# The Seductions of Scala - Applications

Dean Wampler

dean@concurrentthought.com
@deanwampler
polyglotprogramming.com/talks

April 26, 2013



Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

The online version contains more material. You can also find this talk and the code used for many of the examples at github.com/deanwampler/Presentations/tree/master/SeductionsOfScala

Copyright © 2010-2013, Dean Wampler. Some Rights Reserved - All use of the photographs and image backgrounds are by written permission only. The content is free to reuse, but attribution is requested.

http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode

Let's examine two popular Scala toolkits used to implement realworld applications.

- Akka: Actors for highlyconcurrent applications.
- Scalding: Big data the functional way.

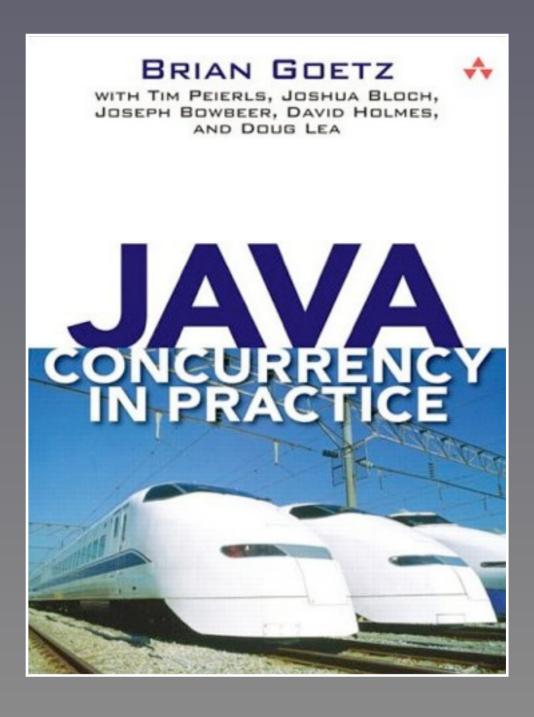


Sunday, July 7, 13

FP is going mainstream because it is the best way to write robust concurrent software. Here's an example...

When you share mutable state...

Hic sunt dracones (Here be dragons)



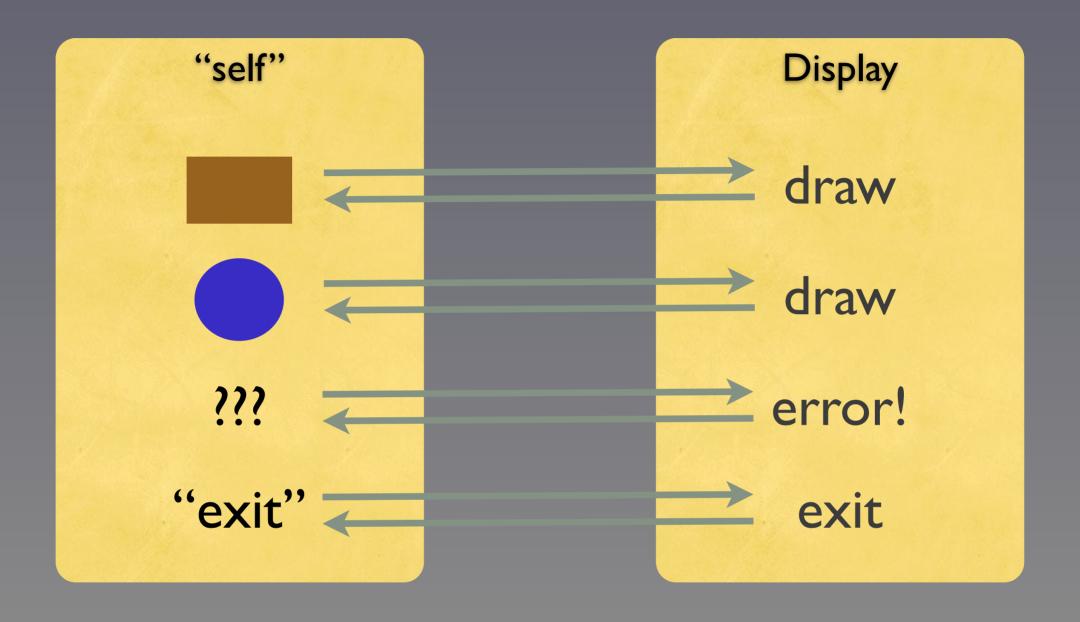
## Actor Model

- Message passing between autonomous actors.
- No shared (mutable) state.

### Actor Model

- First developed in the 70's by Hewitt, Agha, Hoare, etc.
- Made "famous" by Erlang.
  - Scala's Actors patterned after Erlang's.

## 2 Actors:



8

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

#### package shapes

```
case class Point(
x: Double, y: Double)
```

```
abstract class Shape {
  def draw()
  abstract draw method
```

#### Hierarchy of geometric shapes

7

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

"Case" classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape. The "case" keyword makes the arguments "vals" by default, adds factory, equals, etc. methods. Great for "structural" objects.

(Case classes automatically get generated equals, hashCode, toString, so-called "apply" factory methods - so you don't need "new" - and so-called "unapply" methods used for pattern matching.)

NOTE: The full source for this example is at https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala/code-examples/actor.

```
case class Circle(
 center:Point, radius:Double)
    extends Shape {
    draw
                       concrete draw
                         methods
case class Rectangle(
ll:Point, h:Double, w:Double)
    extends Shape {
 def (draw)
```

package shapes import akka.actor.Actor

Use the Akka Actor library

Actor

```
lass Drawer extends Actor
   receive
```

receive and handle each message

Actor for drawing shapes

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

An actor that waits for messages containing shapes to draw. Imagine this is the window manager on your computer. It loops indefinitely, blocking until a new message is received...

Note: This example uses the Akka Frameworks Actor library (see <a href="http://akka.io">http://akka.io</a>), which has now replaced Scala's original actors library. So, some of the basic actor classes are part of Scala's library, but we'll use the full Akka distibution.

Receive method

```
receive = {
case s:Shape =>
   print("-> "); s.draw()
  sender! ("Shape drawn.")
case "exit" =>
   println("-> exiting...")
  sender! ("good bye!")
                  // default
   println("-> Error: "
  sender! ("Unknown: " +
```

12

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

"Receive" blocks until a message is received. Then it does a pattern match on the message. In this case, looking for a Shape object, the "exit" message, or an unexpected object, handled with the last case, the default.

```
receive = {
 case s:Shape
                   S. 0 8W()
               'Shape drawr
   sende
                              þattern
                             matching
   sender
                       default
             -> Error:
   sender! ("Unknown:
```

13

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Each pattern is tested and the first match "wins". The messages we expect are a Shape object, the "exit" string or anything else. Hence, the last "case" is a "default" that catches anything, we we treat as an unexpected error.

```
draw shape
 receive = {
                             & send reply
  case s:Shape =>
              -> "); s.draw()
             ! ("Shape drawn.")
     sender
     println("-> exiting...")
                                          done
     sender! ("good bye!")
                             default
     println("-> Error:
     sender! ("Unknown:
                     unrecognized message
sender! sends a reply
                      14
                              Copyright © 2009-2013, Dean Wampler, All Rights Reserved
```

Sunday, July 7, 13

After handling each message, a reply is sent to the sender, using "self" to get the handle to our actor "nature".

```
package shapes
import akka.actor.Actor
class Drawer extends Actor {
 receive = {
  case s:Shape =>
    print("-> "); s.draw()
    sender ! ("Shape drawn.")
  case "exit" =>
    println("-> exiting...")
    sender ! ("good bye!")
                    // default
  case x =>
    println("-> Error: " + x)
    sender ! ("Unknown: " + x)
```

Altogether

```
import shapes._
import akka.actor._
import com.typesafe.config._

object Driver {
  def main(args:Array[String])={
   val sys = ActorSystem(...)
   val driver=sys.actorOf[Driver]
```

Application driver

16

val drawer=sys.actor0f[Drawer]

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Here's the driver actor. It is declared as an "object" not a class, making it a singleton.

When we start, we send the "go!" message to the Driver actor that is defined on the next slide. This starts the asynchronous message passing. The "!" is the message send method (stolen from Erlang).

driver | Start(drawer)

```
import akka.actor.
import com.typesafe.config.
                  Singleton for main
object Driver {
     main(args:Array[String])={
       sys = ActorSystem(...)
      driver=sys.actorOf[Driver]
       drawer=sys.actor0f[Drawer
            Start (drawer
                               Instantiate
                                 actors
           Send a message to
            start the actors
```

17

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

import shapes.

Here's the driver actor. It is declared as an "object" not a class, making it a singleton.

When we start, we send the "go!" message to the Driver actor that is defined on the next slide. This starts the asynchronous message passing. The "!" is the message send method (stolen from Erlang).

#### Companion class

```
class Driver extends Actor {
  var drawer: Option[Drawer] =
  None
```

```
def receive = {
```

```
....
}
}
```

18

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Here's the driver actor "companion class" for the object on the previous slide that held main.

Normally, you would not do such synchronous call and response coding, if avoidable, as it defeats the purpose of using actors for concurrency.

```
def receive = {
                            sent by
  case Start(d)
                             driver
    drawer = Some(d)
       ! Circle(Point(...),...)
    d | Rectangle(...)
    d! 3.14159
                            sent by
     ! "exit"
                            drawer
  case "good bye!"
    println("<- cleaning up...")
    context.system.shutdown()
  case other =>
    println("<- " + other)</pre>
```

19

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Here's the driver actor, a scala script (precompilation not required) to drive the drawing actor.

Normally, you would not do such synchronous call and response coding, if avoidable, as it defeats the purpose of using actors for concurrency.

```
Circle(Point(...),...)
d ! Rectangle(...)
d! 3.14159
   "exit"
```

```
-> drawing: Circle(Point(0.0,0.0),1.0)
-> drawing: Rectangle(Point(0.0,0.0),
2.0,5.0)
-> Error: 3.14159
-> exiting...
<- Shape drawn.
<- Shape drawn.
<- Unknown: 3.14159
<- cleaning up...
```

"<-" and "->" messages may be interleaved!!

```
// Drawing.receive
receive = {
 case s:Shape
  s.draw() +
  self.reply("...")
 case
 case
```

Functional-style pattern matching

Objectoriented-style polymorphism

"Switch" statements are not (necessarily) evil

21

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

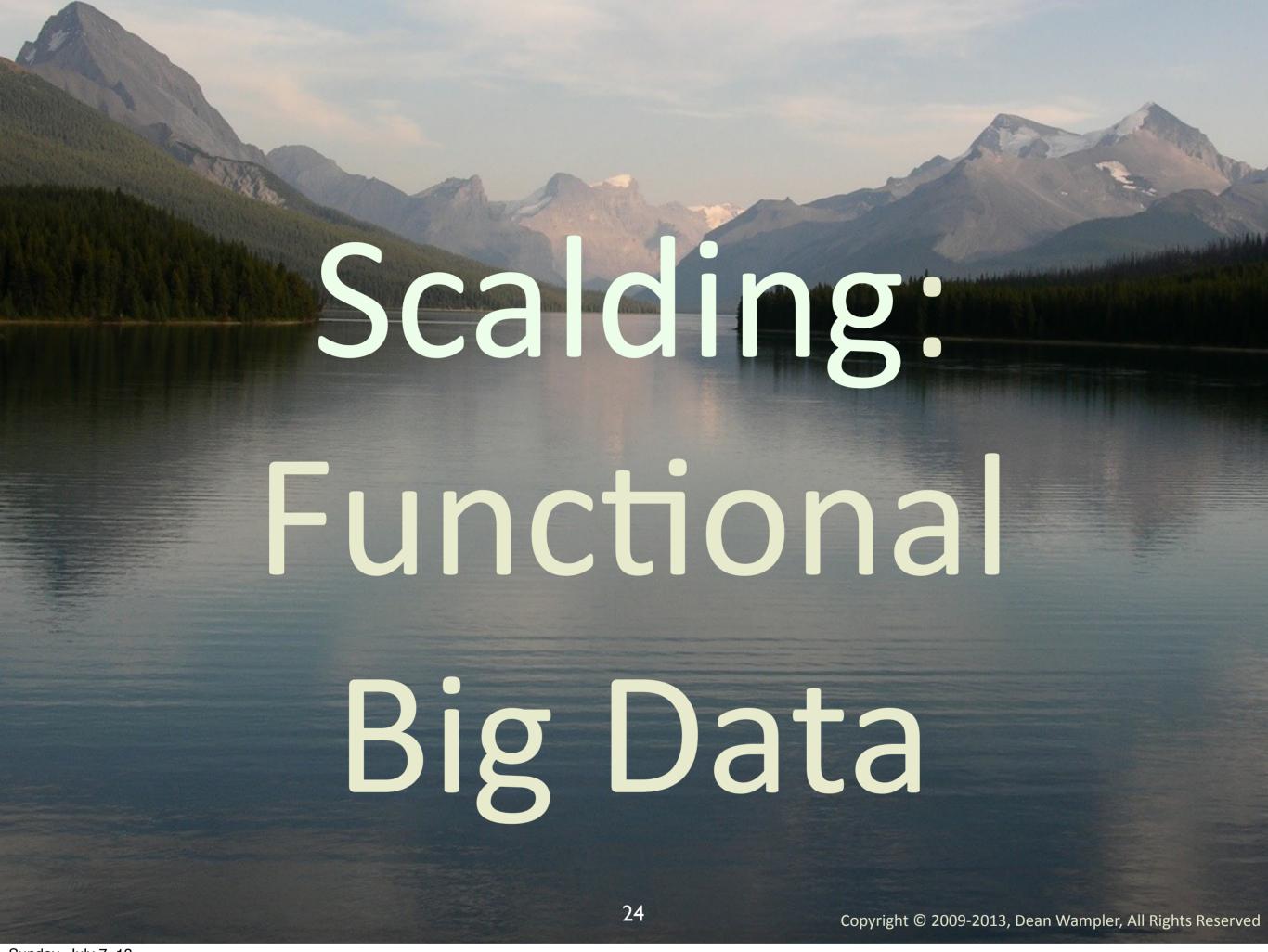
## Exercise

Variations on an Actor theme. drawing-actors.scala



## Refine Features

- Add more geometric shapes
- Add other messages for new behaviors.
  - Redraw all shapes drawn so far.
  - Clear the "screen".
  - ... (see the README).

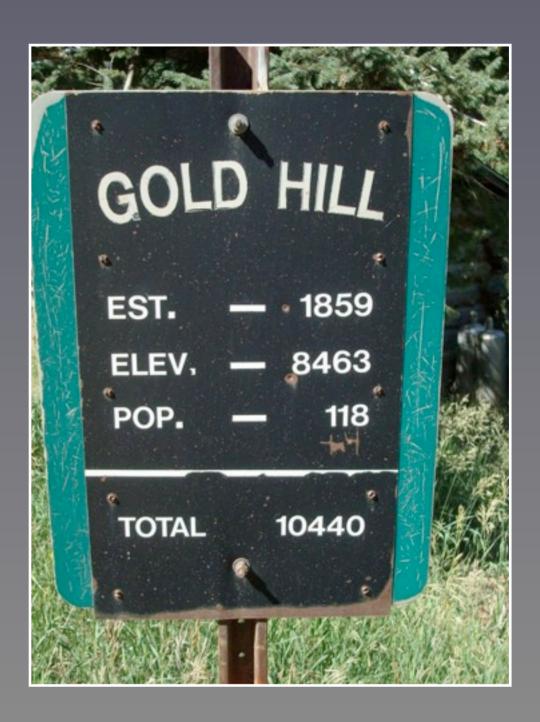


Sunday, July 7, 13

I believe another driver for FP's mainstream adoption will be the realization that it is ideal for data-centric applications, like "Big Data".

## Big Data

Data so big that traditional solutions are too slow, too small, or too expensive to use.



#### Hat tip: Bob Korbus

25

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

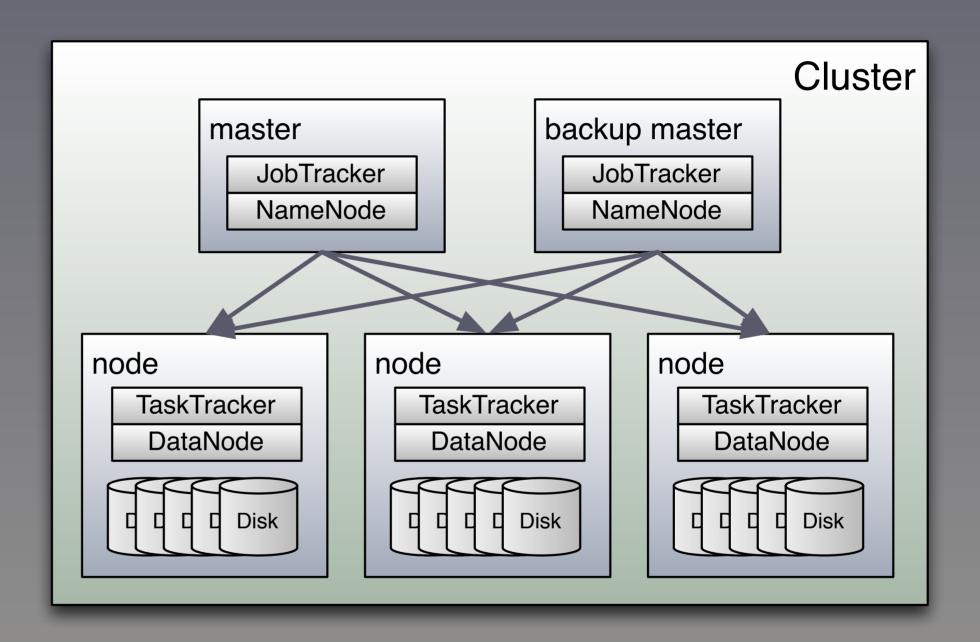
Sunday, July 7, 13

It's a buzz word, but generally associated with the problem of data sets too big to manage with traditional SQL databases. A parallel development has been the NoSQL movement that is good at handling semistructured data, scaling, etc.

## Hadoop Crash Course

 The most popular, generalpurpose, distributed data framework.

## Hadoop Crash Course

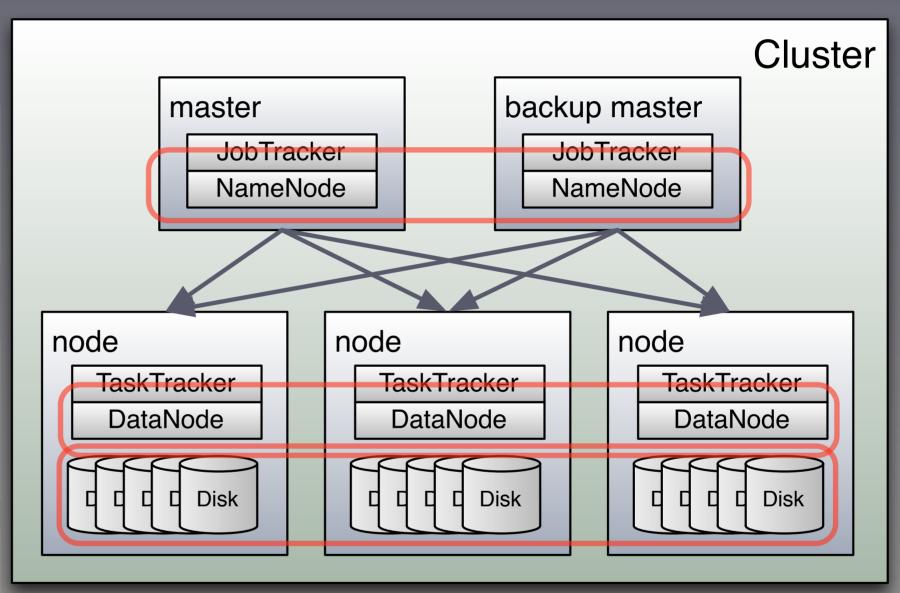


## Hadoop Distributed File System (HDFS)

master daemon

slave daemons

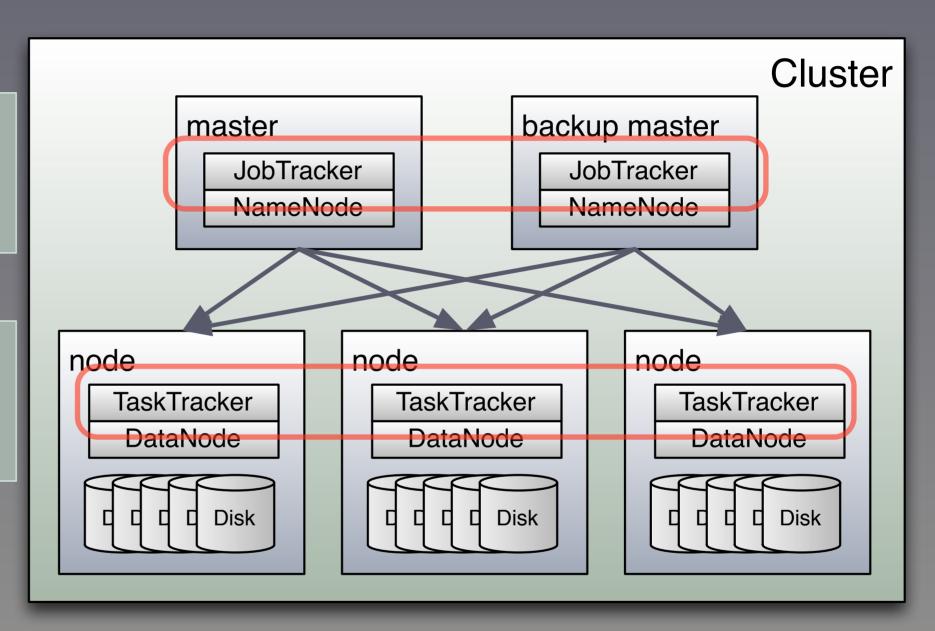
storage



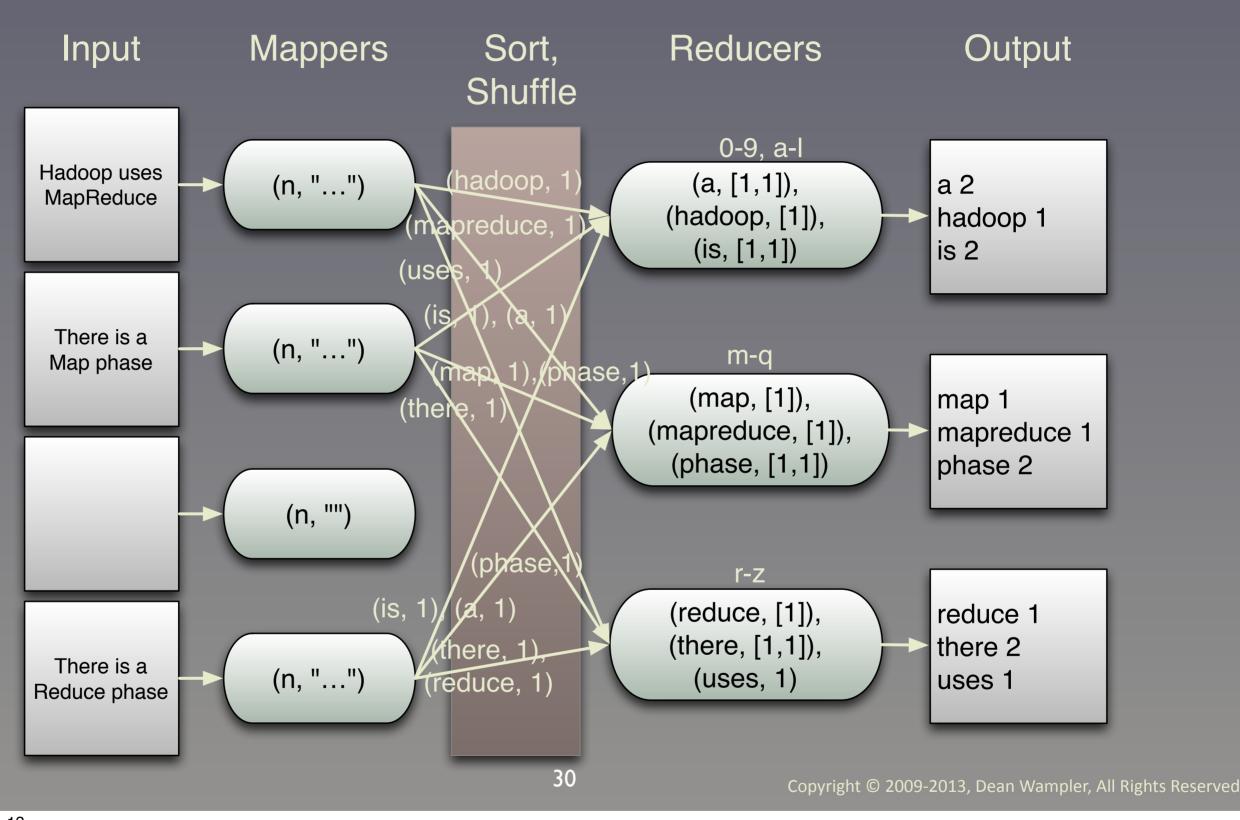
## Hadoop Jobs

master daemon

slave daemons



## What is MapReduce?



Sunday, July 7, 13

The famous Word Count algorithm. We have a corpus of documents on the left. We want to parse and count the words. By default, each document gets a Map task (JVM process) - the rounded bubbled, which we have to implement. Each line of the file is passed in with it's position offset as the "key". We discard the key, tokenize the line, and emit new key-value pairs, each word and a count of 1, for every word found. (There are obvious optimizations to you could use...). Hadoop sorts the keys within each mapper task, then "shuffles" the key-value pairs over the cluster network to the appropriate reducer task (which we also must implement). We have set up the example so that all keys that start with 0 through 9 or a through I go to the 1st reducer, etc. Finally, each reducer is passed the keys, one at a time, with a collection of the counts for each key. The reducer sums the collections and writes the output, at least one file per reducer.

## Hadoop Languages and Tools

- Hive: SQL queries.
- Pig: Dataflow language.
- Java API: "Assembly language".
- Cascading: High-level Java API.
- Scalding: Scala Cascading API.

31

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

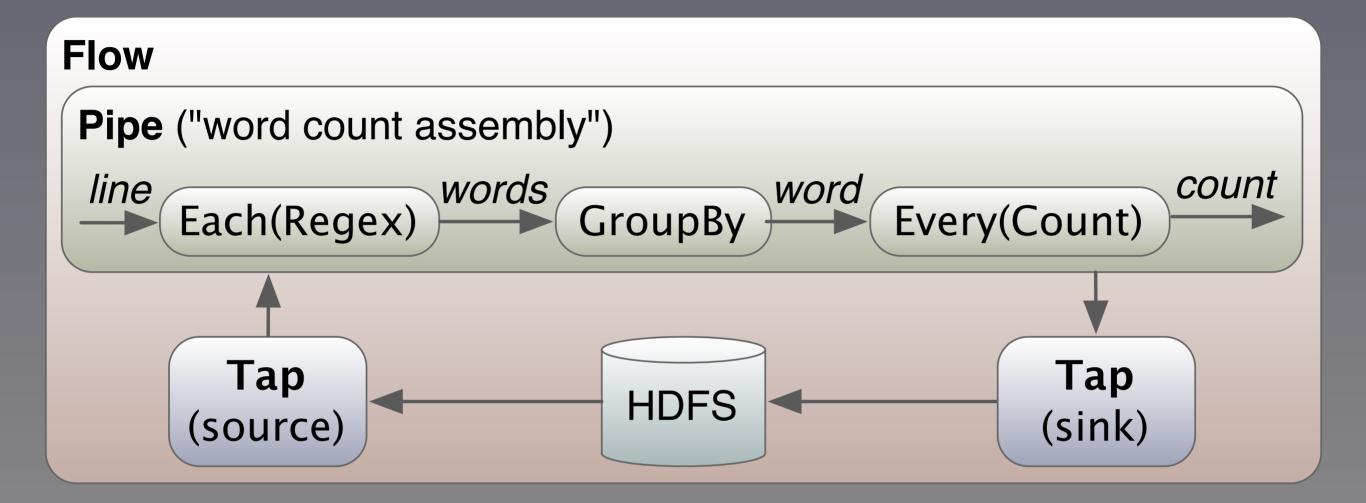
And there are other options... The 1st three are "standard" parts of Hadoop distros, with the Java API being the lowest-level API (and tedious to use). Think of Win32 or X-Windows APIs (if you're old enough;)

Cascading is a popular, 3rd-party Java API with higher abstractions for dataflows and standard relational operations, like joins, group-bys, etc. Scalding provides a Scala API on cascading that really exposed the abstractions beautifully, as we'll see.

## Cascading

Data flows consist of source and sink Taps connected by Pipes.

## Cascading



```
import org.cascading.*;
public class WordCount {
  public static void main(String[] args) {
    String inputPath = args[0];
    String outputPath = args[1];
    Properties properties = new Properties();
    FlowConnector.setApplicationJarClass( properties, Main.class );
   Scheme sourceScheme = new TextLine( new Fields( "line" ) );
   Scheme sinkScheme = new TextLine( new Fields( "word", "count" ) );
   Tap source = new Hfs( sourceScheme, inputPath );
    Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );
    Pipe assembly = new Pipe( "wordcount" );
   String regex = "(?<!\pL)(?=\pL)[^ ]*(?<=\pL)(?!\pL)";
    Function function = new RegexGenerator( new Fields( "word" ), regex );
    assembly = new Each( assembly, new Fields( "line" ), function );
    assembly = new GroupBy( assembly, new Fields( "word" ) );
    Aggregator count = new Count( new Fields( "count" ) );
    assembly = new Every( assembly, count );
    FlowConnector flowConnector = new FlowConnector( properties );
    Flow flow = flowConnector.connect( "word-count", source, sink, assembly);
    flow.complete();
```

34

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Here is the Cascading Java code. It's cleaner than the MapReduce API, because the code is more focused on the algorithm with less boilerplate, although it looks like it's not that much shorter. HOWEVER, this is all the code, where as previously I omitted the setup (main) code. See <a href="http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html">http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html</a> for details of the API features used here; we won't discuss them here, but just mention some highlights.

Note all the green types, but many represent "functions" to apply to the data, such as Count, Each, GroupBy, etc. I won't show the corresponding low-level Java API code, but Cascading does a good job emphasizing composition of behaviors and minimizing framework boilerplate.

## Scalding

Scala API on top of Cascading.

Developed at Twitter.

github.com/twitter/scalding

```
import com.twitter.scalding._
```

```
class WordCountJob(args: Args) extends
Job(args) {
  TextLine( args("input") )
                                    That's it!
    . read
    .flatMap('line -> 'word) {
      line: String =>
        line.trim.toLowerCase
             .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

36

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

#### import com.twitter.scalding.\_

```
class WordCountJob(args: Args) extends
Job(args) {
 TextLine( args("input") )
                                    Imports
    .read
    .flatMap('line -> 'word) {
      line: String =>
        line.trim.toLowerCase
            .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

#### import com.twitter.scalding.\_

```
class WordCountJob(args: Args) extends
Job(args) {
  <u>fextLine(args("input"</u>
    .read
                                       class
    .flatMap('line -> 'word) {
      line: String =>
        line.trim.toLowerCase
             .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

38

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

#### import com.twitter.scalding.\_

```
class WordCountJob(args: Args) extends
Job(args)
                                    Open the
  TextLine( args("input")
                                     text file
    .read
    .tlatMap('line -> 'word)
                                      inbut
      line: String =>
        line.trim.toLowerCase
             .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

39

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

The "args" holds the user-specified arguments used to invoke the job. "--input path" specifies the source of the text. Similarly, "--output path" is used to specify the output location for the results.

```
import com.twitter.scalding.
class WordCountJob(args: Args) extends
Job(args) {
                                    Split each
 TextLine( args("input")
                                     line into
     read
    .flatMap('line -> 'word) {
                                      words
      line: String =>
        line.trim.toLowerCase
             .split("\\W+")
    .groupby( word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

40

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

A flatMap iterates through a collection, calling a function on each element, where the function "converts" each element into a collection. So, you get a collection of collections. The "flat" part flattens those nested collections into one larger collection. The "line: String => line ...("\\W+")" expression is the function called by flatMap on each element, where the argument to the function is a single string variable named line and the body is to the right-hand side of the the =>. The output word field is called "word", from the first argument to flatMap: "line -> 'word".

```
import com.twitter.scalding._
```

```
class WordCountJob(args: Args) extends
Job(args) {
                                     Group by
  TextLine( args("input")
                                     each word
    .read
    .flatMap('line -> 'word) {
                                     and count
      line: String =>
                                     the group
        line.trim.toLowerCase
                                       sizes
             .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

41

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

A flatMap iterates through a collection, calling a function on each element, where the function "converts" each element into a collection. So, you get a collection of collections. The "flat" part flattens those nested collections into one larger collection. The "line: String => line ...("\\W+")" expression is the function called by flatMap on each element, where the argument to the function is a single string variable named line and the body is to the right-hand side of the the =>. The output word field is called "word", from the first argument to flatMap: "line -> 'word".

```
import com.twitter.scalding._
```

```
class WordCountJob(args: Args) extends
Job(args) {
 TextLine( args("input") )
                                     Write tab-
    .read
                                     separated
    .flatMap('line -> 'word) {
                                     words and
      line: String =>
        line.trim.toLowerCase
                                       counts
             .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

42

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Finally, we write the tab-separated words and counts to the output location specified using "--output path".

```
import com.twitter.scalding._
```

```
class WordCountJob(args: Args) extends
Job(args) {
  TextLine( args("input") )
    . read
                                      Profit!!
    .flatMap('line -> 'word) {
      line: String =>
        line.trim.toLowerCase
             .split("\\W+")
    .groupBy('word) {
      group => group.size('count) }
  .write(Tsv(args("output")))
```

# Scalding adds a built-in Matrix library (linear algebra) and a separate Algebra project called Algebird.

github.com/twitter/algebird

## Exercise

Scalding "Workshop":
WordCount2.scala,
StockAverages3.scala



Sunday, July 7, 13

I've provided a snapshot of a Scalding tutorial I've written from <a href="https://github.com/deanwampler/scalding-workshop">https://github.com/deanwampler/scalding-workshop</a>. We'll do a few of the examples from there now.



Sunday, July 7, 13

Doing relational-style joins.

```
import com.twitter.scalding.
class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('symd, 'close, 'volume)
  val divsSchema = ('dymd, 'dividend)
 val stocksPipe = new Tsv(args("stocks"), stockSchema)
    . read
    .project('symd, 'close)
 val divsPipe = new Tsv(args("dividends"), divsSchema)
    . read
  stocksPipe
    .joinWithTiny('symd -> 'dymd, dividendsPipe)
    .project('symd, 'close, 'dividend)
    .write(Tsv(args("output")))
```

#### Inner join of stocks and dividends

import com.twitter.scalding.

#### Define the schemas as values.

```
import com.twitter.scalding.
class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('symd, 'close, 'volume)
 val divsSchema = ('dvmd. 'dividend)
  val stocksPipe = new Tsv(args("stocks"), stockSchema)
    . read
    .project('symd, 'close)
 val divsPipe = new Tsv(args("dividends"), divsSchema)
    . read
  stocksPipe
    .joinWithTiny('symd -> 'dymd, dividendsPipe)
    .project('symd, 'close, 'dividend)
```

Separate pipes for the two sources. Project out the fields we want.

.write(Tsv(args("output")))

```
import com.twitter.scalding.
class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('symd, 'close, 'volume)
 val divsSchema = ('dymd, 'dividend)
 val stocksPipe = new Tsv(args("stocks"), stockSchema)
    read
    .project('symd, 'close)
 val divsPipe = new Tsv(args("dividends"), divsSchema)
    . read
  stocksPipe
    .joinWithTiny('symd -> 'dymd, dividendsPipe)
    .project('symd, 'close, 'dividend)
    .write(Tsv(args("output")))
```

Join stocks with smaller dividends on the year-month-day. Project the output fields.

# Exercise

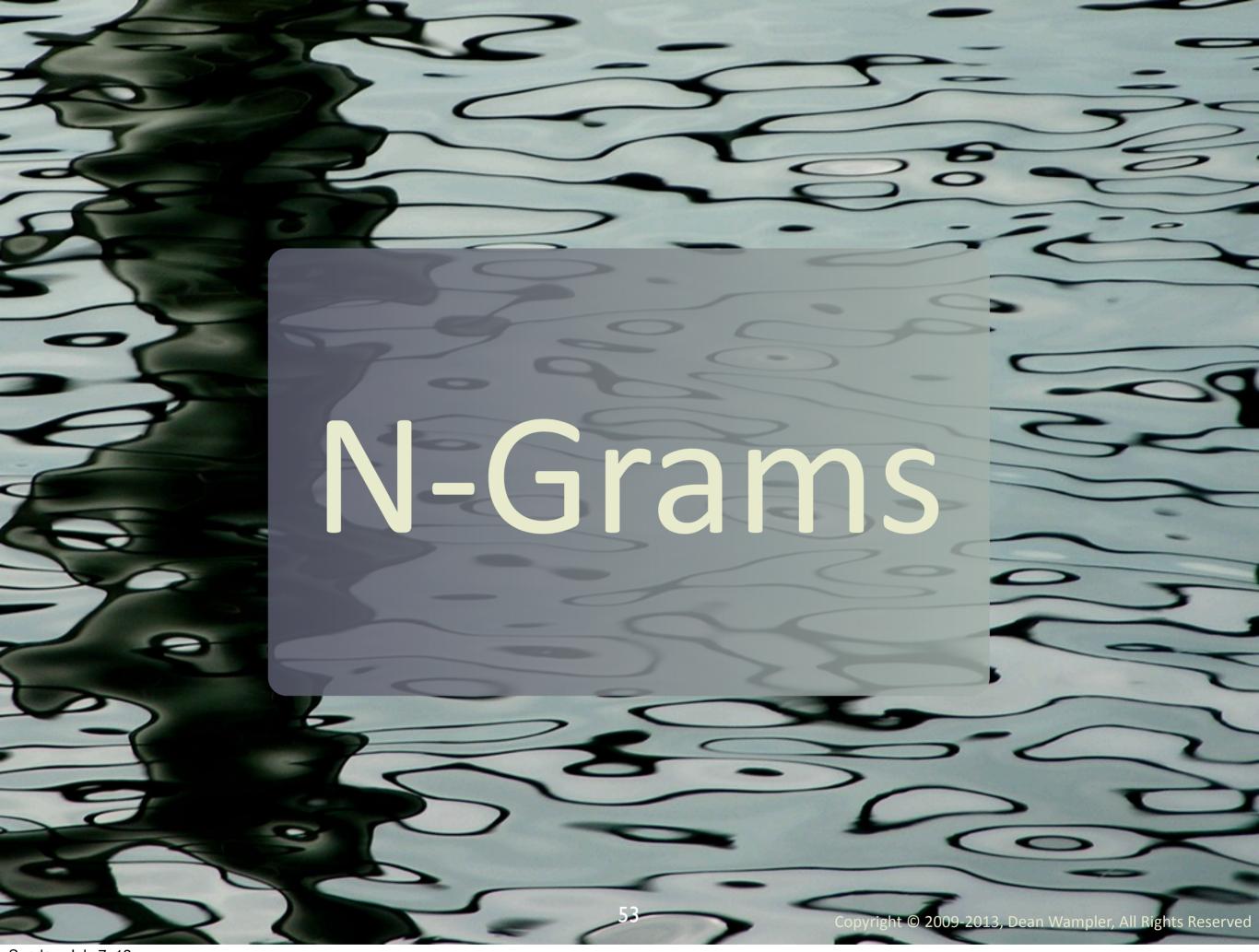
Scalding "Workshop": StocksDividendsJoin4.scala



Sunday, July 7, 13

I've provided a snapshot of a Scalding tutorial I've written from <a href="https://github.com/deanwampler/scalding-workshop">https://github.com/deanwampler/scalding-workshop</a>. We'll do a few of the examples from there now.

```
# Invoking the script (bash)
run.rb StocksDivsJoins.scala \
  --stocks data/stocks/IBM.txt \
  --dividends data/dividends/IBM.txt \
  --output output/IBM-join.txt
# Output (not sorted!)
2010 - 02 - 08
             121.88
                      0.55
2009 - 11 - 06
              123.49 0.55
            117.38 0.55
2009 - 08 - 06
2009-05-06 104.62 0.55
                       0.5
2009 - 02 - 06
           96.14
           85.15 0.5
2008 - 11 - 06
2008-08-06 129.16 0.5
2008 - 05 - 07
                       0.5
          124.14
```



Sunday, July 7, 13 Doing relational-style joins.

```
import com.twitter.scalding.
class ContextNGrams(args: Args) extends Job(args) {
 val ngramPrefix =
    args.list("ngram-prefix").mkString(" ")
 val keepN = args.getOrElse("count", "10").toInt
 val ngramRE = (ngramPrefix + """\s+(\w+)""").r
  // Sort (phrase,count) by count, descending.
  val countReverseComparator =
    (tuple1:(String, Int), tuple2:(String, Int))
      => tuple1. 2 > tuple2. 2
                      Ist half...
```

```
val lines = TextLine(args("input"))
  . read
  .flatMap('line -> 'ngram) { text: String =>
    ngramRE.findAllIn(text).toIterable }
  .discard('num, 'line)
  .groupBy('ngram) { g => g.size('count) }
  .groupAll { g =>
    g.sortWithTake[(String,Int)](
     ('ngram, 'count) -> 'sorted ngrams, keepN)(
     countReverseComparator)
  .write(Tsv(args("output")))
```

```
import com.twitter.scalding.
class ContextNGrams(args: Args) extends Job(args) {
  val ngramPrefix =
    args.list("ngram-prefix").mkString(" ")
 val keepN = args.getOrElse("count", "10").toInt
 val ngramRE = (ngramPrefix + """\s+(\w+)""").r
  // Sort (phrase, count) by count, descending.
 val countReverseComparator =
    (tuple1:(String,Int), tuple2:(String,Int))
      => tuple1. 2 > tuple2. 2
```

```
import com.twitter.scalding._
class ContextNGrams(args: Args) extends Job(args) {
  val ngramPrefix =
    args.list("ngram-prefix").mkString(" ")
  val keepN = args.getOrElse("count", "10").toInt
  val ngramRE = (ngramPrefix + """\s+(\w+)""").r

// Sort (phrase,count) by count, descending.
  val countReverseComparator =
    (tuple1:(String,Int), tuple2:(String,Int))
    => tuple1._2 > tuple2._2
...
```

A regex for finding the ngrams.

```
import com.twitter.scalding._

class ContextNGrams(args: Args) extends Job(args) {
  val ngramPrefix =
    args.list("ngram-prefix").mkString(" ")
  val keepN = args.getOrElse("count", "10").toInt
  val ngramRE = (ngramPrefix + """\s+(\w+)""").r

// Sort (phrase,count) by count, descending.
  val countReverseComparator =
    (tuple1:(String,Int), tuple2:(String,Int))
    => tuple1._2 > tuple2._2
```

Function we'll use to sort n-grams by frequency, descending.

Read lines, find n-grams, discard some fields, group by n-gram.

"Group all" (n-gram, count) together, sort by count, write out.

# Exercise

Scalding "Workshop": ContextNGrams7.scala



Sunday, July 7, 13

I've provided a snapshot of a Scalding tutorial I've written from <a href="https://github.com/deanwampler/scalding-workshop">https://github.com/deanwampler/scalding-workshop</a>. We'll do a few of the examples from there now.

```
# Invoking the script (bash)
run ContextNGrams.scala \
  --input data/shakespeare/plays.txt \
  --output output/context-ngrams.txt \
  --ngram-prefix "I love" \
  --count 10
# Output (reformatted)
(I love thee, 44),
(I love you, 24),
(I love him, 15),
(I love the,9),
(I love her,8),
(I love myself,3),
. . . ,
(I love Valentine, 1),
(I love France, 1), ...
```



Sunday, July 7, 13

Doing relational-style joins.

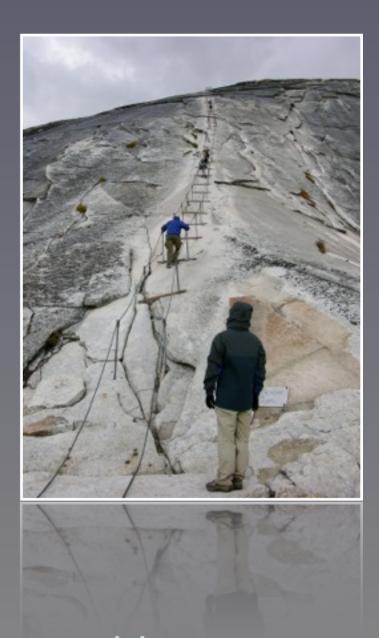
```
import com.twitter.scalding.
import com.twitter.scalding.mathematics.Matrix
// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
 import Matrix.
 val schema = ('user1, 'user2, 'relation)
 val adjacencyMatrix = Tsv(args("input"), schema)
    . read
    .toMatrix[Long,Long,Double](schema)
 val normMatrix = adjacencyMatrix.rowL2Normalize
 // Inner product is equivalent to the cosine:
  // AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
```

```
import com.twitter.scalding.
import com.twitter.scalding.mathematics.Matrix
// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
<del>class MatrixCosine(args: Args) extends Job(args)</del>
  import Matrix.
 val schema = ('user1, 'user2, 'relation)
 val adjacencyMatrix = Tsv(args("input"), schema)
    . read
    .toMatrix[Long,Long,Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize
  // Inner product is equivalent to the cosine:
  // AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
                                            imports
```

```
import com.twitter.scalding.
import com.twitter.scalding.mathematics.Matrix
// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
 import Matrix.
 val schema = ('user1, 'user2, 'relation)
 val adjacencyMatrix = Tsv(args("input"), schema)
    read
    .toMatrix[Long,Long,Double](schema)
 val normMatrix = adjacencyMatrix.rowL2Normalize
 // Inner product is equivalent to the cosine:
  // AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
                                            Cosine!
```

# Final Exercise

 Pick either of the following Akka or Scalding tutorial on GitHub and work start it.



- github.com/henrikengstrom/akka-meetup-sthlm
- github.com/deanwampler/scalding-workshop



Sunday, July 7, 13

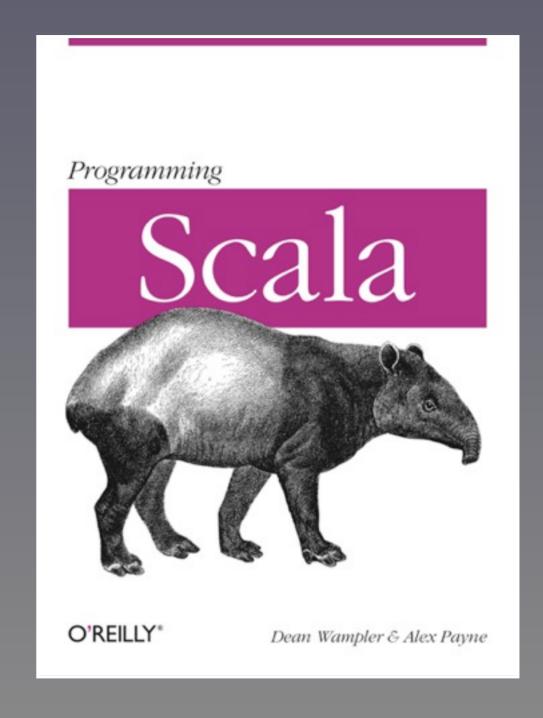
Akka is a powerful toolkit for distributed applications.

Scalding is a great example of why FP is ideal for data problems.

### Thanks!

dean@concurrentthought.com
 @deanwampler

polyglotprogramming.com/ programmingscala.com





71

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Sunday, July 7, 13

Copyright © 2010-2013, Dean Wampler. Some Rights Reserved - All use of the photographs and image backgrounds are by written permission only. The content is free to reuse, but attribution is requested.

http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode