Lambda Architecture

CSCI 5828: Foundations of Software Engineering
Lecture 29 — 12/09/2014

Goals

- Cover the material in Chapter 8 of the Concurrency Textbook
  - The Lambda Architecture
    - Batch Layer
      - MapReduce (e.g. Hadoop, Spark)
    - Speed Layer
      - Stream Processing (e.g. Storm, Spark Streaming)

Lambda Architecture (I)

- The Lambda Architecture refers to an approach for performing “big data” processing developed by Nathan Marz
  - from his experiences at BackType and Twitter
- Everyone has their own definition of “big”
  - gigabytes and terabytes for small research teams and companies
  - terabytes and petabytes for medium and large organizations
    - Google and Microsoft have petabytes of map data
  - exabytes for truly data-intensive organizations
    - Facebook reports having to store 500 TB of new information PER DAY
  - and zettabytes and yottabytes may be in our future some day!

Lambda Architecture (II)

- Everything changes when working at scale
  - many of your assumptions fall over
  - many of the techniques that you are comfortable suddenly reveal that they are not scalable
    - I enjoy giving students their first 60 GB file to work on
- In my research on Project EPIC, we have data sets that are hundreds of gigabytes in size
  - you can’t bring the entire set into memory (at least not on a single machine)
  - you can’t easily store our data sets accumulated over the past five years on a single machine

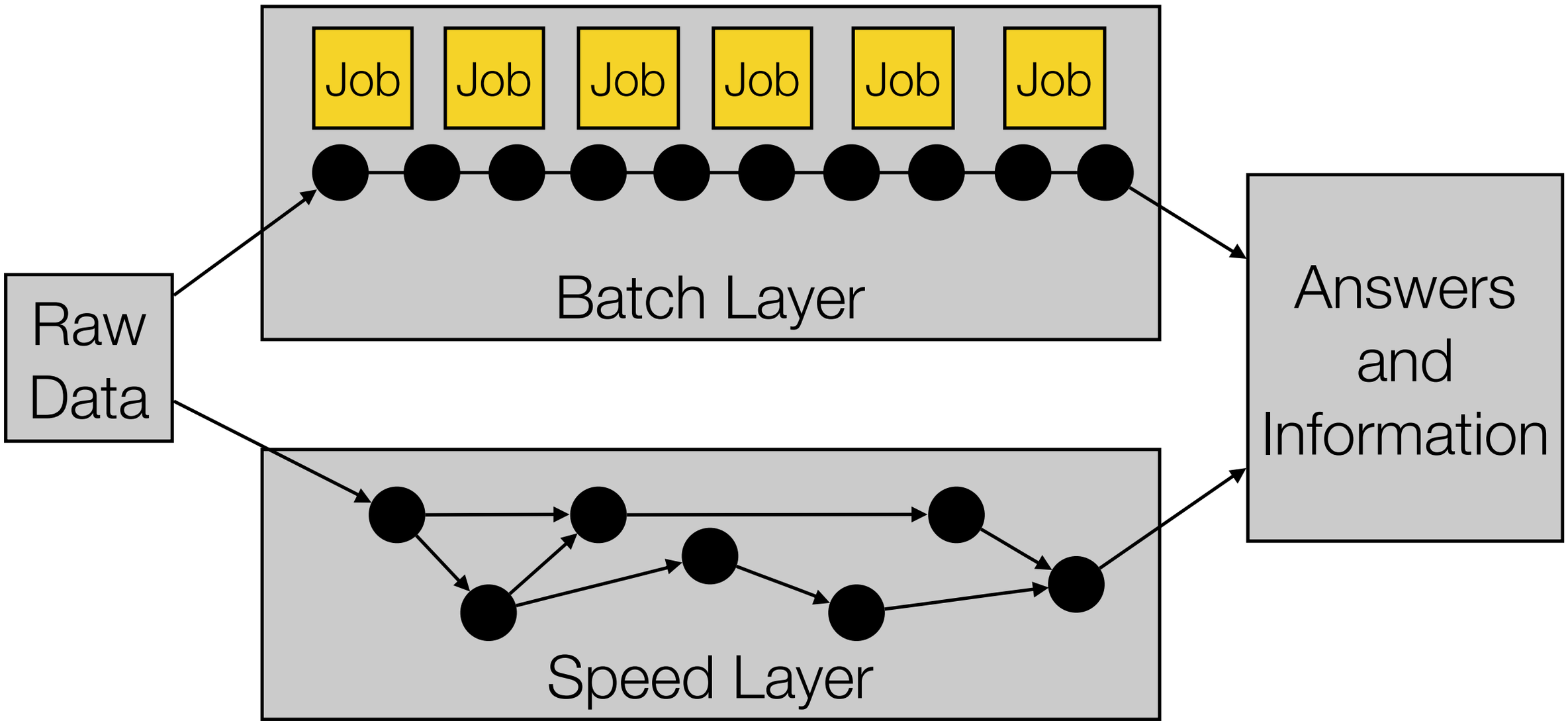
Lambda Architecture (III)

- In a typical data-intensive system, the goal is to
  - while true
    - collect and/or generate raw data
    - store that data in an effective and efficient way
    - **process it to answer questions of some form**
    - use the answers to determine what new questions you have
    - use those questions to determine what new data you need
- In our coverage of the Lambda Architecture, we will be looking primarily at technologies that aid the "process" stage above
  - Additional details can be found in Marz's book on Big Data

Lambda Architecture (IV)

- In processing data to answer questions, the lambda architecture advocates a two-prong approach
  - For large data sets, where the processing can take a significant amount of time (hours) => the batch layer using techniques like MapReduce
  - For more recent generated data, process it as it arrives => the speed layer using techniques like stream processing
- In both cases, these technologies will make use of clusters of machines working together to perform the analysis
  - in this way
    - data is distributed/replicated across machines
    - AND computation is distributed across machines and occurs in parallel

Lambda Architecture (V)



Lambda Architecture (VI)

- For the batch layer, we will make use of techniques that can process large sets of data using “batch jobs”
  - MapReduce, as implemented by Hadoop, has been the 900lb gorilla in this space for a long time
    - it is now being challenged by other implementations (such as Spark)
- For the speed layer, we will make use of techniques that can process streaming data quickly
  - The exemplar in this space is Storm, a streaming technology developed at Twitter
    - Other techniques useful in this space include message queueing systems like RabbitMQ and ActiveMQ

Map Reduce

- The functions **map**, **reduce**, and **filter** have cropped up a lot this semester
  - They represent a fundamental approach to processing data that (via Hadoop) has been shown to apply to extremely large data sets
- **map**: given an array and a function, create a new array that
  - for each element
    - contains the results of the function
      - applied to the corresponding element of the original array
- **filter**: given an array and a boolean function, create a new array that
  - consists of all elements of the original for which the function returns true
- **reduce:** given an array, an initial value, and a binary function
  - return a single value that is the accumulation of the initial value and the elements of the original array as computed by the function

Examples: map, filter, and reduce

- In the subsequent examples, I present examples of map, filter, and reduce that apply to arrays of integers
  - With map, the supplied function has signature: `Int -> Int`
  - With filter, the supplied function has signature: `Int -> Bool`
  - With reduce, the supplied function has signature: `(Int, Int) -> Int`
- But the technique is generic with respect to type
  - For an array of elements of type T
    - the supplied function for map has signature: `T -> U`
    - the supplied function for filter has signature: `T -> Bool`
    - the supplied function for reduce has signature: `(R, T) -> R`

Example: map (I)

```
function map(values, f) {
  var results = [];
  for (var i = 0; i < values.length; i++) {
    results.push(f(values[i]));
  }
  return results;
}
```

- Here's a Javascript implementation of map
  - Create a new array
  - Apply the function f to each element of the input array (values)
    - and append (push in Javascript) the result to the output array

Example: map (II)

```
function triple(x) {
  return x * 3;
}

var data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

console.log(map(data, triple))
```

- Here's an example of using our Javascript version of map
  - Define a function that takes an integer and produces an integer
  - Create an array of integers
  - Call map passing the input array and function; print the result

Example: filter (I)

```
function filter(values, f) {
  var results = [];
  for (var i = 0; i < values.length; i++) {
    if (f(values[i])) {
      results.push(values[i]);
    }
  }
  return results;
}
```

- Here's a Javascript implementation of filter
  - Create a new array
  - Apply the function f to each element of the input array (values)
    - If true, append the element to the output array

Example: filter (II)

```
function isEven(x) {
  return (x % 2) == 0;
}

var data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

console.log(filter(data, isEven))
```

- Here's an example of using our Javascript version of filter
  - Define a function that takes an integer and produces a boolean
  - Create an array of integers
  - Call filter passing the input array and function; print the result

Example: reduce (I)

```
function reduce(values, initial, f) {
  var result = initial;
  for (var i = 0; i < values.length; i++) {
    result = f(result, values[i]);
  }
  return result;
}
```

- Here's a Javascript implementation of reduce
  - Set result equal to the initial value
  - Loop through the input array
    - Update result to be the output of the function applied to result and a[i]
  - Return the final result

Example: reduce (II)

```
function sum(total, value) {
  return total + value;
}

var data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

console.log(reduce(data, 0, sum))
```

- You can even combine them!
  - `console.log(reduce(filter(map(data, triple), isEven), 0, sum))`

Reduce is very powerful

- In some languages, you can implement map and filter using reduce

```
- func rmap<T, U>(xs: [T], f: T -> U) -> [U] {
    - return reduce(xs, []) { result, x in result + [f(x)] }
- }

- func rfilter<T>(xs: [T], check: T -> Bool) -> [T] {
    - return reduce(xs, []) {
        - result, x in return check(x) ? result + [x] : result }
- }
```

- This is an example written in Swift, a new language created by Apple

MapReduce and Hadoop (I)

- Hadoop applies these functions to data at large scale
  - You can have thousands of machines in your cluster
  - You can have petabytes of data
  - The Hadoop file system will replicate your data across the nodes
    - The Hadoop run-time will take your set of map and reduce jobs and distribute across the cluster
      - It will manage the final reduce process and ensure that all results end up in the output file that you specify
- It does all of this and will complete jobs **even if worker nodes go down taking out partially completed tasks**

MapReduce and Hadoop (II)

- Since I showed that map and filter are special cases of reduce, the overarching framework could have been called ReduceReduce
    - but that doesn't sound as good
- But that's just to say that you should be confident that the MapReduce framework provides you with a sufficient range of functionality to perform a wide range of analysis and data manipulation tasks

MapReduce and Hadoop (III)

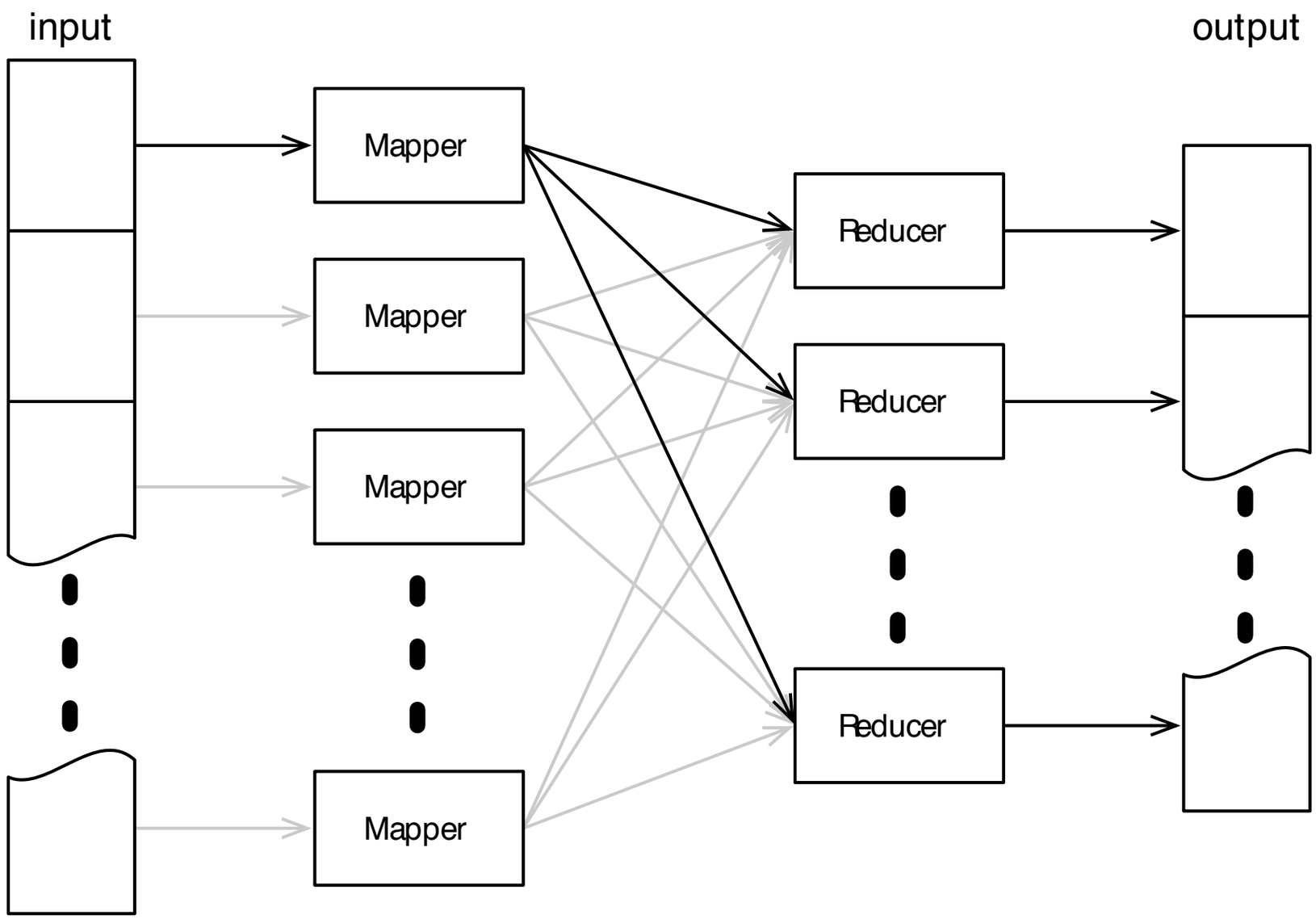
- Hadoop's basic mode of operation is transformation
  - A function (either map or reduce) will receive a document of key-value pairs
    - The result is a set of new documents with key-value pairs
  - In all cases, you are transforming from one set of documents to another
    - Or more generically, one type to another type
- Depending on your algorithm, one input document can produce multiple output documents

MapReduce and Hadoop (IV)

- The difference between map and reduce in Hadoop
  - map functions will receive a single document to transform
    - As I indicated, it will then produce zero or more output documents
      - each with a particular key; the key does not have to be unique
  - reduce functions will receive
    - a key and a set of documents that all had that key
    - It will then typically produce a single document with that key
      - in which all of the documents have had their values combined in some way
- So, if a map phase generates millions of documents across 26 different keys
  - then its reduce phase may end producing 26 final documents

Hadoop Conceptual Structure

Image from our concurrency text book



Hadoop guarantees that all mapper outputs associated with the same key will go to the same reducer

Shuffle Phase

This is called the shuffle phase, while it routes documents to reducers.

Powerful but at a cost

- MapReduce and Hadoop are powerful but they come at a cost
  - latency
- These jobs are NOT quick
  - It takes a lot of time to “spin up” a Hadoop job
  - If your analysis requires multiple MapReduce jobs, you have to “take the hit” of that latency across all such jobs
    - Spark tries to fix exactly this issue with its notion of an RDD
  - But if you have to apply a single algorithm across terabytes or petabytes of data, it will be hard to beat Hadoop once the price of that overhead is paid

Word Count in Hadoop (I)

- The book presents an example of using Hadoop to count the words in a document;
    - (as I mentioned earlier in the semester, this is the “hello world” example of big data)

Word Count in Hadoop (II)

- The high-level design is this:
  - Input document is lines of text
    - Hadoop will split the document into lines and send each line to a mapper
  - The mapper receives an individual line and splits it into words
    - For each word, it creates an output document: (word, 1)
      - e.g. (“you”, 1), (“shall”, 1), (“not”, 1), (“pass”, 1)
- These output documents get shuffled and are passed to the reducer like this: (word, [Int]); e.g. (“shazam!”, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
- The reducer takes this as input and produces (word, <sum>)
  - e.g. (“shazam!”, 10)

Word Count in Hadoop (III)

- There may be multiple phases of reduce depending on how many nodes are in your cluster
  - Depending on the implementation of MapReduce
    - Each node may do their reduce phase first and then the implementation would need to combine (i.e. reduce) the output documents on each node with each other to produce the final output
- OR
  - During the shuffle phase, we make sure that all documents with the same key go to the same reducer, even if that means sending the output document of the map phase on one machine as input to the reduce phase on a second machine

Word Count in Hadoop (IV)

- The code for our map phase looks like this

```
Line 1 public static class Map extends Mapper<Object, Text, Text, IntWritable> {
     -   private final static IntWritable one = new IntWritable(1);
     -
     -   public void map(Object key, Text value, Context context)
     5     throws IOException, InterruptedException {
     -
     -     String line = value.toString();
     -     Iterable<String> words = new Words(line);
     -     for (String word: words)
    10       context.write(new Text(word), one);
     -   }
     - }
```

Word Count in Hadoop (V)

- The code for our reduce phase looks like this

```
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));
  }
}
```

Word Count in Hadoop (VI)

- The main program looks like this

```
Line 1  public class WordCount extends Configured implements Tool {
-
-     public int run(String[] args) throws Exception {
-       Configuration conf = getConf();
5       Job job = Job.getInstance(conf, "wordcount");
-       job.setJarByClass(WordCount.class);
-       job.setMapperClass(Map.class);
-       job.setReducerClass(Reduce.class);
-       job.setOutputKeyClass(Text.class);
10      job.setOutputValueClass(IntWritable.class);
-       FileInputFormat.addInputPath(job, new Path(args[0]));
-       FileOutputFormat.setOutputPath(job, new Path(args[1]));
-       boolean success = job.waitForCompletion(true);
-       return success ? 0 : 1;
15    }
-
-     public static void main(String[] args) throws Exception {
-       int res = ToolRunner.run(new Configuration(), new WordCount(), args);
-       System.exit(res);
20    }
- }
```

MapReduce is a Generic Concept

- Many different technologies can provide implementations of map and reduce
  - We've already seen this with respect to functional programming languages
- But, MapReduce functionality (as in the Hadoop context) is starting to appear in many different tools
  - similar to the way that support for manipulating and searching text via regular expressions appears in many different editors
- In particular, CouchDB and MongoDB provide MapReduce functionality
  - I used MongoDB's MapReduce functionality to find the unique users of a Twitter data set
    - the result of the calculation is that each output document represents a unique user and contains useful information about that user

Twitter Example (I)

- In the unique users example, the high level design is the following
  - The input data set is a set of tweets stored in MongoDB
    - Each tweet is stored as a JSON document of attribute-value pairs
  - The mapper produces an output document with the following fields
    - names => an array of screen names for this user
    - tweets => an array of tweet ids for this user
    - name_count => number of names
    - tweet_count => number of tweets
- That document has as its key, the unique user id for that user

Twitter Example (II)

- The reducer takes a set of mapper-produced documents that all have the same key (i.e. user id) and
  - Combines all screen names and tweets and updates the counts as appropriate
- DEMO

Summary

- Introduced the first part of the Lambda Architecture
  - Covered MapReduce
    - And showed examples in Hadoop and MongoDB

Coming Up Next

- Lecture 30: Lambda Architecture, Part Two