



CHUG, February 12, 2013  
dean.wampler@thinkbiganalytics.com  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)



# Scalding for Hadoop

Thursday, February 14, 13

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.

(All photos are © Dean Wampler, 2005–2013, All Rights Reserved. Otherwise, the presentation is free to use.)

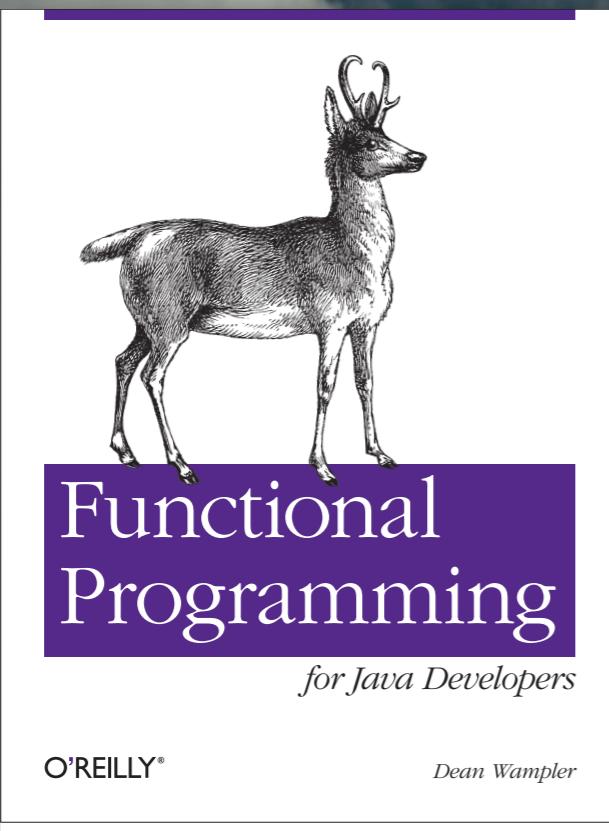
# About Me...

[dean.wampler@thinkbiganalytics.com](mailto:dean.wampler@thinkbiganalytics.com)

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

[github.com/thinkbiganalytics](https://github.com/thinkbiganalytics)

[github.com/deanwampler](https://github.com/deanwampler)



Thursday, February 14, 13

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>



Thursday, February 14, 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));
    }
}

```

## The “simple” Word Count algorithm

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));
    }
}

```

Thursday, February 14, 13

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.

7

Thursday, February 14, 13

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:

Thursday, February 14, 13

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

Your tools should:

*Minimize boilerplate  
by exposing the  
right abstractions.*

Your tools should:

*Maximize expressiveness  
and extensibility.*

Thursday, February 14, 13

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)

10

Thursday, February 14, 13

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();
    }
}

```

||

Thursday, February 14, 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.

```

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" ) );
        String inputPath = args[0];
        String outputPath = args[1];
        Tap source = new Hfs( sourceScheme, inputPath );
        Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.Overwrite );
        Pipe pipe = new Pipe( "wordcount" );
        pipe.setSource( source );
        pipe.setSink( sink );
        pipe.setFlowControl( FlowControl.noControl() );
        pipe.open();
        ...
    }
}

```

Pip “Flow” setup.

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

Thursday, February 14, 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( "line" ) );
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, SinkMode.Overwrite );

Pipe assembly = new Pipe( "wordcount" );

String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function function = new RegexGenerator( new Fields( "line" ) );
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( properties );
flowConnector.setFlow(flow);
flowConnector.connect("word-count", source);
flowConnector.connect("word-count", sink);

}

```

“Flow” setup.

Input and output taps.

Thursday, February 14, 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.

```

Pipe assembly = new Pipe( "wordcount" );

String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?=<=\\pL)(?!\\pL)\\pL";
Function function = new RegexGenerator( new Fields( "line" ) );
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( properties );
Flow flow = flowConnector.connect( "word-count", source );
flow.complete();
}
}

```

“Flow” setup.

Input and  
output taps.

Assemble  
pipes.

Thursday, February 14, 13

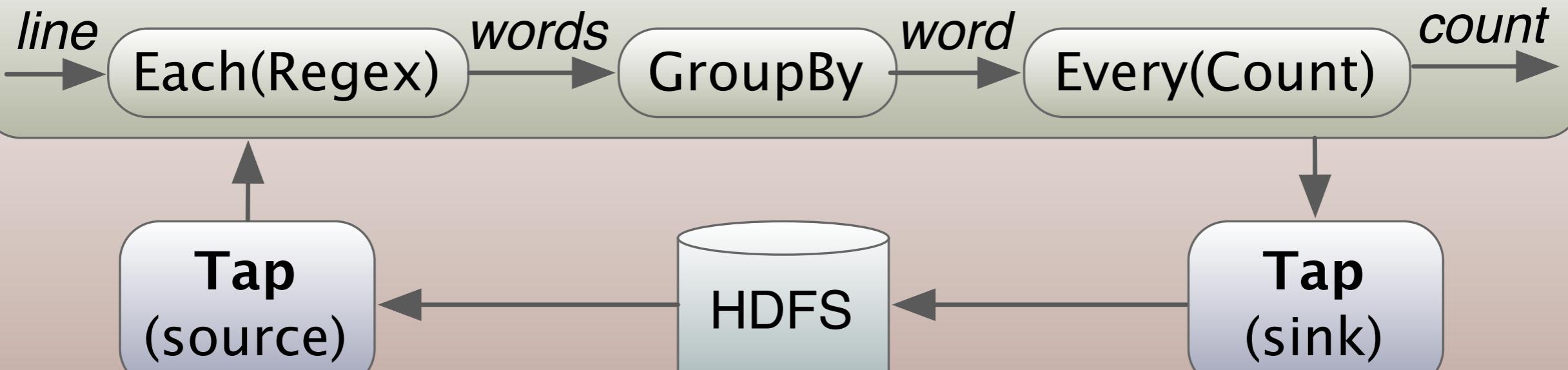
Here is the Cascading Java code. It's cleaner than the MapReduce API, because the code is more focused on the algorithm with less boilerplate, although it looks like it's not that much shorter. HOWEVER, this is all the code, where as previously I omitted the setup (main) code. See <http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html> 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.

# Word Count

## Flow

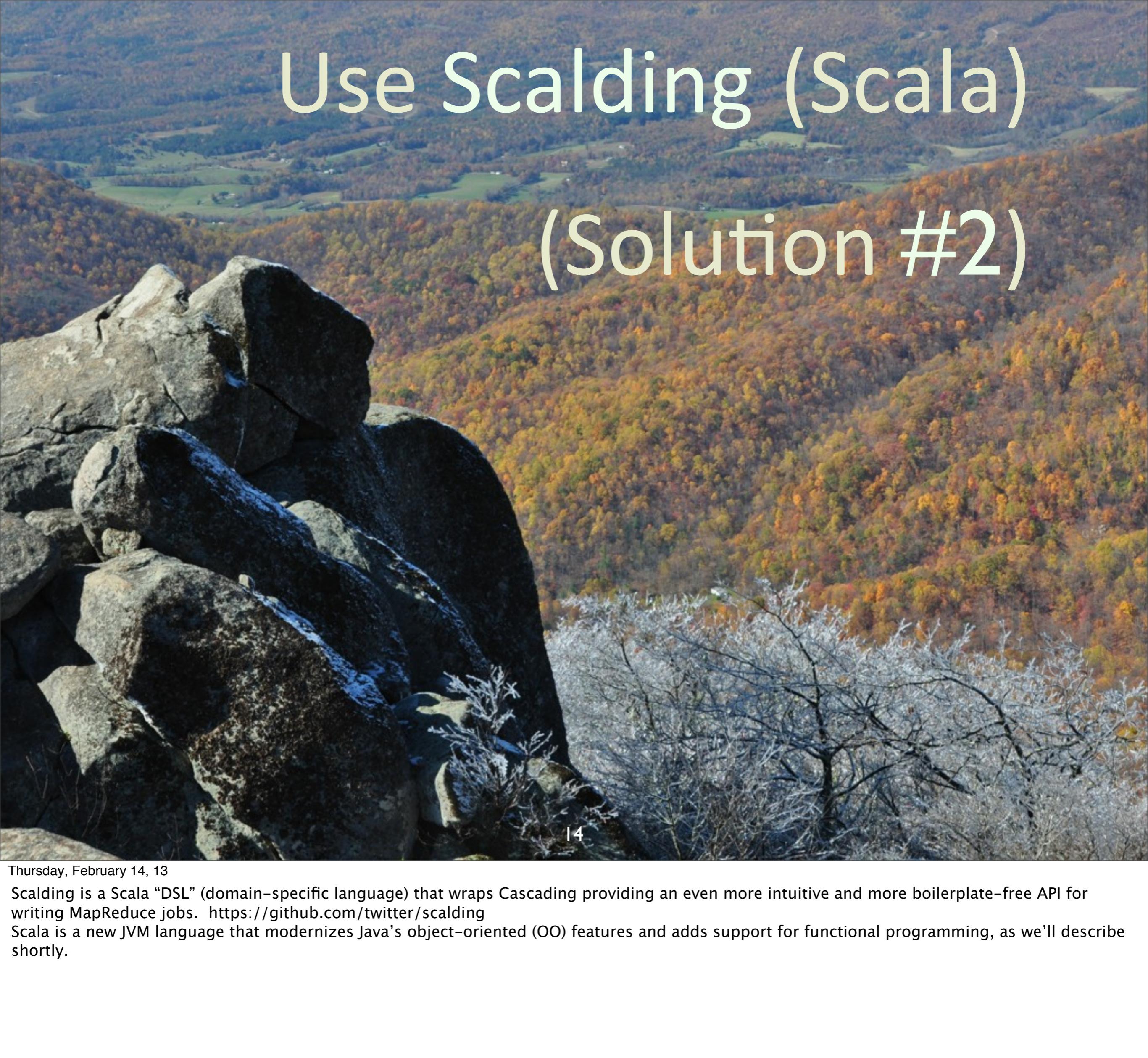
**Pipe ("word count assembly")**



Copyright © 2011-2012, Think Big Analytics, All Rights Reserved

Thursday, February 14, 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)

14

Thursday, February 14, 13

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")))
}
```

That's it!!

15

Thursday, February 14, 13

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;
```

Hive

16

Thursday, February 14, 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

17

Thursday, February 14, 13

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



# What Is Scala?

|8

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

<http://scala-lang.org>

Thursday, February 14, 13

I'll explain what I mean for these terms in the next few slides.

*Modern.*

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

Thursday, February 14, 13

Java was a good compromise of ideas for the state-of-the-art in the early 90's, but that was ~20 years ago! We've learned a lot about language design and effective software development techniques since then.

# *Concise.*

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

Thursday, February 14, 13

Briefly, type inference reduces “noise”, allowing the logic to shine through. Scala has lots of idioms to simplify class definition and object creation, as well as many FP idioms, such as anonymous functions. The collection API emphasizes the important data transformations we need...

# *Object-Oriented Programming.*

Fixes *flaws* in Java's  
*object model.*

Composition/reuse  
through *traits*.

Thursday, February 14, 13

Scala fixes flaws in Java's object model, such as the essential need for a mix-in composition mechanism, which Scala provides via traits. For more on this, see Daniel Spiewak's keynote for NEScala 2013: "[The Bakery from the Black Lagoon](#)".

"

# *Functional Programming.*

*Most natural fit  
for data!*

*Robust, scalable.*

Thursday, February 14, 13

It may not look like it, but this is actually the most important slide in this talk; FP is the most appropriate tool for data, since everything we do with data is mathematics. FP also gives us the idioms we need for robust, scalable concurrency.

*JVM.*

Designed for the *JVM*.

*Interoperates with all  
Java software.*

Thursday, February 14, 13

But Scala is a JVM language. It interoperates with any Java, Clojure, JRuby, etc. libraries.



# What Is Scalding?

25

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

<https://github.com/twitter/scalding>

Thursday, February 14, 13

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.*

Thursday, February 14, 13

I'll show examples.

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

<https://github.com/nathanmarz/cascalog>

Thursday, February 14, 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.



# Moar Scalding!

29

Thursday, February 14, 13

More examples, starting with a deeper look at our first example.

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")))
}
```

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.

```
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.

```
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.

```
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.

```
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.

```
import com.twitter.scalding._

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

Write tab-separated  
words and counts.

```
import com.twitter.scalding._

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

Profit!!

# Joins



Thursday, February 14, 13

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 = ('synd, 'close, 'volume)
  val divsSchema = ('dymd, 'dividend)
  val stocksPipe = new Tsv(args("stocks"), stockSchema)
    .read
    .project('synd, 'close)
  val divsPipe = new Tsv(args("dividends"), divsSchema)
    .read

  stocksPipe
    .joinWithTiny('synd -> 'dymd, dividendsPipe)
    .project('synd, '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 = ('synd, 'close, 'volume)
  val divsSchema = ('dymd, 'dividend)
  val stocksPipe = new Tsv(args("stocks"), stockSchema)
    .read
    .project('synd, 'close)
  val divsPipe = new Tsv(args("dividends"), divsSchema)
    .read

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

```

Capture the schema as values.

```

import com.twitter.scalding._

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

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

```

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

```
import com.twitter.scalding._

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

stocksPipe
  .joinWithTiny('synd -> 'dymd, dividendsPipe)
  .project('synd, 'close, 'dividend)
  .write(Csv(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
...
...
```



43

Thursday, February 14, 13

Compute ngrams in a corpus...

# NGrams



43

Thursday, February 14, 13

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")))
}
```

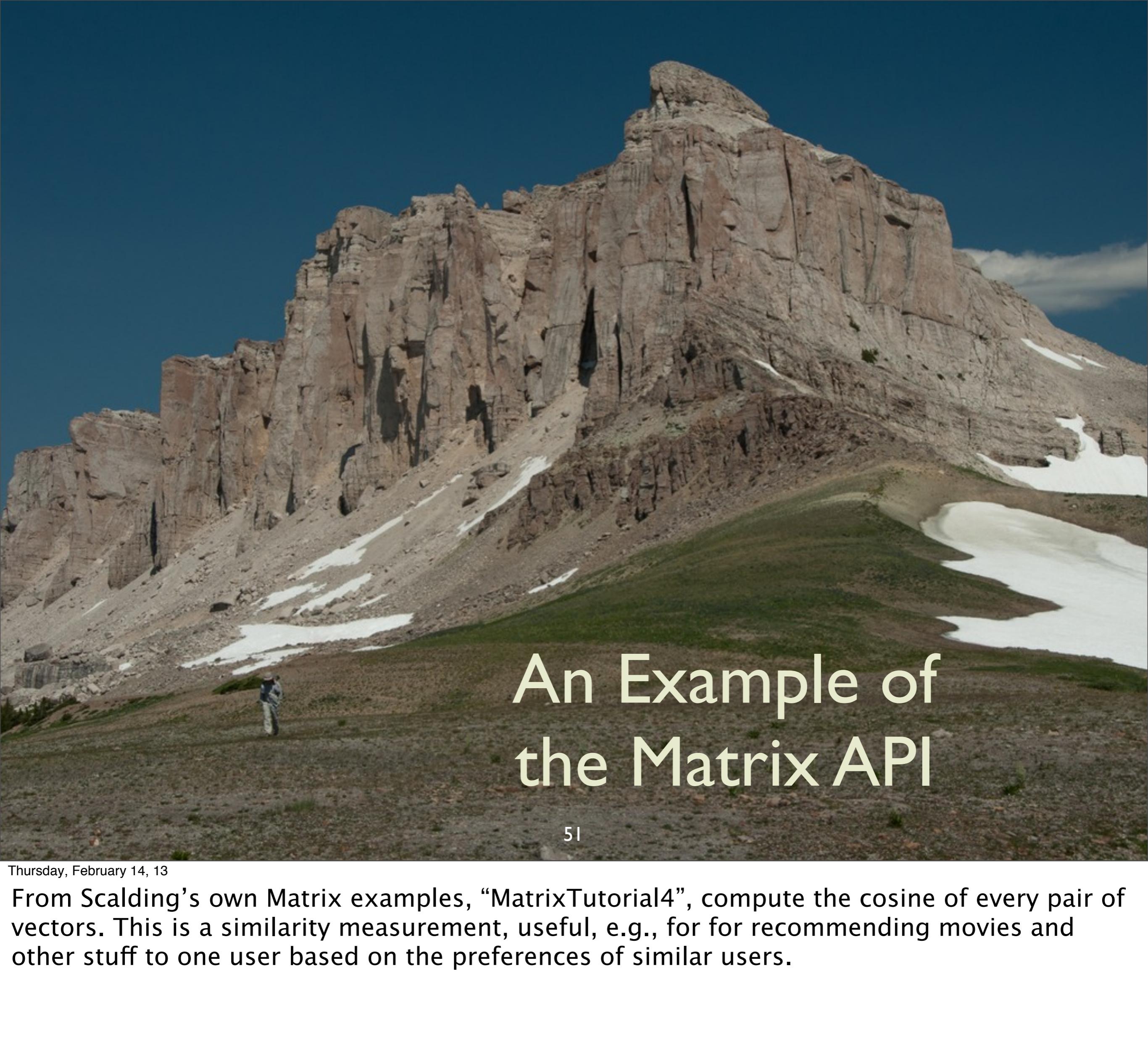
Group all (ngram,count) tuples together  
and sort descending by count. Write...

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

A photograph of a massive, rugged mountain peak under a clear blue sky. The mountain's face is composed of large, vertical rock formations with deep shadows. At the base, there are patches of green grass and small areas of white snow. A lone hiker stands on a grassy slope in the lower-left foreground, providing a sense of scale to the enormous mountain.

# An Example of the Matrix API

51

Thursday, February 14, 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 recommending movies and other stuff to one user based on the preferences of similar users.

```

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix._

  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long, Long, Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

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

```

```

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

// Load a directed graph adjacency matrix where:
// a[i,j] = 1 if there is an edge from a[i] to b[j]
// and computes the cosine of the angle between
// every two pairs of vectors.
class MatrixCosine(args: Args) extends Job(args) {
  import Matrix._

  val schema = ('user1, 'user2, 'relation)
  val adjacencyMatrix = Tsv(args("input"), schema)
    .read
    .toMatrix[Long, Long, Double](schema)
  val normMatrix = adjacencyMatrix.rowL2Normalize

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

```

Import matrix library.

```

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.

```

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:  $\sqrt{x^2 + y^2}$

```

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!



# *Scalding vs. Hive vs. Pig*

57

Thursday, February 14, 13

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



# Use Hive for queries.

58

Thursday, February 14, 13

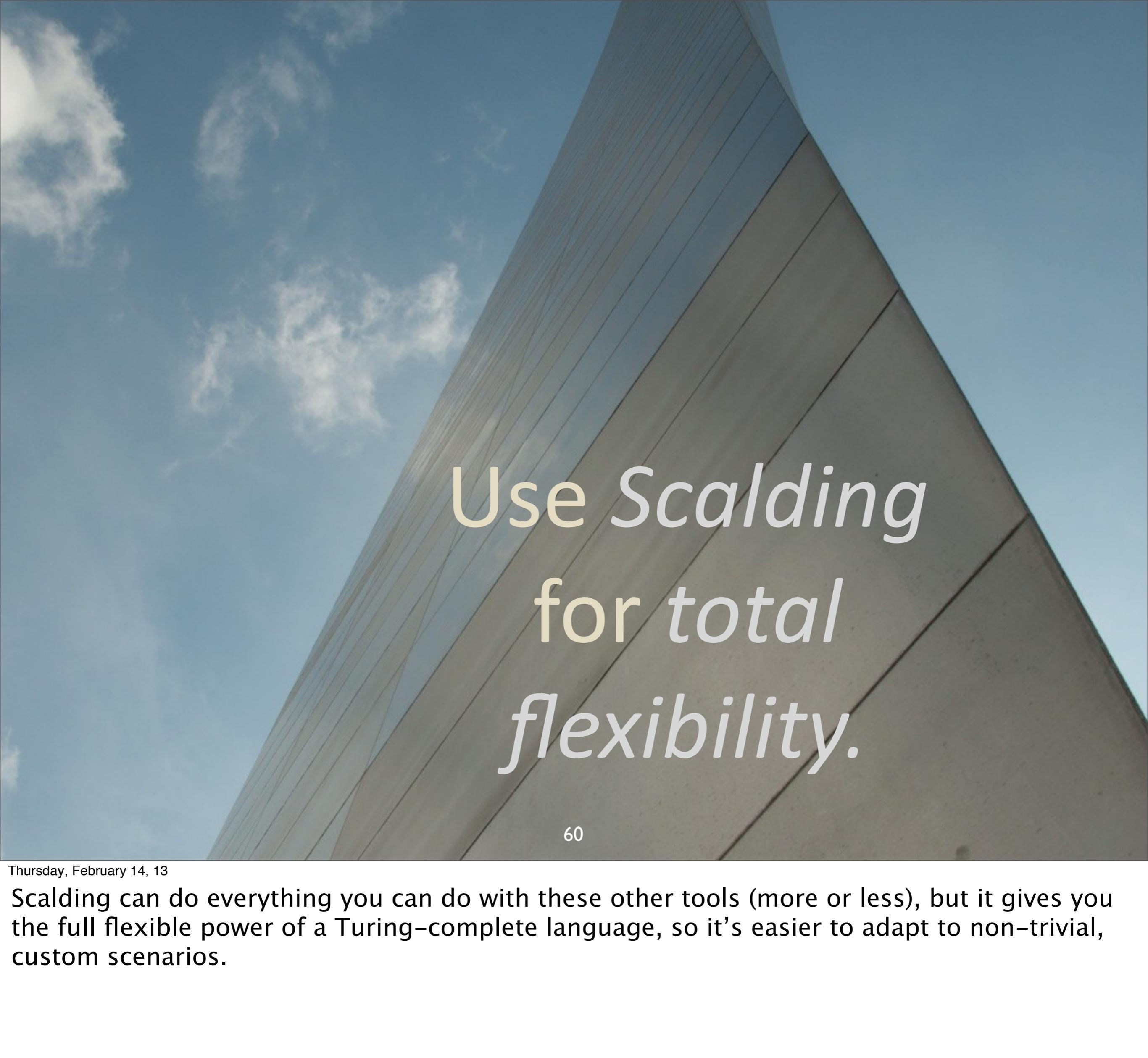
SQL is about as concise as it gets when asking questions of data and doing “standard” analytics. A great example of a “purpose-built” language. It’s great at what it was designed for. Not so great when you need to do something it wasn’t designed to do.



*Use Pig for  
data flows.*

Thursday, February 14, 13

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).



Use *Scalding*  
for *total*  
*flexibility.*

60

Thursday, February 14, 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*:

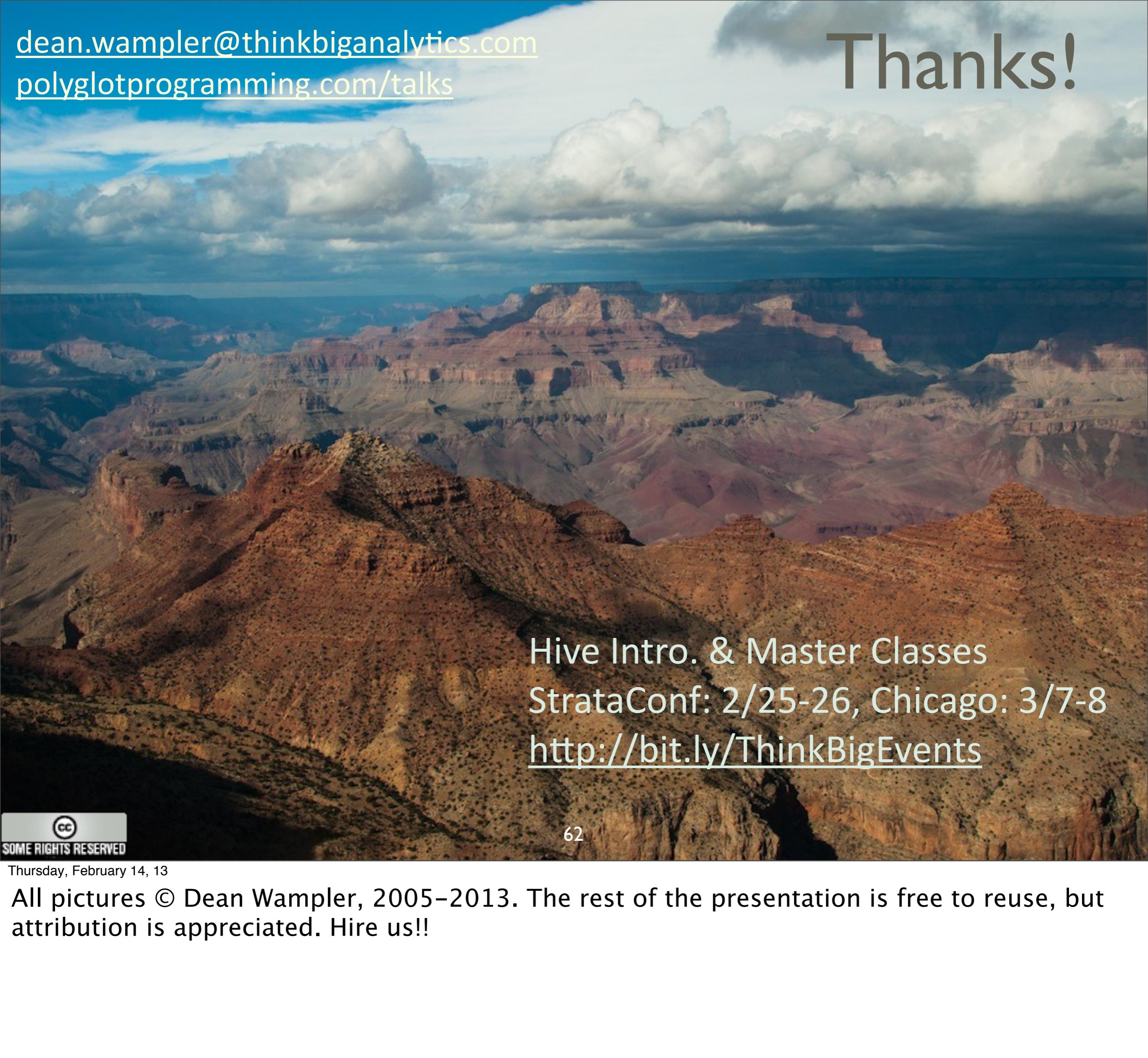
[github.com/twitter/scalding](https://github.com/twitter/scalding)

[github.com/ThinkBigAnalytics/scalding-workshop](https://github.com/ThinkBigAnalytics/scalding-workshop)

[github.com/Cascading/Impatient/tree/master/part8](https://github.com/Cascading/Impatient/tree/master/part8)

[dean.wampler@thinkbiganalytics.com](mailto:dean.wampler@thinkbiganalytics.com)  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)

Thanks!

A wide-angle photograph of the Grand Canyon. The foreground shows rugged, reddish-brown rock formations with distinct horizontal sedimentary layers. In the background, the canyon's depth is visible, with more layers of rock and a bright blue sky filled with large, white, billowing clouds.

Hive Intro. & Master Classes  
StrataConf: 2/25-26, Chicago: 3/7-8  
<http://bit.ly/ThinkBigEvents>



62

Thursday, February 14, 13

All pictures © Dean Wampler, 2005–2013. The rest of the presentation is free to reuse, but attribution is appreciated. Hire us!!