

# The Seductions of Scala - Applications

Dean Wampler  
[dean@concurrentthought.com](mailto:dean@concurrentthought.com)  
[@deanwampler](https://twitter.com/deanwampler)  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)

April 26, 2013



I

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

Friday, April 26, 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/](https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala)

[SeductionsOfScala](https://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 real-  
world applications.

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



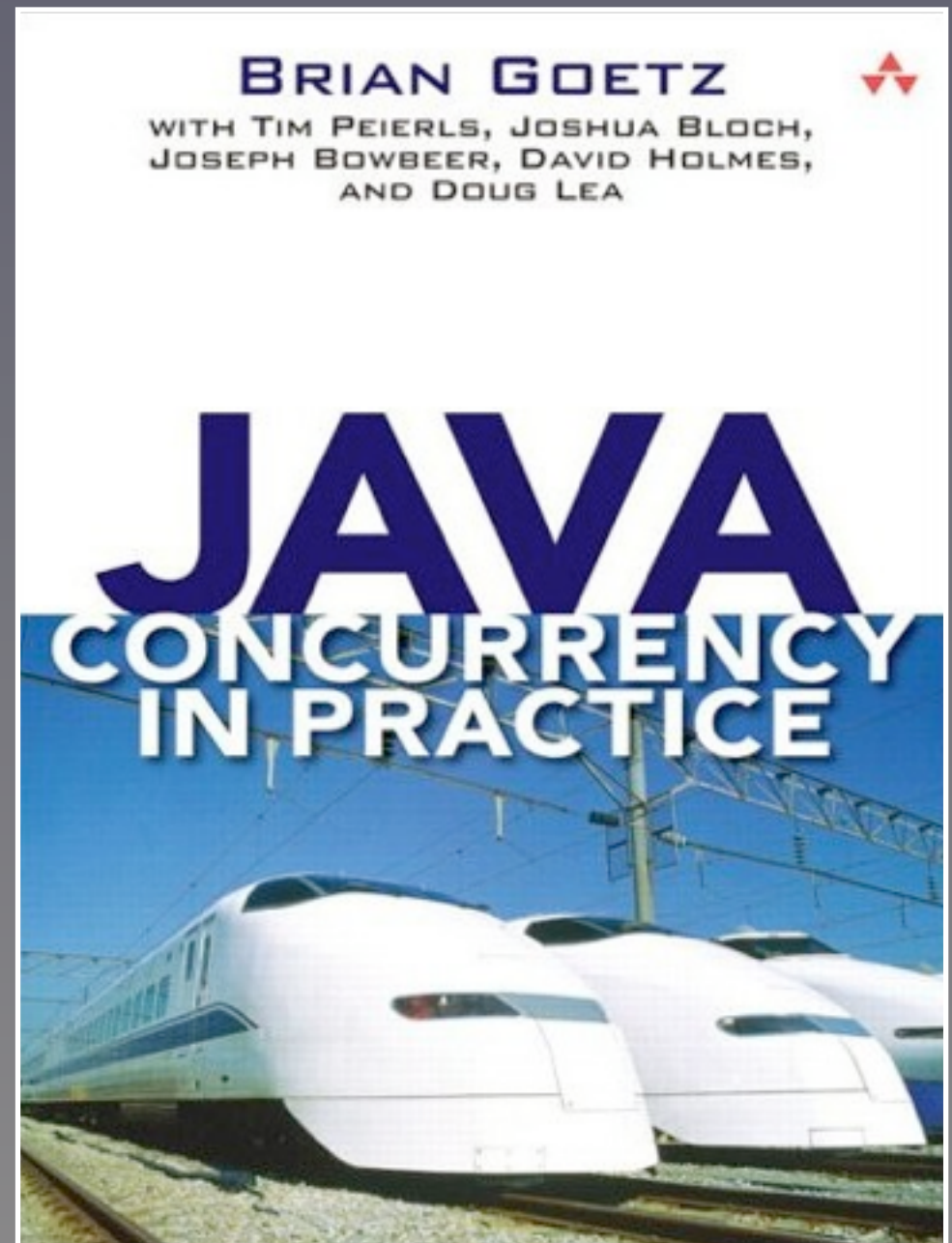


# Akka: Actor Concurrency



# When you share mutable state...

Hic sunt dracones  
(Here be dragons)



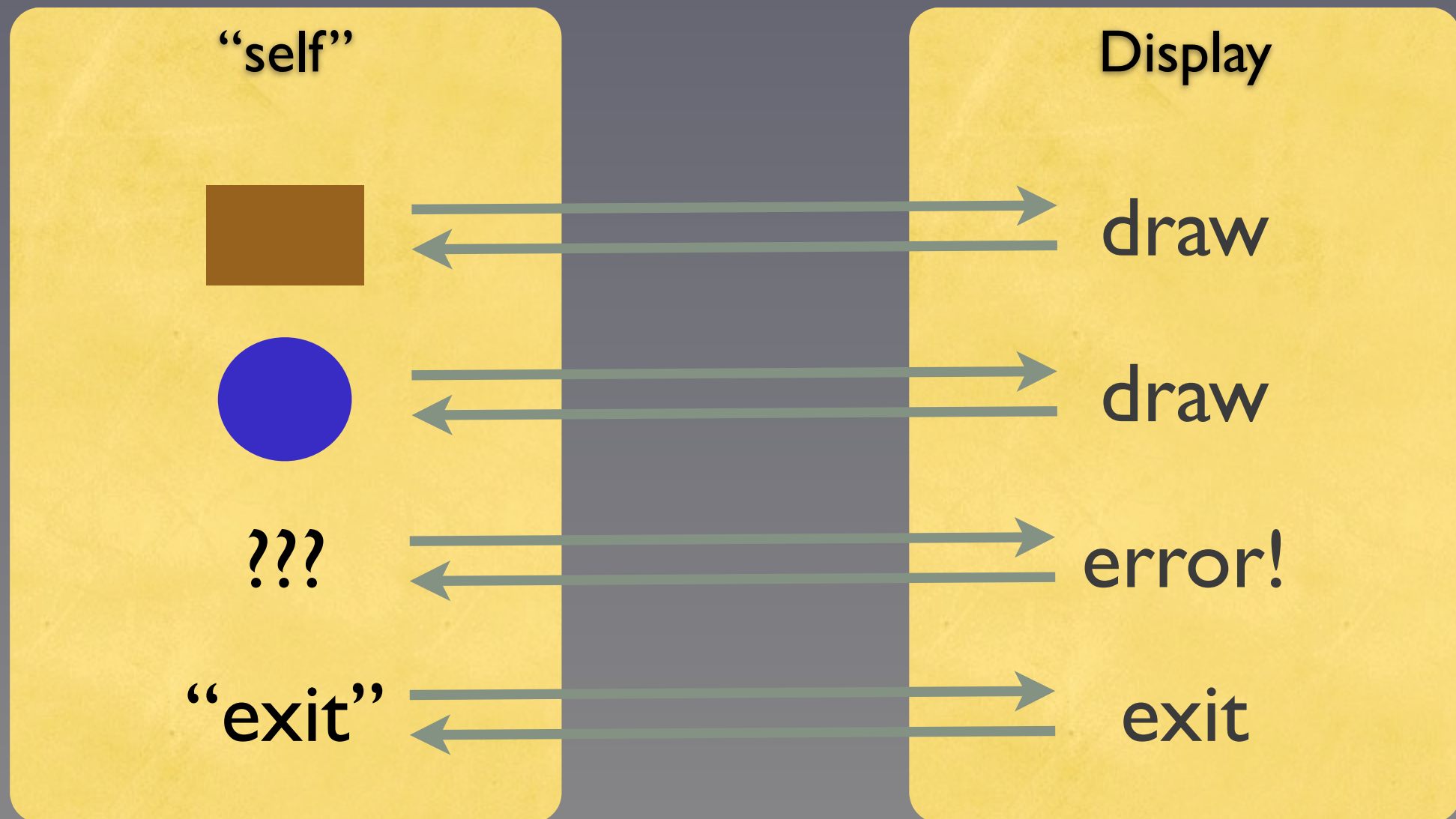
# Actor Model

- 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:





```
package shapes
```

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

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

*abstract **draw** method*

*Hierarchy of geometric shapes*

```
case class Circle(  
  center:Point, radius:Double)  
  extends Shape {  
  def draw() = ...  
}
```

*concrete **draw**  
methods*

```
case class Rectangle(  
  ll:Point, h:Double, w:Double)  
  extends Shape {  
  def draw() = ...  
}
```

*Use the Akka  
Actor library*

```
package shapes
```

```
import akka.actor.Actor
```

*Actor*

```
class Drawer extends Actor {
```

```
  def receive = {
```

```
    ...
```

```
  }
```

*receive and handle  
each message*

```
}
```

*Actor for drawing shapes*

||

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

Friday, April 26, 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...

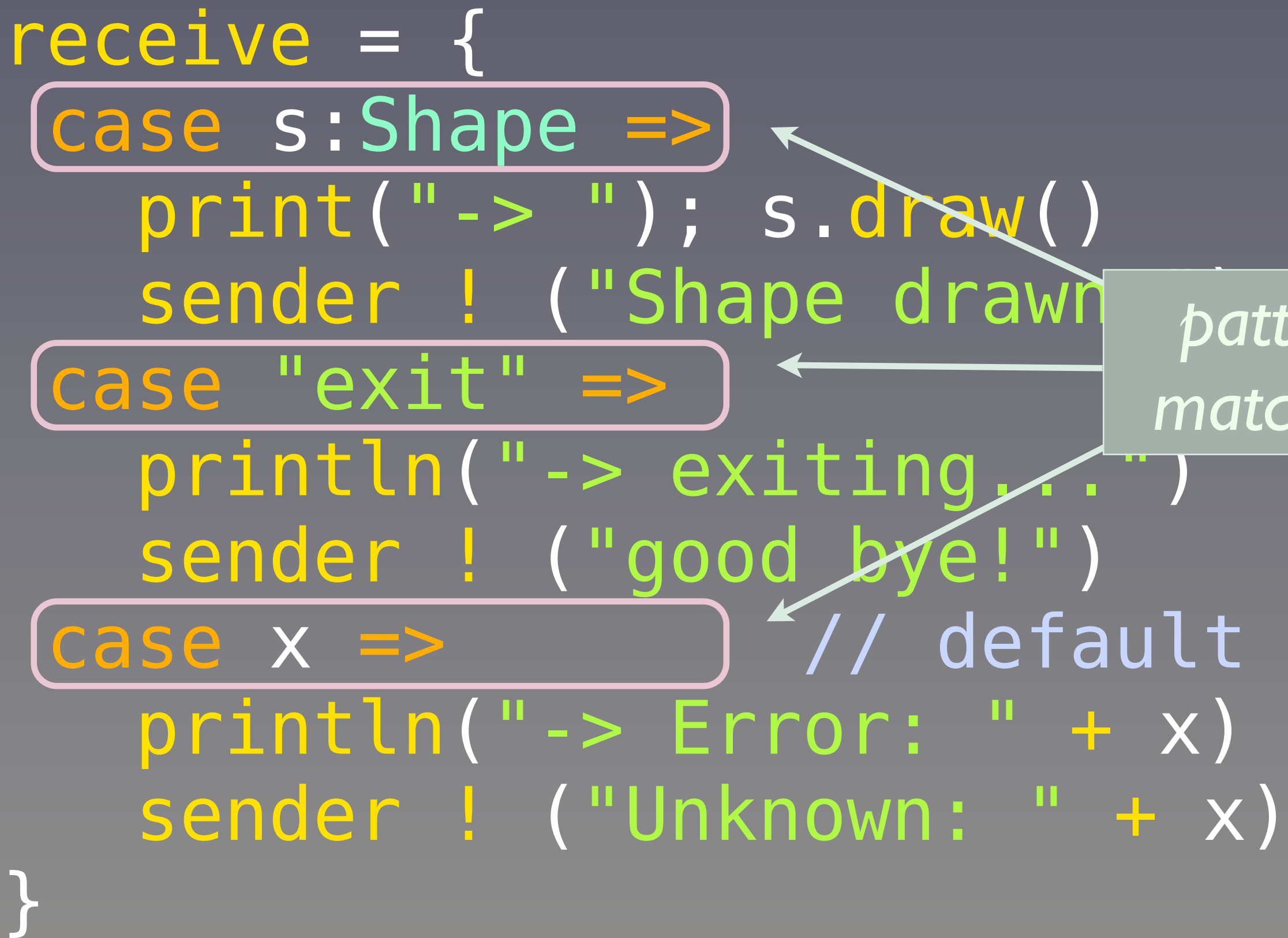
Note: This example uses the Akka Frameworks Actor library (see <http://akka.io>), which has now replaced Scala's original actors library. So, some of the basic actor classes are part of Scala's library, but we'll use the full Akka distribution.

*Receive  
method*

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



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



```
receive = {  
  case s:Shape =>
```

*draw shape  
& send reply*

```
    print("-> "); s.draw()  
    sender ! ("Shape drawn.")
```

```
  case "exit" =>
```

```
    println("-> exiting...")  
    sender ! ("good bye!")
```

*done*

```
  case x => // default
```

```
    println("-> Error: " + x)  
    sender ! ("Unknown: " + x)
```

```
}
```

*sender ! sends a reply*

*unrecognized message*

```
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!")
    case x => // default
      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]
    val drawer=sys.actorOf[Drawer]
    driver ! Start(drawer)
  }
}

```

*Application driver*

...



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

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

*Singleton for **main***

*Instantiate  
actors*

*Send a message to  
start the actors*

...

*Companion class*



```
...  
class Driver extends Actor {  
  var drawer: Option[Drawer] =  
    None  
  
  def receive = {  
    ...  
  }  
}
```

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

The diagram illustrates the flow of messages into the `receive` function. Two grey boxes represent external actors: `sent by driver` and `sent by drawer`. Arrows point from these boxes to specific cases in the `receive` function. The `sent by driver` box has an arrow pointing to the `case Start(d)` block. The `sent by drawer` box has an arrow pointing to the `case "good bye!"` block. Additionally, a long arrow points from the `context.system.shutdown()` line to the `case other` block, indicating a self-message or a message sent to the same actor.

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

*Object-  
oriented-style  
polymorphism*

*“Switch” statements are  
not (necessarily) evil*

# Exercise

Variations on an Actor  
theme.

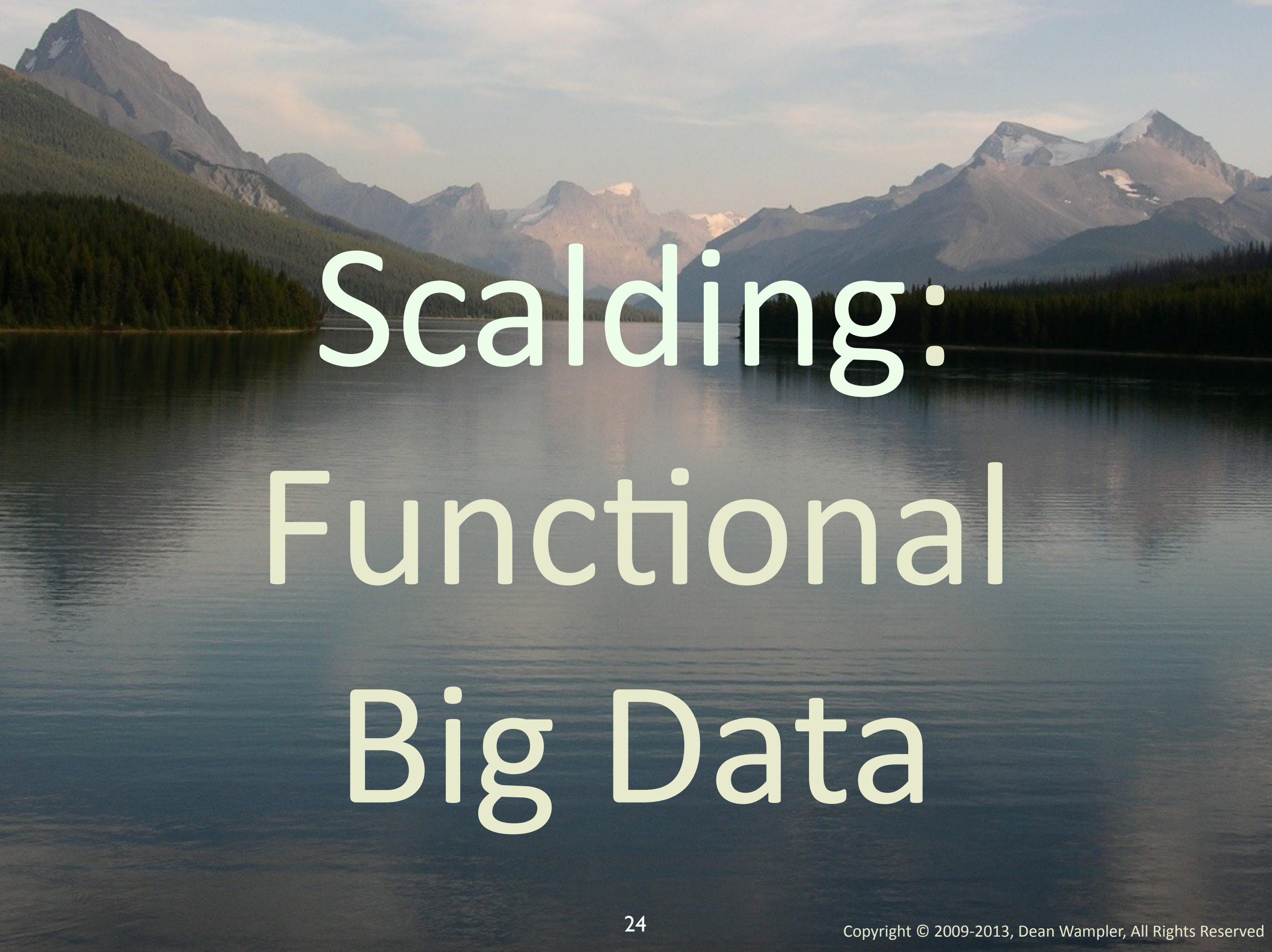
[drawing-actors.scala](http://drawing-actors.scala)



# Refine Features

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





# Scalding: Functional Big Data



# Big Data

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

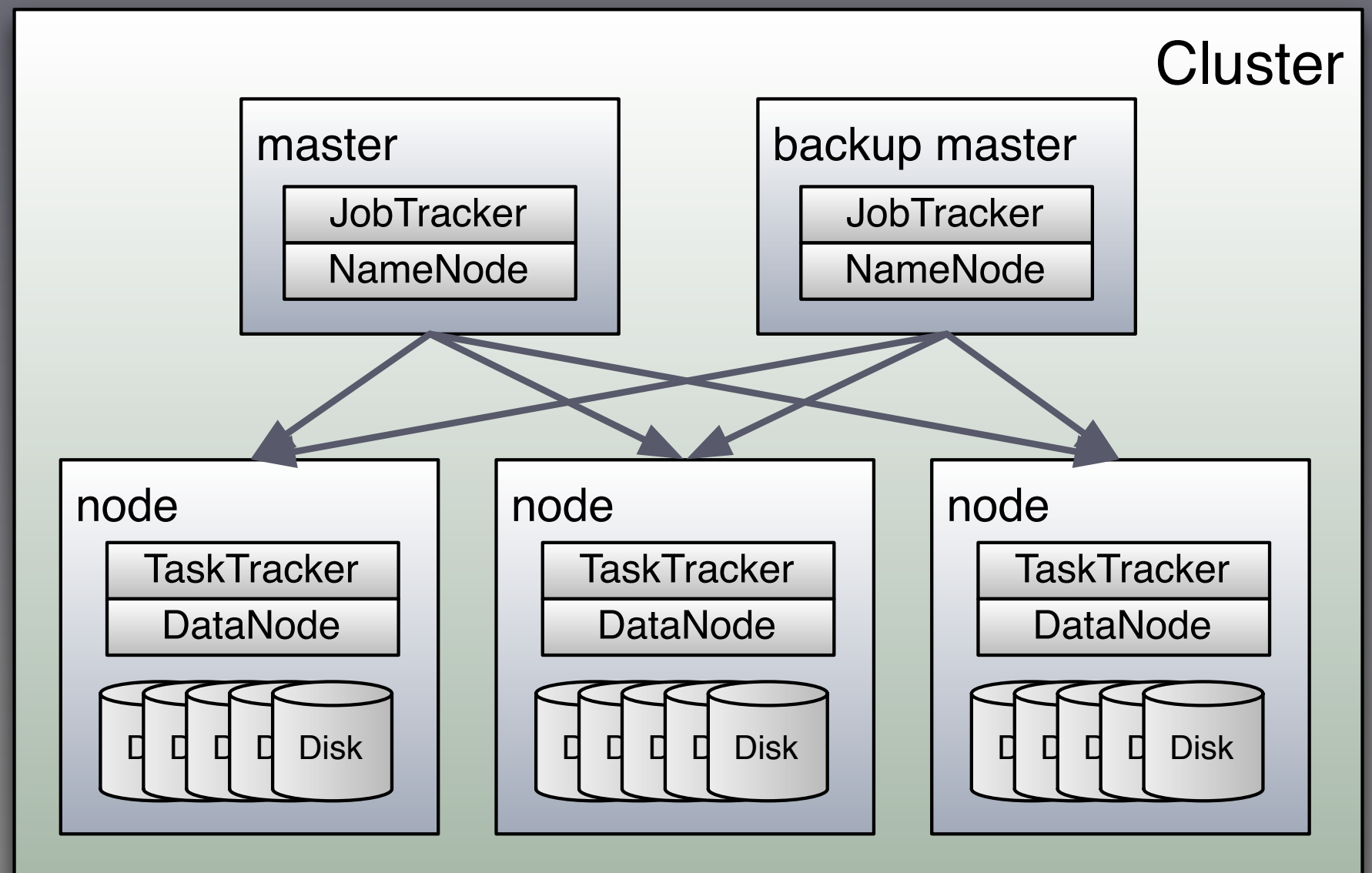


Hat tip: Bob Korbus

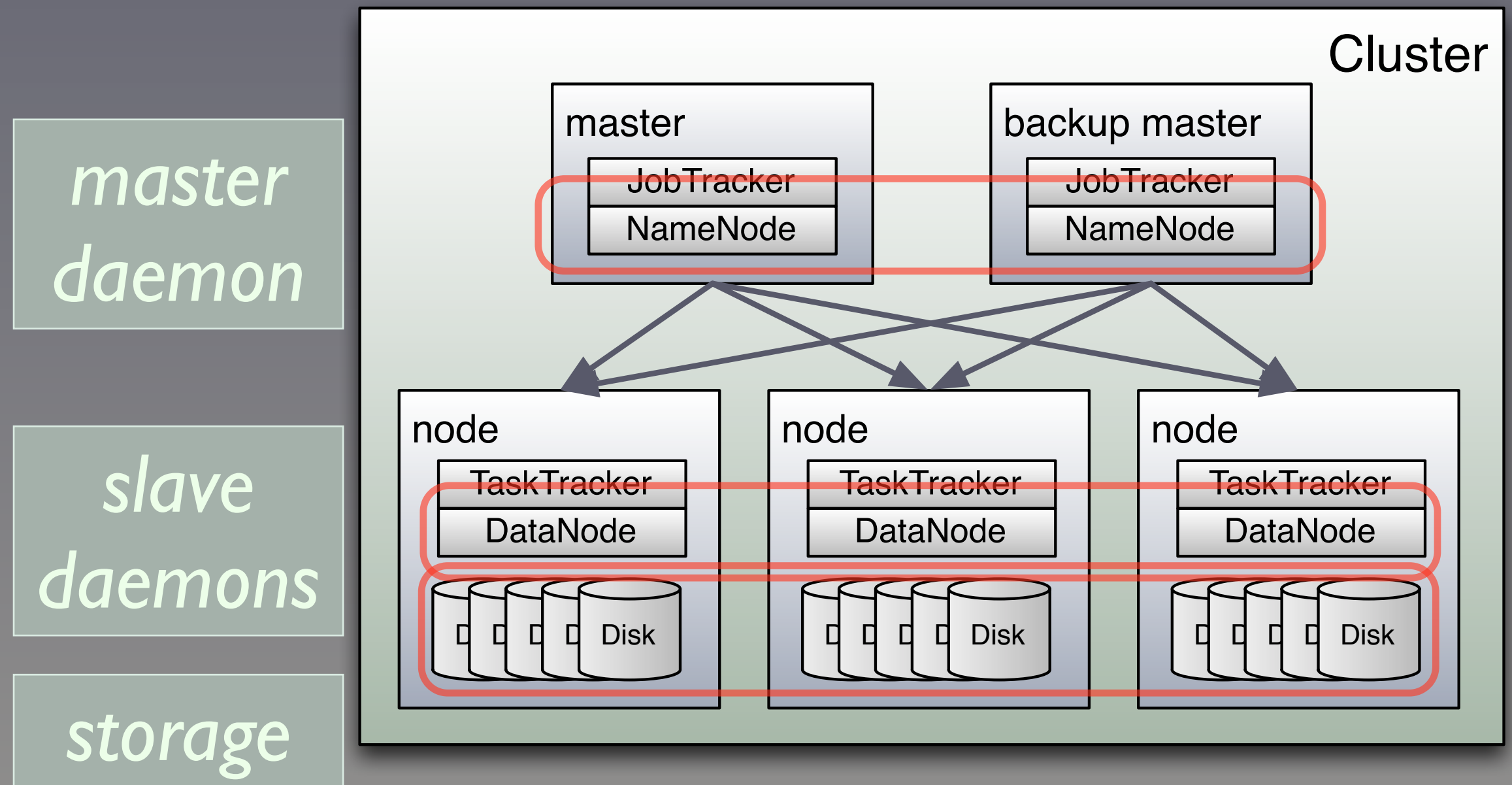
# Hadoop Crash Course

- The most popular, general-purpose, distributed data framework.

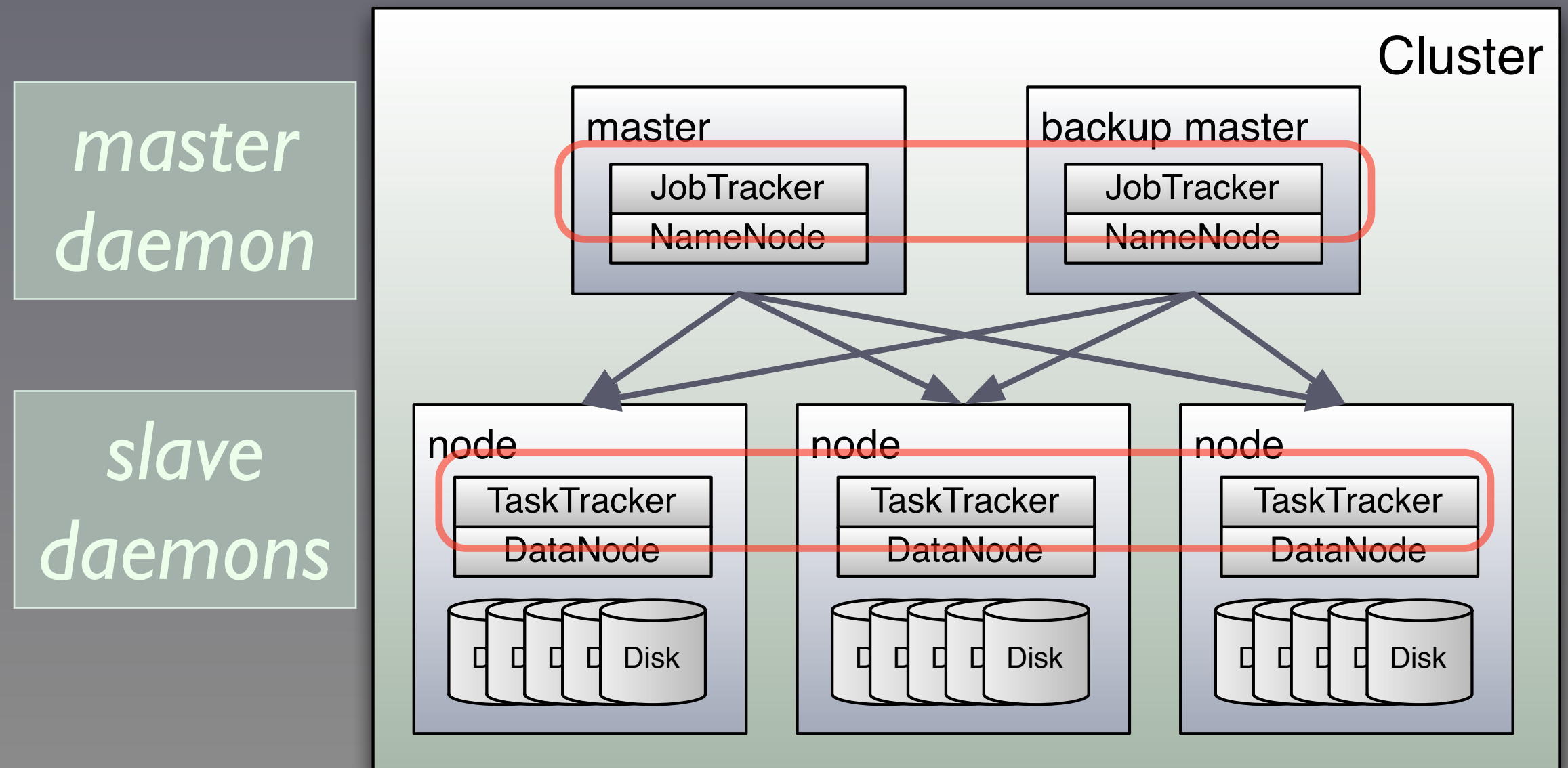
# Hadoop Crash Course



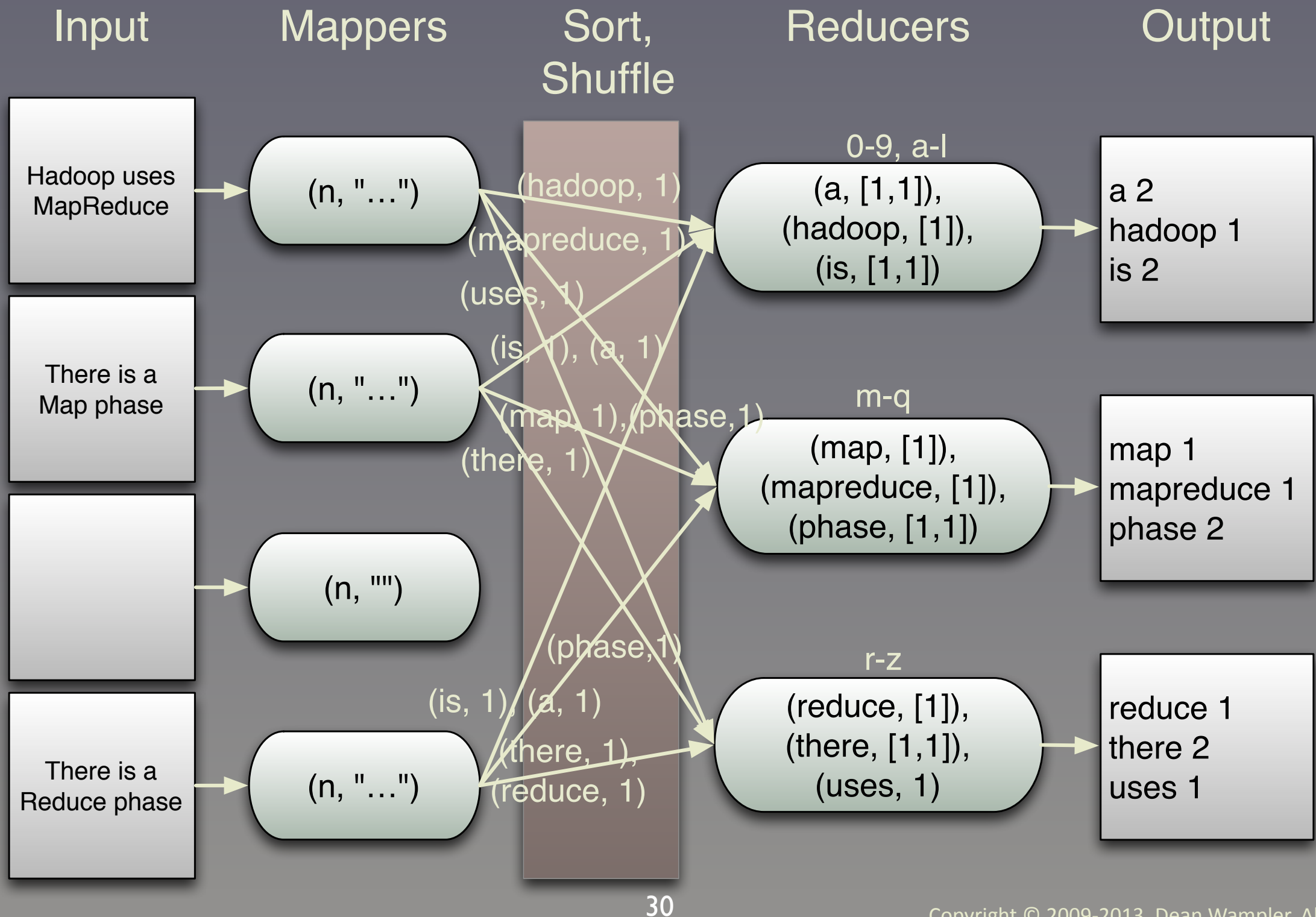
# Hadoop Distributed File System (HDFS)



# Hadoop Jobs



# What is MapReduce?



30

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

Friday, April 26, 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 l 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.

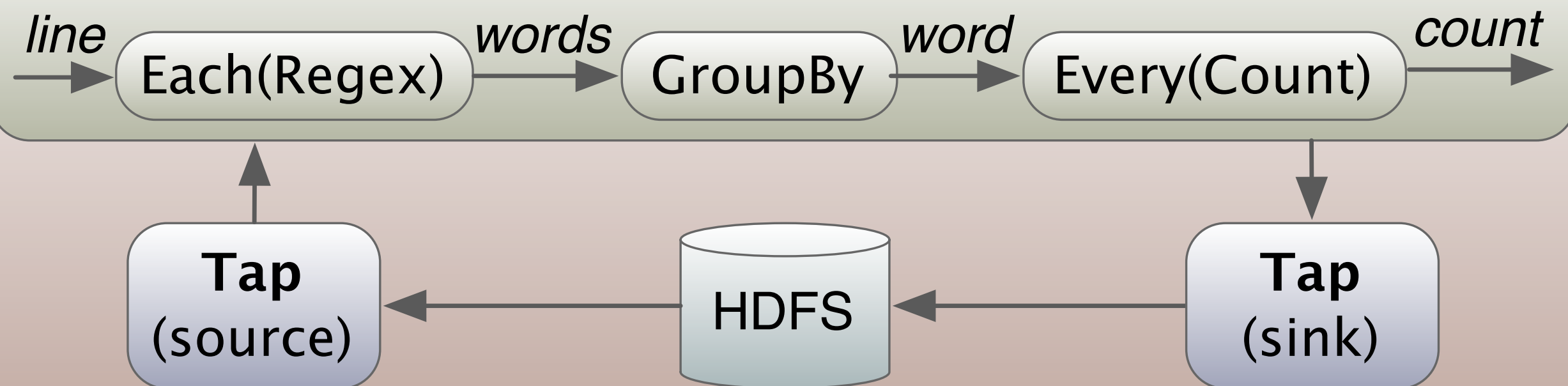
# Cascading

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

# Cascading

## Flow

**Pipe** ("word count assembly")



```

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();
    }
}

```

Friday, April 26, 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 <http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html> 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](https://github.com/twitter/scalding)

```
import com.twitter.scalding._

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

*That's it!*



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

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

*Imports*

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

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

*A Job  
class*

```
import com.twitter.scalding._

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

*Open the  
text file  
input*

```

import com.twitter.scalding._

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

```

*Split each  
line into  
words*

```

import com.twitter.scalding._

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

```

*Group by  
each word  
and count  
the group  
sizes*

```

import com.twitter.scalding._

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

```

*Write tab-separated words and counts*



```
import com.twitter.scalding._

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

*Profit!!*

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

[github.com/twitter/algebird](https://github.com/twitter/algebird)

# Exercise

Scalding “Workshop”:  
WordCount2.scala,  
StockAverages3.scala





# Joins



```
import com.twitter.scalding._

class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('synd, 'close, 'volume)
  val divsSchema = ('dynd, 'dividend)
  val stocksPipe = new Tsv(args("stocks"), stockSchema)
    .read
    .project('synd, 'close)
  val divsPipe = new Tsv(args("dividends"), divsSchema)
    .read

  stocksPipe
    .joinWithTiny('synd -> 'dynd, dividendsPipe)
    .project('synd, 'close, 'dividend)
    .write(Tsv(args("output")))
```

*Inner join of stocks and dividends*



```
import com.twitter.scalding._

class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('synd, 'close, 'volume)
  val divsSchema = ('dynd, 'dividend)
  val stocksPipe = new Isv(args("stocks"), stockSchema)
    .read
    .project('synd, 'close)
  val divsPipe = new Tsv(args("dividends"), divsSchema)
    .read

  stocksPipe
    .joinWithTiny('synd -> 'dynd, dividendsPipe)
    .project('synd, 'close, 'dividend)
    .write(Tsv(args("output")))
```

*Define the schemas as values.*

```
import com.twitter.scalding._

class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('synd, 'close, 'volume)
  val divsSchema = ('dynd, 'dividend)
  val stocksPipe = new Tsv(args("stocks"), stockSchema)
    .read
    .project('synd, 'close)
  val divsPipe = new Tsv(args("dividends"), divsSchema)
    .read
```

```
stocksPipe
  .joinWithTiny('synd -> 'dynd, dividendsPipe)
  .project('synd, 'close, 'dividend)
  .write(Tsv(args("output")))
```

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

```
import com.twitter.scalding._

class StocksDivsJoin(args: Args) extends Job(args){
  val stocksSchema = ('synd, 'close, 'volume)
  val divsSchema = ('dynd, 'dividend)
  val stocksPipe = new Tsv(args("stocks"), stockSchema)
    .read
    .project('synd, 'close)
  val divsPipe = new Tsv(args("dividends"), divsSchema)
    .read
```

```
stocksPipe
  .joinWithTiny('synd -> 'dynd, dividendsPipe)
  .project('synd, '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



```
# 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
2009-08-06	117.38	0.55
2009-05-06	104.62	0.55
2009-02-06	96.14	0.5
2008-11-06	85.15	0.5
2008-08-06	129.16	0.5
2008-05-07	124.14	0.5
...		



# N-Grams



```
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
  ...
}
```

*1st 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")))
}

```

*... 2nd half.*

```

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
  ...

```

*The n-gram prefix (e.g., “I love”) and how many to find (ordered by frequency).*

```
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.*

```

...
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")))
}

```

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



```

...
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")))
}

```

*“Group all” (n-gram, count) together, sort by count, write out.*

# Exercise

Scalding “Workshop”:  
ContextNGrams7.scala



```
# 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), ...
```



# Matrices



```

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.
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")))
}

```

*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](https://github.com/henrikengstrom/akka-meetup-sthlm)
- [github.com/deanwampler/scalding-workshop](https://github.com/deanwampler/scalding-workshop)





# Recap

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](mailto:dean@concurrentthought.com)  
[@deanwampler](https://twitter.com/deanwampler)

[polyglotprogramming.com/](http://polyglotprogramming.com/)  
[programmingscala.com](http://programmingscala.com)

