## Contents

1 Tutorials 3  
1.1 Beginner .......................................................... 3  
1.2 Intermediate ......................................................... 13  
1.3 Advanced ........................................................... 23  

2 Reference 27  
2.1 API Reference ........................................................ 27  
2.2 Calling Scalding from Inside your Application ................. 56  
2.3 Common Exceptions and Possible Reasons ...................... 57  
2.4 Frequently Asked Questions ...................................... 59  
2.5 Pig to Scalding ....................................................... 63  
2.6 REPL Reference ..................................................... 69  
2.7 Rosetta Code ......................................................... 70  
2.8 Scald.rb ............................................................. 74  
2.9 Scalding Commons .................................................. 75  
2.10 Scalding-HBase ...................................................... 75  
2.11 Scalding Sources ................................................... 76  
2.12 Using Scalding with other versions of Scala .................. 76  

3 Other 77  
3.1 Difference between the Fields API of scalding and Scoobi and Scrunch ..... 77  
3.2 Field Rules .......................................................... 78  
3.3 Powered By Scalding ............................................... 78  
3.4 Run in IntelliJ Idea .................................................... 80  
3.5 Scala and SBT for Homebrew users ............................... 80  
3.6 Scala and SBT for Homebrew users ............................... 80  
3.7 Scalding on Amazon Elastic MapReduce ......................... 81  
3.8 Scalding with CDH3U2 in a Maven Project ..................... 82  
3.9 Upgrading to 0.9.0 ................................................... 83  
3.10 Using the Distributed Cache .................................... 83  
3.11 Why “Pack, Unpack” and not “toList” .......................... 84
Contents:
CHAPTER 1

Tutorials

Contents:

1.1 Beginner

Contents:

1.1.1 Getting Started

Installation

To get started with Scalding, first clone the Scalding repository on Github:

```
git clone https://github.com/twitter/scalding.git
```

Next, build the code using sbt (a standard Scala build tool). Make sure you have Scala (download here, see scalaVersion in project/Build.scala for the correct version to download), and run the following commands:

```
./sbt update
./sbt test   # runs the tests; if you do `sbt assembly' below, these tests, which are long
./sbt assembly # creates a fat jar with all dependencies, which is useful when using the scald.rb script
```

Now you're good to go!

Using Scalding with other versions of Scala

Scalding works with Scala 2.9 and 2.10, though a few configuration files must be changed for this to work. In project/Build.scala, ensure that the proper scalaVersion value is set. Additionally, you'll need to ensure the proper version of specs in the same config. Change the following line

```
libraryDependencies += ```org.scala-tools.testing'' % ```specs_2.9.1'' % ```1.6.9'' % ```test''
```

to correspond to the proper version of scala (2.9.1 should work with scala 2.9.2). You can find the published versions here.
IDE Support

Scala’s IDE support is generally not as strong as Java’s, but there are several options that some people prefer. Both Eclipse and IntelliJ have plugins that support Scala syntax. To generate a project file for Scalding in Eclipse, refer to this project, and for IntelliJ files, this (note that with the latter, the 1.1 snapshot is recommended).

1.1.2 Scalding REPL

In addition to production batch jobs, Scalding can be run interactively to help develop ad-hoc queries or give beginners a chance to learn about the platform step-by-step.

The Tutorial will walk through using the REPL in depth. Users already familiar with Scalding may wish to skip to the summary of REPL functionality below.

Quick Start

Starting up the REPL in local mode is as easy as:

> ./sbt ``scalding-repl/run --local''

To run full HDFS mode on a remote machine, you must build the assembly jar and use the scalp.rb script to rsync and ssh to the right machine:

> ./sbt assembly
> ./scripts/scald.rb --repl --hdfs --host <host to ssh to and launch jobs from>

Tutorial

Assuming you’ve checked out the Scalding code, the fastest way to start running Scalding is to launch the REPL in Local mode. Simply run:

> ./sbt ``scalding-repl/run --local''

It will spend some time downloading and compiling, but should eventually show:

[info] Set current project to scalding (in build file:/Users/bholt/hub/scalding/)
import com.twitter.scalding._
import com.twitter.scalding.ReplImplicits._
import com.twitter.scalding.ReplImplicitContext._
... scalding>

As you can see, we’ve imported a bunch of Scalding code, including implicits that make it easier to run jobs interactively. Several of these enhancements are “enrichments” on Scalding’s TypedPipe. The tutorial will go into them in more detail.

Let’s take a look at some.

Viewing pipe contents

First, dump allows you to print what’s in a TypedPipe:

scala> // load a plain text file into a TypedPipe
       | val hello = TypedPipe.from(TextLine(``tutorial/data/hello.txt''))
We can also load the contents into memory as a list:

```
scala> val lst = hello.toList
lst: List[String] = List(Hello world, Goodbye world)
```

In fact, both `dump` and `toList` use another TypedPipe enrichment, `toIterator`, which can be used directly if one wishes to stream over the data and perform some operation.

Remember that each of these operations run locally, so if you are actually working with large datasets, ensure that you do most of your work using TypedPipe operations, which will run through Cascading/Hadoop, and only call `toIterator` (or `dump` or `toList`) once you’ve filtered the results sufficiently. Some helpful TypedPipe operations to downsize datasets are `limit` and `sample`.

### Save

Usually in Scalding, results are written to a Sink using `.write()`. However, the write only happens when the flow is actually run. That way you can build up a complicated flow with multiple sources and sinks and intermediate calculations and run it as one big job. When running interactively, however, we want a way to immediately run part of a job to see if it’s correct before moving on. When running in the REPL, `save` can be used in place of `write` to immediately run that pipe and write out the results.

```
scala> val hello = TypedPipe.from(TextLine(``tutorial/data/hello.txt''))
hello: com.twitter.scalding.typed.TypedPipe[String] = com.twitter.scalding.typed.TypedPipeFactory@6d2e147b

scala> val words = hello.flatMap(_.split(``\s+'')).map(_.toLowerCase)
words: com.twitter.scalding.typed.TypedPipe[String] = com.twitter.scalding.typed.TypedPipeFactory@69f4e6e4
```

Conviently, this returns a new pipe reading from the file which can be used in subsequent computations.

### Snapshots

When working interactively with large datasets, it is often useful to be able to save and re-use data that took significant work to generate. The Scalding REPL provides an easy way to explicitly save intermediate results and use these results in subsequent interactive work.

The `snapshot` enrichment on TypedPipe runs everything necessary to generate the output for the given pipe and saves it to temporary storage. In Local mode, it actually just saves it in memory; in Hadoop, it writes it out to a temporary SequenceFile. Scalding then returns a handle to this new Source as a new TypedPipe.

Snapshots should only be used within a single REPL session. In local mode, you have no choice because they only exist in memory. But even for Hadoop mode, it is not a good idea to try to re-use generated snapshots, as Kryo serialization is unstable and the temporary files may be cleaned up when not in use.

```
scala> // (using `hello' TypedPipe declared above)
   | // split into words, returns TypedPipe all set to do the work (but not run yet)
   | val words = hello.flatMap(_.split(``\s+'')).map(_.toLowerCase)
words: com.twitter.scalding.typed.TypedPipe[String] = com.twitter.scalding.typed.TypedPipeFactory@69f4e6e4

scala> // save a snapshot
   | val s = words.snapshot
s: com.twitter.scalding.TypedPipe[String] = IterablePipe(List(hello, world, goodbye, world))

scala> // now take a look at what's in the snapshot
```
You can see to create the snapshot, it ran a small Cascading job, and returned a TypedPipe with the result to us. Now, rather than spending that enormous amount of time to re-generate “words”, we can use s, our snapshot, to, for a change, count some words:

```scala
scala> val wordCount = s.map((_, 1)).sumByKey
    wordCount: com.twitter.scalding.typed.UnsortedGrouped[String,Int] = IteratorMappedReduce(scala.math.Ordering$String$@2c20bf29,com.twitter.scalding.typed.TypedPipeFactory@6d0d4bf5,<function2>,None,List(.<init>(<console>:20))

scala> wordCount.snapshot.dump
    (goodbye,1)
    (hello,1)
    (world,2)
```

Notice above how running `wordCount` with `snapshot` used the MemoryTap (snapshot) created before as its source. If, instead, we wanted to run the entire job end-to-end, we simply need to bypass the snapshots and use the original pipes:

```scala
scala> // remember, `words` was a TypedPipe which reads from the original text file
    | words
    res9: com.twitter.scalding.typed.TypedPipe[String] = com.twitter.scalding.typed.TypedPipeFactory@69f4e6e4

scala> words.map((_, 1)).sumByKey.dump
    (goodbye,1)
    (hello,1)
    (world,2)
```

See, that time the “source” was the original text file.

**Implicit snapshots** The previous example actually pulled out a subtle additional trick. Calling `dump` on the result of `sumByKey` ran the flow and printed the results. Because you may be dealing with large datasets, we want to run these pipes using Cascading in Hadoop. The `toIterator` method, which is called by `dump`, doesn’t want to have to iterate over excessively large datasets locally, so it calls `snapshot`, which runs the flow in Cascading, and then `toIterator` can iterate over the snapshot. However, each call to `dump` as above will need to re-run, because we aren’t keeping a handle to the snapshot around. If a flow is taking a while to run, try saving the results to a snapshot first, then using that snapshot to further understand the data.

**Running on Hadoop**

So far, we’ve only been running in Cascading’s Local mode, which emulates what Hadoop will do, but cannot handle actually large datasets. The easiest way to launch the REPL in Hadoop mode is to use the `scald.rb` script.

```bash
# must first create the massive assembly jar with all the code
scalding$ ./sbt assembly
# then launch the REPL
scalding$ ./scripts/scald.rb --repl --hdfs
```

**1.1.3 Alice in Wonderland Tutorial**

First, let’s import some stuff.
import scala.io.Source
import com.twitter.scalding._
import com.twitter.scalding.ReplImplicits._
import com.twitter.scalding.ReplImplicitContext._

scala> val alice = Source.fromURL(`https://raw.githubusercontent.com/mihi-tr/reading-alice/master/pg28885.txt'').getLines
alice: Iterator[String] = non-empty iterator

Add the line numbers, which we might want later

scala> val aliceLineNum = alice.zipWithIndex.toList
aliceLineNum: List[(String, Int)] = List((Project Gutenberg`s Alice`s Adventures in Wonderland, by Lewis Carroll,0), ... English,17), (``'',18), (``'',19), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVENTURES IN WONDERLAND **...,16), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,15), (,16), (Language: English,17), (,18), (,19), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,14), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,13), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,12), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,11), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,10), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,9), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,8), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,7), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,6), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,5), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,4), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,3), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,2), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,1), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,0)

Now for scalding. TypedPipe is the main scalding object representing your data.

scala> val alicePipe = TypedPipe.from(aliceLineNum)
alicePipe: com.twitter.scalding.typed.TypedPipe[(String, Int)] = IterablePipe(List((Project Gutenberg`s Alice`s Adventures in Wonderland, by Lewis Carroll,0), ... English,17), (``'',18), (``'',19), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVENTURES IN WONDERLAND **...,16), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,15), (,16), (Language: English,17), (,18), (,19), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,14), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,13), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,12), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,11), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,10), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,9), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,8), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,7), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,6), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,5), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,4), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,3), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,2), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,1), ( *** START OF THIS PROJECT GUTENBERG EBOOK ALICE`S ADVEN...,0))

Three things: map, function, tuples but that’s ugly, so we can use tuple matching the be clearer:

scala> val aliceWordList = alicePipe.map { case (text, lineNum) =>
| text.split(```\s+'`).toList
| }

But we want words, not lists of words. We need to flatten!

scala> val aliceWords = aliceWordList.flatten
aliceWords: com.twitter.scalding.typed.TypedPipe[String] = com.twitter.scalding.typed.TypedPipeFactory@7fa0e6fc

Scala has a common function for this map + flatten == flatMap

scala> val aliceWords = alicePipe.flatMap { case (text, _) => text.split(````\s+'`).toList }

Now lets add a count for each word:

scala> val aliceWithCount = aliceWords.map { word => (word, 1L) }
aliceWithCount: com.twitter.scalding.typed.TypedPipe[(String, Long)] = com.twitter.scalding.typed.TypedPipeFactory@8267443

Let’s sum them for each word:

scala> val wordCount = aliceWithCount.group.sum
wordCount: com.twitter.scalding.typed.UnsortedGrouped[String,Long] = IteratorMappedReduce(scala.math.Ordering$String$@2c20bf29,com.twitter.scalding.typed.TypedPipeFactory@2880d49b,<function2>,None,List(.<init>(<console>:23))

(We could have also used .sumByKey, which is equivalent to .group.sum.)

Let’s print them to the screen (REPL only):

scala> wordCount.toIterator.take(100)
res0: Iterator[(String, Long)] = non-empty iterator

Let’s print just the ones with more that 100 appearances:

scala> wordCount.filter { case (word, count) => count > 100 }.dump
(,,1399)
(```I,120)
(Alice,224)
(I,248)
(a,678)
(all,171)
But which is the biggest word?

Hint: In the Scala REPL, you can turn on `:paste` mode to make it easier to paste multi-line expressions.

```
scala> val top10 = { wordCount
| .groupAll
| .sortBy { case (word, count) => -count }
| .take(10) }
```

Where is Alice? What is with the ()?

```
scala> val top20 = { wordCount
| .groupAll
| .sortBy { case (word, count) => -count }
```
top20: com.twitter.scalding.typed.TypedPipe[(String, Long)] = com.twitter.scalding.typed.TypedPipeFactory@3a2c2b1a

scala> top20.dump
(the,1694)
(,1399)
(to,794)
(and,793)
(a,678)
(of,604)
(she,489)
(said,422)
(in,405)
(it,365)
(was,332)
(you,306)
(I,248)
(as,246)
(that,230)
(with,225)
(Alice,224)
(at,209)
(her,207)
(had,177)

There she is!

Now, suppose we want to know the last line on which each word appears.

How do we solve this? First, we generate (word, lineNum) pairs by flatmapping each line of words to a list of (word, lineNum) pairs.

scala> val wordLine = alicePipe.flatMap { case (text, line) =>
  | text.split(`\s+'`).toList.map { word => (word, line) }
  | }
wordLine: com.twitter.scalding.typed.TypedPipe[(String, Int)] = com.twitter.scalding.typed.TypedPipeFactory@6d08b42d

Next, we group the pairs on the word, and take the max line number for each group.

See all the functions on grouped things here: http://twitter.github.io/scalding/#com.twitter.scalding.typed.Grouped

scala> val lastLine = wordLine.group.max
lastLine: com.twitter.scalding.typed.UnsortedGrouped[String,Int] = IteratorMappedReduce(scala.math.Ordering$String$@2c20bf29,com.twitter.scalding.typed.TypedPipeFactory@641aa1ae,<function2>,None,List(.<init>(<console>:22)))

Finally, we lookup the words from the initial line:

By the way: lastLine.swap is equivalent to lastLine.map { case (word, lastLine) => (lastLine, word) }

scala> val words = {
  | lastLine.map { case (word, lastLine) => (lastLine, word) }
  | .group
  | .join(alicePipe.swap.group)
  | }
words: com.twitter.scalding.typed.CoGrouped[Int,(String, String)] = com.twitter.scalding.typed.CoGroupable$$anon$3@5d67c779

scala> println(words.toIterator.take(30).mkString(`
''))
(0,(Project,Project Gutenberg’s Alice’s Adventures in Wonderland, by Lewis Carroll))
1.1.4 Intro to Scalding Jobs

WordCount in Scalding

Let’s look at a simple WordCount job.

```scala
class WordCountJob(args: Args) extends Job(args) {
  TypedPipe.from(TextLine(args(``input'')))   
    .flatMap { line => line.split(`` '' ``)   
    .groupBy { word => word }   
    .size   
    .write(TypedTsv(args(``output''))) 
}
```

This job reads in a file, emits every word in a line, counts the occurrences of each word, and writes these word-count pairs to a tab-separated file.

To run the job, copy the source code above into a `WordCountJob.scala` file, create a file named `someInputfile.txt` containing some arbitrary text, and then enter the following command from the root of the Scalding repository:

```
```

That's it. You have learned the basics: TypedPipe, map/flatMap/filter groups do reduce/join: max, sum, join, take, sortBy
scripts/scald.rb --local WordCountJob.scala --input someInputfile.txt --output ./someOutputFile.tsv

This runs the WordCount job in local mode (i.e., not on a Hadoop cluster). After a few seconds, your first Scalding job should be done!

**WordCount dissection**

Let’s take a closer look at the job.

**TextLine**

TextLine is an example of a Scalding source that reads each line of a file into a field named line.

```
TextLine(args(``input'')) // args(``input'') contains a filename to read from
```

Another common source is a TypedTsv source that reads tab-delimited files. You can also create sources that read directly from LZO-compressed files on HDFS (possibly containing Protobuf- or Thrift-encoded objects!), or even database sources that read directly from a MySQL table. See the scalding-commons module for lzo, thrift and protobuf support. Also see scalding-avro if you use avro.

**flatMap**

flatMap is an example of a function that you can apply to a stream of tuples.

```
TypedPipe.from(TextLine(args(``input''))) // flat map the `line'' field to a new words separated by one or more space
   .flatMap { line => line.split(``\s+'\s+''``) }
```

The above works just like calling flatMap on a List in scala.

Our tuple stream now contains something like the following:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>this is a line</td>
<td>this</td>
</tr>
<tr>
<td>line 2</td>
<td>is</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>line</td>
</tr>
<tr>
<td></td>
<td>line</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

See the [[Type-safe api reference]] for more examples of flatMap (including how to flat map from and to multiple fields), as well as examples of other functions you can apply to a tuple stream.

**groupBy**

Next, we group the same words together, and count the size of each group.

```
TypedPipe.from(TextLine(args(``input'')))  
   .flatMap { line => line.split(``\s+''``) }
   .groupBy { word => word }
   .size
```

Here, we group the stream into groups of tuples with the same word, and then we make the value for each keyed group the size of that group.

The tuple stream now looks like:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Beginner</td>
<td>11</td>
</tr>
</tbody>
</table>
(line, 2)
(this, 1)
(is, 1)
...
Again, see the [[Type-safe api reference]] for more examples of grouping functions.

write, Tsv

Finally, just as we read from a `TextLine` source, we can also output our computations to a `TypedTsv` source. A `TypedTsv` can see the types it is putting in each column. We could write `TypedTsv[(String, Long)]` below to be sure we are writing what we intend, but scala can usually infer the types if we leave them off (though, when you get in trouble, try adding the types near the compilation error and see if you can get a better message as to what is going on).

```scala
TypedPipe.from(TextLine(args("input")))
  .flatMap { line => line.split("\s+") }
  .groupBy { word => word }
  .size
  .write(TypedTsv(args("output")))
```

scald.rb

The `scald.rb` script in the `scripts/` directory is a handy script that makes it easy to run jobs in both local mode or on a remote Hadoop cluster. It handles simple command-line parsing, and copies over necessary JAR files when running remote jobs.

If you’re running many Scalding jobs, it can be useful to add `scald.rb` to your path, so that you don’t need to provide the absolute pathname every time. One way of doing this is via (something like):

```bash
ln -s scripts/scald.rb $HOME/bin/
```

This creates a symlink to the `scald.rb` script in your `$HOME/bin/` directory (which should already be included in your PATH).

See [[scald.rb]] for more information, including instructions on how to set up the script to run jobs remotely.

Next Steps

You now know the basics of Scalding! To learn more, check out the following resources:

- REPL Example: Try the Alice in Wonderland walkthrough which shows how to use Scalding step by step to learn about the book’s text.
- `tutorial/`: this folder contains an introductory series of runnable jobs.
- [[API Reference]]: includes code snippets explaining different kinds of Scalding functions (e.g., map, filter, project, groupBy, join) and much more.
- [[Matrix API Reference]]: the API reference for the Type-safe Matrix library
- **Cookbook**: Short recipes for common tasks.
1.2 Intermediate

Contents:

1.2.1 Aggregation using Algebird Aggregators

For this tutorial, you need to be using Algebird 0.7.2, 0.8.2 or 0.9 or later. You may need to update your build file (prefer 0.7.2 if you are on scalding 0.11 or scalding 0.12). Scalding 0.13+ comes with algebird 0.9 already.

Aggregators enable creation of reusable and composable aggregation functions. There are three main functions on Aggregator trait.

```scala
trait Aggregator[-A, B, +C] {
  /**
   * Transform the input before the reduction.
   */
  def prepare(input: A): B
  /**
   * Combine two values to produce a new value.
   */
  def reduce(l: B, r: B): B
  /**
   * Transform the output of the reduction.
   */
  def present(reduction: B): C
}
```

## Examples

In this section we will use the data below to show SQL aggregate functions and how to build similar aggregate functions in Scalding. You can run these in the scalding repo by typing: `./sbt "scalding-repl/run --local"` and then use the `.dump` method to print results (or `.get on ValuePipes`).

<table>
<thead>
<tr>
<th>OrderID</th>
<th>OrderDate</th>
<th>OrderPrice</th>
<th>OrderQuantity</th>
<th>CustomerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12/22/2005</td>
<td>160</td>
<td>2</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>08/10/2005</td>
<td>190</td>
<td>2</td>
<td>Johnson</td>
</tr>
<tr>
<td>3</td>
<td>07/13/2005</td>
<td>500</td>
<td>5</td>
<td>Baldwin</td>
</tr>
<tr>
<td>4</td>
<td>07/15/2005</td>
<td>420</td>
<td>2</td>
<td>Smith</td>
</tr>
<tr>
<td>5</td>
<td>12/22/2005</td>
<td>1000</td>
<td>4</td>
<td>Wood</td>
</tr>
<tr>
<td>6</td>
<td>10/2/2005</td>
<td>820</td>
<td>4</td>
<td>Smith</td>
</tr>
<tr>
<td>7</td>
<td>11/03/2005</td>
<td>2000</td>
<td>2</td>
<td>Baldwin</td>
</tr>
</tbody>
</table>

```scala
case class Order(orderId: Int, orderDate: String, orderPrice: Long, orderQuantity: Long, 
                  customerName: String)
val orders = List(
  Order(1, ``12/22/2005'' , 160, 2, ``Smith''),
  Order(2, ``08/10/2005'' , 190, 2, ``Johnson''),
  Order(3, ``07/13/2005'' , 500, 5, ``Baldwin''),
  Order(4, ``07/15/2005'' , 420, 2, ``Smith''),
  Order(5, ``12/22/2005'' , 1000, 4, ``Wood''),
  Order(6, ``10/2/2005''  , 820, 4, ``Smith''),
  Order(7, ``11/03/2005'' , 2000, 2, ``Baldwin''))
```

**Count**

The SQL COUNT function returns the number of rows in a table satisfying the criteria specified in the WHERE clause.
SQL:
SELECT COUNT(*) FROM Orders
WHERE CustomerName = `Smith'

//Scalding:
import com.twitter.algebird.Aggregator.count

TypedPipe.from(orders)
  .aggregate(count(_.customerName == `Smith'))

Output: 3

If you don’t specify a WHERE clause when using COUNT, your statement will simply return the total number of rows in the table

SQL:
SELECT COUNT(*) FROM Orders

//Scalding:
import com.twitter.algebird.Aggregator.size

TypedPipe.from(orders)
  .aggregate(size)

Output: 7

You can also use aggregate functions with Group By.

SQL:
Select CustomerName, Count(CustomerName)
From Orders
Group by CustomerName

//Scalding:
import com.twitter.algebird.Aggregator.size

TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(size)

Output:
(Baldwin,2)
(Johnson,1)
(Smith,3)
(Wood,1)

Sum

The SQL SUM function is used to return the sum of an expression in a SELECT statement

SQL:
SELECT SUM(OrderQuantity)
FROM Orders
GROUP BY CustomerName

//Scalding:
import Aggregator.{ prepareMonoid => sumAfter }

TypedPipe.from(orders)
The SQL MAX function retrieves the maximum numeric value from a column.

SQL:
SELECT CustomerName, MAX(OrderQuantity)
FROM Order
GROUP By CustomerName

//Scalding:
import com.twitter.algebird.Aggregator.max
val maxOp = Aggregator.max[Long].composePrepare { o: Order => o.orderQuantity }

TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(maxOp)

Output:
(Baldwin,5)
(Johnson,2)
(Smith,4)
(Wood,4)

The SQL MIN function selects the smallest number from a column.

SQL:
SELECT CustomerName, MIN(OrderQuantity)
FROM Order
GROUP By CustomerName

//Scalding:
import com.twitter.algebird.Aggregator.minBy
// Rather than using composePrepare, we could also use minBy with andThenPresent:
val minOp = minBy[Order, Long](_.orderQuantity)
  .andThenPresent(_.orderQuantity)

TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(minOp)
**AVG**

The SQL AVG function calculates average value of a numeric column.

SQL:
SELECT CustomerName, AVG(OrderQuantity)
FROM Order
GROUP BY CustomerName

```scala
groupBy(_.customerName)
\times agg(avg)
```

Output:
(Baldwin, 3.5)
(Johnson, 2.0)
(Smith, 2.66)
(Wood, 4.0)

**Distinct**

The SQL DISTINCT function selects distinct values from a column. In Scalding we use a probabilistic data structure called HyperLogLog to calculate distinct values.

SQL:
SELECT DISTINCT CustomerName
FROM Order

//Scalding:
import com.twitter.algebird.HyperLogLogAggregator

```scala
val unique = HyperLogLogAggregator
  .sizeAggregator(bits = 12)
  .composePrepare[Order](_.customerName.getBytes(``UTF-8''))

TypedPipe.from(orders)
  \times agg(unique)
```

Output:
4.0

**Top K**

```scala
import com.twitter.algebird.Aggregator.sortedReverseTake

val topK = sortedReverseTake[Long](2)
  .composePrepare[Order](_.orderQuantity)
```
TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(topK)

Output:
(Baldwin,List(5, 2))
(Johnson,List(2))
(Smith,List(4, 2))
(Wood,List(4))

**Composing Aggregators**

Aggregators can be composed to perform multiple aggregation in one pass.

```scala
import com.twitter.algebird.Aggregator._
val maxOp = maxBy[Order, Long](_.orderQuantity).andThenPresent(_.orderQuantity)
val minOp = minBy[Order, Long](_.orderPrice).andThenPresent(_.orderPrice)
val combinedMetric = maxOp.join(minOp)
```

TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(combinedMetric)

Output:
(Baldwin,(5,500))
(Johnson,(2,190))
(Smith,(4,160))
(Wood,(4,1000))

composition can also be used to combine two or more aggregators to derive a new aggregate function.

```scala
import com.twitter.algebird.Aggregator._
import Aggregator.{ prepareMonoid => sumAfter }
val sumAggregator = sumAfter[Order, Long](_.orderQuantity)
val sizeAggregator = size

/*
 Use more efficient `AveragedValue.aggregator` for AVG calculation. This example
 is only to show how to combine two aggregators.
*/
val avg = sumAggregator.join(sizeAggregator)
  .andThenPresent{ case (sum, count) => sum.toDouble / count.toDouble }
```

TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(avg)

Output:
(Baldwin,3.5)
(Johnson,2.0)
(Smith,2.66)
(Wood,4.0)
you can join up to 22 aggregators by using `GeneratedTupleAggregator`. Example below show calculating Max, Min, Sum, Count, Mean and Standard Deviation in one pass by joining different aggregators.

```scala
import com.twitter.algebird.Aggregator._
import com.twitter.algebird.{GeneratedTupleAggregator, MomentsAggregator, Moments }
import Aggregator.{ prepareMonoid => sumAfter }

val maxOp = maxBy[Order, Long](_.orderPrice)
val minOp = minBy[Order, Long](_.orderPrice)
val sum = sumAfter[Order, Long](_.orderPrice)
val moments = Moments.aggregator.composePrepare[Order](_.orderPrice.toDouble)

val multiAggregator = GeneratedTupleAggregator
  .from4(maxOp, minOp, sum, moments)
  .andThenPresent {
    case (mmax, mmin, ssum, moment) =>
      (mmax.orderPrice, mmin.orderPrice, ssum, moment.count, moment.mean, moment.stddev)
  }

TypedPipe.from(orders)
  .groupBy(_.customerName)
  .aggregate(multiAggregator)

Output:
(Baldwin,(2000,500,2500,2,1250.0,750.0))
(Johnson,(190,190,190,1,190.0,0.0))
(Smith,(820,160,1400,3,466.66,271.46))
(Wood,(1000,1000,1000,1,1000.0,0.0))

1.2.2 SQL to Scalding

Motivation

SQL is a popular language for data analytics. Scalding is a relative newcomer that is more powerful and complex. The goal of this document is to translate commonly used SQL idioms to the Scalding type-safe API (which is preferred over the fields-based API). We are using Vertica SQL variant that is based on PSQL and has support for analytic functions. We have purposely picked trivial example datasets so that it is easy to experiment using the REPL and view intermediate results to get a better understanding of what each method does. More information on how to use the REPL is in [[Scalding REPL]] and Learning Scalding with Alice.

Prerequisites:

- Elementary knowledge of Scala
- Basic ability to decipher types in Scalding methods

You should not expect Scalding to be as intuitive as SQL, but at the same time it is not as hard as it may seem when you see the plethora of classes and methods in the Scalding docs.

To get a deeper understanding of monoids like QTree, please see Learning Algebird Monoids with REPL

```scala
import com.twitter.scalding._
import com.twitter.scalding.ReplImplicitContext._
```
Create datasets

SQL

CREATE TABLE test.allsales(
    state VARCHAR(20),
    name VARCHAR(20),
    sales INT
);

INSERT INTO test.allsales VALUES('CA', 'A', 60);
INSERT INTO test.allsales VALUES('CA', 'A', 20);
INSERT INTO test.allsales VALUES('VA', 'B', 15);
COMMIT;

```
pwagle=> select * from test.allsales;
state | name | sales
-------+------+-------
CA    | A    | 60
VA    | B    | 15
CA    | A    | 20
(3 rows)
```

Scalding

```
scala> case class Sale(state: String, name: String, sale: Int)
defined class Sale
scala> val salesList = List(Sale(``CA'', ``A'', 60), Sale(``CA'', ``A'', 20), Sale(``VA'', ``B'', 15))
salesList: List[Sale] = List(Sale(CA,A,60), Sale(CA,A,20), Sale(VA,B,15))
scala> val salesPipe = TypedPipe.from(salesList)
salesPipe: com.twitter.scalding.typed.TypedPipe[Sale] = IterablePipe(List(Sale(CA,A,60), Sale(CA,A,20), Sale(VA,B,15)))
```

Simple Count

SQL

```
pwagle=> select count(1) from test.allsales;
count
-------
3
```

Scalding

```
scala> salesPipe.groupAll.size.values.dump
3
```

Count distinct

SQL

```
pwagle=> select count(distinct state) from test.allsales;
count
-------
2
```
Scalding Documentation, Release 0.15.0

Scalding

scala> salesPipe.map{ _.state }.distinct.groupAll.size.values.dump
2

Count, Count distinct, Sum in one query

SQL

pwagle=> select count(1), count(distinct state), sum(sales) from test.allsales;
count | count | sum
-------+-------+-----
3 | 2 | 95

Scalding

scala> { salesPipe.map{x => (1, Set(x.state), x.sale) } .groupAll .sum .values .map{ case(count, set, sum) => (count, set.size, sum) } .dump }
(3,2,95)

The above query will have performance issues if count(distinct state) is large. This can be solved in two ways:

- Group by state first (TODO)
- Using an approximate data structure like HyperLogLog (TODO)

Also see [[Aggregation using Algebird Aggregators]].

Where

SQL

select state, name, sales from test.allsales where state = 'CA';

Scalding

scala> salesPipe.filter(sale => (sale.state == `CA'')).dump
Sale(CA,A,60)
Sale(CA,A,20)

Order by X, Y limit N

SQL

select state, name, sale from test.allsales order by state, name limit 1;
Scalding

object SaleOrderingWithState extends Ordering[Sale] {  
    def compare(a: Sale, b: Sale) = a.state compare b.state  
}

implicit val saleOrderingWithState = SaleOrderingWithState

scala> salesPipe.groupAll.sorted.values.dump
Sale(CA,A,60)
Sale(CA,A,20)
Sale(VA,B,15)

scala> salesPipe.groupAll.sorted.take(1).values.dump
Sale(CA,A,20)

scala> salesPipe.groupAll.sortedTake(1).values.dump
List(Sale(CA,A,20))

Union

SQL

select state, name, sales from test.allsales
UNION ALL
select state, name, sales from test.allsales2

Scalding

scala> val salesPipe1 = TypedPipe.from(salesList)
salesPipe1: com.twitter.scalding.typed.TypedPipe[Sale] = IterablePipe(List(Sale(CA,A,60), Sale(CA,A,20), Sale(VA,B,15)))

scala> val salesPipe2 = TypedPipe.from(salesList)
salesPipe2: com.twitter.scalding.typed.TypedPipe[Sale] = IterablePipe(List(Sale(CA,A,60), Sale(CA,A,20), Sale(VA,B,15)))

scala> (salesPipe1 ++ salesPipe2).dump
Sale(CA,A,60)
Sale(CA,A,20)
Sale(VA,B,15)
Sale(CA,A,60)
Sale(CA,A,20)
Sale(VA,B,15)

Group and Aggregate

SQL

pwagle=> select state, count(1), count(discriminate name), sum(sales)
pwagle-> from test.allsales
pwagle-> group by state;

| state | count | count | sum |
|-------+-------+-------+-----|
| CA    | 2     | 1     | 80  |
| VA    | 1     | 1     | 15  |

Scalding

1.2. Intermediate
```scala
scala> {
|   salesPipe.map{ x => (x.state, (1, Set(x.name), x.sale)) }
|     .group
|     .sum
|     .dump
| }
(CA, (2, Set(A), 80))
(VA, (1, Set(B), 15))

scala> {
|   salesPipe.map{ x => (x.state, (1, Set(x.name), x.sale)) }
|     .group
|     .sum
|     .map{ case (state, (count, set, sum)) => (state, (count, set.size, sum))}
|     .dump
| }
(CA, (2, 1, 80))
(VA, (1, 1, 15))

Join

Scalding

case class Table1Row(field1: String, val1: Int)
case class Table2Row(field2: String, val2: Int)

val table1 = TypedPipe.from(List(
  Table1Row(`a'', 1),
  Table1Row(`b'', 2)))
val table2 = TypedPipe.from(List(
  Table2Row(`b'', 3),
  Table2Row(`c'', 4)))

val table1Group = table1.groupBy { _.field1 }
val table2Group = table2.groupBy { _.field2 }

val join = table1Group.join(table2Group)
scala> join.dump
(b, (Table1Row(b, 2), Table2Row(b, 3)))

val leftJoin = table1Group.leftJoin(table2Group)
val outerJoin = table1Group.outerJoin(table2Group)
scala> leftJoin.dump
(a, (Table1Row(a, 1), None))
(b, (Table1Row(b, 2), Some(Table2Row(b, 3))))

scala> outerJoin.dump
(a, (Some(Table1Row(a, 1)), None))
(b, (Some(Table1Row(b, 2)), Some(Table2Row(b, 3))))
(c, (None, Some(Table2Row(c, 4))))
```

---

Chapter 1. Tutorials
Histogram, Ntile

SQL

TODO

Scalding Histogram Fields-based Only

```scala
val inputTp: TypedPipe[Int] = TypedPipe.from(List(5, 2, 3, 3, 4, 4, 4, 1, 15, 30))
val p = inputTp.toPipe(`value)
val p1 = p.groupAll { group => group.histogram(`value -> `histogram) }
  .map(`histogram -> (x.min, x.q1, x.median, x.q3, x.max, x.mean))
val outputTp = p1.toTypedPipe[(Double, Double, Double, Double, Double, Double)]((`min, `q1, `median, `q3, `max, `mean))
```

Output:
```
(1.0, 3.0, 4.0, 5.0, 30.0, 7.1)
```

Scalding QTree

```scala
val inputTp: TypedPipe[Int] = TypedPipe.from(List(5, 2, 3, 3, 4, 4, 4, 1, 15, 30))
implicit val qtSemigroup = new QTreeSemigroup[Long](6)
val v = inputTp.map { x => QTree(x) }.groupAll.sum.values
```

Output:
```
(10, 32.0, 0.0, (4.0, 5.0), (15.0, 16.0))
```

TODO

Analytic / Window Functions (Rank, Ntile, Lag/Lead)

Running Total, Moving Average, Sessionization

1.3 Advanced

Contents:
1.3.1 Introduction to Matrix Library

About

Matrix.scala is a Scalding library that introduces the possibility of treating pipes as sparse matrices and to operate on them using standard matrix operations, such as matrix multiplication:

```scala
//Computing the innerproduct of matrix A
innerProd = A * A.transpose
```

The matrix constructor takes in a pipe containing triples that have the assumed semantics of (row index, column index, matrix value). Additionally, the user can specify the approximate dimensions of the matrix (number of rows, columns, non-zero values) and its skewness (if the distribution of values over the row/column keys is skewed or not). This additional information can help speed-up the computation and improve scalability of the resulting job.

Type restrictions

The matrix row and column indexes can be of any type that is comparable. The usual cases are Int, Long, String. This means that labeled matrices are allowed. For example, we can create a matrix containing the number of users that like specific movie genres per geo without first reindexing the categorical fields to numerical ids:

```scala
//Loading the number of users interested in movie genres per geo from a Tsv source
val interestsMatrix = Tsv(args(`input'')).read
  .toMatrix[String,String,Long](`geo, `movie_genre, `freq)
```

The value type decides what operations can be applied to the matrix.

- Minimally, in order to support **addition**, the value type T must have the trait `Monoid[T]` (as defined in algebird/Monoid).
- In order to support **subtraction**, the value type T must have the trait `Group[T]` (algebird/Group).
- For **multiplication**, the value type T must have the trait `Ring[T]` (algebird/Ring).
- For **division**, the value type T must have the trait `Field[T]` (algebird/Field).

This approach is more powerful than requiring the value type to be Numeric because it allows all of the four operations (addition, subtraction, multiplication, division) to be extended to non-numeric types. For example:

- String addition can be defined as string concatenation
- List addition can be defined as list concatenation
- Set addition can be defined as set union

These operations can be stacked together: For example, Map addition can be defined as set union and the values in the Maps intersection could be aggregated using their own definition of addition. By allowing matrix values to be structured types we can work with higher-order tensors such as cubes or four-tensors with the same library.

For more information on algebraic structures, see the following Wikipedia pages:

- Algebraic structure
- Monoid
- Group
- Ring
- Field
1.3.2 Getting Started. The “Hello World!” example for the Matrix library

Graph nodes outdegrees

Graphs have a straightforward representation in Matrix library as adjacency matrices. We will use the library to compute the outdegrees of the nodes in the graph.

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

import com.twitter.scalding._
import com.twitter.scalding.mathematics.Matrix

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

  import Matrix._

  val adjacencyMatrix = Tsv(args(`input''), (`user1, `user2, `rel))
    .read
    .toMatrix[Long,Long,Double](`user1, `user2, `rel)

  // each row i represents all of the outgoing edges from i
  // by summing out all of the columns we get the outdegree of i
  adjacencyMatrix.sumColVectors.write(Tsv(args(`output'')))
}
```

We convert a pipe of triples to a sparse matrix where element[i,j] represents an edge between row[i] and column[j]. We then sum the values of the columns together into a column vector that has the outdegree of node[i] at row[i].

Next steps

- Read the Matrix API Reference: includes code snippets explaining different kinds of matrix functions (e.g., sumRowVectors, matrix product, element-wise product, diagonal, topRowElems) and much more.
- Go over the Matrix tutorials: the tutorials range from one-liners to more complex examples that show real applications of the Matrix functions to graph problems and text processing.

1.3.3 Building Bigger Platforms with Scalding

As of scalding 0.12, we have an API for this around the Execution type. It is described in [[Calling-Scalding-from-inside-your-application]] This is the recommended approach because it is type-safe, and allows you to compose multiple Executions together.

Using the Fields-API outside of a Job constructor

We consider the Fields-API to be a legacy API which is in maintenance mode. If you really need to use it in new code be aware that sharing code between jobs in the Fields API is a bit challenging because you have to be careful about what fields you leave in the Pipe and there is little help from the compiler.

Generally you will write functions that take Pipes and Fields and return Pipes and Fields. Any time you are reading or writing data, you will need to take (implicit flow: FlowDef, mode: Mode) as implicit arguments to your methods. To get the Dsl syntax, you will want to import com.twitter.scalding.Dsl._ in any file or object that has this shared code.
Customizing Job execution

Mention specialized Job examples (CascadeJob for instance).

Using scalding outside of a com.twitter.scalding.Job (or Tool)

Just do what you would with cascading:

```scala
implicit val mode = Hdfs(new JobConf())
implicit val flowDef = new FlowDef
flowDef.setName(jobName)
val result = myFunctionThatTakesFlowDefAndMode(flowDef, mode)
// Now we have a populated flowDef, time to let Cascading do it's thing:
mode.newFlowConnector(config).connect(flowDef).complete
```

Required settings outside of a com.twitter.scalding.Job (or Tool)

com.twitter.scalding.Job makes changes to the config passed into the Cascading FlowConnector for scalding to function properly. When using scalding outside of a com.twitter.scalding.Job you need to set these.

```scala
val config: Map[AnyRef, AnyRef] = Map(
  `com.twitter.chill.config.configuredinstantiator'' -> `com.twitter.scalding.serialization.KryoHadoop'`
  `cascading.flow.tuple.element.comparator'' -> `com.twitter.scalding.IntegralComparator'`
)
mode.newFlowConnector(config).connect(flowDef).complete
```
2.1 API Reference

There are three APIs:

2.1.1 Type-safe API Reference

There are two main concepts in the type-safe API: a `TypedPipe[T]` which is kind of a distributed list of objects of type `T` and a `KeyedList[K,V]` which represents some sharding of objects of key `K` and value `V`. There are a few `KeyedList` objects: `Grouped[K,V], CoGrouped[K, V]`. The former represents usual groupings, and the latter is used for cogroupings or joins.

Basics

Most of the Typed API is available simply by importing `com.twitter.scalding._`. Most sources, even the simple `TextLine` source, are typed (implement the `TypedSource` trait), which means it is easy to get a `TypedPipe` to begin performing operations on.

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

scala> val lines: TypedPipe[String] = TypedPipe.from(TextLine(``hello.txt''))
lines: com.twitter.scalding.TypedPipe[String] = com.twitter.scalding.typed.TypedPipeFactory@6ec09aa5
scala> // do word count
| (lines.flatMap(_.split(``\s+''))
| .group
| .sum
| .write(TypedTsv[(String, Long)](``output'')))<console>:21: error: Cannot prove that String <::(K, V).
 .group ^
```

The above example generated an error. The problem appears to be that you are running the group function on a `TypedPipe[String]` when it expects a `TypedPipe[(K,V)]`. Essentially you need a pipe of tuples in order to group.
In the above example we show the preferred way to get a `TypedPipe`— using `TypedPipe.from()`, and then demonstrate running a map operation and writing out to a typed sink (`TypedTsv`).

### Map-like functions

#### map

def map[U](f : T => U) : TypedPipe[U]

Converts a `TypedPipe[T]` to a `TypedPipe[U]` via `f : T => U`

case class Bird(name: String, weightInPounds: Double, heightInFeet: Double, color: String)

val birds: TypedPipe[Bird] = TypedPipe.from(Seq(
    Bird(``George'', 12.2, 2.1, ``blue''),
    Bird(``Gatz'', 12.9, 3.21, ``green''),
    Bird(``Jay'', 13.9, 2.7, ``yellow'')))

val britishBirds: TypedPipe[(Double, Double)] =
  birds.map { bird =>
    (0.454 * bird.weightInPounds, 0.305 * bird.heightInFeet)
  }

scala> britishBirds.dump
(5.5388,0.6405)
(5.8566,0.97905)
(6.3106,0.8235)

#### flatMap

def flatMap[U](f : T => Iterable[U]) : TypedPipe[U]


case class Book(title: String, author: String, text: String)

val books: TypedPipe[Book] = TypedPipe.from(Seq(
    Book(``To Kill a Mockingbird'', ``Harper Lee'', ``Atticus Finch''),
    Book(``The Fountainhead'', ``Ayn Rand'', ``Gale Winand''),
    Book(``A Separate Peace'', ``John Knowles'', ``Finny'')))

val words: TypedPipe[String] = books.flatMap { _.text.split(``\s+'') }

scala> words.dump
Atticus Finch
Gale Winand
Finny
Here's an example that uses Option. (Either an animal name passes by in the pipe or nothing.)

```scala
scala> birds.flatMap { b =>
    | if (b.color == `yellow`) { Some(b.name) } else None
    | }.dump
Jay

filter

def filter(f: T => Boolean): TypedPipe[T]

If you return `true` you keep the row, otherwise the row is ignored.

case class Animal(name: String, kind: String)
val animals: TypedPipe[Animal] = TypedPipe.from(Seq(
    Animal(`George', `rabbit'),
    Animal(`Gatz', `bird'),
    Animal(`Joe', `cow'),
    Animal(`Jay', `bird')))
val birds = animals.filter { _.kind == `bird' } 
scala> birds.dump
Animal(Gatz,bird)
Animal(Jay,bird)

filterNot

def filterNot(f : T => Boolean) : TypedPipe[T]

Acts like `filter` with a negated predicate - keeps the rows where the predicate function returns `false`, otherwise the row is ignored.

val notBirds = animals.filterNot { _.kind == `bird' } 
scala> notBirds.dump
Animal(George,rabbit)
Animal(Joe,cow)

collect

def collect(f: PartialFunction[T, U]): TypedPipe[U]

Filters and maps with Scala’s partial function syntax (case):

```scala
val birdNames: TypedPipe[String] = animals.collect { case Animal(name, `bird') => name }
//This is the same as flatMapping an Option.
    | birdNames.dump
Gatz
Jay
```
Creating Groups and Joining (CoGrouping)

Grouping

These are all methods on TypedPipe[T]. Notice that these methods do not return a TypedPipe[T] anymore; instead, they return Grouped[K,T].

groupBy

def groupBy[K](g: T => K)(implicit ord: Ordering[K]) : Grouped[K,T]

Call g : T => K on a TypedPipe[T] to create a Grouped[K,T]. Subsequent aggregation methods use K as the type of the grouping key. We can use any of the functions on Groups specified on the Fields API to transform the Grouped[K, T] to a TypedPipe[U]. Notice that those functions act on T.

Groups need an Ordering (i.e. a comparator) for the key K that we are grouping by. This is implemented for all the standard variable types that we use, in which case no explicit declaration is necessary.

case class Book(title: String, author: String, year: Int)
val books: TypedPipe[Book] = TypedPipe.from(Seq(
    Book(``To Kill a Mockingbird'', ``Harper Lee'', 1960),
    Book(``Go Set a Watchman'', ``Harper Lee'', 2015),
    Book(``The Fountainhead'', ``Ayn Rand'', 1943),
    Book(``Atlas Shrugged'', ``Ayn Rand'', 1957),
    Book(``A Separate Peace'', ``John Knowles'', 1959))

We want to group all the books based on their author

val byAuthor: Grouped[String, Book] = books.groupBy { case book => book.author }

Now, we have Grouped[String, Book].

scala> byAuthor.dump
(Harper Lee,Book(To Kill a Mockingbird,Harper Lee,1960))
(Harper Lee,Book(Go Set a Watchman,Harper Lee,2015))
(Ayn Rand,Book(The Fountainhead,Ayn Rand,1943))
(Ayn Rand,Book(Atlas Shrugged,Ayn Rand,1957))
(John Knowles,Book(A Separate Peace,John Knowles,1959))

scala> byAuthor.size.dump
(Ayn Rand,2)
(Harper Lee,2)
(John Knowles,1)

This creates a KeyedList[String, Int], where the String corresponds to the author and the Int corresponds to the number of books that the author wrote. KeyedList objects are automatically converted to TypedPipes as needed, or you can call .toTypedPipe if you prefer.

group (implicit grouping)

// uses scala's <:< to require that T is a subclass of (K, V).
def group[K, V](implicit ev: T <:< (K, V), ord : Ordering[K]) : Grouped[K, V]

Special case of groupBy that can be called on TypedPipe[(K, V)]. Uses K as the grouping key.
groupAll (send everything to one reducer)

In scala there is a type that has one less value than Boolean, and that is Unit. There is only value in the type Unit. The value is written as ()

```scala
def groupAll: Grouped[Unit, T]
```

Uses Unit as the grouping key. Useful to send all tuples to 1 reducer.

Useful functions on Grouped[K,V].

```scala
val group: Grouped[K, V]
group.keys
//Creates a TypedPipe[K] consisting of the keys in the (key, value) pairs of group.
group.values
//Creates a TypedPipe[V] consisting of the values in the (key, value) pairs of group.
group.mapValues { values => mappingFunction(values) }
//Creates a Grouped[K, V'], where the keys in the (key, value) pairs of group are unchanged.
```

Joining/CoGrouping

These are all methods on CoGroupable[K, V]. TypedPipe[K, V], Grouped[K, V] and even CoGrouped[K, V] are CoGroupable. If possible, put the CoGroupable with the most values per key on the left; this greatly improves performance, but correctness is not impacted. In extreme cases failure to do so can lead to OutOfMemoryErrors. First, we group the pipe by key of type K to get Grouped[K, V]. Then, we join with another group of the same key K, for example Grouped[K, W].

join (inner-join)

```scala
def join[W](smaller: CoGroupable[K, W]): CoGrouped[K, (V, W)]
```

Note that CoGrouped extends KeyedListLike, so any reducing functions you are used to on Grouped will also work on a CoGrouped.

We already know K and V. The only type that could be specified in the join function is W, which is the value in the key-valued group of the smaller group.

Suppose we have two libraries and we want to get a list of the books they have in common. The books of Library 2 have an additional field “copies.”

```scala
import com.twitter.scalding.typed.{CoGroupable, CoGrouped}

case class Book(title: String, author: String)
case class ExtendedBook(title: String, author: String, copies: Long)

val library1: TypedPipe[Book] = TypedPipe.from(Seq(
    Book(``To Kill a Mockingbird'', ``Harper Lee''),
    Book(``The Fountainhead'', ``Ayn Rand''),
    Book(``Atlas Shrugged'', ``Ayn Rand''),
    Book(``A Separate Peace'', ``John Knowles'')))

val library2: TypedPipe[ExtendedBook] = TypedPipe.from(Seq(
    ExtendedBook(``To Kill a Mockingbird'', ``Harper Lee'', 10),
    ExtendedBook(``The Fountainhead'', ``Ayn Rand'', 6),
    ExtendedBook(``Go Set a Watchman'', ``Harper Lee'', 2)))
```
// Group the books of Library 1 by book title.
val group1: CoGroupable[String, Book] = library1.groupBy { _.title }

// Similarly, group the books of Library 2 by book title.
val group2: CoGroupable[String, ExtendedBook] = library2.groupBy { _.title }

// We do group1.join(group2) instead of group2.join(group1)
// because group1 is larger
val theJoin: CoGrouped[String, (Book, ExtendedBook)] = group1.join(group2)
scala> theJoin.dump
(The Fountainhead,(Book(The Fountainhead,Ayn Rand),ExtendedBook(The Fountainhead,Ayn Rand,6)))
(To Kill a Mockingbird,(Book(To Kill a Mockingbird,Harper Lee),ExtendedBook(To Kill a Mockingbird,Harper Lee,10)))

leftJoin

def leftJoin[W](smaller: CoGroupable[K, W]): CoGrouped[K, (V, Option[W])]

Using the definitions from the previous example, assume you are the general manager of Library 1 and you are
interested in a complete list of all the books in your library. In addition, you would like to know, which of those books
can also be found in Library 2, in case the ones in your library are being used:
val theLeftJoin: CoGrouped[String, (Book, Option[ExtendedBook])] = group1.leftJoin(group2)
scala> theLeftJoin.dump
(A Separate Peace,(Book(A Separate Peace,John Knowles),None))
(Atlas Shrugged,(Book(Atlas Shrugged,Ayn Rand),None))
(The Fountainhead,(Book(The Fountainhead,Ayn Rand),Some(ExtendedBook(The Fountainhead,Ayn Rand,6))))
(To Kill a Mockingbird,(Book(To Kill a Mockingbird,Harper Lee),Some(ExtendedBook(To Kill a Mockingbird,Harper Lee,10))))

rightJoin

def rightJoin[W](smaller: CoGroupable[K, W]): CoGrouped[K, (Option[V], W)]

val theRightJoin: CoGrouped[String, (Option[Book], ExtendedBook)] = group1.rightJoin(group2)
scala> theRightJoin.dump
(Atlas Shrugged,(Option(Book(Atlas Shrugged,Ayn Rand)),ExtendedBook(Atlas Shrugged,Ayn Rand,6)))
(To Kill a Mockingbird,(Some(Book(To Kill a Mockingbird,Harper Lee)),ExtendedBook(To Kill a Mockingbird,Harper Lee,10)))

outerJoin

def outerJoin[W](smaller: CoGroupable[K, W]): CoGrouped[K, (Option[V], Option[W])]

val theOuterJoin: CoGrouped[String, (Option[Book], Option[ExtendedBook])] = group1.outerJoin(group2)
scala> theOuterJoin.dump
(A Separate Peace,(Some(Book(A Separate Peace,John Knowles)),None))
(Atlas Shrugged,(Some(Book(Atlas Shrugged,Ayn Rand)),None))
(To Kill a Mockingbird,(None,Some(ExtendedBook(Go Set a Watchman,Harper Lee,2))))
(The Fountainhead,(Some(Book(The Fountainhead,Ayn Rand)),Some(ExtendedBook(The Fountainhead,Ayn Rand,6))))
(To Kill a Mockingbird,(Some(Book(To Kill a Mockingbird,Harper Lee)),Some(ExtendedBook(To Kill a Mockingbird,Harper Lee,10))))

Like all KeyedListLike instances, CoGrouped has toTypedPipe to explicitly convert to TypedPipe. However, this
is automatic (implicit from KeyedListLike[K, V, _] => TypedPipe[(K, V)] in object KeyedListLike).
val myJoin: CoGrouped[K, (V, W)]
val tpipe: TypedPipe[(K, (V, W))] = myJoin.toTypedPipe

Joining multiple streams

Since CoGrouped is CoGroupable it is perfectly legal to do a.join(b).leftJoin(d).outerJoin(e) and it will run in one map/reduce job, but the value type will be a bit ugly: (Option[((A, B), C), Option[D]], Option[E]). To make this cleaner, in scalding 0.12 we introduce the MultiJoin object. MultiJoin(a, b, c, d) does an inner join with a value tuple of (A, B, C, D) as you might expect. You can also do MultiJoin.left or MultiJoin.outer.

Map-side (replicated) joins:

These methods do not require a reduce step, but should only be used on extremely small arguments since each mapper will read the entire argument to do the join.

cross

Suppose we want to send every value from one TypedPipe[U] to each value of a TypedPipe[T]. List(1,2,3) cross List(4,5) gives List((1,4),(1,5),(2,4),(2,5),(3,4),(3,5)). The final size is left.size * right.size.

// Implements a cross product. The right side should be tiny.
def cross[U](tiny: TypedPipe[U]): TypedPipe[(T,U)]

hashJoin

A very efficient join, which works when the right side is tiny, is hashJoin. All the (key, value) pairs from the right side are stored in a hash table for quick retrieval. The hash table is replicated on every mapper and the hashJoin operation takes place entirely on the mappers (no reducers involved).

// Again, the right side should be tiny.

Tip: All groups and joins have .withReducers(n) to explicitly set the number of reducers for that step. For other options, please refer to: http://twitter.github.io/scalding/#com.twitter.scalding.typed.Grouped and http://twitter.github.io/scalding/#com.twitter.scalding.typed.CoGrouped

ValuePipe: working with values that will be computed

Sometimes we reduce everything down to one value:

val userFollowers: TypedPipe[(Long, Int)] = // function to get

val topUsers: TypedPipe[Long] = allUsers
    .collect { case (uid, followers) if followers > 1000000 => uid }

    // put it in a value:
val topUsers: ValuePipe[Set[Long]] = topUsers.map(Set(_)).sum
A value Pipe is a kind of future value: it is a value that will be computed by your job, but is not there yet. TypedPipe.sum returns a ValuePipe.

When you have this, you can then use it on another TypedPipe:

```scala
val allClickers: TypedPipe[Long] = //...
val topClickers = allClickers.filterWithValue(topUsers) { (clicker, optSet) =>
  optSet.get.contains(clicker) // keep the topUsers that are also clickers
}
```

You can also mapWithValue or flatMapWithValue. See ValuePipe.scala for more.

### Records

Suppose you have many fields and you want to update just one or two. Did you know about the `copy` method on all case classes?

Consider this example:

```scala
scala> case class Record(name: String, weight: Double)
defined class Record

scala> List(Record(``Bob'', 180.3), Record(``Lisa'', 154.3))
res22: List[Record] = List(Record(Bob,180.3), Record(Lisa,154.3))

scala> List(Record(``Bob'', 180.3), Record(``Lisa'', 154.3)).map { r =>
  val w = r.weight + 10.0
  r.copy(weight = w)
}
res23: List[Record] = List(Record(Bob,190.3), Record(Lisa,164.3))
```

In exactly the same way, you can update just one or two fields in a case class on scalding with the typed API.

This is how we recommend making records, but WATCH OUT: you need to define case classes OUTSIDE of your job due to serialization reasons (otherwise they create circular references).

### Aggregation and Stream Processing

Both `Grouped[K, R]` and `CoGrouped[K, R]` extend `KeyedListLike[K, R, _]`, which is the class that represents sublists of `R` sharded by `K`. The following methods are the main aggregations or stream processes you can run.

#### sum: Generalized reduction

```scala
def sum[U >: V](implicit s: Semigroup[U]): KeyedListLike[K, U]
```

Scalding uses a type from Algebird called a Semigroup for sums. A semigroup is just a reduce function that has the property that `plus(plus(a, b), c) == plus(a, plus(b, c))`. The default Semigroup is what you probably expect: addition for numbers, union for sets, concatenation for lists, maps do an outer join on their keys and then do the semigroup for their value types.

If there is no sorting on the values, scalding assumes that order does not matter and it will partially apply the sum on the mappers. This can dramatically reduce the communication cost of the job depending on how many keys there are in your data set.
reduce: an ad-hoc Semigroup

def reduce(fn: (V, V) => V): KeyedListLike[K, V]

This defines the plus function for a Semigroup, and then calls sum with that Semigroup. See the documentation there.

aggregate: Using Aggregators for reusability

def aggregate[B,C](a: Aggregator[V, B, C]): KeyedListLike[K, C]

check the aggregator tutorial for more explanation and examples.

foldLeft and fold

def foldLeft[U](init: U)(fn: (U, V) => U): KeyedListLike[K, U]

to use where you might make a loop in some language. A is the some state you are updating every time you
see a new value V. An example might be training a model on some data. A is your model. V are you data points. Your
fn looks like: foldLeft(defaultModel) { (model, data) => updateModel(model, data) }.

def fold[U](f: Fold[V, U]): KeyedListLike[K, U]

def foldWithKey[U](fn: K => Fold[V, U]): KeyedListLike[K, U]

A com.twitter.algebird.Fold is an instance that encapsulates a fold function. The value of this is two fold:

1. Logic can be packaged in a Fold and shared across many jobs, for instance Fold.size

2. Folds can be combined together so many functions can be applied in one pass over the data.

import com.twitter.algebird.Fold
val myWork: Fold[Int, (Long, Boolean, Int)] = {
  Fold.size
  .join(Fold.forall { i: Int => i > 0 })
  .join(Fold.sum[Int])
  .map { case ((size, pos), sum) => (size, pos, sum) }
}

Folds are similar to Aggregators, with the exception that they MUST be run only on the reducers. If you can express
an aggregation in terms of Aggregators, it is worthwhile to do so in that it can give you map-side reduction before
going to the reducers.

mapGroup and mapValueStream: totally general reducer functions

def mapGroup[U](fn: (K, Iterator[V]) => Iterator[U]): KeyedListLike[K, U]

def mapValueStream[U](fn: Iterator[V] => Iterator[U]): KeyedListLike[K, U]

It is pretty rare that you need a reduction that is not a sum, aggregate or fold, but it might occasionally come up. If you
find yourself reaching for this very often, it might be a sign that you have not quite grokked how to use Aggregators
or Folds.

These functions give you an Iterator over the values on your reducer, and in the case of mapGroup the key, and you
can transform just the values, not the key. If you need to change the key, output the new key and value in the U type,
and then discard the keys using the .values method.

Using mapGroup/mapValueStream always forces all the data to reducers. Realizing the entire stream of values at
once (i.e. manually reversing or rescanning the data) can explode the memory, so prefer to operate one at time on the
Iterators you are given.
**Example: Datacubing**

A common pattern is called data-cubing. This is where you have some commutative sum that you want to materialize sums of all possible binary queries where part of the key is present or absent (making each point of the key space into a hyper-cube). Here is an example of how to do this with the typed-API:

The [[Fields-based API Reference]] has a builder-pattern object called GroupBuilder which allows you to easily create a tuple of several parallel aggregations, e.g. counting, summing, and taking the max, all in one pass through the data. The type-safe API has a way to do this, but it involves implementing a type-class for your object and using `KeyedList.sum` on the tuple. Below we give an example.

```scala
import com.twitter.algebird.Monoid

case class Hipster(name: String, ridesFixie: Boolean, rimmedGlasses: Boolean, income: Double) {
  def hipsterScore: Double = List(ridesFixie, rimmedGlasses).map { if(_) 1.0 else 0.0 }.sum + 1.0/income
}

// Monoid which chooses the highest hipster score
implicit val hipsterMonoid = new Monoid[Hipster] {
  def zero = Hipster(``zeroHipster'', false, false, Double.NegativeInfinity)
  def plus(left: Hipster, right: Hipster) =
    List(left, right).maxBy { _.hipsterScore }
}

// Now let's count our fixie riders find the biggest hipster
val people: TypedPipe[Hipster] = TypedPipe.from(Seq(
  Hipster(``Joe'', true, false, 20),
  Hipster(``George'', false, false, 100),
  Hipster(``Grok'', true, true, 1)))

// Now we want to know how many total people, fixie riders, and how many rimmed-glasses wearers,
// as well as the biggest hipster:
val (totalPeople, fixieRiders, rimmedGlasses, biggestHipster) = {
  people.map { person =>
    (1L, if(person.ridesFixie) 1L else 0L, if(person.rimmedGlasses) 1L else 0L, person)
  }.sum
    .toOption
    .get
}

scala> totalPeople
res37: Long = 3

scala> fixieRiders
res38: Long = 2

scala> rimmedGlasses
res39: Long = 1

scala> biggestHipster
res40: Hipster = Hipster(Grok, true, true, 1.0)

Scalding automatically knows how to sum tuples (it does so element-wise, see `GeneratedAbstractAlgebra.scala`).
```
Powerful Aggregation with Algebird

See this example on Locality Sensitive Hashing via @argyris.

Interoperating between Fields API and Type-safe API

If you can avoid the Fields API, we recommend it. But if you have legacy code that you want to keep while you are migrating to the Type-safe API, there are methods to help you.

Generally, all the methods from the [[Fields-based API Reference]] are present with the following exceptions:

1. The mapping functions always replace the input with the output. map and flatMap in the Type safe API are similar to the mapTo and flatMapTo functions (respectively) in the Fields-based API.
2. Due to the previous statement, there is no need to name fields.

If you \texttt{import TDsl._} you get an enrichment on cascading Pipe objects to jump into a Typed block:

\begin{verbatim}
pipe.typed(('in0, 'in1) => 'out) { tpipe : TypedPipe[(Int,Int)] =>
  tpipe.groupBy { x => 1 } //groups on all the input tuples (equivalent to groupAll)
  .mapValues { tup => tup._1 + tup._2 } //sum the two values in each tuple
  .sum //sum all the tuple sums (i.e. sum everything)
  .values // discard the key which is 1
}
\end{verbatim}

In this example, we start off with a cascading Pipe (pipe), which has the \texttt{'in0} and \texttt{'in1} fields. We use the method \texttt{typed} in order to create a new TypedPipe (tpipe). Then, we apply all of our functions on the TypedPipe[(Int, Int)] to obtain a TypedPipe[Int] which has the total sum. Finally, this is converted back into the cascading Pipe (pipe) with the single field \texttt{'out}, which contains a single Tuple holding the total sum.

Converting pipes

- To go from a pipe to a TypedPipe[T]: \texttt{mypipe.toTypedPipe[T](Fields_Kept)}. Fields_Kept specifies the fields in mypipe that we want to keep in the Typed Pipe.
- To go from a TypedPipe[T] to a pipe: \texttt{myTypedPipe.toPipe(f: Fields)} method. Since we go from a Typed to a cascading pipe, we actually need to give names to the fields.

Example:

\begin{verbatim}
import TDsl._

case class Bird(name: String, winLb: Float, color: String)
val birds: TypedPipe[Bird] = getBirdPipe
birds.toPipe(`name, `winLb, `color) //Cascading Pipe with the 3 specified fields.
birds.toTypedPipe[(String, String)](`name, `color) //Typed Pipe (keeping only some fields)
\end{verbatim}

Advanced examples:

\begin{verbatim}
import TDsl._

case class Bird(name: String, winLb: Float, hinFt: Float, color: String)
val birds: TypedPipe[Bird] = getBirdPipe
birds.toPipe(`name, `color)
\end{verbatim}

\begin{verbatim}
val p: TypedPipe[(Double, Double)] =
  TypedTsv[(Double,Double)](input, (\'a, \'b))
  .toTypedPipe[(Double, Double)](\'a, \'b)
\end{verbatim}

\texttt{TypedPipe[MyClass]} is slightly more involved, but you can get it in several ways. One straightforward way is:
object Bird {
    def fromTuple(t: (Double, Double)): Bird = Bird(t._1, t._2)
}

case class Bird(weight: Double, height: Double) {
    def toTuple: (Double, Double) = (weight, height)
}

import TDsl._
val birds: TypedPipe[Bird] =
  TypedTsv[(Double, Double)](path, (`weight, `height))
  .map{ Bird.fromTuple(_) }

### 2.1.2 Fields-based API Reference

Scalding functions can be divided into four types:

- Map-like functions
- Grouping functions
- Reduce operations
- 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, mapTo, flatMapTo**

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

Adds new fields that are transformations of existing ones.

```scala
val fasterBirds = birds.map(`speed -> `doubledSpeed) { speed : Int => speed * 2 }
```

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

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

You can map from a field to itself to update its value:

```scala
items.map(`price -> `price) { price : Float => price * 1.1 }
```

You can use `*` (here and elsewhere) to mean all fields.

```
# pipe.flatMap(existingFields -> additionalFields){function}
```
Maps each element to a list (or an Option), and then flattens that list (emits a Cascading Tuple per each item in the returned list).

```scala
val words =
    books.flatMap(`text -> `word) { text : String => text.split(`\s+'') }
```

# pipe.mapTo(existingFields -> additionalFields){function}

MapTo is equivalent to mapping and then projecting to the new fields, but is more efficient. Thus, the following two lines produce the same result:

```
pipe.mapTo(existingFields -> additionalFields){ ... }
pipe.map(existingFields -> additionalFields){ ... }.project(additionalFields)
```

Here is another example:

```scala
val savings =
    items.mapTo((`price, `discountedPrice) -> `savings) {
        x : (Float, Float) =>
            val (price, discountedPrice) = x
            price - discountedPrice
    }
val savingsSame =
    items.map((`price, `discountedPrice) -> `savings) {
        x : (Float, Float) =>
            val (price, discountedPrice) = x
            price - discountedPrice
    }.
    .project(`savings)
```

# pipe.flatMapTo(existingFields -> additionalFields){function}

The flatMap analogue of mapTo.

```scala
val words =
    books.flatMapTo(`text -> `word) { text : String => text.split(`\s+'') }
```

`project`, `discard`

# pipe.project(fields)

Remove all unspecified fields.

```
// The new pipe contains only two fields: `jobTitle` and `salary`.
val onlyKeepWorkInfo = people.project(`jobTitle, `salary)
```

# pipe.discard(fields)

Removes specified fields. discard is the opposite of project.

```
val forgetBirth = people.discard(`birthplace, `birthday)
```

`insert`, `rename`, `limit`

# pipe.insert(field, value) Insert field(s) with constant value(s)

```
    items.insert((`inflation, `collegeCostInflation), (0.02, 0.10))
```

# pipe.rename(fields -> fields) Rename fields
items.rename((`x, `y) -> (`X, `Y))

# pipe.limit(number)
Allows only a fixed number of items to pass in a pipe.

**filter, filterNot**

# pipe.filter(fields){function}
Filters out rows for which function is false.

val birds = animals.filter(`type) { type : String => type == `bird'' }

You can also filter over multiple fields at once.

val fastAndTallBirds =
  birds.filter(`speed, `height) {
    fields : (Float, Float) =>
      val (speed, height) = fields
      (speed > 100) && (height > 100)
  }

# pipe.filterNot(fields){function}
Works exactly like a negated filter operation. It will filter out the rows for which the predicate function returns true.

val notBirds = animals.filterNot(`type) { type : String => type == `bird'' }

**unique**

# pipe.unique(fields)
Keeps only unique rows based on a specified set of fields.

This looks like a mapping function, but it actually requires a map-reduce pair, so doing this during one of your groupBy operations (if you can structure your algorithm to simultaneously do so) will save work.

// Keep only the unique (firstName, lastName) pairs. All other fields are discarded.
people.unique(`firstName, `lastName)

**pack, unpack**

# pipe.pack(Type)(fields -> object)
You can pack multiple fields into a single object, by using Java reflection. For now this only works for objects that have a default constructor that takes no arguments. The Java reflection only happens once for each field, so the performance should be very good. Basically, the pack and unpack functions are used to group or ungroup fields, respectively, by using Objects.

For example suppose that you have a class called Person, with fields age and height, and setters setAge and setHeight. Then you can do the following to populate those fields:

val people = data.pack[Person]((`age, `height) -> `person)

# pipe.unpack(Type)(object -> fields)
Conversely, you can unpack the contents of an object into multiple fields.
val data = people.unpack[Person](`person -> (`age, `height))

The default reflection-based unpacker works for case classes as well as standard Thrift- and Protobuf-generated classes.

If you want to use tuple packing and unpacking for objects that do not depend on Java reflection, then you need to implement the TuplePacker and TupleUnpacker abstract classes and define implicit conversions in the context of your Job class. See TuplePacker.scala for more.

**Grouping functions**

Grouping/reducing functions operate over groups of rows in a pipe, often aggregating them in some way. They usually involve a reduce phase. These functions are defined in GroupBuilder.scala.

Most of these functions are inspired by the scala.collection.Iterable API.

**groupBy, groupAll, groupRandomly, shard**

```scala
# pipe.groupBy(fields){ group => ... }
Groups your pipe by the values in the specified set of fields, and then applies a set of operations to the group to create a new set of fields. All the entries with the same value (in the field we are grouping by) are sent to the same reduc
for processing. But, different values can be sent to different reducers.
```

```scala
val wordCounts = words.groupBy(`word) { group => group.size }
```

Group operations chain together.

```scala
val demographics = people.groupBy(`country, `sex) { _.size.average(`age) }
```

When the field to group by is an enum or a thrift type, currently it won’t work properly. Please avoid using enum type
for group by.

```scala
# pipe.groupAll{ group => ... }
Creates a single group consisting of the entire pipe.

Think three times before using this function on Hadoop. This removes the ability to do any parallelism in the reducers.
That said, accumulating a global variable may require it. Tip: if you need to bring this value to another pipe, try
crossWithTiny (another function you should use with great care).
```

```scala
val vocabSize = wordCounts.groupAll { _.size }
```

groupAll is also useful if you want to sort a pipe immediately before outputting it.

```scala
val sortedPeople = people.groupAll { _.sortBy(`lastName, `firstName) }
```

As we mentioned earlier, groupBy splits the various groups among different reducers, which do not collaborate. Therefore, if we want to sort everything we use groupAll, which basically sends everything to 1 reducer (since it creates a single group of the entire pipe). Then, the sorting can happen on the reducer.

**Group/Reduce Functions**

Here is an overview of some of the most popular:

# group.size(name)
Counts the number of rows in this group. By default, the name of the new field is `size`, but you can pass in a new name as well.

```scala
// The new `wordCounts` pipe contains `word' and `size' fields.
val wordCounts = words.groupBy(`word) { _.size }
```

```scala
// Same, but calls the new field `count' instead of the default `size'.
val wordCounts = words.groupBy(`word) { _.size(`count) }
```

# group.average(field)
Computes the mean over a field. By default, the new field has the same name as the original field, but you can pass in a new name as well.

```scala
// Find the mean age of boys vs. girls. The new pipe contains `sex' and `age' fields.
val demographics = people.groupBy(`sex) { _.average(`age) }
```

```scala
// Same, but call the new field `meanAge'.
val demographics = people.groupBy(`sex) { _.average(`age -> `meanAge) }
```

# group.sizeAveStdev(field, fields)
Computes the count, average and standard deviation over a field. You must pass new fields to accommodate the output data

```scala
// Find the count of boys vs. girls, their mean age and standard deviation. The new pipe contains `sex', `count', `meanAge' and `stdevAge' fields.
val demographics = people.groupBy(`sex) { _.sizeAveStdev(`age -> (`count, `meanAge, `stdevAge) ) }
```

# group.mkString(field, joiner)
Turns a column in the group into a string. Again, the new field has the same name as the original field by default, but you can also pass in a new name.

```scala
// Take all the words with a given count and join them with a comma.
wordCounts.groupBy(`count) { _.mkString(`word, ',', '') }
```

```scala
// Same, but call the new column `words'.
wordCounts.groupBy(`count) { _.mkString(`word -> `words, ',', '') }
```

# group.toList(field)
Turns a column in the group into a list. An idiosyncracy about this is that null items in the list are removed. It is equivalent to first filtering null items. Be careful about depending on this behavior as it may be changed before scalding 1.0.

```scala
// Take all the words with this count and join them into a list.
wordCounts.groupBy(`count) { _.toList[String](`word) }
```

```scala
// Same, but call the new column `words'.
wordCounts.groupBy(`count) { _.toList[String](`word -> `words) }
```

# group.sum(field)
Sums over a column in the group.

```scala
expenses.groupBy(`shoppingLocation) { _.sum[Double](`cost) }
```

```scala
// Same, but call the summed column `totalCost'.
```
expenses.groupBy(`shoppingLocation) { _.sum[Double](`cost -> `totalCost) }

# group.max(field), group.min(field)
Computes the largest or smallest element of a group.

expenses.groupBy(`shoppingLocation) { _.max(`cost) }

# group.count(field){function}
Counts the number of rows in a group that satisfy some predicate.

val usersWithImpressions =
  users
  .groupBy(`user) { _.count(`numImpressions) { x : Long => x > 0 } }

# group.sortBy(fields)
Using sortBy you can sort the output before writing it into some output sink.

users.groupAll { _.sortBy(`age) }

Note: When reading from a CSV, the data types are set to String, hence the sorting will be alphabetically, therefore to sort by age, an int, you need to convert it to an integer. For example,

val users = Csv(file_source, separator = '',' ', fields = Schema)
  .read
  .map (`age-> `ageInt) {x: Int => x}
  .groupAll { _.sortBy(`ageInt) } // will sort age as a number.

# group.sortBy(fields).reverse
You can also reverse the sort-order used (descending, instead of ascending):

users.groupAll { _.sortBy(`age).reverse }

At the moment it is a limitation that reverse must be called after a sortBy, so this: _.reverse.sortBy(`age)
/* wrong */ would compile, but would throw an “Cannot sort when reducing” exception during the planning phase.

reduce, foldLeft

# group.reduce(field){function}
Applies a reduce function over grouped columns. The reduce function is required to be associative, so that the work can be done on the map side and not solely on the reduce side (like a combiner).

// This example is equivalent to using `sum`, but you can also supply other reduce functions.
expenses.groupBy(`shoppingLocation) {
  _.reduce(`cost -> `totalCost) {
    (costSoFar : Double, cost : Double) => costSoFar + cost
  }
}

# group.foldLeft(field){function}
Like reduce, but all the work happens on the reduce side (so the fold function is not required to be associative, and can in fact output a type different from what it takes in). Fold is the fundamental reduce function.

// for the sake of example, assume we want to discount cost so far by specified amounts
// and that items are in the order we want
expenses.groupBy(`shoppingLocation) {
  val init_cost_so_far = 0.0
```scala
_.foldLeft((`cost, `inflation) -> `discountedCost)(init_cost_so_far) {
  (discountedCostSoFar: Long, cost_infl: (Double, Double)) =>
  val (cost, inflation) = cost_infl
  discountedCostSoFar * inflation + cost
}

# take & sorting

take(n) keeps the first n elements of the group.

### group.take(number)

groupBy(`shoppingLocation) {
  _.take(100)
}

# group.takeWhile[T](f : Fields)(fn : (T) => Boolean)

Take while the predicate is true; stop at the first false.

# group.drop(number)

drop(n) drops the first n elements of the group.

# group.sortWithTake( fields -> result_field, take_number)

Equivalent to sorting by a comparison function, then taking 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.

sortWithTake( ('clicks, `tweet) -> `results, 5) {
  comparison_function : ( clickTweet1 :(Long,Long), clickTweet2:(Long,Long) =>
  clickTweet1._1 < clickTweet2._1
}

# group.sortedReverseTake[Field Types](fields -> temporary_field_tuple, number)

Reverse stands for decreasing order.

In this example, we first sort by 'CountNewBorns, then by 'CityName and finally by'StateName. Since it is in decreasing order, the entry with most newborns will be the first one. All the fields are stored as a tuple in the 'top field, which we then flatten to get the original fields.

reducers

# group.reducers(number)

Override the number of reducers used in the groupBy. Useful when outputting fewer files is desired.

pipe.groupBy(`key) {
  _.sortBy(`count).reverse.reducers(6)
}
```
**Chained group operations**

Chain together multiple GroupBuilder operations to apply different reductions to different fields:

```scala
group.sum[Long]('x).max('y)
```

**pivot, unpivot**

Pivot and unpivot are similar to SQL and Excel functions that change data from a row-based representation to a column-based one (in the case of `pivot`) or vice-versa (in the case of `unpivot`).

```scala
# group.pivot

Converts data from a row-based representation to a column-based one.

```scala
pipe.groupBy(`key) { _.pivot((`col, `val) -> (`x, `y, `z)) }
```

In the first example, you need to have rows like:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><code>x</code></td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td><code>y</code></td>
<td>3.4</td>
</tr>
<tr>
<td>4</td>
<td><code>z</code></td>
<td>4</td>
</tr>
</tbody>
</table>

and after the pivot you will have:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.2</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>null</td>
<td>4</td>
</tr>
</tbody>
</table>

When pivoting, you can provide an explicit default value instead of replacing missing rows with null:

```scala
pipe.groupBy(`key) { _.pivot((`col, `val) -> (`x, `y, `z), 0.0) }
```

This will result in:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.2</td>
<td>3.4</td>
</tr>
<tr>
<td>4</td>
<td>0.0</td>
<td>4</td>
</tr>
</tbody>
</table>

```scala
# pipe.unpivot

Converts data from a column-based representation to a row-based one. (Strictly speaking, `unpivot` is a map-like function which appears in RichPipe.scala and does not require a reduce-phase.)

```scala
pipe.unpivot((`x, `y, `z) -> (`col, `val))
```

**Join operations**

Join operations merge two pipes on a specified set of keys, similar to SQL joins. They are defined in `JoinAlgorithms.scala`.

All the expected joining modes are present: inner, outer, left, and right. Cascading implements these as CoGroup operations which are implemented in a single map-reduce job.

**joins**

Since it is important to hint at the relative sizes of your data, Scalding provides three main types of joins. All of them are inner joins:

- `joinWithSmaller`
- `joinWithLarger`
• joinWithTiny: this is a special map-side join that does not move the left-hand side from mappers to reducers. Instead, the entire right hand side is replicated to the nodes holding the left side. By, “right hand side,” we mean the significantly smaller pipe that we are passing as an argument to this function.

When in doubt, choose joinWithSmaller and optimize if that step seems to be taking a very long time.

# pipe1.joinWithSmaller(fields, pipe2)

Joins two pipes on a specified set of fields. Use this when pipe2 has fewer values per key than pipe1.

// `people` is a pipe with a `birthCityId` field.
// It is `larger` because there are many people and many share the same birthCityId
// Join it against the `cities` pipe, which contains an `id` field.
// Cities is `smaller` because it has a smaller number of values per id (in this case 1)
val peopleWithBirthplaces = people.joinWithSmaller(`birthCityId -> `id, cities)

// Join on both city.id and state.id
val peopleWithBirthplaces = people.joinWithSmaller( (`birthCityId, `birthStateID) -> (`id, `StateID) , cities)

# pipe1.joinWithLarger(fields, pipe2)

Joins two pipes on a specified set of fields. Use this when pipe2 has more values per key than pipe1.

// `cities` is a pipe with an `id` field.
// `cities` is `smaller` because it has a smaller number of values per id (in this case 1)
// Join it against the `people` pipe, which contains a `birthCityId` field.
// `people` is `larger` because there are many people and many share the same birthCityId
val peopleWithBirthplaces = cities.joinWithLarger(`id -> `birthCityId, people)

# pipe1.joinWithTiny(fields, pipe2)

Joins two pipes on a specified set of fields. As explained above, this is a special map-side join that does not move the left-hand side from mappers to reducers. Instead, the entire right hand side is replicated to the mappers (nodes) holding the left side. joinWithTiny is appropriate when you know that # of rows in bigger pipe > mappers * # rows in smaller pipe, where mappers is the number of mappers in the job.

// Assume this is a small pipe containing at most couple thousand rows.
val celebrities = ...

val celebrityBirthplaces = cities.joinWithTiny(`id -> `birthCityId, celebrities)

join modes

By default, all joins are inner joins. You can also specify that you want a left join, a right join, or an outer join. left join: It keeps all the rows/entries from the left pipe and attaches the entries that have matching keys from the right pipe. The entries of the left pipe that do not have any matches with the right pipe have null for the new fields introduced by the right pipe. right join: Similar to the left join; it keeps all the rows/entries from the right pipe. outer join: This join keeps all entries from both pipes. Again, if there is no match the empty fields contain null.

import cascading.pipe.joiner._

people.joinWithSmaller(`birthCityId -> `id, cities, joiner = new LeftJoin)
people.joinWithSmaller(`birthCityId -> `id, cities, joiner = new RightJoin)
people.joinWithSmaller(`birthCityId -> `id, cities, joiner = new OuterJoin)

Note that when performing an inner join, the left and right pipes are allowed to join on common field names.

// This is allowed. Only a single `ssn` field will be left in the resulting merged pipe.
people.joinWithSmaller(`ssn -> `ssn, teachers)
// Instead
people.joinWithSmaller(`ssn_left -> `ssn_right, teachers)
// Both fields are kept after the join.

However, joining on common field names is not allowed for the left joins, right joins, or outer joins (since it is useful to know whether a missing field value comes from the left pipe or the right pipe).

// This is not allowed.
people.joinWithSmaller(`ssn -> `ssn, teachers, joiner = new OuterJoin)

crossWithTiny

# pipe1.crossWithTiny(pipe2)
Performs the cross product of two pipes. The right (pipe2) is replicated to all the nodes, and the left is not moved at all. Therefore, the “tiny” part should be on the right.

existingFields

Miscellaneous functions

On pipes

All this and more in RichPipe.scala:

• pipe1 ++ pipe2 to union two pipes that have the same fields
• p.addTrap to capture any exceptions thrown on the pipe

val peopleWithBirthplaces = people.joinWithSmaller(`birthCityId -> `id, cities)
    .addTrap(Tsv(``/home/data/error_folder/''))

• p.debug to see rows on stdout/stderr
• p.name("myPipe") to name your pipe
• p.partition(fields_to_apply_function -> field_based_on_function_output)
{function} {group} Given a function, it partitions the pipe into several groups based on the output of the function. Then applies a GroupBuilder function on each of the groups.

pipe.mapTo(()->(`age, `weight) { ... }
    .partition(`age -> `isAdult) { _ > 18 } { _.average(`weight) }
// pipe now contains the average weights of adults (above 18) and minors.
• p.sample(percentage) where 0.00 < percentage < 1.00, note that percentage is actually a decimal
• p.thenDo{ p : Pipe => if(scala.util.Random.nextInt(2) > 0) p.insert('foo, 1) else p }
• p.write(Tsv("myfile"))

On groups

All this and more in ReduceOperations.scala:

• group.dot(a, b, a_dot_b) dot product

groupBy(`x) { _ .dot(`y,`z, `ydotz) }
// First do "`times" on each pair, then "`plus" them all together.
• group.head Return the first element; useful mostly for sorted case.
• group.histogram
Scalding Documentation, Release 0.15.0

- `group.last` Return the last element; again, useful mostly for sorted case.

More functions can be found at `StreamOperations` for stream-like functions (e.g. `take`, `drop`) and `[FoldOperations]` (http://twitter.github.io/scalding/#com.twitter.scalding.FoldOperations) for fold/reduce-like functions (e.g. `foldLeft`).

### About functions on multiple fields at once

In many places in the scalding fields-based API can functions be applied to multiple fields at once. For example:

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

The parameter `x` is a tuple of size 2 which is consistent with the number of fields the function is expected to operate on. As an alternative you can also import `FunctionImplicits._` and use a regular function with multiple input arguments:

```scala
import com.twitter.scalding.FunctionImplicits._
val britishBirds =
  birds.map((`weightInLbs, `heightInFt) -> (`weightInKg, `heightInMeters)) {
    (weightInLbs: Float, heightInFt: Float) =>
    (0.454 * weightInLbs, 0.305 * heightInFt)
  }
```

### 2.1.3 Fields API: Reduce Functions of GroupBuilder

`com.twitter.scalding.GroupBuilder`

- `mapStream`

  Full access to the Iterator of values. Avoid it if you can as this cannot be optimized much

- `scanLeft`

  A particular, but common, pattern to processes an iterator and return a new iterator.

- `foldLeft`

  Exactly like the above, but we only keep the last value.

- `reduce / mapReduceMap`

  Preferred operations, which are partially executed on the mappers.
2.1.4 Matrix API Reference

Matrix library functions can be divided into three types:

- Value operations
- Vector operations
- Matrix operations

In addition, matrix library contains

- ConversionFrom functions
- ConversionTo functions
- Other functions

Value operations

Value operations work over individual values in a matrix, usually transforming them in some way.

**mat.mapValues( mapping function ): Matrix**

Maps the values of the matrix to new values using a function

```scala
// The new matrix contains the elements from the original matrix multiplied by 2
val doubledMatrix = matrix.mapValues{ value : Int => value * 2 }
```

**mat.filterValues( filter function ): Matrix**

Keep only the values of the matrix that set the function to true

```scala
// The new matrix contains only the positive elements from the matrix
val filteredMatrix = matrix.filterValues{ _ > 0 }
```

**mat.binarizeAs[NewValT]: Matrix**

Sets all of the non-zero values of the matrix to the one element of the type

```scala
// The new matrix contains ones as integers for all non-zero values from the matrix
val binMatrix = matrix.binarizeAs[Int]
```

Vector operations

Vectors are represented as pipes of pairs of indexes and values. There are two classes of vectors, one for row and one for column vectors respectively. Objects from the two classes can convert to one another through `vector.transpose` and to a two-dimensional matrix using `vector.toMatrix` or `vector.diag`. 
RowVector operations

mat.getRow( rowNumber ): RowVector

Returns the row indexed with the specified value from the original matrix

// Returns the 3rd row from the matrix
val row = matrix.getRow(3)
// Returns the row from the matrix that is indexed with "France"
val row = matrix.getRow("France")

matrix.reduceRowVectors( reduce function ): RowVector

Reduces all row vectors into a single row vector using a associative pairwise aggregation function

// Produces a row vector containing the products of each column's values
// matrix = 1 1 1
//     2 2 1
//     3 0 1
// rowProd = 6 0 1
val rowProd = matrix.reduceRowVectors { (x,y) => x * y }

matrix.sumRowVectors: RowVector

Reduces all row vectors into the sum row vector

// Returns the row vector of all per-column sums
// -------
// 1 | 1 2
// 2 | 1
// 3 | 3 5
val rowSum = matrix.sumRowVectors

// rowSum (a 1x2 vector) is
// -------
// | 4 8

matrix.mapRows( mapping function ): Matrix

Maps each of the row vectors into a new row vector using a function over the list of non-zero elements in the original rows

// Returns the mean-centered rows of the original matrix
val rowSum = matrix.mapRows{ meanCenter }

def meanCenter[T](vct: Iterable[(T,Double)]) : Iterable[(T,Double)] = {
  val valList = vct.map { _._2 }
  val sum = valList.sum
  val count = valList.size
  val avg = sum / count
  vct.map { tup => (tup._1, tup._2 - avg) }
}
matrix.topRowElems( numberOfElements ): Matrix

Returns the matrix containing only the top K greatest elements in each row as specified by the type-specific comparator. The new matrix pipe has the elements sorted in decreasing order per row key.

// Returns the matrix with top 10 elements
// Given a matrix:
// matrix = jim apple 3
//         jim orange 6
//         jim banana 10
//         jim grapefruit 4
//         bob apple 7
//         bob orange 0
//         bob banana 1
//         bob grapefruit 3

// topkMatrix = jim banana 10
//             jim orange 6
//             bob apple 7
//             bob grapefruit 3
val topkMatrix = matrix.topRowElems(2)

matrix.rowL1Normalize: Matrix

Returns the matrix containing the L1 row-normalized elements of the original matrix

// Returns the adjacency matrix of the follow graph normalized by outdegree
// matrix = 1 0 1
//         1 1 1
//         1 0 0
// matrixL1Norm = 0.5 0 0.5
//                0.33 0.33 0.33
//                1 0 0
val matrixL1Norm = matrix.rowL1Normalize

matrix.rowL2Normalize: Matrix

Returns the matrix containing the L2 row-normalized elements of the original matrix

// Returns the adjacency matrix of the follow graph normalized by outdegree
// matrix = 3 0 4
//         0 0 0
//         2 0 0
// matrixL2Norm = 0.6 0 0.8
//                0 0 0
//                1 0 0
val matrixL2Norm = matrix.rowL2Normalize
matrix.rowMeanCentering: Matrix

Returns the matrix containing the row-mean centered elements of the original matrix, by computing the mean per each row and substracting it from the row elements, such that the original mean value is now zero. (the code is shown above in the mapRows example)

```scala
// Substracts all of the row-wise means from all the elements
val matrixCentered = matrix.rowMeanCentering
```

matrix.rowSizeAveStdev: Matrix

Computes the row size, average and standard deviation returning a new kx3 matrix where each row is mapped to the 3 values.

```scala
// Computes the row stats
// TODO: add example
val matrixRowStats = matrix.rowSizeAveStdev
def rowColValSymbols : List[Symbol] = List(rowSym, colSym, valSym)
```

ColVector operations

Column vector operations are similar to the row vector ones, where all function names are renamed from row to col and the return type is in general ColVector.

Matrix operations

matrix1 * matrix2: Matrix

Computes the product of two matrices

```scala
// Computes the product of two matrices
val matrixProd = matrix1 * matrix2
```

matrix / scalar(LiteralScalar): Matrix

Computes the element-wise division of a matrix by a scalar

```scala
// Computes the element-wise division of a matrix by 100
val matrixDiv = matrix / 100
```

matrix1.elemWiseOp(matrix2){ function }

Computes the element-wise merging of two matrices

```scala
// Computes the element-wise division of a matrix by another
matrix1.elemWiseOp(matrix2)((x,y) => x/y)
```
**matrix1 + matrix2: Matrix**

Computes the sum of two matrices

```java
// Computes the sum of two matrices
matrixSum = matrix1 + matrix2
```

**matrix1 - matrix2: Matrix**

Computes the difference between two matrices

```java
// Computes the difference between two matrices
matrixDiff = matrix1 - matrix2
```

**matrix1.hProd(matrix2): Matrix**

Computes the element-wise product of two matrices

```java
// Computes the element-wise product of two matrices
matrixProd = matrix1.hProd(matrix2)
```

**matrix1.zip(matrix2/row/column): Matrix**

Merges the elements of the two matrices creating a matrix that has as values the pair tuples. Similarly, when zipping a matrix with a row or a vector, it zips the values of the row/column across all of the rows/columns of the matrix. The resulting value type is a pair tuple that has defined a Monoid trait.

```java
// Returns the matrix with elements of the type ( 0, elem2), ( elem1, 0) and (elem1, elem2)
// matrix = 0 1 1
// 1 2 0
// rowVct = 1 0 1
// matrixVctPairs = (0, 1) (1, 0) (1, 1)
// (1, 1) (2, 0) (0, 1)
matrixPairs = matrix1.zip(matrix2)
matrixVctPairs = matrix.zip(rowVct)
```

**matrix.nonZerosWith(Scalar)**

Zips the scalar on the right with all non-zeros in this matrix

```java
// Similar to zip, but combine the scalar on the right with all non-zeros in this matrix:
matrixProd = matrix1.nonZerosWith(100)
```

**matrix.trace: Scalar**

Computes the trace of a matrix that is equal to the sum of the diagonal values of the matrix.

```java
// Computes the trace of a matrix
trace = matrix.trace
```
matrix.sum: Scalar

Computes the sum of the elements of a matrix

```scala
// Computes the sum of the elements of a matrix
sum = matrix.sum
```

matrix.transpose: Matrix

Computes the transpose of the matrix

```scala
// Computes the transpose of the matrix
matrixTranspose = matrix.transpose
```

matrix.diagonal: DiagonalMatrix

Returns the diagonal of the matrix

```scala
// Returns the diagonal of the matrix
matrixDiag = matrix.diagonal
```

ConversionFrom functions

PipeExtensions

pipe.toMatrix(field ): Matrix

Constructs a matrix from a pipe.

```scala
// The matrix will contain all of the data from the pipe and will have the names of the row, col and val. Again, the row and column types do not have to be numeric.
val matrix = pipe.toMatrix('row,'col,'val)
```

pipe.mapToMatrix(fields)(mapping function ): Matrix

Constructs a matrix from a pipe by applying first a mapping function

```scala
// The new pipe contains all of the triples data in matrix with squared values
val matrix = pipe.mapToMatrix('a, 'b, 'c){ (x, y, z) => (x, y, z * z) }
```

pipe.flatMapToMatrix(fields)(mapping function ): Matrix

Constructs a matrix from a pipe by applying first a mapping function

```scala
// The new pipe contains all of the triples data in matrix with log-values
val matrix = pipe.flatMapToMatrix('a, 'b, 'c){ (x, y, z) => (x, y, scala.math.log(z)) }
```
Companion object methods

ConversionTo functions

matrix.pipe: RichPipe

Returns the pipe associated with the matrix. The pipe contains tuples of the three (row,column,value) elements that can be accessed either using the field names or by using the Matrix class variables matrix.rowSym, matrix.colSym, matrix.valSym.

// The new pipe contains all of the triples data in matrix
val newPipe = matrix.pipe

matrix.pipeAs(toFields): RichPipe

Returns the matrix pipe with the fields renamed

// The new pipe contains all of the triples data in matrix renamed to (`user,'movie,'similarity)
val pipeFields = matrix.pipeAs( 'user, 'movie, 'similarity)

matrix.write(src, outFields): Matrix

Writes to a sink and returns the matrix data for further processing.

// Writes to a sink and returns the matrix data for further processing.
matrix.write( Tsv(``output'') )

Other functions

matrix.fields: List[Symbol]

Returns the list of the fields of the pipe associated with the matrix: matrix.rowSym, matrix.colSym, matrix.valSym.

// The new pipe contains all of the triples data in matrix
val pipeFields = matrix.fields

matrix.withSizeHint: Matrix

Adds a SizeHint to the matrix. The size hint specifies the approximate value of the dimensions of the matrix and impacts the type of joins that. If set to true, the skewness flag triggers an additional split of the data over a set of random keys to evenly distribute the computation.

// The new matrix has a new SizeHint specifying that there are around 1000 rows, 4000 columns, 1000 non-zeroes and that there is a skew of the values over the indexes.
val newMatrix = matrix.withSizeHint( 1000, 4000, 1000, true )

matrix.hasHint: SizeHint

Returns true if the matrix has a size hint.

// Check if the matrix has a sizeHint set.
if (!matrix.hasHint) matrix.withSizeHint(newHint)
### 2.2 Calling Scalding from Inside your Application

Starting in scalding 0.12, there is a clear API for doing this. See `Execution[T]`, which describes a set of map/reduce operations that when executed return a `Future[T]`. See the scaladocs for `Execution`. Below is an example.

```scala
val job: Execution[Unit] =
  TypedPipe.from(TextLine(``input'''))
  .flatMap(_.split(``\s+''))
  .map { word => (word, 1L) }
  .sumByKey
  .writeExecution(TypedTsv(``output'''))
// Now we run it in Local mode
val u: Unit = job.waitFor(Config.default, Local(true))

// Or for Hadoop:
val jobConf = new JobConf
val u: Unit = job.waitFor(Config.hadoopWithDefaults(jobConf), Hdfs(true, jobConf))
// If you want to be asynchronous, use run instead of waitFor and get a Future in return
```

For testing or cases where you aggregate data down to a manageable level, `.toIterableExecution` on `TypedPipe` is very useful:

```scala
val job: Execution[Iterable[(String, Long)]] =
  TypedPipe.from(TextLine(``input'''))
  .flatMap(_.split(``\s+''))
  .map { word => (word, 1L) }
  .sumByKey
  .toIterableExecution
// Now we run it in Local mode
val counts: Map[String, Long] = job.waitFor(Config.default, Local(true)).toMap
```

To run an `Execution` as a stand-alone job, see:

1. **ExecutionApp** Make an object `MyExJob` extends `ExecutionApp` for a job you can run like a normal java application (by using java on the classname).
2. **ExecutionJob** - use this only if you have an existing tooling around launching `scalding.Job` subclasses.

#### 2.2.1 Some rules

1. When using `Execution` NEVER use `.write` or `.toPipe` (or call any method that takes an implicit `flowDef`). Instead use `.writeExecution`, `.toIterableExecution`, or `.forceToDiskExecution`. (see scaladocs).
2. Avoid calling `waitFor` or `run` AS LONG AS POSSIBLE. Try to compose your entire job into on large Execution using `.zip` or `.flatMap` to combine Executions. `waitFor` is the same as `run` except it waits on the future. There should be at most 1 calling to `.waitFor` or `.run` in each Execution App/Job.

3. Only mutate vars or perform side effects using `.onComplete`. If you `run` the result of `onComplete`, your function you pass will be run when the result up to that point is available and you will get the Try[T] for the result. Avoid this if possible. It is here to deal with external IO, or existing APIs, and designed for experts that are comfortable using `.onComplete` on scala Futures (which is all this method is doing under the covers).

### 2.2.2 Running Existing Jobs Inside A Library

We recommend the above approach to build composable jobs with Executions. But if you have an existing Job, you can also run that:

**Working example:**

**WordCountJob.scala**

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

**Runner.scala**

```scala
object Runner extends App {
  val hadoopConfiguration: Configuration = new Configuration
  hadoopConfiguration.set(`mapred.job.tracker'',``hadoop-master:8021'')
  hadoopConfiguration.set(`fs.defaultFS'',``hdfs://hadoop-master:8020''

  val hdfsMode = Hdfs(strict = true, hadoopConfiguration)
  val arguments = Mode.putMode(hdfsMode, Args(`--input in.txt --output counts.tsv''))

  // Now create the job after the mode is set up properly.
  val job: WordCountJob = new WordCountJob(arguments)
  val flow = job.buildFlow
  flow.complete()
}
```

And then you can run your App on any server, that have access to Hadoop cluster

### 2.3 Common Exceptions and Possible Reasons

#### 2.3.1 java.lang.NoClassDefFoundError

This is probably when you are missing a dependency, or have a problem with the binary version of a dependency. Some libraries required at run time, such as Hadoop and many of Hadoop’s includes, are expected to be on the classpath and are not bundled by Scalding itself. So check for that.
2.3.2 java.lang.AbstractMethodError

This probably is also a jar version problem.

2.3.3 java.lang.NoSuchMethodError

This probably is also a jar version problem.

2.3.4 java.lang.ClassNotFoundException

1. Your classpath or jar is missing a needed class. Check your build, or perhaps the name of the job you are trying to run.
2. Make sure you specified the fully packaged name of the class you wish to run package.JobName and not just JobName.

2.3.5 java.lang.UnsupportedClassVersionError

You have compiled your code with a later version of javac/scala than is supported by the java vm you are attempting to run with. You need to either downgrade the compiler or upgrade the java installation.

2.3.6 java.util.zip.ZipException

: invalid CEN header (bad signature)
This is the jar file exceeding 64k problem. The solution is to use –bundle option of scald.rb.

2.3.7 cascading.flow.planner.PlannerException

Cascading requires all sources to have final sinks on disk. This exception happens when you miss an output for an input.
It also could signify an attempt to write an unserializable datatype.

2.3.8 com.twitter.scalding.InvalidSourceException

This happens when data is missing from the path you provided.

2.3.9 java.lang.RuntimeException: Please only provide a single value for –some input key here

Try putting quotes around your input value

2.3.10 cascading.flow.FlowException: step failed: (1/2) .../algos/4/data/userIds.tsv, with job id: ...

Common exception when running hadoop commands with sudo. Check that you have given permissions for the user (root) in hdfs: sudo addgroup supergroup; sudo adduser root supergroup
2.4 Frequently Asked Questions

Feel free to add new questions and to ping @Scalding for an answer.

2.4.1 Running Scalding

Who actually uses Scalding?

Twitter uses it in production all over the place!
Check out our [[Powered By]] page for more examples.

I'm having trouble with scald.rb, and I just want to run jars in my own system:

See this conversation on Twitter.

Can Scalding be run on Amazon's Elastic MapReduce?

Yes! See the cascading-user group discussion. We would like to see someone prepare a patch for scald.rb to handle submission of scalding jobs to EMR.

Scalding complains when I use a TimePathedSource and some of the data is missing. How can I ignore that error?

Pass the option --tool.partialok to your job and it will ignore any missing data. It’s safer to work around by either filling with place-holder empty files, or writing sources that will skip known-missing dates. Using that option by default is very dangerous.

I receive this error when running sbt update: Error occurred during initialization of VM. Incompatible minimum and maximum heap sizes specified

In your sbt script, set local min=$( ( $mem / 2 ) )

2.4.2 Writing Jobs

How do I perform windowed calculations (for example, moving average) in Scalding?

You want to use GroupBuilder.scanLeft. A scanLeft is like a foldLeft except that you output each intermediate value. Both of these functions are part of the standard Scala library as well. See StackOverflow for scanLeft examples. For the specific example of moving averages in Scalding, see the cascading-user group discussion.

How do I read a single reduced value from a pipe?

You can’t do that. Instead you should use RichPipe.crossWithTiny to efficiently do a cartesian product of a small set of values to a larger set. The small set might be a single output, from say pipe.groupAll { _.size }. Alternatively, you might kick off a subsequent job in Job.next, and use Source.readAtSubmitter to read the value before you get going (or even in Job.next to see if you need to kick off the next job).
How do I make simple records for use in my Scalding job?

We recommend cases classes defined outside of your Job. Case classes defined inside your job capture an $outer member variable that references the job that is wasteful for serialization. If you have a use case this doesn’t cover, email the cascading-user list or mention @scalding. Dealing with serialization issues well in systems like Hadoop is tricky, and we’re still improving our approaches.

See the discussion on cascading-user.

How do I pass parameters to my hadoop job (number of reducers, memory options, etc.)?

```
hadoop jar myjar \
com.twitter.scalding.Tool \
-D mapred.output.compress=false \
-D mapred.child.java.opts=-Xmx2048m \
-D mapred.reduce.tasks=20 \
com.class.myclass \
--hdfs \
--input $input \
--output $output
```

How do I access the jobConf?

If you want to update the jobConf in your job, the way to do it is to override the config method in Job:

```
https://github.com/twitter/scalding/blob/cee3bb99ebb00db9622c387bee0b2718ab9cea61/scalding-core/src/main/scala/com/twitter/scalding/Job.scala#L163
```

If you really want to just read from the jobConf, you can do it with code like:

```
implicitly[Mode] match {
  case Hdfs(_, configuration) => {
    // use the configuration which is an instance of Configuration
  }
  case _ => error("\'Not running on Hadoop! (maybe cascading local mode?)\'')
}
```

See this discussion: https://groups.google.com/forum/?fromgroups=#!topic/cascading-user/YppTLebWds8

How do I append my parameters to jobConf?

```
class WordCountJob(args : Args) extends Job(args) {

  // Prior to 0.9.0 we need the mode, after 0.9.0 mode is a def on Job.
  override def config(implicit m: Mode): Map[AnyRef,AnyRef] = {
    super.config ++ Map (`\'my.job.name'' -> `\'my new job name''
  }
}
```

What if I have more than 22 fields in my data-set?

Warning: this answer refers to the DEPRECATED Fields API.
Many of the examples (e.g. in the tutorial/ directory) show that the fields argument is specified as a Scala Tuple when reading a delimited file. However Scala Tuples are currently limited to a maximum of 22 elements. To read-in a data-set with more than 22 fields, you can use a List of Symbols as fields specifier. E.g.

```scala
val mySchema = List(`first, `last, `phone, `age, `country)
val input = Csv(``/path/to/file.txt'', separator = '' ,''', fields = mySchema)
val output = TextLine(``/path/to/out.txt'')
input.read
  .project(`age, `country)
  .write(Tsv(output))
```

Another way to specify fields is using Scala Enumerations, which is available in the develop branch (as of Apr 2, 2013), as demonstrated in Tutorial 6:

```scala
object Schema extends Enumeration {
  val first, last, phone, age, country = Value // arbitrary number of fields
}
import Schema._

Csv(``tutorial/data/phones.txt'', separator = '' ,''', fields = Schema)
  .read
  .project(first,age)
  .write(Tsv(``tutorial/data/output6.tsv''))
```

**How do I increase the spill threshold?**

The spilling is controlled with the same hadoop option as cascading:

-Dcascading.spill.list.threshold=1000000

Would keep 1 million items in memory.

The rule of thumb is use as much as you can without getting OOM.

**How do I increase the AggregateBy threshold value?**

You can’t set a default for AggregateBy, you need to set it in each reducer by calling spillThreshold function on GroupBuilder. https://github.com/twitter/scalding/blob/develop/scalding-core/src/main/scala/com/twitter/scalding/GroupBuilder.scala#L97

**Q. My Hadoop job is erroring out with AbstractMethodError or IncompatibleClassChangeError.**

A. If your job has dependencies that clash with Hadoop’s, Hadoop can replace your version of a library (like log4j or ASM) with its own native version. You can fix this with an environment flag that makes sure that your jars show up on the classpath before Hadoop’s. Set these environment variables:

```bash
export HADOOP_CLASSPATH=<your_jar_file>
export HADOOP_USER_CLASSPATH_FIRST=true
```
Q. I’m getting a NotSerializableException on Hadoop job submission.

A. All fields in Job get serialized and sent to Hadoop. Your job contains an object that is not serializable, even with Kryo. This issue may exhibit itself as other exceptions, such as InvocationTargetException, KryoException, or IllegalAccessException. What all these potential exceptions have in common is being related to serialization failures during Hadoop job submission.

First, try to figure out which object is causing the problem.

For a better stacktrace than the usual opaque dump, try submitting your job again with the extendedDebugInfo flag set:

```bash
export HADOOP_OPTS=''-Dsun.io.serialization.extendedDebugInfo=true''; hadoop <your-commands>
```

You should see a much larger stacktrace, with many entries like this:

```
- field (class `com.twitter.scalding.MapsideReduce`, name: `commutativeSemigroup`, type: `interface com.twitter.algebird.Semigroup`)
- object (class `com.twitter.scalding.MapsideReduce`, MapsideReduce[decl:'key', `value'])
- field (class `cascading.pipe.Operator`, name: `operation`, type: `interface cascading.operation.Operation`)
- object (class `cascading.pipe.Each`, Each(_pipe_2*_pipe_3)[MapsideReduce[decl:'key', `value`]])
- field (class `org.jgrapht.graph.IntrusiveEdge`, name: `target`, type: `class java.lang.Object`)
- object (class `org.jgrapht.graph.IntrusiveEdge`, org.jgrapht.graph.IntrusiveEdge@6ed95e60)
- custom writeObject data (class `java.util.HashMap`)
- object (class `java.util.LinkedHashMap`, [{?}:UNKNOWN], [{2}:0:1]
```

Typically, if you start reading from the bottom of these entries upward, the first familiar class you see will be the object that’s being unexpectedly serialized and causing you issues. In this case, the error was with Scalding’s `=MapsideReduce= class.

Once you know which object is causing the problem, try one of the following remedies:

1. Put the object in a lazy val
2. Move it into a companion object, which will not be serialized.
3. If the item is only needed at submission, but not on the Mappers/Reducers, make it `@transient`.

If you see a common case we overlooked, let us know. Some common issues are inner classes to the Job (don’t do that), Logger objects (don’t put those in the job, put them in a companion), and some mutable Guava objects have given us trouble (we’d love to see this ticket closed: https://github.com/twitter/chill/issues/66)

### 2.4.3 Issues with Testing

#### How do I get my tests working with Spec2?

from Alex Dean, @alexatkeplar

The problem was in how I was defining my tests. For Scalding, your Specs2 tests must look like this:

```scala
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```scala
```
```
```
```
A job which trys to do blah'' should {
  `successfully do blah'' in {
    <<RUN JOB>>
    expected.blah must_== actual.blah
  }
}
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
2.5.1 LOAD

Pig:
A = LOAD `foo`

Scalding:

```scala
// The TextLine source splits the input by lines.
val textSource = TextLine(args(``input''))
// Create a type-safe pipe from the TextLine.
val lines: TypedPipe[Array[String]] = TypedPipe.from[String](textSource)
```

2.5.2 STORE

Pig:
STORE B INTO `bar`

Scalding:

```scala
b.write(TypedTsv[String](args(``output'')))  // using pattern matching to name elements of a tuple
```

2.5.3 FOREACH

Pig:
B = FOREACH A GENERATE /* expression */

Scalding:

```scala
val b = a.map((t) => /* expression */)
```

2.5.4 FILTER

Pig:
B = FILTER A BY foo == 0

Scalding:

```scala
a.filter{ case (foo, bar) => foo == 0 }  // using pattern matching to name elements of a tuple
// if you don't need to name an element you can use the _ wildcard instead
a.filter{ case (foo, _) => foo == 0 }
```

2.5.5 FOREACH A GENERATE FLATTEN(...)

in Scalding the use of flatMap is similar to the following in Pig:
B = FOREACH A GENERATE FLATTEN(Tokenize(text))

in Scalding:

```scala
def tokenize(s: String) = s.split(``\\s+'`).toList
b = a.flatMap(tokenize(_))  // which produces the same result as:
b = a.map(tokenize(_)).flatten()
// and the same as
```
b = a.map(tokenize(_)).flatten // empty parens are usually omitted

### 2.5.6 Aggregating

**Pig:**

B = FOREACH (GROUP A BY $0) GENERATE COUNT(A)

**Scalding:**

val b = a.groupBy(_.label).size

notice the _ shorthand used here.

### 2.5.7 Join

**Pig:**

C = JOIN A BY $0, B BY $0

**Scalding:** assuming a and b are both a Pipe[(K,V)], you can join them as follows

val c = a.join(b)

### 2.5.8 Scala cheat sheet

It is recommended to know the basics of Scala when trying out Scalding. Here are some common things Scala noobs may become confused about coming from Java and Pig.

**Primitive types**

Scala uses the java primitive type names but with the first letter capitalized. (Scala uses the boxed type automatically when needed.)

For example:

**Java:**

```java
final int a = 1
```

**Scala:**

```scala
val a = 1 // (val means it's a constant. Type is inferred. use var for variables)
val a: Int = 1 // same thing with explicit type declaration
```

**Functions**

```scala
def f(x:Int) = x * 2 // return type inferred

def f(x:Int): Int = x * 2 // same thing with explicit return type
```
Common types

Case classes

A case class is an immutable data class that can be used in pattern matching. For example:

case class User(val firstname: String, val lastname: String)

is kind of similar to the following Java code (plus the added benefit of pattern matching):

```scala
final class User {
   // these are immutable so it's fine to make them public
   public final String firstname;
   public final String lastname;
   public User(String firstname, String lastname) {
      this.firstname = firstname;
      this.lastname = lastname;
   }
}
```
t match {
    case (a, _) => a
}

Similarly with case classes:

// This is the same as User.apply(``Jack'', ``Jackson''). Not a constructor call
val u = User(``Jack'', ``Jackson'')

val v = u match {
    case User(firstname, lastname) => firstname
    ... // other cases
}

More advanced pattern matching

case class Name(first: String, middle: String, last: String)
case class Address(street: String, zip: String, city: String)
case class Person(name: Name, age: Int, address: Address)

val p = Person(Name(``Bob'', ``E.'', ``Roberts''), 42, Address(``23 colorado st.'', ``99999''))

// unwrap Person
p match { case Person(a,b,c) => (a,b,c) }

// unwrap Person and Name
p match { case Person(Name(f,m,l), b, c) => (f, m, l, b, c) }

// multiple case statements (anonymizing minors not in the ``Roberts'' family)
p match {
    // matches only when lastname in Name is ``Roberts''
    case Person(Name(first, _,``Roberts''), _, _) => first

    // predicate can be applied as well
    case Person(Name(first, _, _), age, _) if (age > 21) => first

    // default case if none of the above applies
    case _ => ``anonymous''
} 

// just extracting age
val age = p match { case Person(_, age, _) => age }

// The previous line does the same thing as
val age = p.age

// flattening the entire structure
p match {
    case Person(Name(f,m,l), age, Address(street, zip, city)) =>
        (f, m, l, age, street, zip, city)
}
Typed pipes basics

Map

If we have the following:

- p1 of type TypedPipe[T]
- f of type Function1[T,U]

then we can do

```scala
val p2: Pipe[U] = p1.map(f)
```
p2 is of type Pipe[U]

Lambda syntax

When defining a function inline we use the following syntax:

```scala
(param1, param2, ...) => /* expression */
```
which can be used in map

```scala
p.map( (a) => a + 1 )
```
Here we are defining a function that takes one parameter named `a` and apply it to all elements of `p`

Map variations

With `p1` of type `Pipe[(Int, String)]` (a `Pipe` of `Tuple2[Int, String]`) mapping elements in `p1`:

```scala
p1.map( (t) => t._1 )
```
When a function takes only one parameter and is extremely simple, we can use the following shorthand:

```scala
p1.map( _._1 )
```
This syntax defines a function that takes one parameter on which we call `.1` (get the first element of the tuple)

**WARNING:** `_` expands only to the expression directly around it. `.1` works but `(._1)._2` does not. (It turns into `((t) => (t._1))._2` which does not compile.) Always fallback to the full syntax when in doubt: `(t) => (t._1)._2` works.

In Scala the syntax for getting a field is the same as for calling a parameter-less method (parentheses are omitted). In fact, getting a fields is calling a parameter-less methods.

Operator notation to call a function

```scala
p1 map f
```
is the same as

```scala
p1.map(f)
```
In Scala every method can be used as an operator. In fact, this is how operators are implemented as symbols are allowed in method names.

```scala
p1 filter { _._1 == 0 } map { _._2 }
```
also:
pl.map { (t) => t._1 }

Notice the curly braces, we’re executing a block of code that returns a function. The result (last statement) of `{ }` is passed to map

```scala
// ``foo'' is printed once (before passing the function to map)
pl.map { println(``foo''); (t) => t._1 }
```

```scala
// ``foo'' is printed once (before passing the function to map)
pl.map { println(``foo''); _._1 }
```

```scala
// ``foo'' is printed for each element
pl.map{ (t) => { println(``foo''); t._1 } }
```

**Pattern matching shorthand**

```scala
pl.map { case (a,b) => a }
```

Passing a block of code that returns a partial function is a short hand for:

```scala
pl.map( (t) => t match { case (a,b) => a } )
```

## 2.6 REPL Reference

### 2.6.1 Enrichments available on TypedPipe/Grouped/CoGrouped objects:

- `.save(dest: TypedSink[T]): TypedPipe[T]`: runs just the flow to produce this pipe and saves to the given sink, returns a new pipe reading from the source
- `.snapshot: TypedPipe[T]`: Runs just the part of the flow needed to produce the output for this pipe, saves the results to a SequenceFile (if in Hadoop mode), or to memory (in Local mode), and returns a new TypedPipe which reads from the new source.
- `.toIterator`: Returns an iterator. If there is any work to be done to produce the values, it will automatically call `snapshot` first and return an iterator on the snapshot instead.
- `.dump`: Print the contents to stdout (uses `.toIterator`)
- `.toList`: Load the contents into an in-memory List (uses `.toIterator`)

Additionally, `ValuePipe` gets the enrichment `.toOption` which is like `.toIterator` but for a single value.

### 2.6.2 Repl globals

Everything in a REPL session shares a common FlowDef to allow users to build up a large complicated flow step-by-step and then run it in its entirety. When building a job this way, use `write` (rather than `save`) to add sinks to the global flow.

Importing `ReplImplicits._` makes the following commands available:

- `.run`: Runs the overall flow. Trims anything not feeding into a persistent sink (created by `write`). Note: any sinks created but not connected up will cause this to fail. Extra sources, however, are trimmed.
- `.resetFlowDef`: Clear the global FlowDef. Will require that all pipes/sources/sinks be re-created.

`ReplImplicits` also includes the execution context, which can be tweaked manually:
• mode: Local or Hdfs mode (determined initially by how the repl was launched but can be changed later)
• config: Various configuration parameters, including Hadoop configuration parameters specified on the command line, by the Hadoop cluster, or by Scalding.

2.6.3 Customizing the REPL

• On startup, the REPL searches for files named .scalding_repl from the current directory up to the filesystem root and automatically loads them into the current session in order of specificity. Common definitions, values, etc can be put in these files. For instance, one can specify global defaults in their home directory (~/.scalding_repl), and project-specific settings in the project directory (~/workspace/scalding/.scalding_repl).
• ScaldingShell.prompt: () => String: A function to produce the prompt displayed by Scalding. This can also be customized. For instance, to print the current directory at the prompt:

```javascript
ScaldingShell.prompt = { () =>
    Console.GREEN +
    System.getProperty(``user.dir'').replace(System.getProperty(``user.home''),``~'') +
    Console.BLUE + '' scalding> `` + Console.RESET
}
```

2.7 Rosetta Code

A collection of MapReduce tasks translated (from Pig, Hive, MapReduce streaming, etc.) into Scalding. For fully runnable code, see the repository here.

2.7.1 Word Count

Hadoop Streaming (Ruby)

```ruby
# Emit (word, count) pairs.
def mapper
    STDIN.each_line do |line|
        line.split.each do |word|
            puts [word, 1].join(``	``)
        end
    end
end

# Aggregate all (word, count) pairs for a particular word.
#
# In Hadoop Streaming (unlike standard Hadoop), the reducer receives
# rows from the mapper *one at a time*, though the rows are guaranteed
# to be sorted by key (and every row associated to a particular key
# will be sent to the same reducer).
def reducer
    curr_word = nil
    curr_count = 0
    STDIN.each_line do |line|
        word, count = line.strip.split(``	``)
        if word != curr_word
```
puts [curr_word, curr_count].join(`	`)  
curr_word = word  
curr_count = 0  
end  
curr_count += count.to_i  
end

puts [curr_word, curr_count].join(`	`) unless curr_word.nil?

Hive

# tokenizer.py
import sys

for line in sys.stdin:  
    for word in line.split():  
        print word

SQL

CREATE TABLE tweets (text STRING);
LOAD DATA LOCAL INPATH `tweets.tsv' OVERWRITE INTO TABLE tweets;

SELECT word, COUNT(*) AS count  
FROM (  
    SELECT TRANSFORM(text) USING `python tokenizer.py' AS word  
    FROM tweets  
) t  
GROUP BY word;

Pig

tweets = LOAD `tweets.tsv' AS (text:chararray);
words = FOREACH tweets GENERATE FLATTEN(TOKENIZE(text)) AS word;
word_groups = GROUP words BY word;
word_counts = FOREACH word_groups GENERATE group AS word, COUNT(words) AS count;
STORE word_counts INTO `word_counts.tsv';

Cascalog 2.0

(cascalog.repl/bootstrap)

(\<-(hfs-textline `\word_counts.tsv\') [?word ?count]  
  (hfs-textline `\tweets.tsv\') ?text)  
  ((mapcatfn [text] (.split text `\s+')) ?text => ?word)  
  (c/count ?count))
import com.twitter.scalding._
import com.twitter.scalding.source.TypedText

class ScaldingTestJob(args: Args) extends Job(args) {
    TypedText.tsv[String](``tweets.tsv'')
        .flatMap(_.split(``\s+'''))
        .groupBy(_.size)
        .write(TypedText.tsv[String](``word_counts.tsv''))
}

### 2.7.2 Distributed Grep

#### Hadoop Streaming (Ruby)

PATTERN = /.*hello.*/

# Emit words that match the pattern.
def mapper
    STDIN.each_line do |line|
        puts line if line =~ PATTERN
    end
end

# Identity reducer.
def reducer
    STDIN.each_line do |line|
        puts line
    end
end

#### Pig

%declare PATTERN `.*hello.*';

tweets = LOAD `tweets.tsv' AS (text:chararray);
results = FILTER tweets BY (text MATCHES `$PATTERN');

#### Cascalog

(def pattern #'`.*hello.*')

(deffilterop matches-pattern? [text pattern]
    (re-matches pattern text))

(defn distributed-grep [input pattern]
    (<- [?textline]
        (input ?textline)
        (matches-pattern? ?textline pattern))

    (?- (stdout) (distributed-grep (hfs-textline `tweets.tsv'') pattern))
Scalding

import com.twitter.scalding.source.TypedText

val Pattern = ".*hello.*".r

TypedText.tsv[String](``tweets.tsv'').filter { _.matches(Pattern) }

2.7.3 Inverted Index

Hadoop Streaming (Ruby)

# Emit (word, tweet_id) pairs.
def mapper
  STDIN.each_line do |line|
    tweet_id, text = line.strip.split(``	``)
    text.split.each do |word|
      puts [word, tweet_id].join(``	``)
    end
  end
end

# Aggregate all (word, tweet_id) pairs for a particular word.
# In Hadoop Streaming (unlike standard Hadoop), the reducer receives
# rows from the mapper *one at a time*, though the rows are guaranteed
# to be sorted by key (and every row associated to a particular key
# will be sent to the same reducer).
def reducer
  curr_word = nil
  curr_inv_index = []
  STDIN.each_line do |line|
    word, tweet_id = line.strip.split(``	``)
    if word != curr_word
      # New key.
      puts [curr_word, curr_inv_index.join('','')].join(``	``)
      curr_word = word
      curr_inv_index = []
    end
    curr_inv_index << tweet_id
  end
  unless curr_word.nil?
    puts [curr_word, curr_inv_index.join('',' '')].join(``	``)
  end
end

Pig

tweets = LOAD `tweets.tsv' AS (tweet_id:int, text:chararray);

words = FOREACH tweets GENERATE tweet_id, FLATTEN(TOKENIZE(text)) AS word;
word_groups = GROUP words BY word;
inverted_index = FOREACH word_groups GENERATE group AS word, words.tweet_id;

Cascalog

;;; define the data
(def index [
  [0 "Hello World"]
  [101 "The quick brown fox jumps over the lazy dog"]
  [42 "Answer to the Ultimate Question of Life, the Universe, and Everything"]
])

;;; the tokenize function
(defmapcatop tokenize [text]
  (seq (.split text "\s+'")))

;;; ensure inverted index is distinct per word
(defbufferop distinct-vals [tuples]
  (list (set (map first tuples))))

;;; run the query on data
(?<- (stdout) [?word ?ids]
  (index ?id ?text)
  (tokenize ?text => ?word)
  (distinct-vals ?id => ?ids))

Scalding

import com.twitter.scalding.source.TypedText

val invertedIndex =
  TypedText.tsv[Int,String](``tweets.tsv'')
  .flatMap { case (tweetId, text) => text.split(``\s+'').map((_, tweetId)) }
  .group

2.8 Scald.rb

The scald.rb script in the scripts/ directory is a handy script that makes it easy to run jobs in both local mode or on a remote Hadoop cluster. It handles simple command-line parsing, and copies over necessary JAR files when running remote jobs.

If you’re running many Scalding jobs, it can be useful to add scald.rb to your path, so that you don’t need to provide the absolute pathname every time. One way of doing this is via (something like):

```
ln -s scripts/scald.rb $HOME/bin/
```

This creates a symlink to the scald.rb script in your $HOME/bin/ directory (which should already be included in your PATH).

More information coming soon.
2.9 Scalding Commons

(This page contains the README for the former scalding-commons library. All scalding-commons code has been merged into the main Scalding repo as of June 6th, 2013.)

Common extensions to the Scalding MapReduce DSL.

2.9.1 Dfs-Datastores Integration

Scalding-Commons includes Scalding Sources for use with the dfs-datastores project.

This library provides a VersionedKeyValSource that allows Scalding to write out key-value pairs of any type into a binary sequencefile. Serialization is handled with the bijection-core library’s Injection trait.

VersionedKeyValSource allows multiple writes to the same path, as write creates a new version. Optionally, given a Monoid on the value type, VersionedKeyValSource allows for versioned incremental updates of a key-value database.

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

// ## Sink Example

// The bijection library provides implicit Injections
// from String -> Array[Byte] and Int -> Array[Byte].
val versionedSource = VersionedKeyValSource[String,Int](``path'')

// creates a new version on each write
someScaldingFlow.write(versionedSource)

// because Scalding provides an implicit Monoid[Int],
// the writeIncremental method will add new integers into
// each value on every write:
someScaldingFlow.writeIncremental(versionedSource)

// ## Source Examples

// This Source produces the most recent set of kv pairs from the VersionedStore
// located at `path':
VersionedKeyValSource[String,Int](``path'')

// This source produces version 12345:
VersionedKeyValSource[String,Int](``path'', Some(12345))
```

2.10 Scalding-HBase

2.10.1 Resources

- Running Scalding with HBase support - a github example project.
- Spy Glass - Advanced featured HBase wrapper for Cascading and Scalding
- Maple - a collection of Cascading Taps, including a simple HBase tap. Spy Glass appears to be the more advanced option.
2.11 Scalding Sources

Scalding sources are how you get data into and out of your scalding jobs. There are several useful sources baked into the project and a few more in the scalding-commons repository. Here are a few basic ones to get you started:

- To read a text file line-by-line, use TextLine(filename). For every line in filename, this source creates a tuple with two fields:
  - line contains the text in the given line
  - offset contains the byte offset of the given line within filename

- To read or write a tab- or comma-separated values file, use Tsv or Csv.
  - When reading a Tsv or Csv, Scalding will choose field names based on the input file’s headers.
  - When writing a Tsv or Csv, Scalding will write out headers with the field names.

- To create a pipe from data in a Scala Iterable, use the IterableSource. For example, IterableSource(List(4,8,15,16,23,42), 'foo) will create a pipe with a field 'foo. IterableSource is especially useful for unit testing.

- A NullSource is useful if you wish to create a pipe for only its side effects (e.g., printing out some debugging information). For example, although defining a pipe as Csv("foo.csv").debug without a sink will create a java.util.NoSuchElementException, adding a write to a NullSource will work fine: Csv("foo.csv").debug.write(NullSource).

2.12 Using Scalding with other versions of Scala

There are two places you need to change in order to make this run correctly:

1. Change the scala version in the build.sbt in the project root.

2. Since scald.rb calls the compiler on your script, it needs to match the version you use above. Edit the top of scald.rb to change SCALAC to point to the version of the compiler that matches the project (sorry about this, patches accepted).
Other

Contents:
Scalding comes with an executable tutorial set that does not require a Hadoop cluster. If you’re curious about Scalding, why not invest a bit of time and run the tutorial yourself and make your own judgement?

The fact is, the APIs of these systems are all very similar if we compare the TypedAPI of Scalding to Scrunch or Scoobi (or Spark). Scalding is probably the most heavily used in terms of total data processed, given the heavy use at Twitter, Stripe, Etsy and many other places.

3.1 Difference between the Fields API of scalding and Scoobi and Scrunch

Scalding was developed before either of those projects were announced publicly and has been used in production at Twitter for more than six months (though it has been through a few iterations internally). The main difference between Scalding (and Cascading) and Scrunch/Scoobi is that Cascading has a record model where each element in your distributed list/table is a table with some named fields. This is nice because most common cases are to have a few primitive columns (ints, strings, etc...). This is discussed in detail in the two answers to the following question: http://www.quora.com/Apache-Hadoop/What-are-the-differences-between-Crunch-and-Cascading

Scoobi and Scrunch stress types and do not use field names to build ad-hoc record types. Cascading’s fields are very convenient, and our users have been very productive with Scalding. Fields do present problems for type inference because Cascading cannot tell you the type of the data in Fields(“user_id”, “clicks”) at compile time. This could be surmounted by building a record system in scala that allows the programmer to express the types of the fields, but the cost of this is not trivial, and the win is not so clear.

Scalding supports using any scala object in your map/reduce operations using Kryo serialization, including scala Lists, Sets, Maps, Tuples, etc. It is not clear that such transparent serialization is present yet in scrunch. Like Scoobi, Scalding has a form of MSCR fusion by relying on Cascading’s AggregateBy operations. Our Reduce primitives (see GroupBuilder.reduce and .mapReduceMap) are comparable to Scoobi’s combine primitive, which by default uses Hadoop combiners on the map side.

Lastly, Scalding comes with a script that allows you to write a single file and run that single file locally or on your Hadoop cluster by typing one line: scald.rb [--local] myJob.scala. It is really convenient to use the same language/tool to run jobs on Hadoop and then to post-process the output locally.
3.2 Field Rules

3.2.1 Map-phase

In the map phase (map/flatMap/pack/unpack) the rule is: if the target fields are new (disjoint from the input fields), they are appended. If source or target fields are a subset of the other, only the results are kept. Otherwise, you get an exception at flow planning stage (there is some overlap but not subset relationship).

If you use mapTo/flatMapTo/packTo/unpackTo, only the results are kept.

3.2.2 GroupBy

In the groupBy, the keys are always kept and only the target fields. So, `groupBy('x) { _.sum('y -> 'ys).sum('z -> 'zs) }` will return a pipe with three columns: ('x, 'ys, 'zs).

3.2.3 Joins

Joins keep all the fields. For inner joins, the field names can collide on the fields you are joining, and in that case only one copy is kept in the result. Otherwise, all field names must be distinct.

3.2.4 Cascading Experts

The documentation in FieldConversions.scala might be helpful for Cascading experts.

```scala
/**
 * Rather than give the full power of cascading's selectors, we have
 * a simpler set of rules encoded below:
 * 1) if the input is non-definite (ALL, GROUP, ARGS, etc...) ALL is the output.
 * 2) If one of from or to is a strict super set of the other, SWAP is used.
 * 3) If they are equal, REPLACE is used.
 * 4) Otherwise, ALL is used.
 */
```

3.3 Powered By Scalding

Want to be added to this page? Send a tweet to @scalding or open an issue.
<table>
<thead>
<tr>
<th>Company</th>
<th>Scalding Use Case</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>We use Scalding often, for everything from custom ad targeting algorithms, market insight, click prediction, traffic quality to PageRank on the Twitter graph. We hope you will use it too!</td>
<td></td>
</tr>
<tr>
<td>Etsy</td>
<td>We’re starting to use Scalding alongside the JRuby Cascading stack described <a href="https://github.com/twitter/scalding/issues/1264">here</a>. More to come as we use it further.</td>
<td></td>
</tr>
<tr>
<td>Ebay</td>
<td>We use Scalding in our Search organization for ad-hoc data analysis jobs as well as more mature data pipelines that feed our production systems.</td>
<td></td>
</tr>
<tr>
<td>Snowplow Analytics</td>
<td>Our data validation &amp; enrichment process for event analytics is built on top of Scalding.</td>
<td>GitHub</td>
</tr>
<tr>
<td>PredictionIO</td>
<td>Machine-learning algorithms build on top of Scalding.</td>
<td>GitHub</td>
</tr>
<tr>
<td>Gatling</td>
<td>We’ve just rebuilt our reports generation module on top of Scalding. Handy API on top of an efficient engine.</td>
<td>GitHub</td>
</tr>
<tr>
<td>SoundCloud</td>
<td>We use Scalding in our search and recommendations production pipelines to pre and post-process data for various machine learning and graph-based learning algorithms. We also use Scalding for ad-hoc and regular jobs run over production logs for things like click tracking and quality evaluation on search results and recommendations.</td>
<td></td>
</tr>
<tr>
<td>Sonar</td>
<td>Our platform is built on Hadoop, Scalding, Cassandra and Storm. See Sonar’s job listings.</td>
<td></td>
</tr>
<tr>
<td>Sky</td>
<td>Sky is using Scalding on Hadoop and utilizing HBase through the SpyGlass library for statistical analysis, content related jobs and reporting.</td>
<td></td>
</tr>
<tr>
<td>LivePerson</td>
<td>LivePerson’s data science group is using Scalding on Hadoop, to develop machine learning algorithms and big data analysis.</td>
<td></td>
</tr>
<tr>
<td>Sharethrough</td>
<td>Sharethrough uses Scalding throughout our production data infrastructure. We use it for everything from advertiser reporting and ML feature engineering, to ad targeting and click forecasting.</td>
<td></td>
</tr>
<tr>
<td>LinkedIn</td>
<td>Scalding is being used at LinkedIn both at the Product Data Science team and the Email Experience team.</td>
<td></td>
</tr>
<tr>
<td>Stripe</td>
<td>Stripe uses Scalding for ETL and machine learning to support our analytics and fraud prevention teams.</td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td>Move uses Scalding on Hadoop for advanced analytics and personalization for Realtor.com and its mobile real estate apps.</td>
<td></td>
</tr>
<tr>
<td>Tapad</td>
<td>Tapad uses scalding to manage productized analytics and reporting, internal ad-hoc data mining, and to support our data science team’s research and development efforts.</td>
<td></td>
</tr>
<tr>
<td>CrowdStrike</td>
<td>CrowdStrike employs Scalding in our data science and data mining pipelines as part of our big data security platforms in research, development, product and customer endpoints. We have plans to open source our Scalding API (AWS, EMR) on github.</td>
<td></td>
</tr>
<tr>
<td>Tumblr</td>
<td>Tumblr uses scalding as a sort of MVC framework for Hadoop. Applications include recommendations/discovery, spam detection, and general ETL.</td>
<td></td>
</tr>
<tr>
<td>Elance</td>
<td>Elance uses scalding for constructing data sets for search ranking, recommendation systems, other modeling problems.</td>
<td></td>
</tr>
<tr>
<td>Commonwealth Bank Of Australia</td>
<td>Commbank uses scalding as a key component within its big data infrastructure. Both on the ETL side, and for the implementation of data science pipelines for building various predictive models.</td>
<td>GitHub</td>
</tr>
<tr>
<td>Sabre Labs</td>
<td>Sabre Labs uses Scalding for ETL and ad hoc data analysis of trip information.</td>
<td></td>
</tr>
<tr>
<td>gutefrage.net</td>
<td>gutefrage.net uses Scalding for its Data Products and general ETL flows.</td>
<td></td>
</tr>
<tr>
<td>MediaMath</td>
<td>MediaMath uses Scalding to power its Data Platform, the centralized data store that powers our ad hoc analytics, client log delivery and new optimization/insight-based products.</td>
<td></td>
</tr>
<tr>
<td>The Search Party</td>
<td>The Search Party is using Scalding to build production machine learning libraries for clustering, recommendation and text analysis of recruitment related data. Scalding is a breath of fresh air!</td>
<td></td>
</tr>
<tr>
<td>Opower</td>
<td>Opower uses Scalding and KijiExpress to analyze the world’s energy data and extract machine learning-based insights that power behavior change.</td>
<td></td>
</tr>
<tr>
<td>Barclays</td>
<td>Barclays uses Scalding for Data Warehousing, ETL and data transformation into (query optimized) data formats.</td>
<td></td>
</tr>
</tbody>
</table>
3.4 Run in IntelliJ Idea

You can run your job from IDEA locally. Run -> Edit configurations -> New Application

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main class</td>
<td>com.twitter.scalding.Tool</td>
</tr>
<tr>
<td>VM options</td>
<td>-XX:MaxPermSize=512M -Xmx1024M</td>
</tr>
<tr>
<td>Program arguments</td>
<td>job.class.name -hdfs -param1 value -param2 value -input local-path</td>
</tr>
<tr>
<td>Working directory</td>
<td>~/projects/scalding-jobs</td>
</tr>
<tr>
<td>Use classpath of module</td>
<td>first-party</td>
</tr>
</tbody>
</table>

Note that `--output` is optional usually. When not specified we write to NullSource which prints to stdout.

Here is sample scalding job:

```scala
package com.sample

import com.twitter.scalding._
import com.twitter.scalding.Tsv

class SampleJob(args: Args) extends Job(args) {
  val input = args("input")
  val output = args.getOrElse("output", null)

  val results = Tsv(input).read
  //todo do something here

  if (output != null)
    results.write(Tsv(output))
  else
    results.debug.write(NullSource)
}
```

3.5 Scala and SBT for Homebrew users

For Mac users who use Homebrew, this will get their Scala and sbt versions right:

```bash
cd /usr/local/Library/Formula
git checkout 0e16b9d /usr/local/Library/Formula/scala.rb
brew install scala
brew install sbt
```

3.6 Scala and SBT for Homebrew users

As of this writing, MacPorts (2.1.2) installs sbt 0.12, which is not supported by either master or develop. Applying this patch was found to work:

```bash
diff --git a/build.sbt b/build.sbt
index 4363ee8..e19a2de 100644
--- a/build.sbt
+++ b/build.sbt
@@ -43,6 +43,8 @@ excludedJars in assembly <= (fullClasspath in assembly) map { cp =>
          // Some of these files have duplicates, let's ignore:
          mergeStrategy in assembly <= (mergeStrategy in assembly) { (old) =>
```
3.7 Scalding on Amazon Elastic MapReduce

I copied this from the Google Group Discussion about how to get Scalding running in EMR:

I was able to successfully execute the WordCountJob Scalding’s example on Amazon EMR. To recap, here are the steps I took:

- I excluded Hadoop from the scalding assembly by changing the last lines in build.sbt:

```scala
excludedJars in assembly <<= (fullClasspath in assembly) map { cp =>
  cp filter { Set(``janino-2.5.16.jar'', ``hadoop-core-0.20.2.jar'') contains _.data.getName}
}
```

- I uploaded the resulting scalding jar and the hello.txt file to Amazon S3

- I created an EMR job using a custom jar. From the command line, it looks like this:

```bash
elastic-mapreduce --create --name `Test Scalding'' --jar s3n://<bucket-and-path-to-scalding-assembly-0.3.5.jar> --arg ...
```

Voilà! :-)

For an example of a standalone Scalding job which can be run on Amazon EMR, please see:

https://github.com/snowplow/scalding-example-project
3.8 Scalding with CDH3U2 in a Maven Project

3.8.1 Introduction

Aim

This wiki describes a procedure that should allow the dedicated reader to create an executable jar file implementing Scalding, using Maven, that is readily available for deployment on CDH3U2 cluster.

Hadoop Flavors and Compatibility Issues

To deploy a MapReduce job on any Hadoop cluster, since the different Hadoop versions are not necessarily compatible with each other, one has to ensure that the core Hadoop libraries the client code uses are identical to those found throughout the entire cluster. Roughly said, client code that is planned to be deployed as an executable jar, should use the same exact jars as are used by the server nodes on the cluster. See http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/ for a walk down the Hadoop and Cloudera version road of chaos.

3.8.2 Protocol

Prerequisites

- Scalding source - here we used v0.5.3
- SBT - to build Scalding
- Cloudera’s Hadoop (CDH) - binaries are fine, e.g. hadoop-0.20.2-cdh3u2.tar.gz. Other versions are cool, just use the same version your cluster uses.
- IDE with Maven support - here I use Eclipse. There is no need for an IDE if you are a Maven wizard. I am not one of those.

Procedure

1. CD to your Scalding source directory
2. Edit build.sbt to exclude the hadoop-core jar from being packaged in Scalding:
   
   ```scala
   excludedJars in assembly <<= (fullClasspath in assembly) map { cp =>
     cp.filter { Set(``janino-2.5.16.jar'', ``hadoop-core-0.20.2.jar'') contains _.data.getName }
   }
   ```
   
   (https://gist.github.com/238d74b081d9f2c6e5f1)
3. sbt -29 update (-29 is a flag for SBT to build with Scala 2.9.1 libraries. Use if you intend to implement your code with this version of Scala)
4. sbt -29 assembly (creates scalding-assembly.0.5.3.jar)
5. My own preference is to install self compiled jars in my local Maven repository. Therefore I use mvn install:install-file target (see http://maven.apache.org/plugins/maven-install-plugin/usage.html) to install the created scalding-assembly.0.5.3.jar locally. From hereon this jar’s spec are groupId=com.twitter artifactId=scalding-assembly version=0.5.3.cdh3u2
6. Download Cloudera’s hadoop-0.20.2-cdh3u2.tar.gz
7. As in 5, install locally your hadoop-core-cdh3u2.jar, or alternatively you can embed Cloudera’s parent pom in your project’s pom (in the following steps) - they have instructions somewhere on their website.

8. In your IDE, create a new Scala project using/based on this pom: https://gist.github.com/40f1838bddd15cc25b21

9. Create the file src/assembly/job.xml and edit: https://gist.github.com/9c5e6f04da287667983a

10. The fun part! - Create your Scala class implementing Scalding’s Job

```scala
class SomethingCool(args: Args) extends Job(args)
```

11. `mvn package` (creates a fat jar)

12. The generated jar would be placed under your project’s target folder, named like: YOURPROJECT-0.0.1-SNAPSHOT-job.jar

13. CD to your hadoop-0.20-cdh3u2 folder

14. Setup your Hadoop configuration files (most importantly, your conf/core-site.xml file) and edit

```xml
<property>
  <name>fs.default.name</name>
  <value>hdfs://namenode.somethingcool.com:8020/</value>
</property>
```

15. Run

```
bin/hadoop jar YOURPROJECT-0.0.1-SNAPSHOT-job.jar com.twitter.scalding.Tool your.package.your.class --hdfs --input hdfs://namenode.somethingcool.com/user/hdfs/tmp/hello.txt --output hdfs://namenode.somethingcool.com/user/hdfs/tmp/hello_out.txt
```

### 3.9 Upgrading to 0.9.0

1. `def config` in Job does not accept a mode. Job has `Job.mode` use that if you need it. `def listeners` in Job no longer accepts a mode either.

2. `sum` takes a type parameter in the fields API. `sum[Double]` is equivalent to the old behavior, but you might want `sum[Long]` or any other `T` that has an `algebird.Semigroup[T]`. Without the type parameter you get diverging implicit expansion for type `com.twitter.algebird.Semigroup[T]`

3. TypedSink and Mappable need setter and converter defined. Using `TupleSetter.asSubSetter`, or `TupleConverter.asSuperConverter` can help here. (add better docs, please if you get confused).

4. RichDate parsing needs an implicit scalding.DateParser. Job has one in scope that follows the old rules (minus natty), but you may need to set an implicit DateParser outside of a Job. (See `DateParser.default`).

5. `JsonLine` has been extracted into scalding-json module.

These `sed` rules may help.

### 3.10 Using the Distributed Cache

The distributed cache is simply hadoop’s method for allowing each node local access to a specific file. In the example, I am mapping ip addresses to geographical locations (country, city, etc.). The heavy lifting is done my Maxmind’s geop LookupService. LookupService requires random access to a local file, GeoLiteCity.dat, which defines a mapping from ip ranges to a location object.

The DistributedCacheFile object provides a simple API for dealing with registering files to be copied into the file cache at configuration time, and accessing them later during map/reduce time.
Firstly, we need to place the file in question on the hadoop file system. I used `s3distcp` to put a copy of GeoLiteCity.dat to hdfs://cache/GeoLiteCity.dat. If your copy is not on s3, you can use the normal distcp command. So now GeoLiteCity.dat is distributed across the data nodes. So let now go to the scalding code.

```scala
import com.twitter.scalding.filecache.DistributedCacheFile

val geoIPDataFile = DistributedCacheFile(`hdfs://path/to/maxmind.dat'')
```

This handles registering the file in the DistributedCache for us at configuration time, and will work in both local and hdfs modes (in local mode, it simply uses the local path and creates no symlinks).

Next, we need to be very careful about how we instantiate LookupService. Lookup service is not serializable, so we cannot be holding a reference to the nodes when the job get serialized to the nodes. Also we don’t want to be initializing it more than once per task attempt.

```scala
@transient private lazy val lookupService =
    new LookupService(geoIPDataFile.path, LookupService.GEOIP_MEMORY_CACHE)
```

By making lookupService transient, we avoid serialization, but it means that it will be null when received by the nodes. Making it lazy ensures that the nodes will initialize it when needed. We use the auto-generated symlink path created by DistributedCacheFile. This does not require the jobConf as it will not be available, when lookupService is lazily evaluated.

Lastly, we write a function to be used by our mappers to do the lookup. This is included for completeness:

```scala
def getLocationInfo(ip: String): (String, String, String, String, String, Int) = {
  val unknown = (``'', ``'', ``'', ``'', ``'', 0)
  if (ip == ``'' || ip == null)
    return unknown
  val location = lookupService.getLocation(ip)
  if (location == null)
    return unknown
  (location.countryName,
  location.countryCode,
  location.region,
  location.city,
  location.postalCode,
  location.metro_code)
}
```

### 3.11 Why “Pack, Unpack” and not “toList”

The field based API `toList` should not be used if the size of the list in a `groupBy` is very large/not known in advance. `toList` doesn’t decrease the data size significantly, and it stands a good chance of creating OOM errors if the lists get too long. A good alternative to `toList` is to use pack/unpack and reduce. Use pack to convert the tuples into an object, then do a `groupBy` with a reduce function inside it and have your logic to process the grouped items, combine them etc.

**Example 1:**

```scala
val res_pipe= inputpipe.groupBy(`firstname){
  .toList[`lastname]
}
```

**Example 2:**

```scala
case class Person(firstname: String=``'', lastname: String = ``'')
```
val res_pipe = inputpipe.flatMap((`firstname,`lastname)->(`firstname,`person)) {
  in: (String, String) =>
  val (firstname,lastname) = in
  val person = Person(firstname= firstname, lastname= lastname)
  (firstname, person)
}
  .groupBy(`firstname)
  .reduce(`person->`combinedperson) {
    (personAccumulated: Person, person: Person) =>
    val combined_lastname_person = Person(
      firstname = personAccumulated.firstname,
      lastname = personAccumulated.lastname + ',' + person.lastname,
    )
    combined_lastname_person
  }.unpack[Person](`combinedperson -> (`firstname,`lastname))
//comma separated last names

3.11. Why “Pack, Unpack” and not “toList”