@Scalding

https://github.com/twitter/scalding

Argyris Zymnis @argyris
Oscar Boykin @posco
Twitter

- What is Scalding?
- Why Scala for Map/Reduce?
- How is it used at Twitter?

- An API to write Hadoop jobs in Scala
- Scala is a more natural language choice for manipulating data in Hadoop
- Extensively used in Twitter's revenue team and other teams that use Scala

I use Pig. Why should I care?

- Pig is good for quick and dirty tasks. Not so good for running production pipelines
- Pig is not a programming language
- Pig scripts can quickly get complicated (lots of copy pasting involved)
- UDFs are a pain

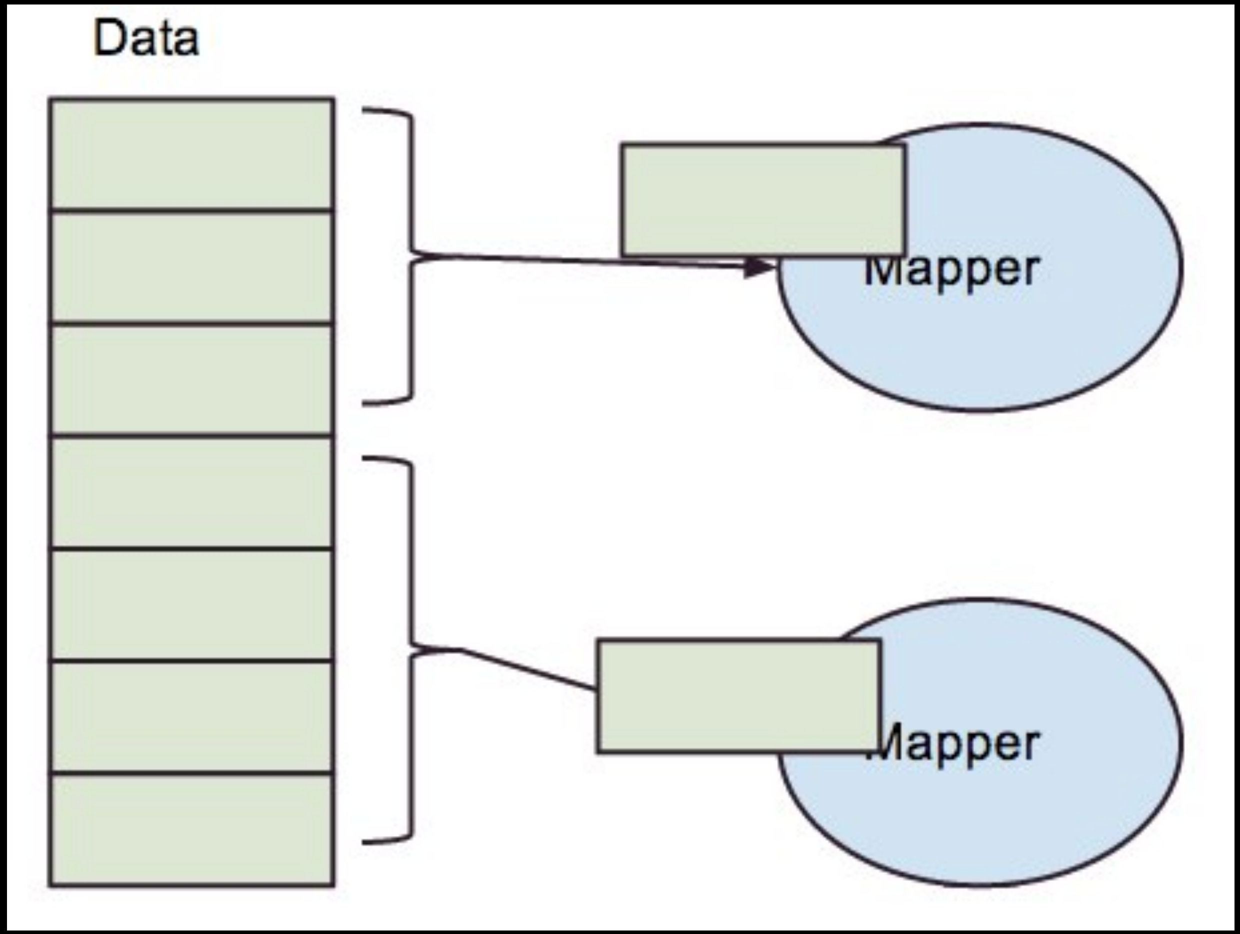
Wordcount in Java

```java
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
           word.set(tokenizer.nextToken());
           context.write(word, one);
       }
   }
}

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));
   }
}
```

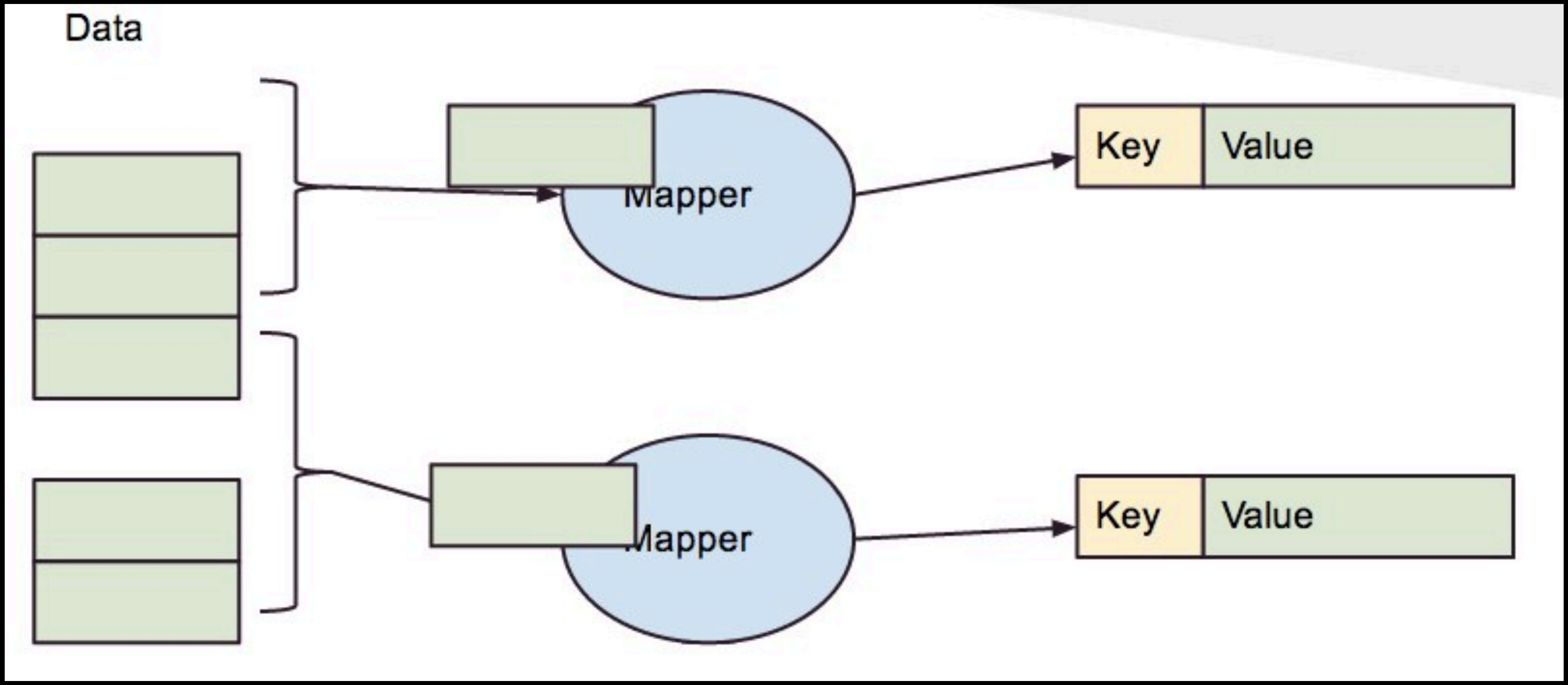
Wordcount in Scalding

```scala
class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String => tokenize(line) }
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").split("\\s+")
  }
}
```

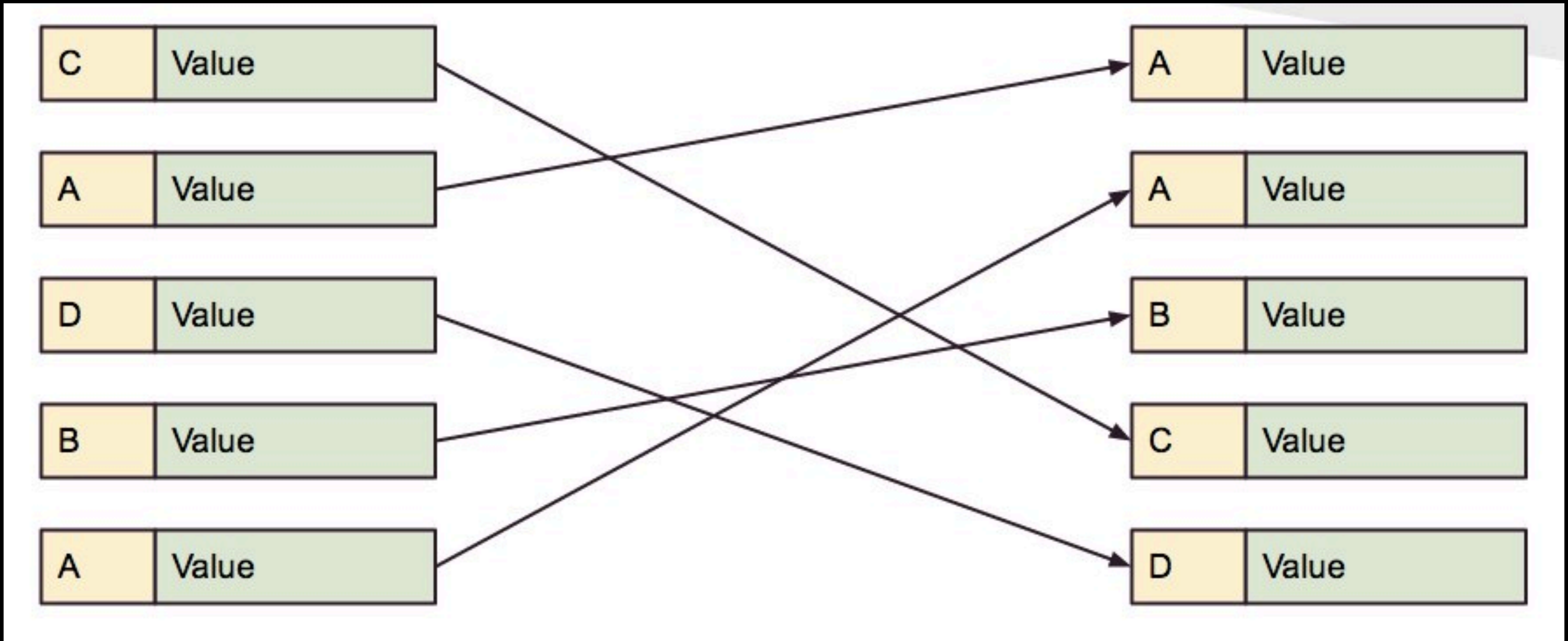
Recap: MapReduce



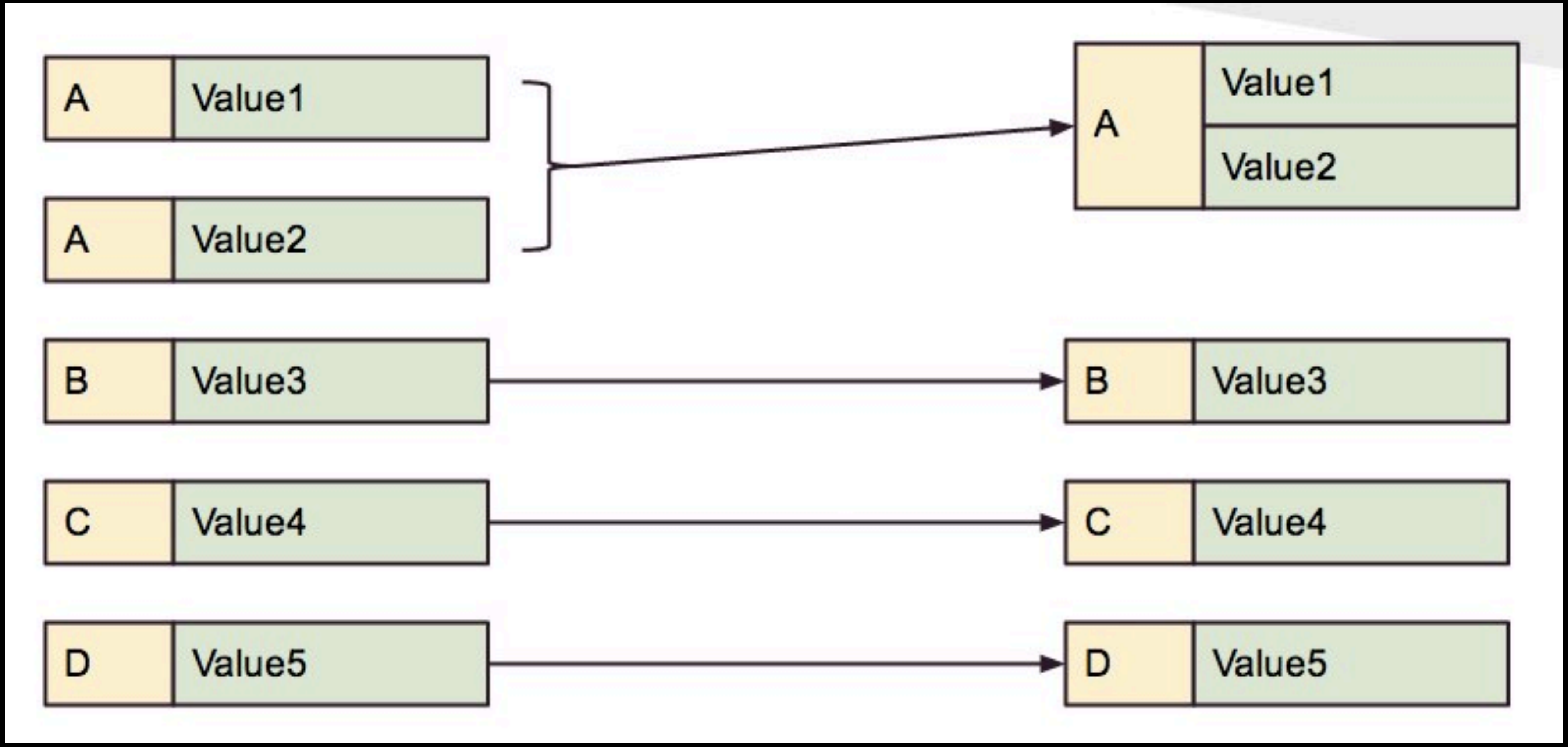
Map



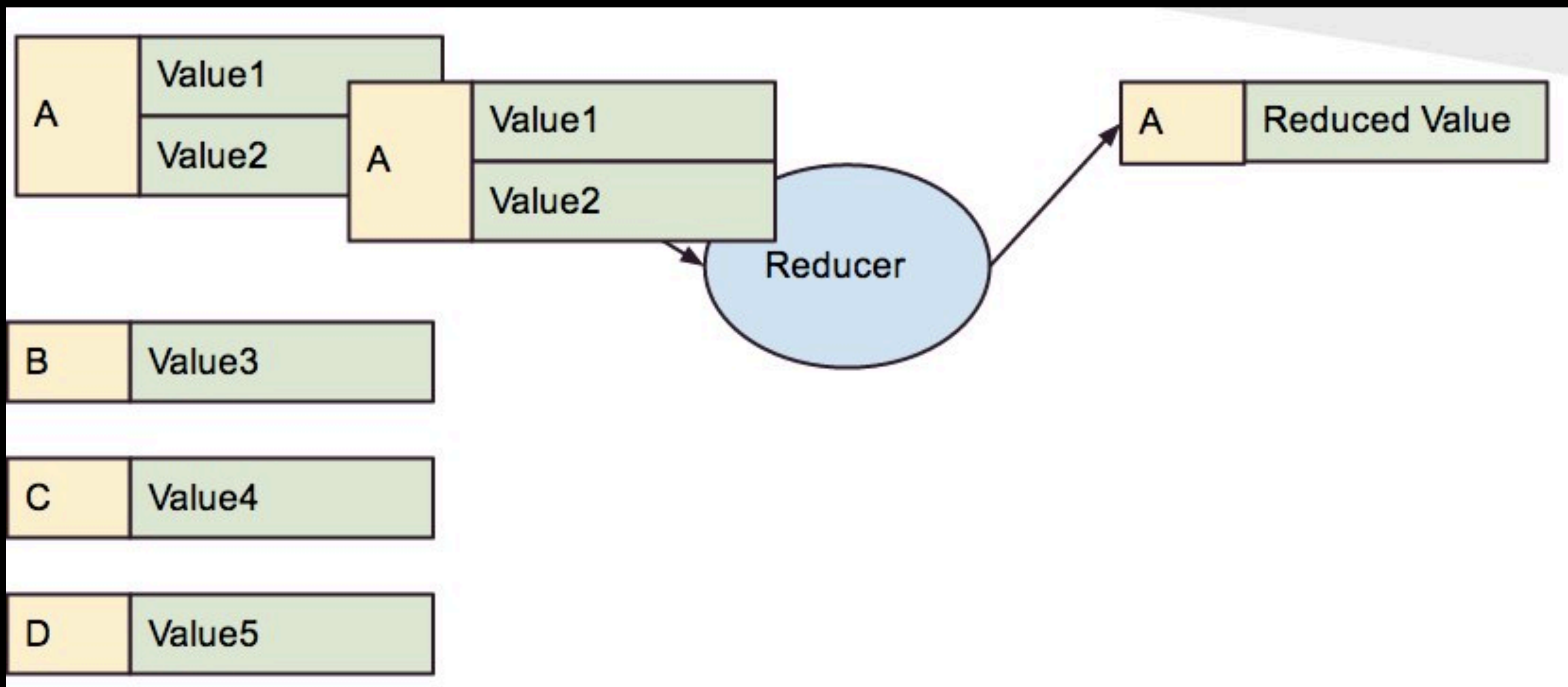
Sort


C
Value
A
Value
D
Value
B
Value
A
Value
A
Value
A
Value
B
Value
C
Value
D
Value

Combine


A
Value1
A
Value2
B
Value3
C
Value4
D
Value5
A
Value1
Value2
B
Value3
C
Value4
D
Value5

Reduce



A
Value1
Value2
A
Value1
Value2
Reducer
A
Reduced Value
B
Value3
C
Value4
D
Value5

Too low level

- Hard to build giant data processing pipelines if you can only think in terms of map and reduce
- There's a need for a different language to express high level user defined functions
- However going to Hive or Pig means that you lose flexibility

Yep, we're counting words:

Scalding jobs subclass Job

```
package com.twitter.scalding.examples

import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String => tokenize(li
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").spli
  }
}
```

Yep, we're counting words:

Logic is in the constructor

```
package com.twitter.scalding.examples

import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String => tokenize(
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").sp
  }
}
```

Yep, we're counting words:

```scala
package com.twitter.scalding.examples

import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String => tokenize(line) }
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").split("\\s+"
  }
}
```

Functions can be called or defined inline

Scalding Model

- Source objects read and write data (from HDFS, DBs, MemCache, etc...)
- Pipes represent the flows of the data in the job. You can think of Pipe as a distributed list.

Yep, we're counting words:

```
package com.twitter.scalding.examples

import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String =>
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words
  def tokenize(text : String) : Array[String] =
    // Lowercase each word and remove punctuatio
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]
  }
}
```

Read and Write data through Source objects

Yep, we're counting words:

Data is modeled as streams of named Tuples (of objects)

```
package com.twitter.scalding.examples

import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String => tok
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]",
  }
}
```

Why Scala

- The scala language has a lot of built-in features that make domain-specific languages easy to implement.
- Map/Reduce is already within the functional paradigm.
- Scala’s collection API covers almost all usual use cases.

Word Co-occurrence

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

class WordCooccur(args : Args) extends Job(args) {

  val interestingWords = Set("alice", "hatter", "rabbit")

  TextLine(args("input"))
    .flatMapTo('word, 'seenwith) { line =>
      val words = line.split("\\s+")
        .map { _.toLowerCase }

      for(word0 <- words.zipWithIndex if interestingWords.contains(word0._1);
          word1 <- words.zipWithIndex if (word1._2 != word0._2))
        yield (word0._1, Map(word1._1 -> 1))
    }
    // plus for maps does plus on the values for each key (merge the maps)
    // It's better to groupBy the pair of words and count, but we're showing off
    .groupBy('word) { _.plus[Map[String, Int]]('seenwith) }
    // Flatten it out
    .flatMap('seenwith -> ('otherWord, 'count)) { seenwith : Map[String,Int] =>
      seenwith.toStream
    }
    .project('word, 'otherWord, 'count)
    .write(Tsv(args("output")))
}
```

Word Co-occurrence

We can use standard scala containers

```
import com.twitter.scalding._

class WordCooccur(args : Args) extends Job(args) {

  val interestingWords = Set("alice", "hatter", "rab

  TextLine(args("input"))
    .flatMapTo('word, 'seenwith) { line =>
      val words = line.split("\\s+")
        .map { _.toLowerCase }

      for(word0 <- words.zipWithIndex if interesting
          word1 <- words.zipWithIndex if (word1._2 !
        yield (word0._1, Map(word1._1 -> 1))
    }
    // plus for maps does plus on the values for each
    // It's better to groupBy the pair of words and
    .groupBy('word) { _.plus[Map[String, Int]]('seen
    // Flatten it out
```

Word Co-occurrence

We can do real logic in the mapper without external UDFs.

```
import com.twitter.scalding._

class WordCooccur(args : Args) extends Job(args

  val interestingWords = Set("alice", "hatter",

  TextLine(args("input"))
    .flatMapTo('word, 'seenwith) { line =>
      val words = line.split("\\s+")
        .map { _.toLowerCase }

      for(word0 <- words.zipWithIndex if intere
          word1 <- words.zipWithIndex if (word1
        yield (word0._1, Map(word1._1 -> 1))
    }
    // plus for maps does plus on the values fo
    // It's better to groupBy the pair of words
    .groupBy('word) { _.plus[Map[String, Int]](
```

Word Co-occurrence

Generalized "plus" handles lists/sets/maps and can be customized (implement Monoid[T])

```
import com.twitter.scalding._

class WordCooccur(args : Args) extends Job(args) {

  val interestingWords = Set("alice", "hatter", "rabbit")

  TextLine(args("input"))
    .flatMapTo('word, 'seenwith) { line =>
      val words = line.split("\\s+")
        .map { _.toLowerCase }

      for(word0 <- words.zipWithIndex if interestingWords.c
          word1 <- words.zipWithIndex if (word1._2 != word0
        yield (word0._1, Map(word1._1 -> 1))
    }
    // plus for maps does plus on the values for each key (
    // It's better to groupBy the pair of words and count, 
    .groupBy('word) { _.plus[Map[String, Int]]('seenwith) }
    // Flatten it out
    .flatMap('seenwith -> ('otherWord, 'count)) { seenwith 
      seenwith.toStream
    }
    .project('word, 'otherWord, 'count)
    .write(Tsv(args("output")))
}
```

GroupBuilder: enabling parallel reductions

```
class SizeAveStdJob(args : Args) extends Job(args) {
  TextLine(args("input")).mapTo('x,'y) { line =>
    val p = line.split(" ").map { _.toDouble }.slice(0,2)
    (p(0),p(1))
  }.map('x -> 'x) { (x : Double) => (4 * x).toInt }
  .groupBy('x) {
    _.sizeAveStdev('y->('size,'yave,'ystdev))
    //Make sure this doesn't ruin the calculation
    .sizeAveStdev('y->('size2,'yave2,'ystdev2))
    .average('y)
  }
  .project('x,'size,'yave,'ystdev,'y)
  .write(Tsv(args("output")))
}
```

- groupBy takes a function that mutates a GroupBuilder.
- GroupBuilder adds fields which are reductions of (potentially different) inputs.
- On the left, we add 7 fields.

scald.rb

- driver script that compiles the job and runs it locally or transfers and runs remotely.
- we plan to add EMR support.

```
[oscar@Macintosh-040cce219376 ~/years/2012/proj/scalding_talk/code]$ scald.rb --local WordCount.scala --input pg11.txt --output wordcnt0.out
Apr 10, 2012 3:28:21 PM cascading.util.Version printBanner
INFO: Concurrent, Inc - Cascading 2.0.0 [hadoop-0.20.2+]
Apr 10, 2012 3:28:21 PM cascading.flow.Flow logInfo
INFO: [] starting
Apr 10, 2012 3:28:21 PM cascading.flow.Flow logInfo
INFO: []  source: FileTap["TextLine[['num', 'line']->[ALL]]"]["pg11.txt"]"]
Apr 10, 2012 3:28:21 PM cascading.flow.Flow logInfo
INFO: []  sink: FileTap["TextDelimited[[UNKNOWN]->[ALL]]"]["wordcnt0.out"]"]
Apr 10, 2012 3:28:21 PM cascading.flow.Flow logInfo
INFO: []  parallel execution is enabled: true
Apr 10, 2012 3:28:21 PM cascading.flow.Flow logInfo
INFO: []  starting jobs: 1
Apr 10, 2012 3:28:21 PM cascading.flow.Flow logInfo
INFO: []  allocating threads: 1
Apr 10, 2012 3:28:21 PM cascading.flow.planner.FlowStep logInfo
[oscar@Macintosh-040cce219376 ~/years/2012/proj/scalding_talk/code]$
```

API Reference

Scalding functions can be divided into three types:

- **Map-like** functions
- **Grouping/reducing** functions
- **Join** operations

Map-like functions

Map-like functions operate over individual rows in a pipe, usually transforming them in some way. They are defined in `RichPipe.scala`.

map, flatMap

pipe.**map**(*existingFields -> additionalFields*){*function*}

Adds new fields that are transformations of existing ones.

```
// In addition to the existing `speed` field, the new `fasterBirds`
// pipe will contain a new `doubledSpeed` field (plus any other
// fields that `birds` already contained).
val fasterBirds = birds.map('speed -> 'doubledSpeed) { speed : Int => speed * 2 }
```

You can also map from and to multiple fields at once.

```
val britishBirds =
  birds.map(('weightInLbs, 'heightInFt) -> ('weightInKg, 'heightInMeters)) {
    x : (Float, Float) =>
    val (weightInLbs, heightInFt) = x
    (0.454 * weightInLbs, 0.305 * heightInFt)
  }
```

pipe.**flatMap**(*originalFields -> newFields*){*function*}

Maps each element to a list (or an `Option`), and then flattens that list

Most functions in the API have very close analogs in scala.collection.Iterable.

mapReduceMap

- We abstract Cascading's map-side aggregation ability with a function called mapReduceMap.
- If only mapReduceMaps are called, map-side aggregation works. If a foldLeft is called (which cannot be done map-side), scalding falls back to pushing everything to the reducers.

Most Reductions are mapReduceMap

```
// This is count with a predicate: only counts the tuples for which fn(tuple) is true
def count[T:TupleConverter](fieldDef : (Fields, Fields))(fn : T => Boolean) : GroupBuilder = {
  mapReduceMap[T,Long,Long](fieldDef)(arg => if(fn(arg)) 1L else 0L)((s1 : Long, s2 : Long) => s1+s2)(s => s)
}
```

```
def forall[T:TupleConverter](fieldDef : (Fields,Fields))(fn : (T) => Boolean) : GroupBuilder = {
  mapReduceMap(fieldDef)(fn)({(x : Boolean, y : Boolean) => x && y})({ x => x })
}
```

```
def average(f : (Fields, Fields)) : GroupBuilder = {
  mapReduceMap(f){(x:Double) =>
    (1L, x)
  } {(cntAve1, cntAve2) =>
    val (big, small) = if (cntAve1._1 >= cntAve2._1) (cntAve1, cntAve2) else (cntAve2, cntAve1)
    val n = big._1
    val k = small._1
    val an = big._2
    val ak = small._2
    val newCnt = n+k
    val scaling = k.toDouble/newCnt
    // a_n + (a_k - a_n)*(k/(n+k)) is only stable if n is not approximately k
    val newAve = if (scaling < STABILITY_CONSTANT) (an + (ak - an)*scaling) else (n*an + k*ak)/newCnt
    (newCnt, newAve)
  } { res => res._2 }
}
```

```scala
// Equivalent to sorting by a comparison function
// then take-ing k items.  This is MUCH more efficient than doing a total sort followed by a take,
// since these bounded sorts are done on the mapper, so only a sort of size k is needed.
// example:
// sortWithTake( ('clicks, 'tweet) -> 'topClicks, 5) { fn : (t0 :(Long,Long), t1:(Long,Long) => t0._1 < t1._1 }
// topClicks will be a List[(Long,Long)]
def sortWithTake[T:TupleConverter](f : (Fields, Fields), k : Int)(lt : (T,T) => Boolean) : GroupBuilder = {
  assert(f._2.size == 1, "output field size must be 1")
  mapReduceMap(f) /* map1 */ { (tup : T) => List(tup) }
  /* reduce */ { (l1 : List[T], l2 : List[T]) =>
    mergeSorted(l1, l2, lt, k)
  } /* map2 */ {
    (lout : List[T]) => lout
  }
}
```

```scala
def toList[T](fieldDef : (Fields, Fields))(implicit conv : TupleConverter[T]) : GroupBuilder = {
  val (fromFields, toFields) = fieldDef
  conv.assertArityMatches(fromFields)
  val out_arity = toFields.size
  assert(out_arity == 1, "toList: can only add a single element to the GroupBuilder")
  mapReduceMap[T, List[T], List[T]](fieldDef) { //Map
    // TODO this is questionable, how do you get a list including nulls?
    x => if (null != x) List(x) else Nil
  } { //Reduce, note the bigger list is likely on the left, so concat into it:
    (prev, current) => current ++ prev
  } { //Map
    t => t
  }
}
```

Scalding @Twitter

- Revenue quality team (ads targeting, market insight, click-prediction, traffic-quality) uses scalding for all our work.
- Scala engineers throughout the company use it (i.e. storage, platform).
- More than 100 in-production scalding jobs, hundreds of ad-hoc jobs.
- Not our only tool: Pig, PyCascading, Cascalog, Hive are also used.

Example: finding similarity

- A simple recommendation algorithm is cosine similarity.
- Represent user-tweet interaction as a vector, then find the users whose vectors point in directions near the user in question.
- We’ve developed a Matrix library on top of scalding to make this easy.

Cosine Similarity

Matrices are strongly typed.

```scala
import com.twitter.scalding._
// Matrix is not *yet* on github
import com.twitter.pluck.mathematics.Matrix._

class Cosine(args : Args) extends Job(args) {

  val matrix = Tsv(args("input"))
    .toMatrix[Int,Int,Double](0,1,2)
    .rowL2Normalize

  (matrix * matrix.transpose)
    .write(Tsv(args("output")))
}
```

Cosine Similarity

Col,Row types (Int,Int) can be anything comparable. Strings are useful for text indices.

```
import com.twitter.scalding._
// Matrix is not *yet* on github
import com.twitter.pluck.mathematics.Matrix._

class Cosine(args : Args) extends Job(args) {

  val matrix = Tsv(args("input"))
    .toMatrix[Int,Int,Double](0,1,2)
    .rowL2Normalize

  (matrix * matrix.transpose)
    .write(Tsv(args("output")))
}
```

Cosine Similarity

Value (Double) can be anything with a Ring[T] (plus/times)

```scala
import com.twitter.scalding._
// Matrix is not *yet* on github
import com.twitter.pluck.mathematics.Matrix._

class Cosine(args : Args) extends Job(args) {

  val matrix = Tsv(args("input"))
    .toMatrix[Int,Int,Double](0,1,2)
    .rowL2Normalize

  (matrix * matrix.transpose)
    .write(Tsv(args("output")))
}
```

Cosine Similarity

Operator overloading gives intuitive code.

```
import com.twitter.scalding._
// Matrix is not *yet* on github
import com.twitter.pluck.mathematics.Matrix._

class Cosine(args : Args) extends Job(args) {

  val matrix = Tsv(args("input"))
    .toMatrix[Int,Int,Double](0,1,2)
    .rowL2Normalize

  (matrix * matrix.transpose)
    .write(Tsv(args("output")))
}
```

Matrix in foreground, map/reduce behind

With this syntax, we can focus on logic, not how to map linear algebra to Hadoop

```
class ScalarOps(args: Args) extends Job(args) {
  import Matrix._
  val p1 = Tsv("mat1",('x1,'y1,'v1)).read
  val mat1 = new Matrix[Int,Int,Double]('x1,'y1,'v1, p1)
  (mat1 * 3.0).pipe.write(Tsv("times3"))
  (mat1 / 3.0).pipe.write(Tsv("div3"))
  (3.0 * mat1).pipe.write(Tsv("3times"))
  // Now with Scalar objects:
  (mat1.trace * mat1).pipe.write(Tsv("tracetimes"))
  (mat1 * mat1.trace).pipe.write(Tsv("timestrace"))
  (mat1 / mat1.trace).pipe.write(Tsv("divtrace"))
}
```

Example uses:

- Do random-walks on the following graph. Matrix power iteration until convergence: (m * m * m * m).
- Dimensionality reduction of follower graph (Matrix product by a lower dimensional projection matrix).
- Triangle counting: (M*M*M).trace / 3

That’s it.

- follow and mention: @scalding @argyris @posco
- pull reqs: http://github.com/twitter/scalding

Demo