Parallel and Distributed Computing: MapReduce

Alona Fyshe

About Me

- I worked @Google for 3 years
  - And used MapReduce almost every day
- Now I'm a PhD student in Machine Learning
  - I work on brains

Overview

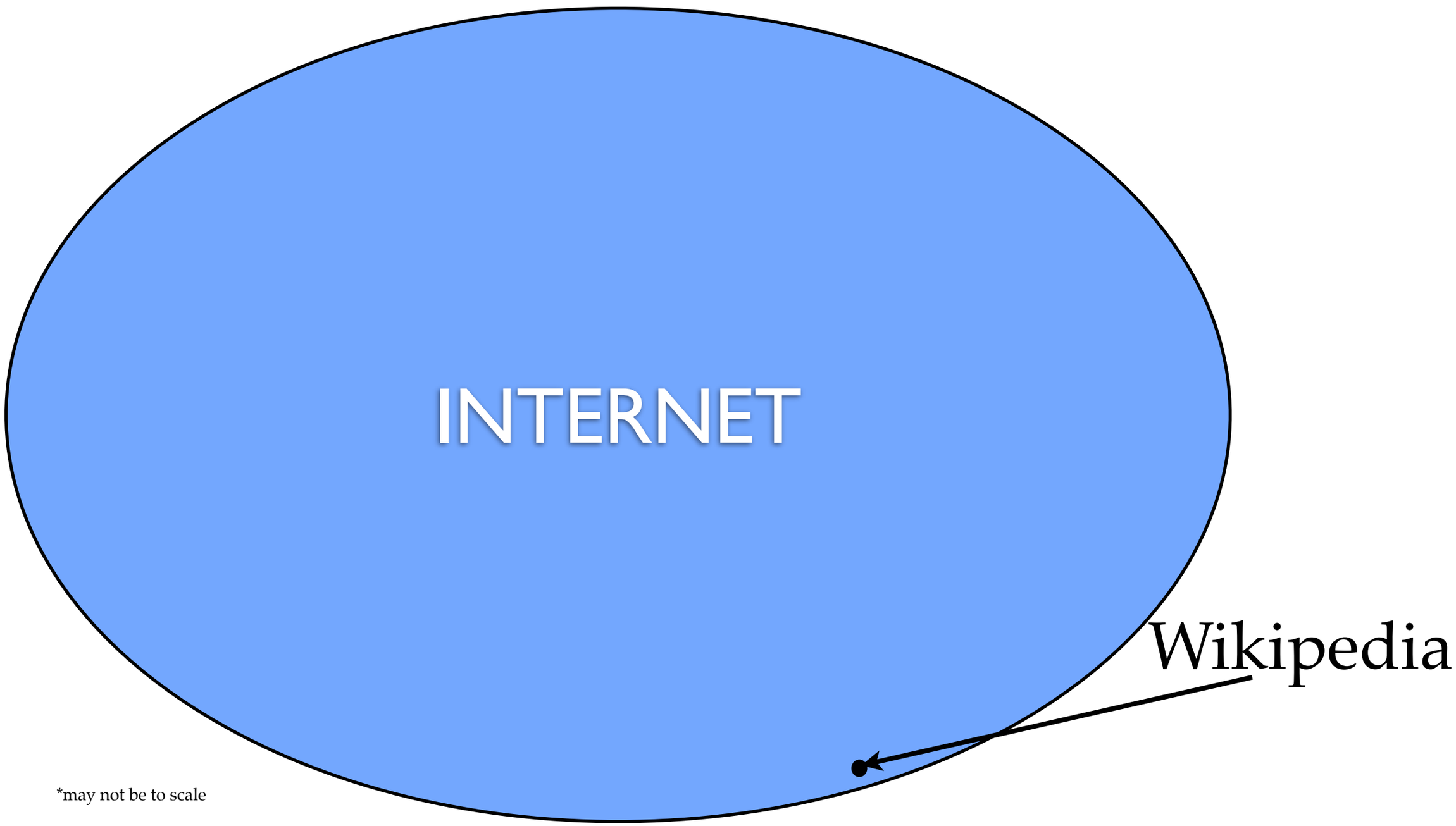
- Motivating example
  - considerations for distributed computation
- MapReduce
  - the idea
  - some examples

Inspiration not Plagiarism

- This is not the first lecture ever on Mapreduce
- I borrowed from:
  - Jimmy Lin
    - http://www.umiacs.umd.edu/~jimmylin/cloud-computing/SIGIR-2009/Lin-MapReduce-SIGIR2009.pdf
  - Google
    - http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html
    - http://code.google.com/edu/submissions/mapreduce/listing.html
  - Cloudera
    - http://vimeo.com/3584536

Motivating Example

- Wikipedia is a very small part of the internet*



*may not be to scale

Scaling up

- Let's do assignment 2 for the whole internet

Distributing NB

Distributing NB

- How can you handle new data?

Distributing NB

- How can you handle new data?
  - What about deleted data?

Distributing NB

- How can you handle new data?
  - What about deleted data?
- What happens at classification time?

Distributing NB

- How can you handle new data?
  - What about deleted data?
- What happens at classification time?
  - Can we distribute classification too?

Distributing NB

- How can you handle new data?
  - What about deleted data?
- What happens at classification time?
  - Can we distribute classification too?
    - Can we avoid merging the dictionaries then?

Distributing NB

- Questions:
  - How will you know when each machine is done?
    - Communication overhead
  - How will you know if a machine is dead?

Failure

- How big of a deal is it really?
  - A huge deal. In a distributed environment disks fail ALL THE TIME.
  - Large scale systems must assume that any process can fail at any time.

    Ken Arnold (Sun, CORBA designer):

> Failure is the defining difference between distributed and local programming, so you have to design distributed systems with the expectation of failure. Imagine asking people, "If the probability of something happening is one in 10^{13}, how often would it happen?" Common sense would be to answer, "Never." That is an infinitely large number in human terms. But if you ask a physicist, she would say, "All the time. In a cubic foot of air, those things happen all the time."
>
> When you design distributed systems, you have to say, "Failure happens all the time." So when you design, you design for failure. It is your number one concern. What does designing for failure mean? One classic problem is partial failure. If I send a message to you and then a network failure occurs, there are two possible outcomes. One is that the message got to you, and then the network broke, and I just didn't get the response. The other is the message never got to you because the network broke before it arrived.
>
> So if I never receive a response, how do I know which of those two results happened? I cannot determine that without eventually finding you. The network has to be repaired or you have to come up, because maybe what happened was not a network failure but you died. How does this change how I design things? For one thing, it puts a multiplier on the value of simplicity. The more things I can do with you, the more things I have to think about recovering from. [2]

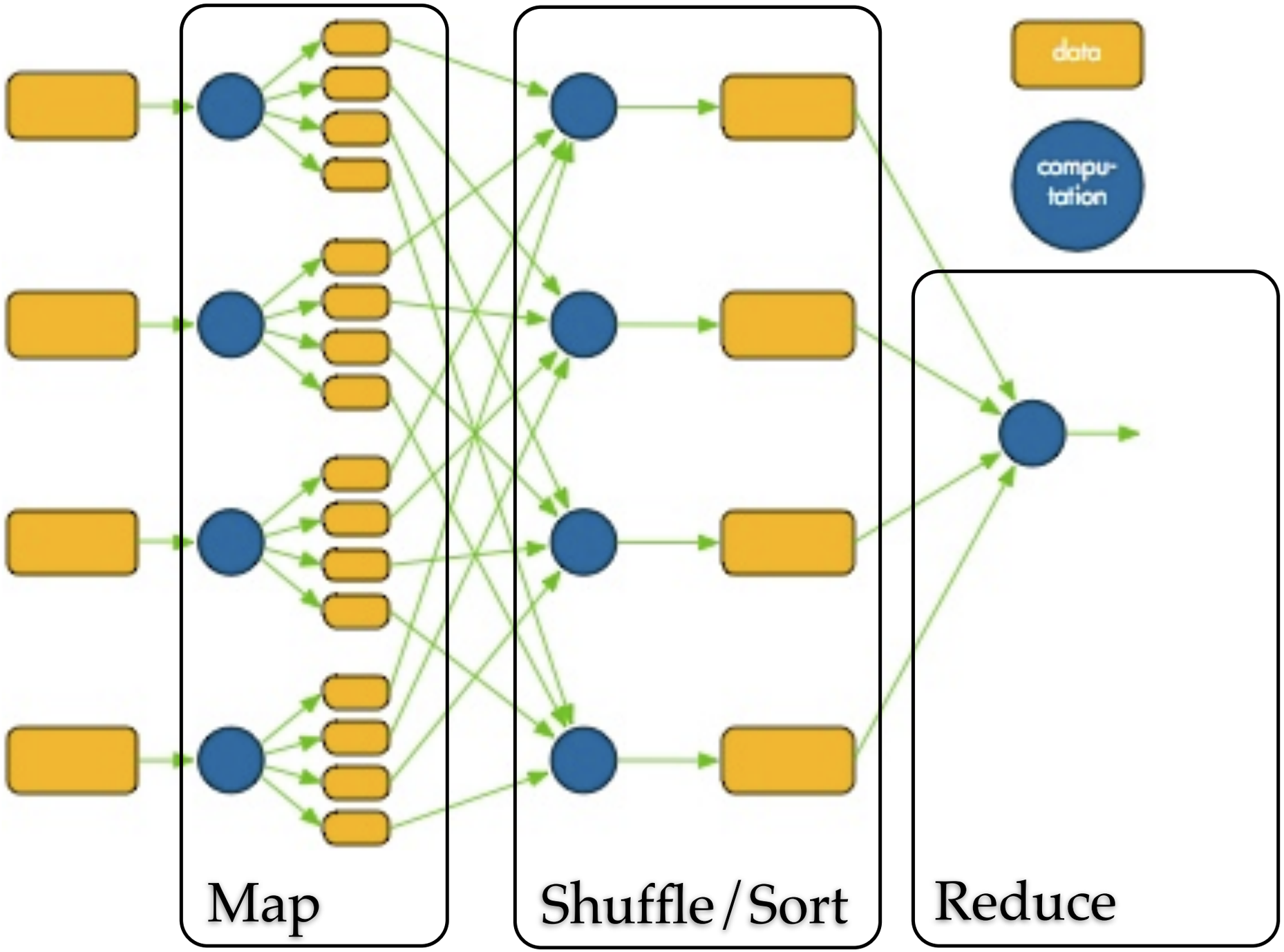
Well, that's a pain

- What will you do when a task fails?

Surprise, you mapreduced!

- Mapreduce has three main phases
  - Map (send each input record to a key)
  - Sort (put all of one key in the same place)
    - handled behind the scenes
  - Reduce (operate on each key and its set of values)

Mapreduce overview



Mapreduce: slow motion

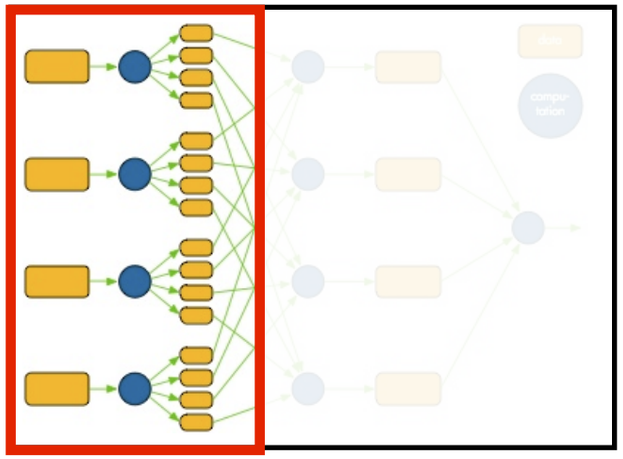
- The canonical mapreduce example is word count
- Example corpus:

  Joe likes toast

  Jane likes toast with jam

  Joe burnt the toast

MR: slow motion: Map



Input

Joe likes toast

Map 1

Jane likes toast with jam

Map 2

Joe burnt the toast

Map 3

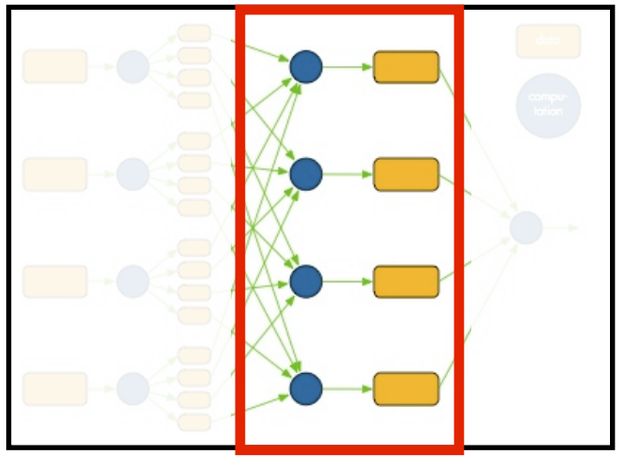
Output

Joe	1
likes	1
toast	1

Jane	1
likes	1
toast	1
with	1
jam	1

Joe	1
burnt	1
the	1
toast	1

MR: slow motion: Sort



Input

Joe	1
likes	1
toast	1

Jane	1
likes	1
toast	1
with	1
jam	1

Joe	1
burnt	1
the	1
toast	1

Output

Joe	1
Joe	1

Jane	1

likes	1
likes	1

toast	1
toast	1
toast	1

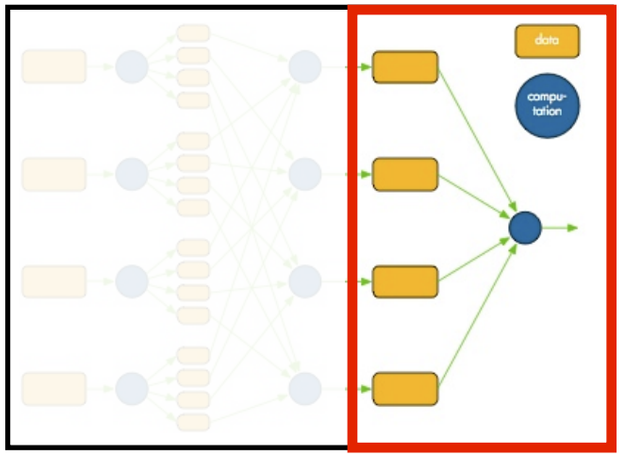
with	1

jam	1

burnt	1

the	1

MR: slow mo: Reduce



Input

Joe	1	Reduce 1
Joe	1	
Jane	1	Reduce 2
likes	1	Reduce 3
likes	1	
toast	1	Reduce 4
toast	1	
toast	1	
with	1	Reduce 5
jam	1	Reduce 6
burnt	1	Reduce 7
the	1	Reduce 8

Output

Joe	2
Jane	1
likes	2
toast	3
with	1
jam	1
burnt	1
the	1

Word count = Naive Bayes

- Building your NB dictionary was word count
- The interesting bit will be classifying using a map reduce
  - where interesting = homework

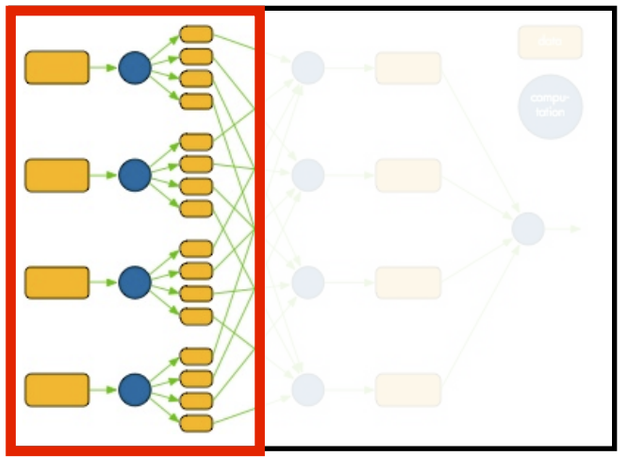
MR ~ functional programming

- If you're familiar with Functional Programming:
  - Map = map
  - Reduce ~ fold (acts on a subset of the map output, not all)
- If you're not familiar with FP:
  - Map = divide
  - Reduce = conquer
    - (kind of)

MR Code: Word Count Map

```
  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);
    }
  }
}
```

Reminder



Input

Joe likes toast
Map 1

Jane likes toast with jam
Map 2

Joe burnt the toast
Map 3

Output

Joe	1
likes	1
toast	1

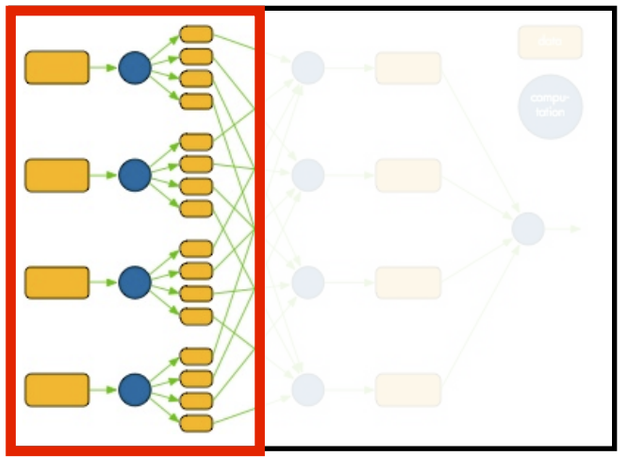
Jane	1
likes	1
toast	1
with	1
jam	1

Joe	1
burnt	1
the	1
toast	1

MR code: Word count Reduce

```
  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));
  }
}
```

Reminder



Input

Joe	1	Reduce 1
Joe	1	
Jane	1	Reduce 2
likes	1	Reduce 3
likes	1	
toast	1	Reduce 4
toast	1	
toast	1	
with	1	Reduce 5
jam	1	Reduce 6
burnt	1	Reduce 7
the	1	Reduce 8

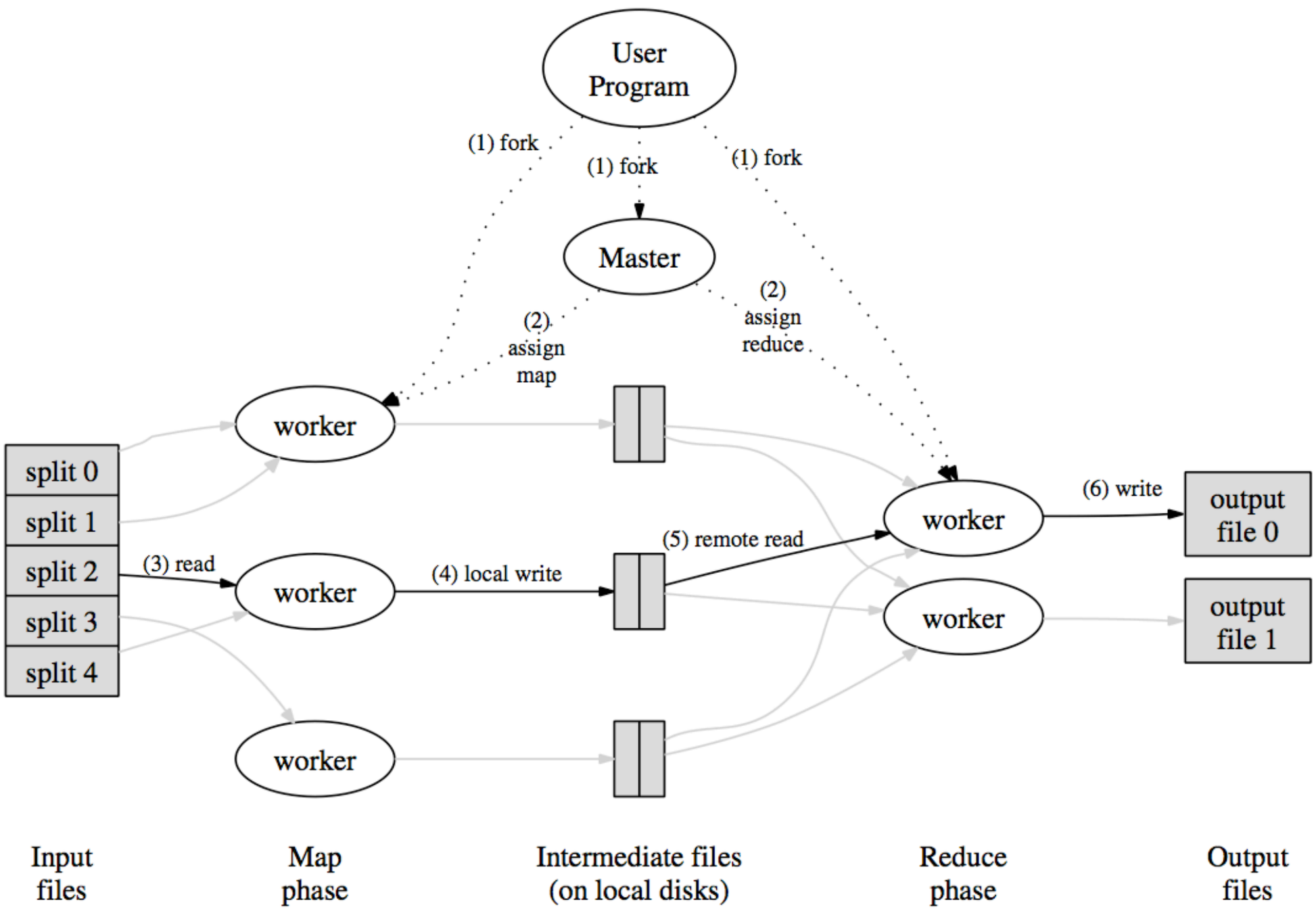
Output

Joe	2
Jane	1
likes	2
toast	3
with	1
jam	1
burnt	1
the	1

MR code: Word count Main

```
public static void main(String[] args) throws Exception {

 Configuration conf = new Configuration();

 Job job = new Job(conf, "wordcount");

 job.setMapperClass(Map.class);
 job.setReducerClass(Reduce.class);

     job.setInputFormatClass(TextInputFormat.class);
     job.setOutputFormatClass(TextOutputFormat.class);

      job.setOutputKeyClass(Text.class);

     job.setOutputValueClass(IntWritable.class);

     FileInputFormat.addInputPath(job, new Path(args[0]));
     FileOutputFormat.setOutputPath(job, new Path(args[1]));

     job.waitForCompletion(true);
 }
```

MR Overview


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Is any part of this wasteful?

Is any part of this wasteful?

- Remember - moving data around and writing to/reading from disk are very expensive operations

Is any part of this wasteful?

- Remember - moving data around and writing to/reading from disk are very expensive operations
- No reducer can start until:

Is any part of this wasteful?

- Remember - moving data around and writing to/reading from disk are very expensive operations
- No reducer can start until:
    - all mappers are done

Is any part of this wasteful?

- Remember - moving data around and writing to/reading from disk are very expensive operations
- No reducer can start until:
  - all mappers are done
  - data in its partition has been sorted

Combiners

- Sits between the map and the shuffle
  - Do some of the reducing while you're waiting for other stuff to happen
  - Avoid moving all of that data over the network
- Only applicable when
  - order of reduce values doesn't matter
  - effect is cumulative

MR code: Word count Combiner

```
public static class Combiner 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));
   }
 }
```

Deja vu: Combiner = Reducer

- Often the combiner is the reducer.
  - like for word count
  - but not always

Some common pitfalls

- You have no control over the order in which reduces are performed
- You have “no” control over the order in which you encounter reduce values
    - More on this later
- The only ordering you should assume is that Reducers always start after Mappers

Some common pitfalls

- You should assume your Maps and Reduces will be taking place on different machines with different memory spaces
- Don't make a static variable and assume that other processes can read it
  - They can't.
  - It appear that they can when run locally, but they can't
  - No really, don't do this.

Some common pitfalls

- Do not communicate between mappers or between reducers
  - overhead is high
  - you don't know which mappers/reducers are actually running at any given point
  - there's no easy way to find out what machine they're running on
    - because you shouldn't be looking for them anyway

When mapreduce doesn't fit

- The beauty of mapreduce is its separability and independence
- If you find yourself trying to communicate between processes
    - you're doing it wrong

      or
    - what you're doing is not a mapreduce

When mapreduce doesn't fit

- Not everything is a mapreduce
- Sometimes you need more communication
  - We'll talk about other programming paradigms later

Thinking in Mapreduce

- A new task: Word co-occurrence statistics (simplified)
- Input:

  Sentences

- Output:

  P(Word B is in sentence | Word A started the sentence)

Thinking in mapreduce

- We need to calculate

  P(B in sentence | A started sentence) =

  P(B in sentence & A started sentence)/P(A started sentence)=

  count<A,B>/count<A,*>

Word Co-occurrence: Solution 1

- The **Pairs** paradigm:
    - For each sentence, output a pair
    - E.g Map("Machine learning for big data") creates:

      <Machine, learning>:1

      <Machine, for>:1

      <Machine, big>:1

      <Machine, data>:1

      <Machine,*>:1

Word Co-occurrence: Solution 1

- Reduce would create, for example:

  <Machine, learning>:10

  <Machine, for>:1000

  <Machine, big>:50

  <Machine, data>:200

  ...

  <Machine,*>:12000

Word Co-occurrence: Solution 1

P(B in sentence | A started sentence) =

P(B in sentence & A started sentence)/P(A started sentence)=

<A,B>/<A,*>

- Do we have what we need?
  - Yes!

Word Co-occurrence: Solution 1

- But wait!
  - There's a problem .... can you see it?

Word Co-occurrence: Solution 1

Word Co-occurrence: Solution 1

- Each reducer will process all counts for a <word1,word2> pair

Word Co-occurrence: Solution 1

- Each reducer will process all counts for a <word1,word2> pair
- We need to know <word1,*> at the same time as <word1,word2>

Word Co-occurrence: Solution 1

- Each reducer will process all counts for a <word1,word2> pair
- We need to know <word1,*> at the same time as <word1,word2>
- The information is in different reducers!

Word Co-occurrence: Solution 1

- Solution 1 a)
  - Make the first word the reduce key
  - Each reducer has:
    - key: word_i
    - values:
      <word_i,word_j>....<word_i,word_b>.....<word_i,*>....

Word Co-occurrence: Solution 1

- Now we have all the information in the same reducer
- But, now we have a new problem, can you see it?

- Hint: remember - we have no control over the order of values

Word Co-occurrence: Solution 1

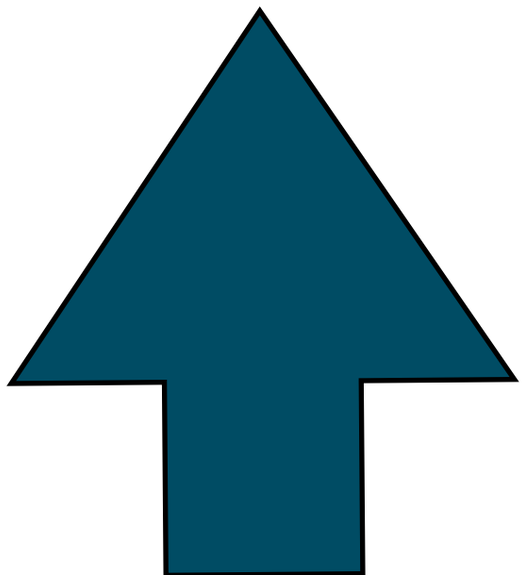
- There could be too many values to hold in memory
- We need <word_i,*> to be the first value we encounter
- Solution 1 b):
  - Keep <word_i,word_j> as the reduce key
  - Change the way Hadoop does its partitioning.

Word Co-occurrence: Solution 1

```
  /**

 * Partition based on the first part of the pair.
 */
public static class FirstCommaPartitioner extends Partitioner<Text, Text> {
    @Override
    public int getPartition(Text key, Text value, int numPartitions) {
        String[] s = key.toString().split(",");
        return (s[0].hashCode() & Integer.MAX_VALUE) % numPartitions;
    }
}
```

Word Co-occurrence: Solution 1

```
  /**
 * Partition based on the first part of the pair.
 */
public static class FirstCommaPartitioner extends Partitioner<Text, Text> {
    @Override
    public int getPartition(Text key, Text value, int numPartitions) {
        String[] s = key.toString().split(",");
        return (s[0].hashCode() & Integer.MAX_VALUE) % numPartitions;
    }
}
```



Removes the sign bit, if set

Word Co-occurrence: Solution 1

- Ok cool, but we still have the same problem.
  - The information is all in the same reducer, but we don't know the order
- But now, we have all the information we need in the reduce key!

Word Co-occurrence: Solution 1

- We can use a comparator to sort the keys we encounter in the reducer
    - See `KeyFieldBasedComparator`

```
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {}
```

Word Co-occurrence: Solution 1

- Now the order of key, value pairs will be as we need:

<Machine,*>:12000

<Machine, big>:50

<Machine, data>:200

<Machine, for>:1000

<Machine, learning>:10

...

$$P(\text{“big” in sentence} \mid \text{“Machine” started sentence}) = 50/12000$$

Word Co-occurrence: Solution 2

Word Co-occurrence: Solution 2

- The **Stripes** paradigm

Word Co-occurrence: Solution 2

- The **Stripes** paradigm
- For each sentence, output a key, record pair
  - E.g Map("Machine learning for big data") creates:

    <Machine>:<*:1,learning:1, for:1, big:1, data:1>

  - E.g Map("Machine parts are for machines") creates:

    <Machine>:<*:1,parts:1,are:1, for:1,machines:1>

Word Co-occurrence: Solution 2

- Reduce combines the records:
  - E.g Reduce for key <Machine> receives values:

    <*:1,learning:1, for:1, big:1, data:1>

    <*:1,parts:1,are:1, for:1,machines:1>

    - And merges them to create
    - <*:2,learning:1, for:2, big:1, data:1,parts:1,are:1,machines:1>

Word Co-occurrence: Solution 2

- This is nice because we have the * count already created
  - we just have to ensure it always occurs first in the record

Word Co-occurrence: Solution 2

Word Co-occurrence: Solution 2

- There is a really big (ha ha) problem with this solution
  - Can you see it?

Word Co-occurrence: Solution 2

- There is a really big (ha ha) problem with this solution
  - Can you see it?
- The value may become too large to fit in memory

Performance

- IMPORTANT
  - You may not have room for all reduce values in memory
    - In fact you should PLAN not to have memory for all values
    - Remember, small machines are much cheaper
      - you have a limited budget

Performance

Performance

- Which is faster, stripes vs pairs?

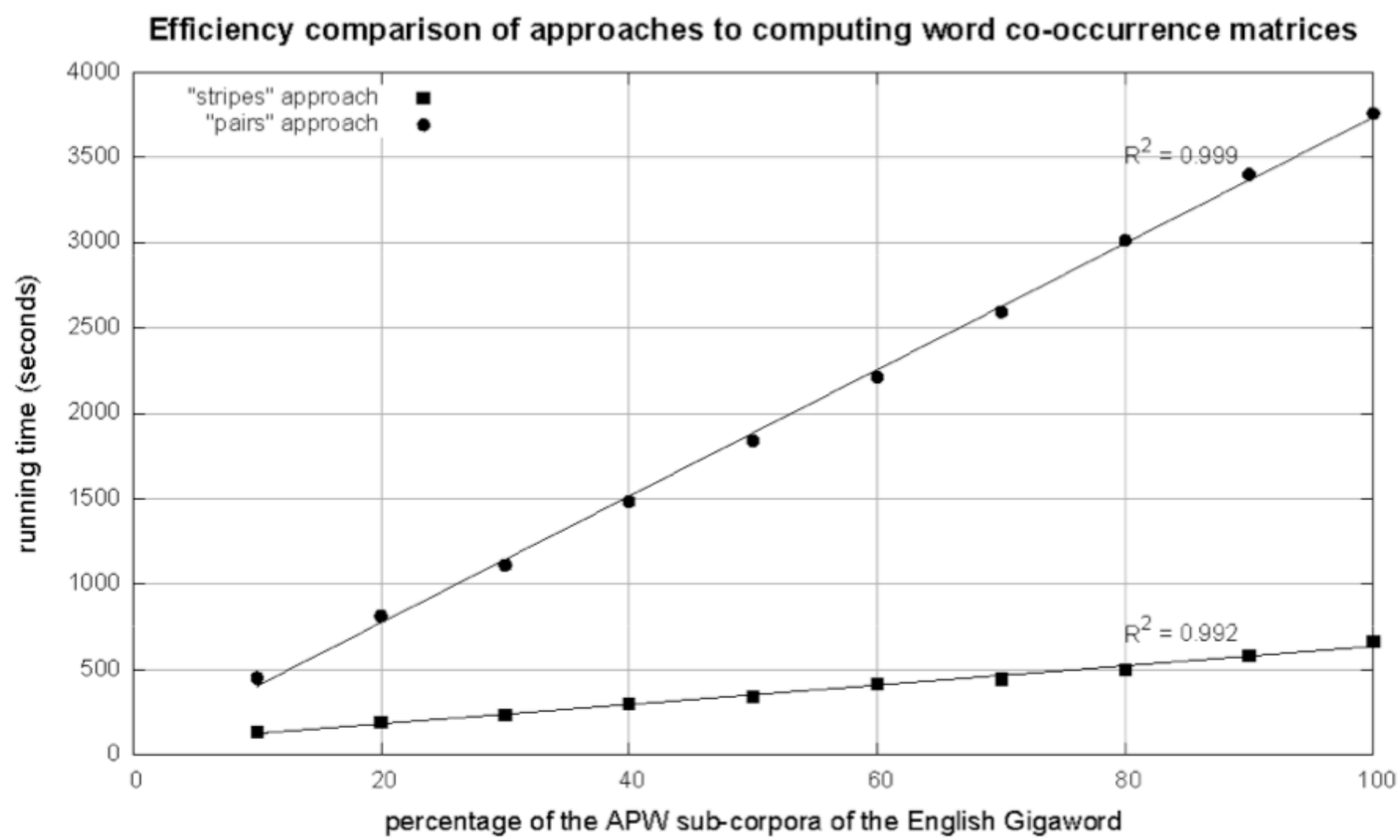
Performance

- Which is faster, stripes vs pairs?
  - Stripes has a bigger value per key

Performance

- Which is faster, stripes vs pairs?
  - Stripes has a bigger value per key
  - Pairs has more partition/sort overhead

Performance



Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

Conclusions

- Mapreduce
  - Can handle big data
  - Requires minimal code-writing

Tuesday

- On Tuesday
  - Mapreduce Tips
  - Shared memory programming

Done!

- Questions?

- I'll be having special office hours
  - Friday Feb 16, 10-11, outside GHC 8009
  - Tuesday Feb 22, 12-1 via Skype/Google hangout
  - Monday Feb 27, 10-11 outside GHC 8009