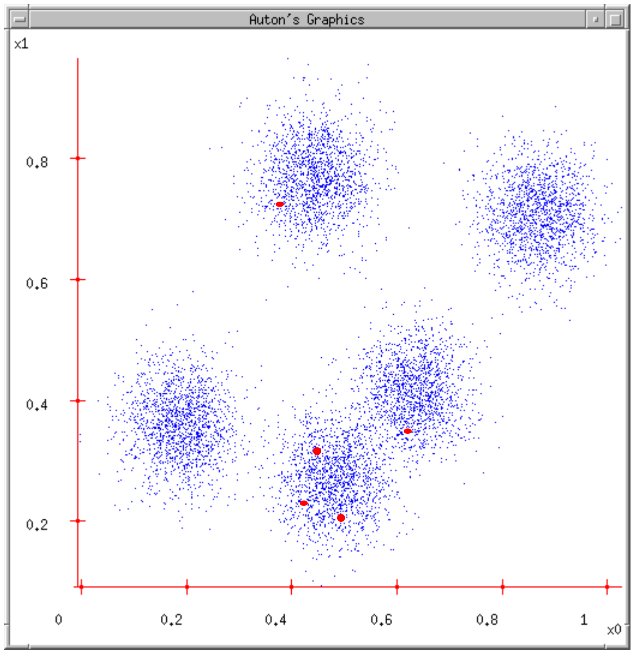
1. Ask user how many clusters they'd like. *(e.g. k=5)*



 6

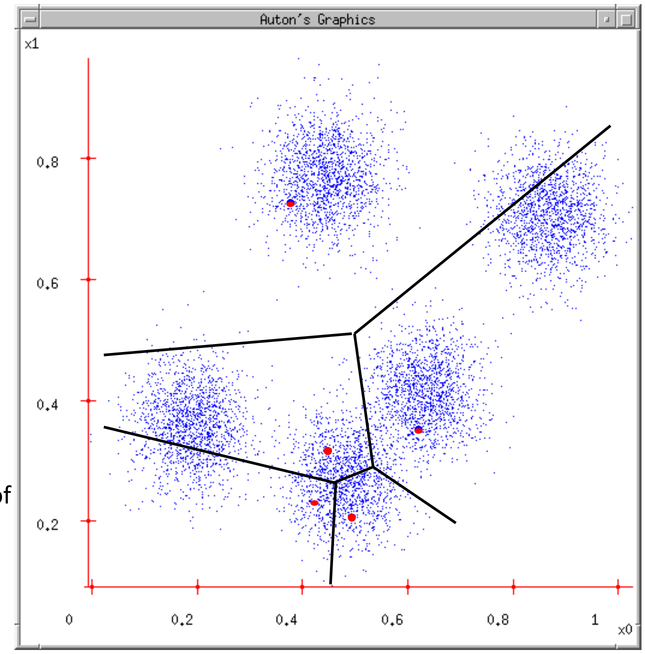
K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations



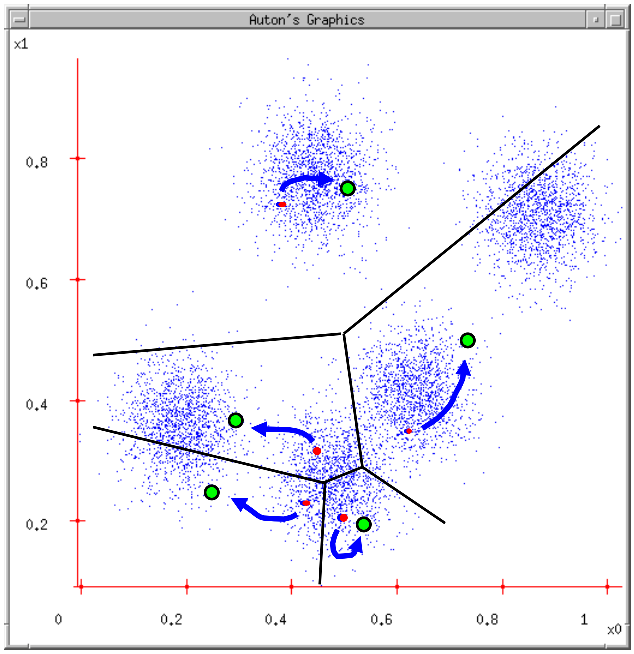
K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to. (Thus each Center "owns" a set of datapoints)



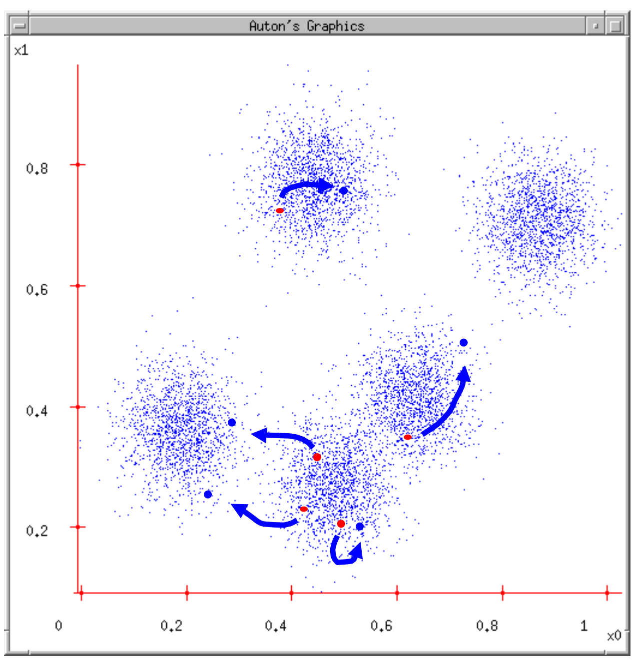
K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to.
4. Each Center finds the centroid of the points it owns



K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to.
4. Each Center finds the centroid of the points it owns…
5. …and jumps there
6. …Repeat until terminated!



K-means

- Randomly initialize *k* centers

  $\mu^{(0)} = \mu_1^{(0)}, \ldots, \mu_k^{(0)}$

- **Classify**: Assign each point $j \in \{1,\ldots N\}$ to nearest center:

$$z^j \leftarrow \arg\min_i ||\mu_i - \mathbf{x}^j||_2^2$$

- **Recenter**: μ_i becomes centroid of its point:

$$\mu_i^{(t+1)} \leftarrow \arg\min_\mu \sum_{j:z^j=i} ||\mu - \mathbf{x}^j||_2^2$$

  – Equivalent to $\mu_i \leftarrow$ average of its points!

Case Study 2: Document Retrieval

Parallel Programming
Map-Reduce

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Sham Kakade
April , 2017

Needless to Say, We Need Machine Learning for Big Data

6 Billion
Flickr Photos

28 Million
Wikipedia Pages

1 Billion
Facebook Users

72 Hours a Minute
YouTube

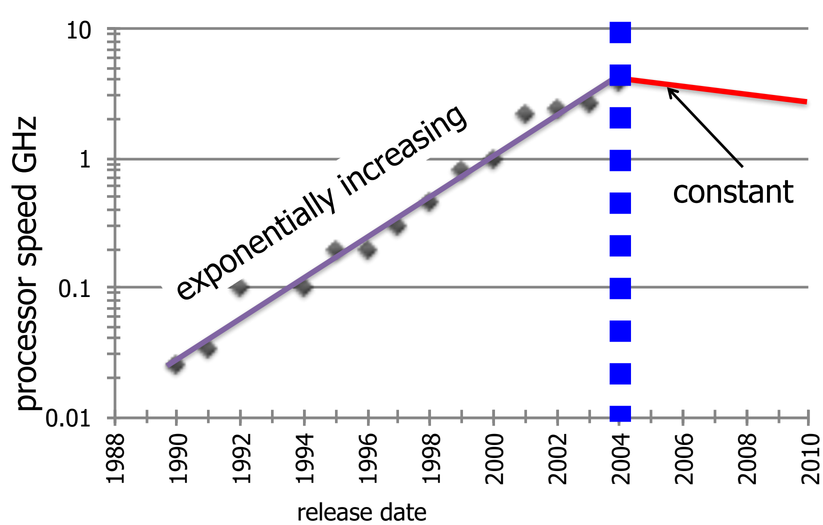
The New York Times
SundayReview

WORLD U.S. N.Y. / REGION BUSINESS TEC

NEWS ANALYSIS
The Age of Big Data
By STEVE LOHR
Published: February 11, 2012

"… data a new class of economic asset, like currency or gold."

CPUs Stopped Getting Faster…



ML in the Context of Parallel Architectures

GPUs Multicore Clusters Clouds Supercomputers

- But scalable ML in these systems is hard, especially in terms of:
  1. Programmability
  2. Data distribution
  3. Failures

Programmability Challenge 1: Designing Parallel Programs

- SGD for LR:
  - For each data point $\mathbf{x}^{(t)}$:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta_t \left\{ -\lambda w_i^{(t)} + \phi_i(\mathbf{x}^{(t)})[y^{(t)} - P(Y=1|\phi(\mathbf{x}^{(t)}), \mathbf{w}^{(t)})] \right\}$$

Programmability Challenge 2: Race Conditions

- We are used to sequential programs:
  - Read data, think, write data, read data, think, write data, read data, think, write data, read data, think, write data, read data, think, write data, read data, think, write data…
- But, in parallel, you can have non-deterministic effects:
  - One machine reading data while other is writing
- Called a race-condition:
  - Very annoying
  - One of the hardest problems to debug in practice:
    - because of non-determinism, bugs are hard to reproduce

 17

Data Distribution Challenge

- Accessing data:
  - Main memory reference: 100ns (10^{-7}s)
  - Round trip time within data center: 500,000ns (5 * 10^{-4}s)
  - Disk seek: 10,000,000ns (10^{-2}s)
- Reading 1MB sequentially:
  - Local memory: 250,000ns (2.5 * 10^{-4}s)
  - Network: 10,000,000ns (10^{-2}s)
  - Disk: 30,000,000ns (3*10^{-2}s)
- Conclusion: Reading data from local memory is **much** faster ➔ Must have data locality:
  - Good data partitioning strategy fundamental!
  - "Bring computation to data" (rather than moving data around)

 18

Robustness to Failures Challenge

- From Google's Jeff Dean, about their clusters of 1800 servers, in first year of operation:
  - 1,000 individual machine failures
  - thousands of hard drive failures
  - one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours
  - 20 racks will fail, each time causing 40 to 80 machines to vanish from the network
  - 5 racks will "go wonky," with half their network packets missing in action
  - the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span
  - 50% chance cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover
- How do we design distributed algorithms and systems robust to failures?
  - It's not enough to say: run, if there is a failure, do it again… because you may never finish

Move Towards Higher-Level Abstraction

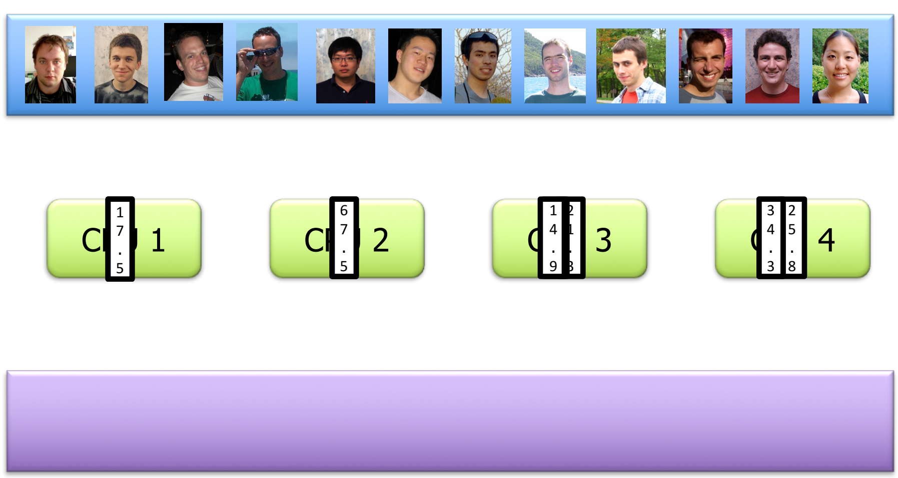
- Distributed computing challenges are hard and annoying!
  1. Programmability
  2. Data distribution
  3. Failures
- High-level abstractions try to simplify distributed programming by hiding challenges:
  - Provide different levels of robustness to failures, optimizing data movement and communication, protect against race conditions…
  - Generally, you are still on your own WRT designing parallel algorithms
- Some common parallel abstractions:
  - Lower-level:
    - Pthreads: abstraction for distributed threads on single machine
    - MPI: abstraction for distributed communication in a cluster of computers
  - Higher-level:
    - Map-Reduce (Hadoop: open-source version): mostly data-parallel problems
    - GraphLab: for graph-structured distributed problems

Simplest Type of Parallelism: Data Parallel Problems

- You have already learned a classifier
  - What's the test error?
- You have 10B labeled documents and 1000 machines

- Problems that can be broken into independent subproblems are called data-parallel (or embarrassingly parallel)
- Map-Reduce is a great tool for this…
  - Focus of today's lecture
  - but first a simple example

Data Parallelism (MapReduce)



*Solve a huge number of **independent** subproblems, e.g., extract features in images*

Counting Words on a Single Processor

- (This is the "Hello World!" of Map-Reduce)
- Suppose you have 10B documents and 1 machine
- You want to count the number of appearances of each word in this corpus
  - Similar ideas useful for, e.g., building Naïve Bayes classifiers and computing TF-IDF
- Code:

Naïve Parallel Word Counting

- Simple data parallelism approach:

- Merging hash tables: annoying, potentially not parallel ➔ no gain from parallelism???

Counting Words in Parallel & Merging Hash Tables in Parallel

- Generate pairs (word,count)
- Merge counts for each word in parallel
  - Thus parallel merging hash tables

 25

Map-Reduce Abstraction

- Map:
  - Data-parallel over elements, e.g., documents
  - Generate (key,value) pairs
    - "value" can be any data type
- Reduce:
  - Aggregate values for each key
  - Must be commutative-associate operation
  - Data-parallel over keys
  - Generate (key,value) pairs
- Map-Reduce has long history in functional programming
  - But popularized by Google, and subsequently by open-source Hadoop implementation from Yahoo!

 26

Map Code (Hadoop): Word Count

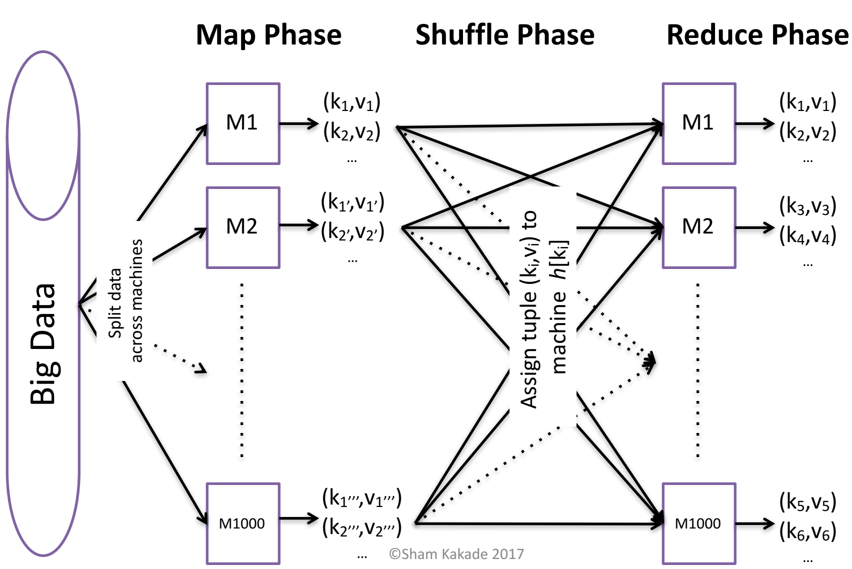
```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws <stuff>
 {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
     }
 }
```


Reduce Code (Hadoop): Word Count

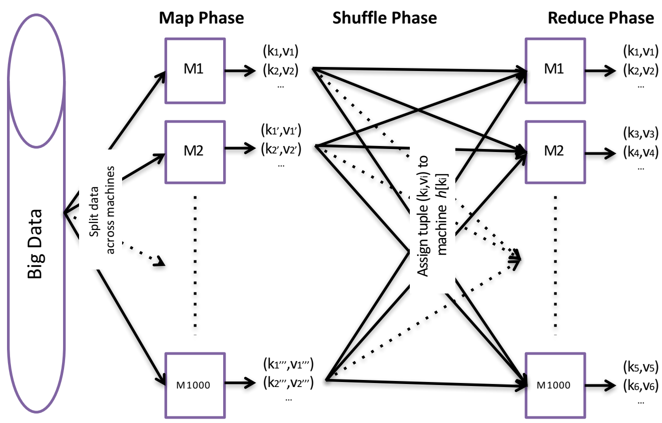
```
public static class Reduce extends Reducer<Text, IntWritable,
Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }
```

Map-Reduce Parallel Execution

Map-Reduce – Execution Overview



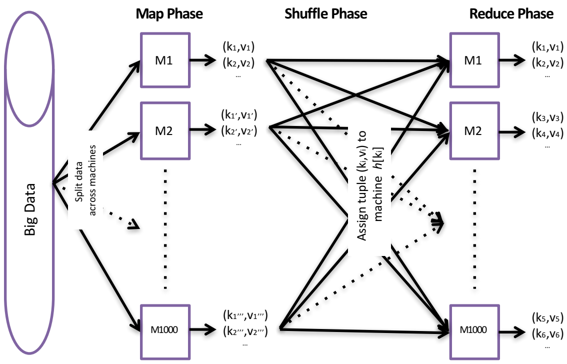
Map-Reduce – Robustness to Failures 1: Protecting Data: **Save To Disk Constantly**



Distributed File Systems

- Saving to disk locally is not enough ➔ If disk or machine fails, all data is lost
- Replicate data among multiple machines!
- Distributed File System (DFS)
  – Write a file from anywhere ➔ automatically replicated
  – Can read a file from anywhere ➔ read from closest copy
    - If failure, try next closest copy
- Common implementations:
  – Google File System (GFS)
  – Hadoop File System (HDFS)
- Important practical considerations:
  – Write large files
    - Many small files ➔ becomes way too slow
  – Typically, files can't be "modified", just "replaced" ➔ makes robustness much simpler

Map-Reduce – Robustness to Failures 2: Recovering From Failures: **Read from DFS**



- Communication in initial distribution & shuffle phase "automatic"
  - Done by DFS
- If failure, don't restart everything
  - Otherwise, never finish
- Only restart Map/Reduce jobs in dead machines

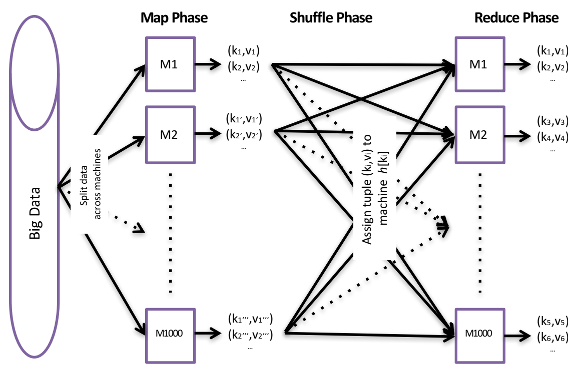
Improving Performance: Combiners

- Naïve implementation of M-R very wasteful in communication during shuffle:

- **Combiner**: Simple solution, perform reduce locally before communicating for global reduce
  - Works because reduce is commutative-associative

(A few of the) Limitations of Map-Reduce

- Too much synchrony
  - E.g., reducers don't start until all mappers are done
- "Too much" robustness
  - Writing to disk all the time
- Not all problems fit in Map-Reduce
  - E.g., you can't communicate between mappers
- Oblivious to structure in data
  - E.g., if data is a graph, can be much more efficient
    - For example, no need to shuffle nearly as much
- Nonetheless, extremely useful; industry standard for Big Data
  - Though many many companies are moving away from Map-Reduce (Hadoop)



What you need to know about Map-Reduce

- Distributed computing challenges are **hard** and **annoying**!
  1. Programmability
  2. Data distribution
  3. Failures
- High-level abstractions help a lot!
- Data-parallel problems & Map-Reduce
- Map:
  - Data-parallel transformation of data
    - Parallel over data points
- Reduce:
  - Data-parallel aggregation of data
    - Parallel over keys
- Combiner helps reduce communication
- Distributed execution of Map-Reduce:
  - Map, shuffle, reduce
  - Robustness to failure by writing to disk
  - Distributed File Systems

Case Study 2: Document Retrieval

Parallel K-Means on Map-Reduce

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Sham Kakade
April , 2017

 37

Map-Reducing One Iteration of K-Means

- **Classify**: Assign each point $j \in \{1,\ldots N\}$ to nearest center:

$$z^j \leftarrow \arg\min_i ||\mu_i - \mathbf{x}^j||_2^2$$

- **Recenter**: μ_i becomes centroid of its point:

$$\mu_i^{(t+1)} \leftarrow \arg\min_\mu \sum_{j:z^j=i} ||\mu - \mathbf{x}^j||_2^2$$

  - Equivalent to $\mu_i \leftarrow$ average of its points!

- **Map**:

- **Reduce**:

 38

Classification Step as Map

- **Classify**: Assign each point $j \in \{1,\ldots,N\}$ to nearest center:

$$z^j \leftarrow \arg\min_i ||\mu_i - \mathbf{x}^j||_2^2$$

- **Map**:

Recenter Step as Reduce

- **Recenter**: μ_i becomes centroid of its point:

$$\mu_i^{(t+1)} \leftarrow \arg\min_\mu \sum_{j:z^j=i} ||\mu - \mathbf{x}^j||_2^2$$

  - Equivalent to $\mu_i \leftarrow$ average of its points!

- **Reduce**:

Some Practical Considerations

- K-Means needs an iterative version of Map-Reduce
  - Not standard formulation
- Mapper needs to get data point and all centers
  - A lot of data!
  - Better implementation: mapper gets many data points

What you need to know about Parallel K-Means on Map-Reduce

- Map: classification step;
  data parallel over data points
- Reduce: recompute means;
  data parallel over centers