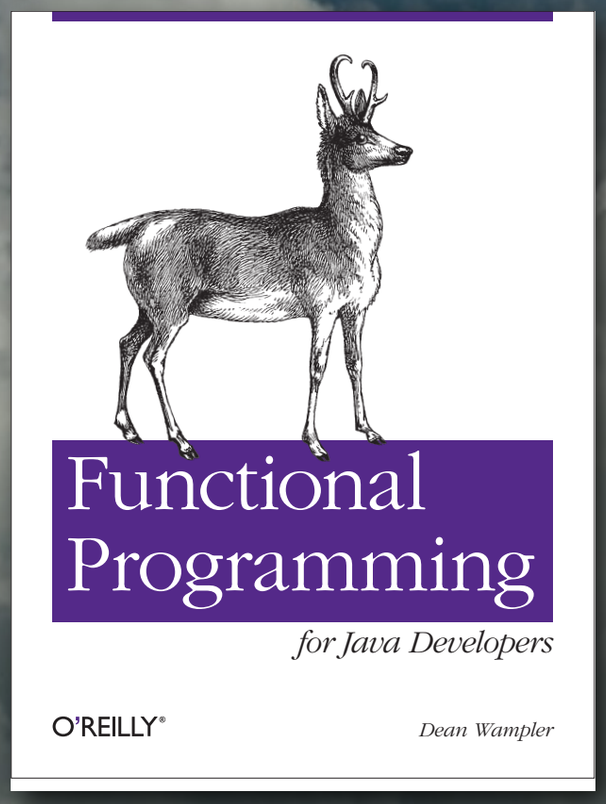
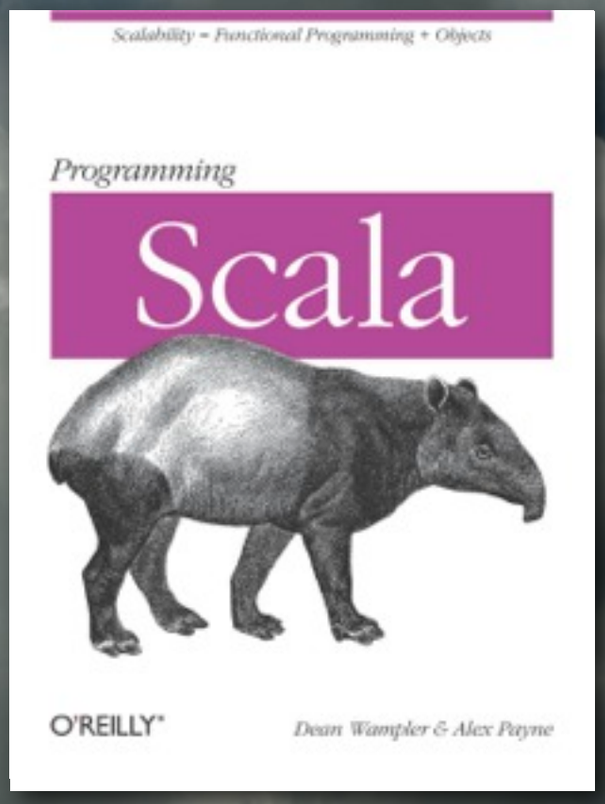
Using Scalding to leverage Cascading to write MapReduce jobs. Note: This talk was presented in tandem with a Cascading introduction presented by Paco Nathan (@pacoid), so it assumes you know some Cascading concepts.

About Me...

dean.wampler@thinkbiganalytics.com
@deanwampler
github.com/thinkbiganalytics
github.com/deanwampler

THINKBIG ANALYTICS





My books and contact information.

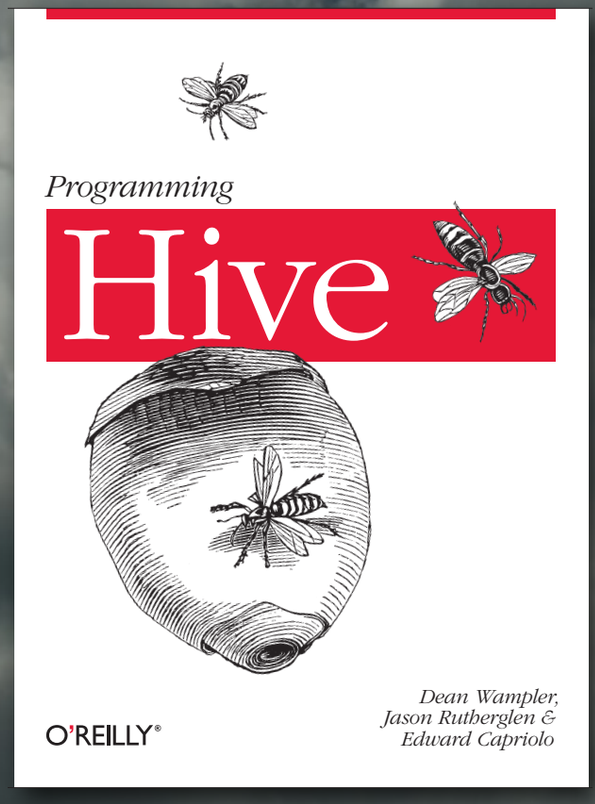
Hive Training in Chicago!

Introduction to Hive
March 7

Hive Master Class
March 8 (also before StrataConf, February 25)

http://bit.ly/ThinkBigEvents

THINKBIG ANALYTICS



Tuesday, February 12, 13

I'm teaching two, one-day Hive classes in Chicago in March. I'm also teaching the Hive Master Class on Monday before StrataConf, 2/25, in Mountain View.

How many of you have
all the time in the world
to get your work *done*?

How many of you have
all the time in the world
to get your work *done*?

Then why are you
doing *this*:

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import java.util.StringTokenizer;

class WCMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

  static final IntWritable one  = new IntWritable(1);
  static final Text word = new Text;   // Value will be set in a non-thread-safe way!

  @Override
  public void map(LongWritable key, Text valueDocContents,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    String[] tokens = valueDocContents.toString.split("\\s+");
    for (String wordString: tokens) {
      if (wordString.length > 0) {
        word.set(wordString.toLowerCase);
        output.collect(word, one);
      }
    }
  }
}
```

```
class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritable> valuesCounts,
                     OutputCollector<Text, IntWritable> output, Reporter reporter) {
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```


This is intentionally too small to read and we're not showing the main routine, which roughly doubles the code size. The "Word Count" algorithm is simple, but the Hadoop MapReduce framework is in your face. It's very hard to see the actual "business logic". Plus, your productivity is terrible. Yet, many Hadoop developers insist on working this way...

The main routine I've omitted contains boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets very complex and tedious very fast!

Notice the green, which I use for all the types, most of which are infrastructure types we don't really care about. There is little yellow, which are function calls that do work. We'll see how these change...

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import java.util.StringTokenizer;

class WCMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

  static final IntWritable one  = new IntWritable(1);
  static final Text word = new Text;   // Value will be set in a non-thread-safe way!

  @Override
  public void map(LongWritable key, Text valueDocContents,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    String[] tokens = valueDocContents.toString.split("\\s+");
    for (String wordString: tokens) {
      if (wordString.length > 0) {
        word.set(wordString.toLowerCase);
        output.collect(word, one);
      }
    }
  }
}
```

The "simple" *Word Count* algorithm

```
class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritable> valuesCounts,
                     OutputCollector<Text, IntWritable> output, Reporter reporter) {
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```


This is intentionally too small to read and we're not showing the main routine, which roughly doubles the code size. The "Word Count" algorithm is simple, but the Hadoop MapReduce framework is in your face. It's very hard to see the actual "business logic". Plus, your productivity is terrible. Yet, many Hadoop developers insist on working this way...

The main routine I've omitted contains boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets very complex and tedious very fast!

Notice the green, which I use for all the types, most of which are infrastructure types we don't really care about. There is little yellow, which are function calls that do work. We'll see how these change...

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import java.util.StringTokenizer;

class WCMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

  static final IntWritable one  = new IntWritable(1);
  static final Text word = new Text;   // Value will be set in a non-thread-safe way!

  @Override
  public void map(LongWritable key, Text valueDocContents,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    String[] tokens = valueDocContents.toString.split("\\s+");
    for (String wordString: tokens) {
      if (wordString.length > 0) {
        word.set(wordString.toLowerCase);
        output.collect(word, one);
      }
    }
  }
}
```

```
class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritable> valuesCounts,
                     OutputCollector<Text, IntWritable> output, Reporter reporter) {
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```


This is intentionally too small to read and we're not showing the main routine, which roughly doubles the code size. The "Word Count" algorithm is simple, but the Hadoop MapReduce framework is in your face. It's very hard to see the actual "business logic". Plus, your productivity is terrible. Yet, many Hadoop developers insist on working this way...

The main routine I've omitted contains boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets very complex and tedious very fast!

Notice the green, which I use for all the types, most of which are infrastructure types we don't really care about. There is little yellow, which are function calls that do work. We'll see how these change...

```
        output.collect(word, one);
      }
    }
  }
}

class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritab
                     OutputCollector<Text, IntWritable> output, Reporte
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```


This is intentionally too small to read and we're not showing the main routine, which roughly doubles the code size. The "Word Count" algorithm is simple, but the Hadoop MapReduce framework is in your face. It's very hard to see the actual "business logic". Plus, your productivity is terrible. Yet, many Hadoop developers insist on working this way...

The main routine I've omitted contains boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets very complex and tedious very fast!

Notice the green, which I use for all the types, most of which are infrastructure types we don't really care about. There is little yellow, which are function calls that do work. We'll see how these change...

```
        output.collect(word, one);
      }
    }
  }
}

class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritab
                     OutputCollector<Text, IntWritable> output, Reporte
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```

Green types.

This is intentionally too small to read and we're not showing the main routine, which roughly doubles the code size. The "Word Count" algorithm is simple, but the Hadoop MapReduce framework is in your face. It's very hard to see the actual "business logic". Plus, your productivity is terrible. Yet, many Hadoop developers insist on working this way...

The main routine I've omitted contains boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets very complex and tedious very fast!

Notice the green, which I use for all the types, most of which are infrastructure types we don't really care about. There is little yellow, which are function calls that do work. We'll see how these change...

```
        output.collect(word, one);
      }
    }
  }
}

class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritab
                     OutputCollector<Text, IntWritable> output, Reporte
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```

Green types.

Yellow operations.

This is intentionally too small to read and we're not showing the main routine, which roughly doubles the code size. The "Word Count" algorithm is simple, but the Hadoop MapReduce framework is in your face. It's very hard to see the actual "business logic". Plus, your productivity is terrible. Yet, many Hadoop developers insist on working this way...

The main routine I've omitted contains boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets very complex and tedious very fast!

Notice the green, which I use for all the types, most of which are infrastructure types we don't really care about. There is little yellow, which are function calls that do work. We'll see how these change...

Your tools should:

We would like a “domain-specific language” (DSL) for data manipulation and analysis.

Your tools should:

Minimize *boilerplate*
by exposing the
right abstractions.

We would like a “domain-specific language” (DSL) for data manipulation and analysis.

Your tools should:

Maximize *expressiveness* and *extensibility*.

Expressiveness - ability to tell the system what you need done.
Extensibility - ability to add functionality to the system when it doesn't already support some operation you need.

Use Cascading (Java)

(Solution #1)

Cascading is a Java library that provides higher-level abstractions for building data processing pipelines with concepts familiar from SQL such as a joins, group-bys, etc. It works on top of Hadoop's MapReduce and hides most of the boilerplate from you.
See http://cascading.org.

```
import org.cascading.*;
...
public class WordCount {
  public static void main(String[] args) {
    Properties properties = new Properties();
    FlowConnector.setApplicationJarClass( properties, WordCount.class );

    Scheme sourceScheme = new TextLine( new Fields( "line" ) );
    Scheme sinkScheme = new TextLine( new Fields( "word", "count" ) );
    String inputPath  = args[0];
    String outputPath = args[1];
    Tap source = new Hfs( sourceScheme, inputPath );
    Tap sink   = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

    Pipe assembly = new Pipe( "wordcount" );

    String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
    Function function = new RegexGenerator( new Fields( "word" ), regex );
    assembly = new Each( assembly, new Fields( "line" ), function );
    assembly = new GroupBy( assembly, new Fields( "word" ) );
    Aggregator count = new Count( new Fields( "count" ) );
    assembly = new Every( assembly, count );

    FlowConnector flowConnector = new FlowConnector( properties );
    Flow flow = flowConnector.connect( "word-count", source, sink, assembly);
    flow.complete();
  }
}
```


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

Note that there is still a lot of green for types, but now they are almost all domain-related concepts and less infrastructure types. The infrastructure is less "in your face." There is still little yellow, because Cascading has to wrap behavior in Java classes, like GroupBy, Each, Count, etc. Also, the API emphasizes composing behaviors together in an intuitive and powerful way.

```
import org.cascading.*;
...
public class WordCount {
  public static void main(String[] args) {
    Properties properties = new Properties();
    FlowConnector.setApplicationJarClass( properties, WordCount.class );

    Scheme sourceScheme = new TextLine( new Fields( "line" ) );
    Scheme sinkScheme = new TextLine( new Fields( "word", "count" ) );
    String inputPath  = args[0];
    String outputPath = args[1];
    Tap source = new Hfs( sourceScheme, inputPath );
    Tap sink   = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

    Pipe assembly = new Pipe( "wordcount" );

    String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
    Function function = new RegexGenerator( new Fields( "word" ), regex );
    assembly = new Each( assembly, new Fields( "line" ), function );
    assembly = new GroupBy( assembly, new Fields( "word" ) );
    Aggregator count = new Count( new Fields( "count" ) );
    assembly = new Every( assembly, count );

    FlowConnector flowConnector = new FlowConnector( properties );
    Flow flow = flowConnector.connect( "word-count", source, sink, assembly);
    flow.complete();
  }
}
```


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

Note that there is still a lot of green for types, but now they are almost all domain-related concepts and less infrastructure types. The infrastructure is less "in your face." There is still little yellow, because Cascading has to wrap behavior in Java classes, like GroupBy, Each, Count, etc. Also, the API emphasizes composing behaviors together in an intuitive and powerful way.

```
import org.cascading.*;
...
public class WordCount {
  public static void main(String[] args) {
    Properties properties = new Properties();
    FlowConnector.setApplicationJarClass( properties, W

    Scheme sourceScheme = new TextLine( new Fields( "lin
    Scheme sinkScheme = new TextLine( new Fields( "word
    String inputPath  = args[0];
    String outputPath = args[1];
    Tap source = new Hfs( sourceScheme, inputPath );
    Tap sink   = new Hfs( sinkScheme, outputPath, SinkM

    Pip                w Pipe( "wordcount" );

    String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\
```

"Flow" setup.

Tuesday, February 12, 13

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

Note that there is still a lot of green for types, but now they are almost all domain-related concepts and less infrastructure types. The infrastructure is less "in your face." There is still little yellow, because Cascading has to wrap behavior in Java classes, like GroupBy, Each, Count, etc. Also, the API emphasizes composing behaviors together in an intuitive and powerful way.

```
    Scheme sourceScheme = new TextLine( new Fields( "li
    Scheme sinkScheme = new TextLine( new Fields( "word
    String inputPath  = args[0];
    String outputPath = args[1];
    Tap source = new Hfs( sourceScheme, inputPath );
    Tap sink   = new Hfs( sinkScheme, outputPath, SinkM

    Pipe assembly = new Pipe( "wordcount" );

    String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\
    Function function = new RegexGenerator( new Fields(
    assembly = new Each( assembly, new Fields( "line" )
    assembly = new GroupBy( assembly, new Fields( "word
    Aggregator count = new Count( new Fields( "count" )
    assembly = new Every( assembly, count );

    FlowConnector flowConnector = new FlowConnector( pr
    Flo                                  "word-count", so
    flo
  }
}
```

"Flow" setup.

Input and output taps.

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

Note that there is still a lot of green for types, but now they are almost all domain-related concepts and less infrastructure types. The infrastructure is less "in your face." There is still little yellow, because Cascading has to wrap behavior in Java classes, like GroupBy, Each, Count, etc. Also, the API emphasizes composing behaviors together in an intuitive and powerful way.

```
    Pipe assembly = new Pipe( "wordcount" );

    String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\
    Function function = new RegexGenerator( new Fields(
    assembly = new Each( assembly, new Fields( "line" )
    assembly = new GroupBy( assembly, new Fields( "word
    Aggregator count = new Count( new Fields( "count" )
    assembly = new Every( assembly, count );

    FlowConnector flowConnector = new FlowConnector( pr
    Flow flow = flowConnector.connect( "word-count", so
    flow.complete();
  }
}
```

"Flow" setup.

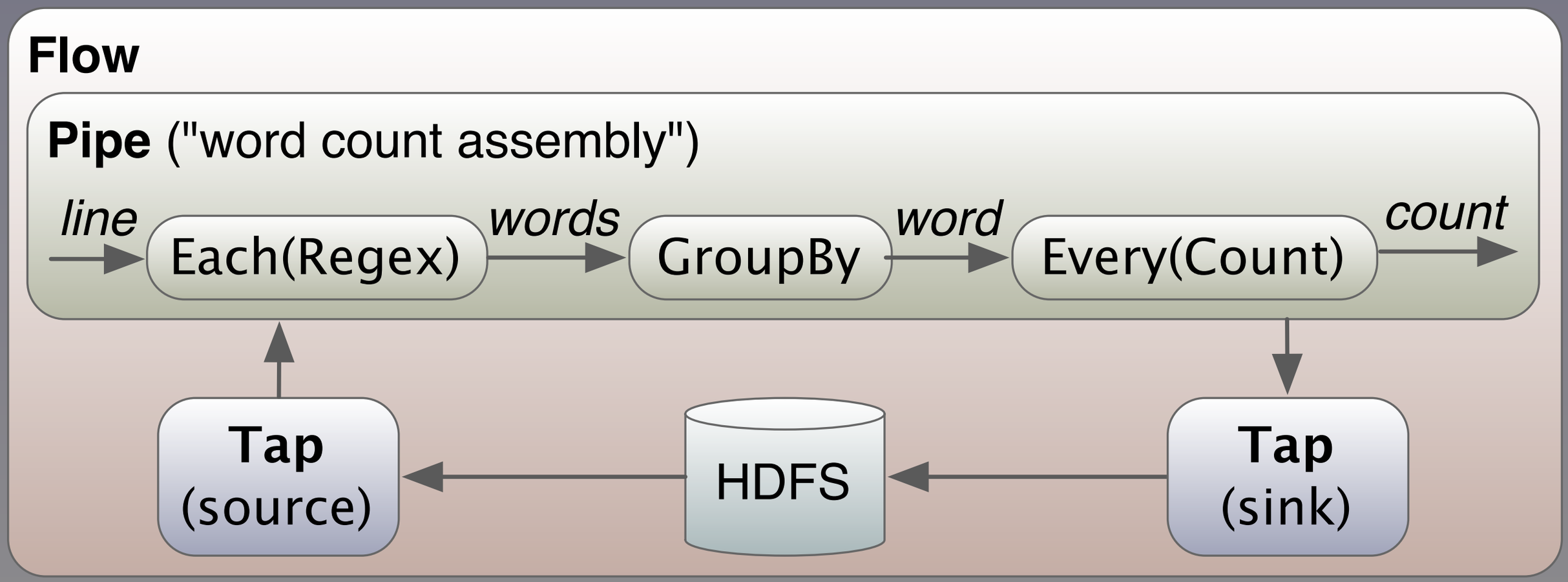
Input and output taps.

Assemble pipes.

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

Note that there is still a lot of green for types, but now they are almost all domain-related concepts and less infrastructure types. The infrastructure is less "in your face." There is still little yellow, because Cascading has to wrap behavior in Java classes, like GroupBy, Each, Count, etc. Also, the API emphasizes composing behaviors together in an intuitive and powerful way.

Word Count



Tuesday, February 12, 13

Schematically, here is what Word Count looks like in Cascading. See http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html for details.

Use Scalding (Scala)

(Solution #2)

Scalding is a Scala “DSL” (domain-specific language) that wraps Cascading providing an even more intuitive and more boilerplate-free API for writing MapReduce jobs. https://github.com/twitter/scalding

Scala is a new JVM language that modernizes Java’s object-oriented (OO) features and adds support for functional programming, as we’ll describe shortly.

```
import com.twitter.scalding._

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


This Scala code is almost pure domain logic with very little boilerplate. There are a few minor differences in the implementation. You don't explicitly specify the "Hfs" (Hadoop Distributed File System) taps. That's handled by Scalding implicitly when you run in "non-local" model. Also, I'm using a simpler tokenization approach here, where I split on anything that isn't a "word character" [0-9a-zA-Z_].
There is much less green, in part because Scala infers types in many cases, but also because fewer infrastructure types are required. There is a lot more yellow for the functions that do real work!
What if MapReduce, and hence Cascading and Scalding, went obsolete tomorrow? This code is so short, I wouldn't care about throwing it away! I invested little time writing it, testing it, etc.

```
import com.twitter.scalding._

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

That's it!!

This Scala code is almost pure domain logic with very little boilerplate. There are a few minor differences in the implementation. You don't explicitly specify the "Hfs" (Hadoop Distributed File System) taps. That's handled by Scalding implicitly when you run in "non-local" model. Also, I'm using a simpler tokenization approach here, where I split on anything that isn't a "word character" [0-9a-zA-Z_].
There is much less green, in part because Scala infers types in many cases, but also because fewer infrastructure types are required. There is a lot more yellow for the functions that do real work!
What if MapReduce, and hence Cascading and Scalding, went obsolete tomorrow? This code is so short, I wouldn't care about throwing it away! I invested little time writing it, testing it, etc.

```
CREATE EXTERNAL TABLE docs (line STRING)
LOCATION '/path/to/corpus';

CREATE TABLE word_counts
AS SELECT word, count(1) AS count FROM
(SELECT explode(split(line, '\s')) AS word
 FROM docs) w GROUP BY word
ORDER BY word;
```


Here is the Hive equivalent (more or less, we aren't splitting into words using a sophisticated regular expression this time). Note that Word Count is a bit unusual to implement with a SQL dialect (we'll see more typical cases later), and we're using some Hive SQL extensions that we won't take time to explain (like ARRAYs), but you get the point that SQL is wonderfully concise… when you can express your problem in it!!

```
CREATE EXTERNAL TABLE docs (line STRING)
LOCATION '/path/to/corpus';

CREATE TABLE word_counts
AS SELECT word, count(1) AS count FROM
(SELECT explode(split(line, '\s')) AS word
 FROM docs) w GROUP BY word
ORDER BY word;
```

Hive

Tuesday, February 12, 13

Here is the Hive equivalent (more or less, we aren't splitting into words using a sophisticated regular expression this time). Note that Word Count is a bit unusual to implement with a SQL dialect (we'll see more typical cases later), and we're using some Hive SQL extensions that we won't take time to explain (like ARRAYs), but you get the point that SQL is wonderfully concise… when you can express your problem in it!!

```
inpt = load '/path/to/corpus'
  using TextLoader as (line: chararray);
words = foreach inpt generate
  flatten(TOKENIZE(line)) as word;
grpd = group words by word;
cntd = foreach grpd generate group, COUNT(words);
dump cntd;
```


Pig is more expressive for non-query problems like this.

```
inpt = load '/path/to/corpus'
  using TextLoader as (line: chararray);
words = foreach inpt generate
  flatten(TOKENIZE(line)) as word;
grpd = group words by word;
cntd = foreach grpd generate group, COUNT(words);
dump cntd;
```

Pig

Tuesday, February 12, 13

Pig is more expressive for non-query problems like this.


What Is
Scala?

Scala is a *modern, concise, object-oriented* and *functional* language for the *JVM*.

http://scala-lang.org

Modern.

Reflects the *20 years* of *language evolution* since *Java's invention.*

Concise.

Type *inference*, syntax for *common* OOP and FP *idioms*, *APIs*.

Object-Oriented
Programming.
Fixes *flaws* in Java's
object model.
Composition/reuse
through *traits*.

Functional
Programming.

Most *natural fit*
for *data*!
Robust, scalable.

JVM.

Designed for the *JVM*.

Interoperates with all *Java* software.

What Is Scalding?

Scalding is a *Scala* Big Data library based on *Cascading*, developed by *Twitter*.

https://github.com/twitter/scalding

I say “based on” Cascading, because it isn’t just a thin Scala veneer over the Java API. It adds some different concepts, hiding some Cascading API concepts (e.g., IO specifics), but not all (e.g., groupby function <=> GroupBy class).

Scalding adds a *Matrix API* useful for *graph* and *machine-learning algorithms.*

I'll show examples.

Note: *Cascalog* is a *Clojure* alternative for *Cascading*. Also adds *Logic-Programming* based on *Datalog*.

https://github.com/nathanmarz/cascalog

Tuesday, February 12, 13

For completeness, note that Cascalog is a Clojure alternative. It is also notable for its addition of Logic-Programming features, adapted from Datalog. Cascalog was developed by Nathan Marz, now at Twitter, who also invented Storm.

More Scalding examples

Returning to our first example.

```
import com.twitter.scalding._

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


Tuesday, February 12, 13

Let's walk through the code to get the "gist" of what's happening.
First, import the Scalding API.

Returning to our first example.

```
import com.twitter.scalding._

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

Import the library.

Tuesday, February 12, 13

Let’s walk through the code to get the “gist” of what’s happening.
First, import the Scalding API.

```
import com.twitter.scalding._

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

Create a "Job" class.

Wrap the logic in a class (OOP!) that extends the Job abstraction.

```
import com.twitter.scalding._

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

Open the user-specified input as text.

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

```
import com.twitter.scalding._

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

Split each `'line` into `'words`, a collection and flatten the collections.

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

```
import com.twitter.scalding._

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

Group by each `'word` and
determine the size of each group.

Now we group the stream of words by the word value, then count the size of each group. Give the output field the name “count”, rather than the default name “size” (if no argument is passed to size()).

```
import com.twitter.scalding._

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

Write tab-separated
words and counts.

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

```
import com.twitter.scalding._

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

Profit!!

For all you South Park fans out there...

Joins

An example of doing a simple inner join between stock and dividend data.

```
import com.twitter.scalding._

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

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

Load stocks and dividends data in separate pipes for a given stock, e.g., IBM.

```
import com.twitter.scalding._

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

Capture the schema as values.

Tuesday, February 12, 13

“symd” is “stocks year-month-day” and similarly for dividends. Note that Scalding requires unique names for these fields from the different tables.

```
import com.twitter.scalding._

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

Load the stock and dividend records into separate pipes, project desired stock fields.

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

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

  stocksPipe
    .joinWithTiny('symd -> 'dymd, dividendsPipe)
    .project('symd, 'close, 'dividend)
    .write(Tsv(args("output")))
}
```

Join stocks with smaller dividends on dates, project desired fields.

```
# Invoking the script (bash)
scald.rb StocksDivsJoins.scala \
  --stocks    data/stocks/IBM.txt \
  --dividends data/dividends/IBM.txt \
  --output output/IBM-join.txt
```

```
# Invoking the script (bash)
scald.rb StocksDivsJoins.scala \
  --stocks    data/stocks/IBM.txt \
  --dividends data/dividends/IBM.txt \
  --output output/IBM-join.txt


# Output (not sorted!)
2010-02-08      121.88  0.55
2009-11-06      123.49  0.55
2009-08-06      117.38  0.55
2009-05-06      104.62  0.55
2009-02-06      96.14   0.5
2008-11-06      85.15   0.5
2008-08-06      129.16  0.5
2008-05-07      124.14  0.5
...
```

Compute ngrams in a corpus...

NGrams

Compute ngrams in a corpus...

```
import com.twitter.scalding._

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

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

Find, count, sort, all "I love _" in Shakespeare's plays...

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

```
import com.twitter.scalding._

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

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

From user-specified args, the ngram prefix ("I love"), the # of ngrams desired, a matching regular expression.

```
import com.twitter.scalding._

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

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

A function "value" we'll use to sort ngrams by frequency, descending.

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

Read, find desired ngram phrases, project desired fields, then group by ngram.

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

Group all (ngram,count) tuples together and sort descending by count. Write...

```
# Invoking the script (bash)
scald.rb ContextNGrams.scala \
  --input  data/shakespeare/plays.txt \
  --output output/context-ngrams.txt \
  --ngram-prefix "I love" \
  --count 10
```

```
# Invoking the script (bash)
scald.rb ContextNGrams.scala \
  --input  data/shakespeare/plays.txt \
  --output output/context-ngrams.txt \
  --ngram-prefix "I love" \
  --count 10

# Output (reformatted)
(I love thee,44),
(I love you,24),
(I love him,15),
(I love the,9),
(I love her,8),
...,
(I love myself,3),
...,
(I love Valentine,1),
...,
(I love France,1), ...
```

An Example of the Matrix API

Tuesday, February 12, 13

From Scalding's own Matrix examples, "MatrixTutorial4", compute the cosine of every pair of vectors. This is a similarity measurement, useful, e.g., for for recommending movies and other stuff to one user based on the preferences of similar users.

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix._
  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long,Long,Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

  // Inner product is equivalent to the cosine:
  //   AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
}
```

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix._
  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long,Long,Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

  // Inner product is equivalent to the cosine:
  //   AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
}
```

Import matrix library.

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix.
  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long,Long,Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

  // Inner product is equivalent to the cosine:
  //   AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
}
```

Load into a matrix.

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix._
  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long,Long,Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

  // Inner product is equivalent to the cosine:
  //   AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
}
```

Normalize: sqrt(x^2+y^2)

L2 norm is the usual Euclidean distance: sqrt(a*a + b*b + …)

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix._
  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long,Long,Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

  // Inner product is equivalent to the cosine:
  //   AA^T/(||A||*||A||)
  (normMatrix * normMatrix.transpose)
  .write(Tsv(args("output")))
}
```

Compute cosine!

L2 norm is the usual Euclidean distance: sqrt(a*a + b*b + …)
(I won't show an example invocation here.)

Tuesday, February 12, 13

Some concluding thoughts on Scalding (and Cascading) vs. Hive vs. Pig, since I've used all three a lot.

Tuesday, February 12, 13

SQL is about as concise as it gets when asking questions of data and doing “standard” analytics.

Pig is great for data flows, such as sequencing transformations typical of ETL (extract, transform, and load). As you might expect for a purpose-built language, it is very concise for these purposes, but writing extensions is a non-trivial step (same for Hive).

Tuesday, February 12, 13

Scalding can do everything you can do with these other tools (more or less), but it gives you the full flexible power of a Turing-complete language, so it's easier to adapt to non-trivial, custom scenarios.

For more on *Scalding*:

github.com/twitter/scalding
github.com/ThinkBigAnalytics/scalding-workshop
github.com/Cascading/Impatient/tree/master/part8