Big Data Part One: An Intro to MapReduce

Chris Jermaine

Rice University

15 Years Ago...

- Say you had a big data set, wanted platform to analyze it
- What is "big"?
  - Too large to fit in RAM of an expensive server machine

You Might Roll Your Own Software

- Costly, time consuming
  - A $10M software feature might eat up most of the IT budget for a single firm
- Requires expertise not always found in house
- Risky: high potential for failure

Or, You Might Buy a DB System

- Costs a LOT of money
- Performance often unpredictable, or just flat out poor
- Software insanely complicated to use correctly
- Software stack too big/deep, not possible to unbundle
  - If you are doing analysis, ACID not important
  - And yet, you pay for it (money, complexity, performance)
- Difficult to put un- or semi-structured data into an SQL DB

Plus, Many People Just Don't Like SQL

- People uncomfortable with declarative programming
  - We love it!
  - But user doesn't really know what's happening under the hood
  - Makes many programmers uncomfortable

- Also, not easy/natural to specify important computations
  - Especially data mining and machine learning
  - Not to mention HPC-style computations

By Early-Mid 2000's...

- The Internet companies (Google, Yahoo, etc.)...
  - ...had some of the largest databases in the world
  - But they never used classical SQL databases for webscale
- How'd they do it?
  - Many ways...
  - But paradigm with most widespread impact was MapReduce
  - First described in a 2004 academic paper, appeared in OSDI
  - Easy read! Do a search on "Google MapReduce paper"

What Is MapReduce?

- It is a simple data processing paradigm

- To process a data set:
  - You have two pieces of user-supplied code
  - A Map code
  - And a Reduce code

machines that don't share anything; just in the same network

- These are run in a huge shared-nothing compute cluster
  - Using three data processing phases
  - A Map phase
  - A Shuffle phase   moves data around
  - And a Reduce phase

First: What Is Shared-Nothing?

- Store/analyze data on a large number of commodity machines
  - Local, non-shared storage attached to each of them
  - Only link is via a LAN
  - Shared nothing refers to no sharing of RAM, storage
  - Note: NAS is common now, “pure” shared-nothing rarer
- Why good?
  - Inexpensive, built out of commodity components
  - Compute resources scales nearly linearly with money
  - Contrast to shared RAM machine with uniform memory access

MapReduce: The Map Phase

- Input data are stored in a huge file
  - Contains a simple list of pairs of type $(key1, value1)$

user defined function

- Have a UDF of the form $Map(key1, value1)$
  - outputs a list of pairs of the form $(key2, value2)$

- In the Map phase of the MapReduce computation
  - The Map function is called for every record in the input
  - Instances of Map run in parallel all over the cluster

Example: WordCount

- Large text corpus
- Want to count number of occurs of each word
- Ex output: ('The', 1832321), ('An', 1732432), etc.
- To power the Map phase:
  - ▷ MapReduce software automatically breaks corpus into large number of $(lineNo, text)$ pairs

Example Map Phase…

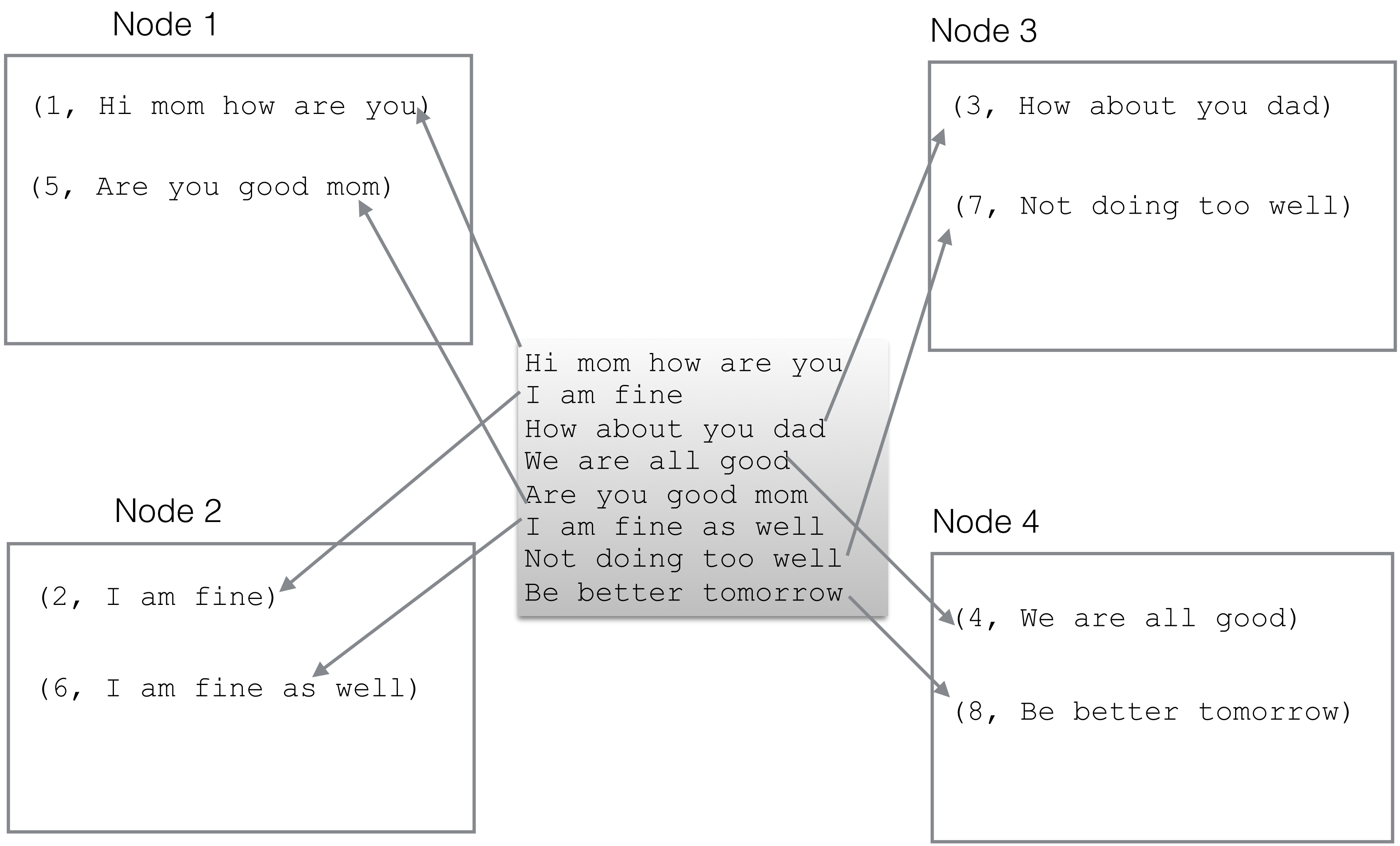
Node 1

Node 3

```
Hi mom how are you
I am fine
How about you
We are all good
Are you good
I am fine as well
Not doing too well
Be better tomorrow
```

Node 2

Node 4



Node 1
(1, Hi mom how are you)
(5, Are you good mom)
Node 3
(3, How about you dad)
(7, Not doing too well)
Hi mom how are you
I am fine
How about you dad
We are all good
Are you good mom
I am fine as well
Not doing too well
Be better tomorrow
Node 2
(2, I am fine)
(6, I am fine as well)
Node 4
(4, We are all good)
(8, Be better tomorrow)

Node 1

(1, Hi mom how are you)

Apply mapper

(5, Are you good mom)

Node 3

(3, How about you dad)

(7, Not doing too well)

```
Hi mom how are you
I am fine
How about you dad
We are all good
Are you good mom
I am fine as well
Not doing too well
Be better tomorrow
```

Node 2

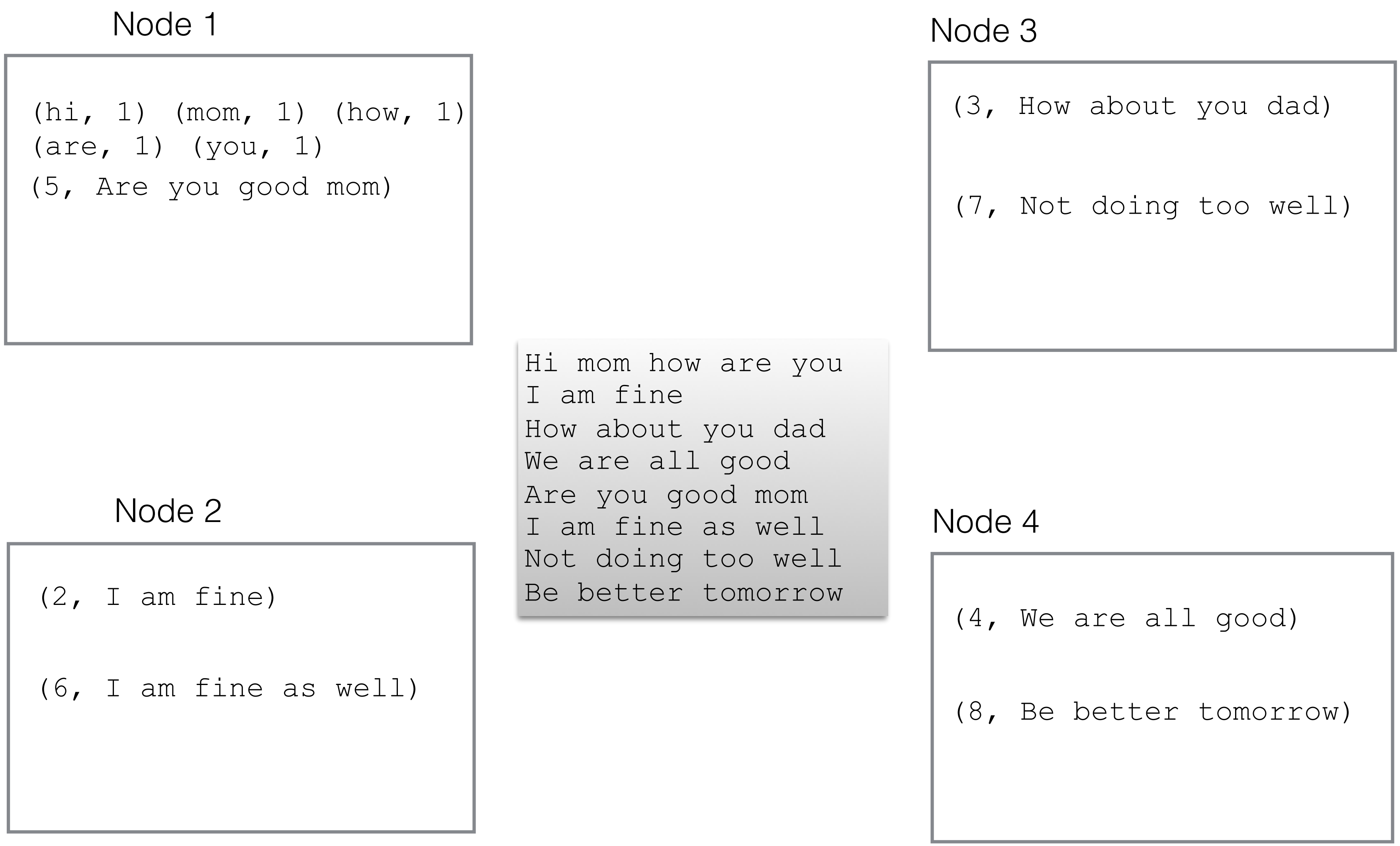
(2, I am fine)

(6, I am fine as well)

Node 4

(4, We are all good)

(8, Be better tomorrow)



Node 1
(hi, 1) (mom, 1) (how, 1)
(are, 1) (you, 1)
(5, Are you good mom)
Node 2
(2, I am fine)
(6, I am fine as well)
Hi mom how are you
I am fine
How about you dad
We are all good
Are you good mom
I am fine as well
Not doing too well
Be better tomorrow
Node 3
(3, How about you dad)
(7, Not doing too well)
Node 4
(4, We are all good)
(8, Be better tomorrow)

Node 1

(hi, 1) (mom, 1) (how, 1)
(are, 1) (you, 1)

(5, Are you good mom)

Apply mapper

Node 2

(2, I am fine)

(6, I am fine as well)

```
Hi mom how are you
I am fine
How about you dad
We are all good
Are you good mom
I am fine as well
Not doing too well
Be better tomorrow
```

Node 3

(3, How about you dad)

(7, Not doing too well)

Node 4

(4, We are all good)

(8, Be better tomorrow)

Node 1

(hi, 1) (mom, 1) (how, 1)
(are, 1) (you, 1)
(are, 1) (you, 1)
(good, 1) (mom, 1)

Node 3

(3, How about you dad)

(7, Not doing too well)

```
Hi mom how are you
I am fine
How about you dad
We are all good
Are you good mom
I am fine as well
Not doing too well
Be better tomorrow
```

Node 2

(2, I am fine)

(6, I am fine as well)

Node 4

(4, We are all good)

(8, Be better tomorrow)

Done in parallel all over the cluster…

Node 1

(hi, 1) (mom, 1) (how, 1)
(are, 1) (you, 1)
(are, 1) (you, 1)
(good, 1) (mom, 1)

Node 3

(how, 1) (about, 1)
(you, 1) (dad, 1)
(not, 1) (doing, 1)
(too, 1) (well, 1)

```
Hi mom how are you
I am fine
How about you dad
We are all good
Are you good mom
I am fine as well
Not doing too well
Be better tomorrow
```

Node 2

(i, 1) (am, 1) (fine, 1)
(i, 1) (am, 1) (fine, 1)
(as, 1) (well, 1)

Node 4

(we, 1) (are, 1)
(all, 1) (good, 1)
(be, 1) (better, 1)
(tomorrow, 1)

MapReduce: The Shuffle Phase

- Accepts all of the $(key2, value2)$ pairs from the Map phase
  - And it groups them together
- After grouping, all of the pairs
  - From all over the cluster having the same $key2$ value
  - Are merged into a single $(key2, list\langle value2\rangle)$ pair
- Called a "Shuffle"...
  - Because this is where a potential all-to-all data transfer happens

Example Shuffle Phase…

Node 1

(hi, 1) (mom, 1) **(how, 1)**
(are, 1) (you, 1)
(are, 1) (you, 1)
(good, 1) (mom, 1)

Node 3

(how, 1) (about, 1)
(you, 1) (dad, 1)
(not, 1) (doing, 1)
(too, 1) (well, 1)

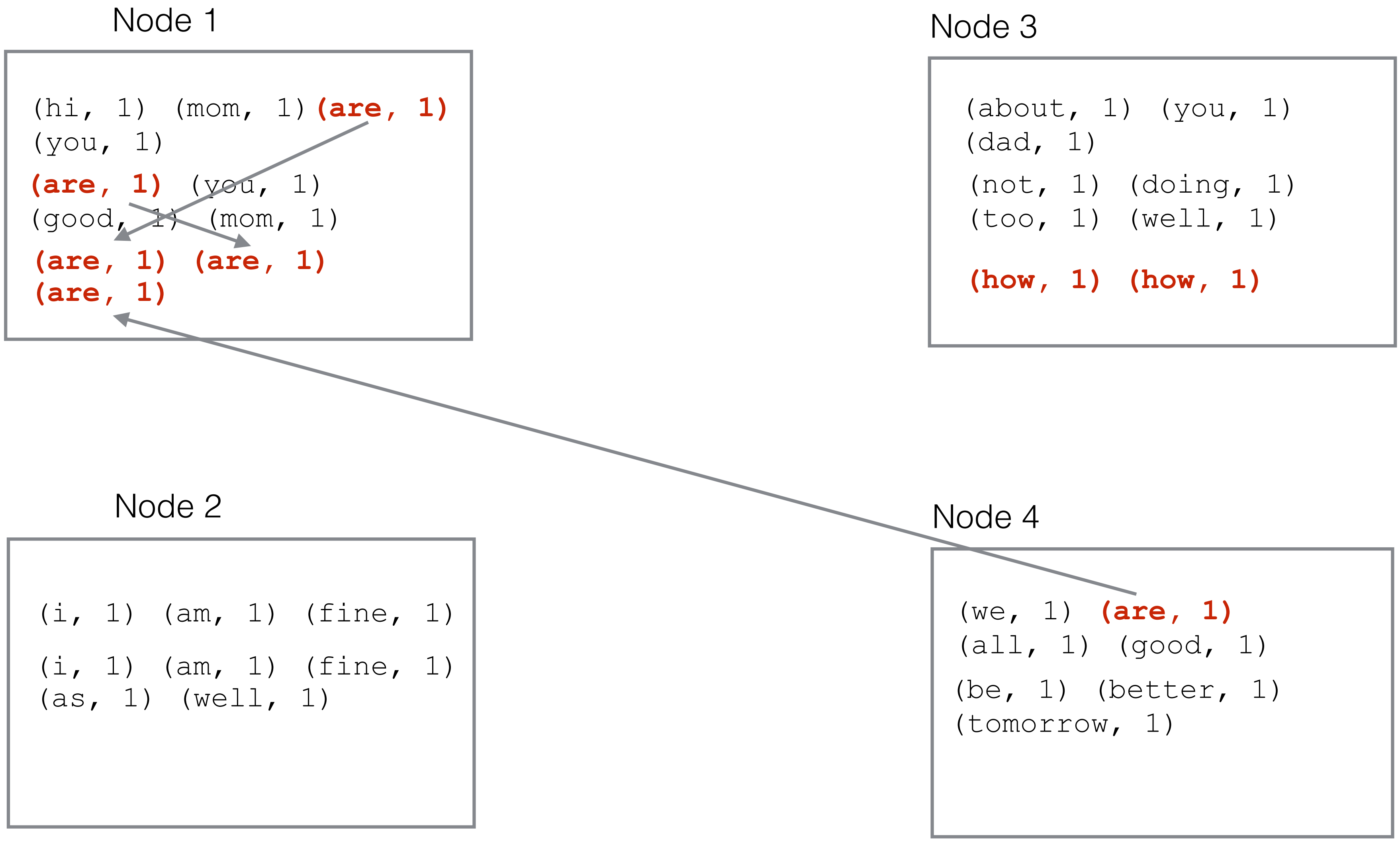
(how, 1) **(how, 1)**

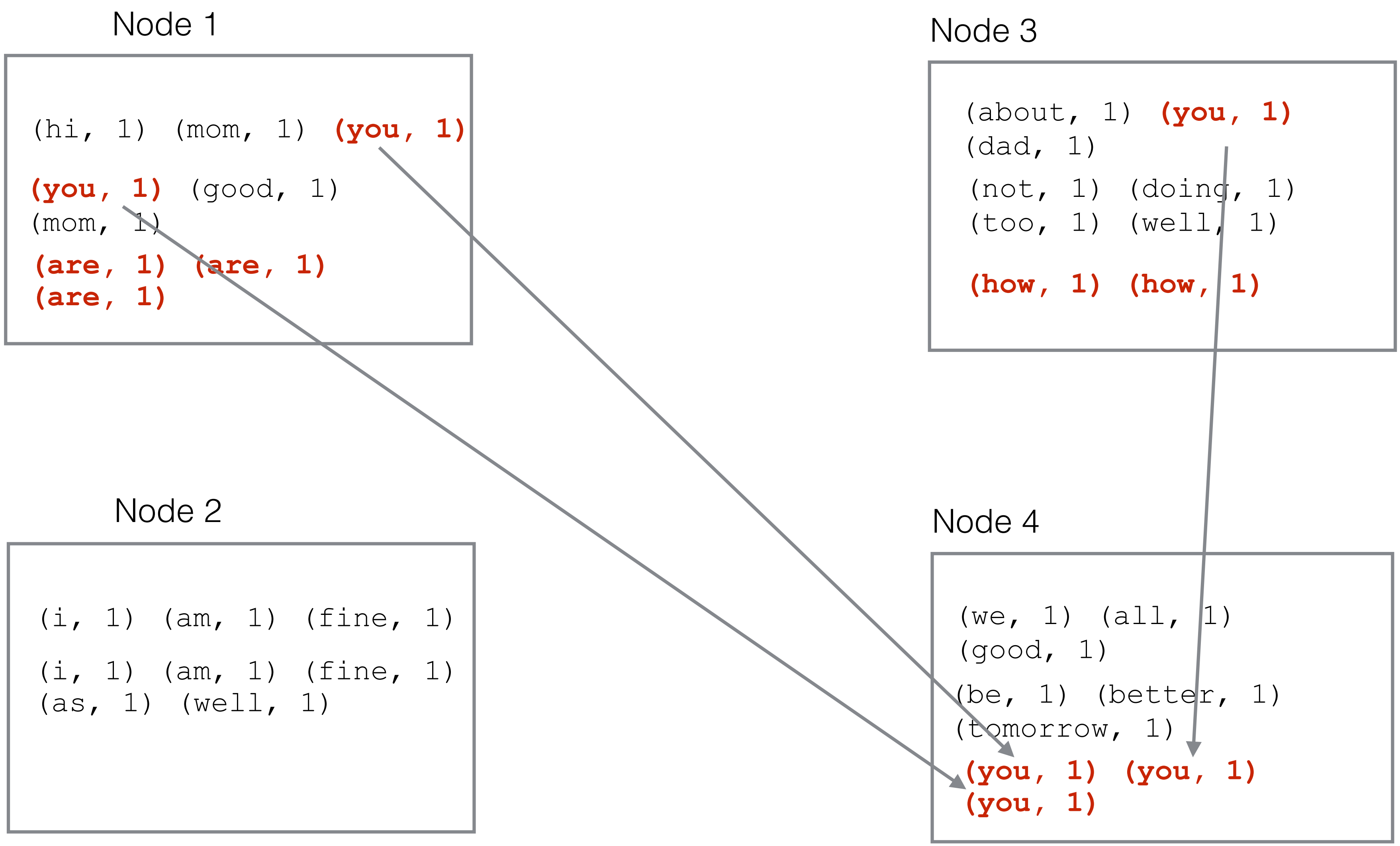
Node 2

(i, 1) (am, 1) (fine, 1)
(i, 1) (am, 1) (fine, 1)
(as, 1) (well, 1)

Node 4

(we, 1) (are, 1)
(all, 1) (good, 1)
(be, 1) (better, 1)
(tomorrow, 1)


Node 1
(hi, 1) (mom, 1) (are, 1)
(you, 1)
(are, 1) (you, 1)
(good, 1) (mom, 1)
(are, 1) (are, 1)
(are, 1)
Node 3
(about, 1) (you, 1)
(dad, 1)
(not, 1) (doing, 1)
(too, 1) (well, 1)
(how, 1) (how, 1)
Node 2
(i, 1) (am, 1) (fine, 1)
(i, 1) (am, 1) (fine, 1)
(as, 1) (well, 1)
Node 4
(we, 1) (are, 1)
(all, 1) (good, 1)
(be, 1) (better, 1)
(tomorrow, 1)



Node 1
(hi, 1) (mom, 1) (you, 1)
(you, 1) (good, 1)
(mom, 1)
(are, 1) (are, 1)
(are, 1)
Node 3
(about, 1) (you, 1)
(dad, 1)
(not, 1) (doing, 1)
(too, 1) (well, 1)
(how, 1) (how, 1)
Node 2
(i, 1) (am, 1) (fine, 1)
(i, 1) (am, 1) (fine, 1)
(as, 1) (well, 1)
Node 4
(we, 1) (all, 1)
(good, 1)
(be, 1) (better, 1)
(tomorrow, 1)
(you, 1) (you, 1)
(you, 1)

After all data have been xferred…

Node 1

```
(am, 1) (am, 1)
(as, 1)
(not, 1)
(good, 1) (good, 1)
(are, 1) (are, 1)
(are, 1)
```

Node 3

```
(better, 1)
(be, 1)
(about, 1)
(too, 1)
(all, 1)
(how, 1) (how, 1)
(hi, 1)
```

Node 2

```
(fine, 1) (fine, 1)
(doing, 1)
(well, 1) (well, 1)
(dad, 1)
(i, 1) (i, 1)
```

Node 4

```
(mom, 1) (mom, 1)
(we, 1)
(tomorrow, 1)
(you, 1) (you, 1)
(you, 1)
```

Arranged into (key, list) pairs…

Node 1

```
(am, <1, 1>)
(as, <1>)
(not, <1>)
(good, <1, 1>)
(are, <1, 1, 1>)
```

Node 3

```
(better, <1>)
(be, <1>)
(about, <1>)
(too, <1>)
(all, <1>)
(how, <1, 1>)
(hi, <1>)
```

Node 2

```
(fine, <1, 1>)
(doing, <1>)
(well, <1, 1>)
(dad, <1>)
(i, <1, 1>)
```

Node 4

```
(mom, <1, 1>)
(we, <1>)
(tomorrow, <1>)
(you, <1, 1, 1>)
```

MapReduce: The Reduce Phase

- Have a user-supplied function of the form
  - $Reduce(key2, list\langle value2 \rangle)$
  - Outputs a list of $value3$ objects
- In the Reduce phase of the MapReduce computation
  - $Reduce$ function is called for every $key2$ value output by the Shuffle
  - Instances of $Reduce$ run in parallel all over the compute cluster
  - The output of all of those instances is collected
  - Put in a (potentially) huge output file

Finally, Reduce Phase…

Node 1

```
(am, <1, 1>)
(as, <1>)
(not, <1>)
(good, <1, 1>)
(are, <1, 1, 1>)
```

Node 3

```
(better, <1>)
(be, <1>)
(about, <1>)
(too, <1>)
(all, <1>)
(how, <1, 1>)
(hi, <1>)
```

Node 2

```
(fine, <1, 1>)
(doing, <1>)
(well, <1, 1>)
(dad, <1>)
(i, <1, 1>)
```

Node 4

```
(mom, <1, 1>)
(we, <1>)
(tomorrow, <1>)
(you, <1, 1, 1>)
```

Node 1

```
(am, 3)
(as, 1)
(not, 1)
(good, 2)
(are, 3)
```

"am" should be "2"

Node 3

```
(better, 1)
(be, 1)
(about, 1)
(too, 1)
(all, 1)
(how, 2)
(hi, 1)
```

Node 2

```
(fine,2)
(doing, 1)
(well, 2)
(dad, 1)
(i, 2)
```

Node 4

```
(mom, 2)
(we, 1)
(tomorrow,1)
(you, 3)
```

MapReduce Is a Compute Paradigm

- It is not a data storage paradigm
  - ▷ But any MapReduce system
  - ▷ Must read/write data from some storage system
- So MapReduce strongly linked with the idea of a distributed file system (DFS)
  - ▷ Allows data to be stored/accessed across machines in a network
  - ▷ Abstracts away differences between local and remote data
  - ▷ Same API to read/write data
  - ▷ No matter where data is located in the network

Distributed File Systems for MR

- DFSs have been around for a long time
  - ▷ First widely used DFS was Sun's NFS, first introduced in 1985
- Unlike classical DFS...
  - ▷ MapReduce DFS sits on top of each machine's OS
  - ▷ Lives in "user space"
  - ▷ The OS is not aware of the DFS
  - ▷ You can't mount it anywhere
- Why on top of, not in the OS?
  - ▷ Heterogeneity no problem
  - ▷ Easily portable (JVM)

MapReduce / Hadoop getting less popular, but the DFS is popular

- Even as MapReduce becomes less popular, MR DFS lives on!

MapReduce vs. HPC

- MapReduce pros
  - MUCH lower programmer burden than HPC
  - No synchronization, parallelism implicit
  - Data and task placement automatic
  - Built-in fault tolerance
  - Works with (almost!) arbitrarily-sized data
  - Out-of-core execution is no problem

MapReduce vs. HPC

- MapReduce cons
  - Standard softwares are JVM-based
  - Not suitable for communication-heavy tasks...
  - ...only communication is via the shuffle
  - Assumes BIG data... always reads/writes data from DFS

Questions?

Big Data Two: Beyond MapReduce

Chris Jermaine

Rice University

Most Popular "Pure" MapReduce Software

- Is called Hadoop
  - Runs on JVM (like most Big Data software)
  - Includes MapReduce functionality
  - Plus the Hadoop distributed file system (HDFS)
- Hadoop popularity peaked around 2015...
- Has been declining since then
- Why? Were several issues...

Hadoop MR Word Count Java Code

```java
import java.util.*;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;

public class WordCount {

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

    // if we got the wrong number of args, then exit
    if (args.length != 4 || !args[0].equals ("-r")) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // Get the default configuration object
    Configuration conf = new Configuration ();

    // now create the MapReduce job
    Job job = new Job (conf);
    job.setJobName ("WordCount");

    // we'll output text/int pairs (since we have words as keys and counts as values)
    job.setMapOutputKeyClass (Text.class);
    job.setMapOutputValueClass (IntWritable.class);

    // again we'll output text/int pairs (since we have words as keys and counts as values)
    job.setOutputKeyClass (Text.class);
    job.setOutputValueClass (IntWritable.class);

    // tell Hadoop the mapper and the reducer to use
    job.setMapperClass (WordCountMapper.class);
    job.setCombinerClass (WordCountReducer.class);
    job.setReducerClass (WordCountReducer.class);

    // we'll be reading in a text file, so we can use Hadoop's built-in TextInputFormat
    job.setInputFormatClass (TextInputFormat.class);

    // we can use Hadoop's built-in TextOutputFormat for writing out the output text file
    job.setOutputFormatClass (TextOutputFormat.class);

    // set the input and output paths
    TextInputFormat.setInputPaths (job, args[2]);
    TextOutputFormat.setOutputPath (job, new Path (args[3]));

    // set the number of reduce paths
    try {
      job.setNumReduceTasks (Integer.parseInt (args[1]));
    } catch (Exception e) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // force the mappers to handle one megabyte of input data each
    TextInputFormat.setMinInputSplitSize (job, 1024 * 1024);
    TextInputFormat.setMaxInputSplitSize (job, 1024 * 1024);

    // this tells Hadoop to ship around the jar file containing "WordCount.class" to all of
the different
    // nodes so that they can run the job
    job.setJarByClass(WordCount.class);

    // submit the job and wait for it to complete!
    int exitCode = job.waitForCompletion (true) ? 0 : 1;
    return exitCode;
```

Not pretty!
Programmer burden too high...

Many Felt MapReduce Too Slow

- Data reread from DFS for each MR job
- Bad for iterative data processing
  - Example: most machine learning uses gradient descent
  - Need to make 100's or 1000's of passes over data
  - Re-evaluating gradient at various points

MR API is Too Restrictive

- Can only do Map
- Or MapReduce
- Everything else in terms of those operations
- Unless you cheat
  - Ex: Mappers/Reducers talk to each other using sockets
  - But then why not just go with C/MPI?

Result: MapReduce Used Less and Less

- Are now an entire ecosystem of alternative softwares
  - Spark, Flink, etc.
- For use in both streaming and batch processing applications
- Generally oriented more towards in-memory computing
- Have far more expressive APIs
- We will focus on Spark

Apache Spark

- #1 Hadoop MapReduce killer
- What is Spark?
  - Platform for efficient distributed data analytics
- Runs on the JVM
- Written in Scala
  - But has bindings for Java, Scala, Python, R
  - Python nice for data analytics (NumPy, SciPy)... will focus there
- Doesn't do storage
  - Focus exclusively on compute
  - Commonly used with HDFS, S3, HBase, etc.

RDDs

- Basic abstraction: Resilient Distributed Data Set (RDD)
- RDD is a data set buffered in RAM by Spark
  - Distributed across machines in cluster
  - To create and load an RDD (in Python shell):

```
myRDD = sc.textFile (someFileName) # sc is the Spark context
# or else...
data = [1, 2, 3, 4, 5]
myRDD = sc.parallelize (data) # or
myRDD = sc.parallelize (range (20000)) # or...
```

Computations: Series of Xforms Over RDDs

- Example: word count

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split("␣"))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
     return aggCounts.top (200, key=lamda p: p[1])
```

- What transforms do we see here?
  - ▷ flatMap, map, reduceByKey, top

- Let's go through them

- But first, quick review of lambdas...
  - ▷ Fundamental to programming in Spark

What's a Lambda?

important to Spark

- Basically, a function that that we can pass like a variable
- Key ability: can "capture" its surroundings at creation

```
def addTwelveToResult (myLambda):
     return myLambda (3) + 12

a = 23
aCoolLambda = lambda x : x + a
addTwelveToResult (aCoolLambda) # prints 38

a = 45
addTwelveToResult (aCoolLambda) # prints ???
```

Lambdas and Comprehensions

- Lambdas can return many items

```
def sumThem (myLambda):
     tot = 0
     for a in myLambda ():
          tot = tot + a
     return tot

x = np.array([1, 2, 3, 4, 5])
iter = lambda : (j for j in x)
sumThem (iter) # prints 15
```

Spark Operation: flatMap ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split("␣"))
```

split every time you see a space

- Process every data item in the RDD
- Apply lambda to it
- Lambda argument to return zero or more results
- Each result added into resulting RDD

creates a new RDD list of words

Spark Operation: map ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split("␣"))
     counts = lines.map (lambda word: (word, 1))
```

- Process every data item in the RDD
- Apply lambda to it
- The lambda must return exactly one result

could have used flatmap instead of map, but there is some performance benefit to using map which states there is 1 result (vs. zero or more)

map expects to produce single output = list?
flatmap provide a list and create a new RDD?

Spark Operation: reduceByKey ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split("␣"))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
```

- Data must be $(Key, Value)$ pairs
- Shuffle so that all (K, V) pairs with same K on same machine
- Organize into $(K, (V_1, V_2, ..., V_n))$ pairs
- Use the lambda to "reduce" the list to a single value

Spark Operation: top ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split("␣"))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
     retrun aggCounts.top (200, key=lamda p: p[1])
```

- Data must be $(Key, Value)$ pairs
- Takes two params... first is number to return
- Second (optional): lambda to use to obtain key for comparison
- Note: top collects an RDD, moving from cloud to local

top is a collection operation

- So result is not an RDD

top returns some sort of a list data structure

An Important Note

- Spark uses lazy evaluation...
- If I run this code:

```
textFile = sc.textFile (fileName)
lines = textFile.flatMap (lambda line: line.split("␣"))
counts = lines.map (lambda word: (word, 1))
aggCounts = counts.reduceByKey (lambda a, b: a + b)
```

- Nothing happens! (Other than Spark remembers the ops)
  - ▷ Spark does not execute until an attempt made to collect an RDD
  - ▷ When we hit top(), then all of these are executed
- Why do this?
  - ▷ By waiting until last possible second, opportunities for "pipelining" exploited
  - ▷ Only ops that require a shuffle can't be pipelined

Some Other, More Advanced Ops

- groupByKey (), join (), reduce (), aggregate ()

groupByKey ()

- Data must be $(Key, Value)$ pairs
- Shuffle so that all (K, V) pairs with same K on same machine
- Organize into $(K, \langle V_1, V_2, ..., V_n \rangle)$ pairs
- Store each list as a `ResultIterable` for future processing
- Like `reduceByKey ()` but without the reduce

join ()

- Given two data sets $rddOne$, $rddTwo$ of $(Key, Value)$ pairs
- We join them using:

```
rddOne.join (rddTwo)
```

- Returns $(K, (V_1, V_2))$ pairs
- Constructed from all (K_1, V_1) from $rddOne$, (K_2, V_2) from $rddTwo$, where $K_1 = K_2$

join () (Continued)

- Example:
  - $rddOne$ is $\{(red, 9), (blue, 7), (red, 12), (green, 4)\}$
  - $rddTwo$ is $\{(blue, up), (green, down), (green, behind)\}$
  - Result of join is $\{(blue, (7, up)), (green, (4, down)), (green, (4, behind))\}$

- Can blow up RDD size if join is many-to-many

- Requires expensive shuffle!

Why Do We Join?

- Allows "communication"

- Imagine fluid simulation
  - ▷ State stored in RDD of $(gridCell, (xPos, yPos, state))$ pairs

- Join this RDD with itself
  - ▷ To get $(gridCell, ((xPos_1, yPos_1, state_1), (xPos_2, yPos_2, state_2)))$ pairs
  - ▷ Then map () to get $((xPos_1, yPos_1, state_1), (xPos_2, yPos_2, state_2))$ pairs
  - ▷ groupByKey () to get $(xPos_1, yPos_1, state_1)$ with all states in same grid cell
  - ▷ Finally, evoke a lambda to update $(xPos_1, yPos_1, state_1)$ based on neighbors

reduce ()

- Unlike past operations, this is not a transform from RDD to RDD
  - This is like `top ()`, moves result back to Python

- Repeatedly apples a lambda to each item in RDD to get single result

```
>>> myData = sc.parallelize (range(20000))
>>> myData.reduce (lambda a, b: a + b)
199990000
```

aggregate ()

- With `reduce ()` you aggregate directly, can be restrictive...
- Example: RDD is $\{(red, 9), (blue, 7), (red, 12), (green, 4)\}$
  - ▷ Want dictionary where value is sum for each unique color
  - ▷ Cannot use `reduce ()`
  - ▷ It only "sums" up the items in the input RDD directly
  - ▷ Two inputs and output must be the same type
  - ▷ How do I get the desired output dictionary by defining $+$ in $(((red, 9) + (blue, 7)) + (red, 12)) + (green, 4)$?

aggregate () (Continued)

- Data must be $(Key, Value)$ pairs
- Organize into $(K, \langle V_1, V_2, ..., V_n \rangle)$ pairs
- Then aggregate the list, like `reduce ()`
- `aggregate ()` takes three args
  - The "zero" to init the aggregation
  - Lambda that takes X_1, X_2 and aggs them, where X_1 already aggregated, X_2 not
  - Lambda that takes X_1, X_2 and aggs them, where both aggregated

aggregate () (Example)

```
def add (dict, tuple):
  result = {}
  for key in dict:
    result[key] = dict[key]
  if (tuple[0] in result):
    result[tuple[0]] += tuple[1]
  else:
    result[tuple[0]] = tuple[1]
  return result
```

aggregate () (Example)

```
def combine (dict1, dict2):
  result = {}
  for key in dict1:
    result[key] = dict1[key]
  for key in dict2:
    if (key in result):
      result[key] += dict2[key]
    else:
      result[key] = dict2[key]
  return result
```

aggregate () (Example)

```
>>> myRdd = sc.parallelize ([('red', 9), ('blue', 7),
... ('red', 12), ('green', 4)])
>>> myRdd.aggregate ({}, lambda x, y: add (x, y),
... lambda x, y: combine (x, y))

{'blue': 7, 'green': 4, 'red': 21}
```

Closing Thoughts

- When is Spark/MapReduce a better option than HPC?
  - ▷ When your pipeline is heavily data-oriented
  - ▷ Or when your compute is (relatively) loosely coupled

- Key benefits compared to HPC
  - ▷ Built in fault tolerance
  - ▷ Better support for BIG data
  - ▷ Much higher programmer productivity

- Will continue to take market share from HPC
  - ▷ You see academic papers with both MPI, Spark implementations
  - ▷ But not everything can move to Spark

Questions?