Typesafe

Scala eXchange 9-Dec-2014

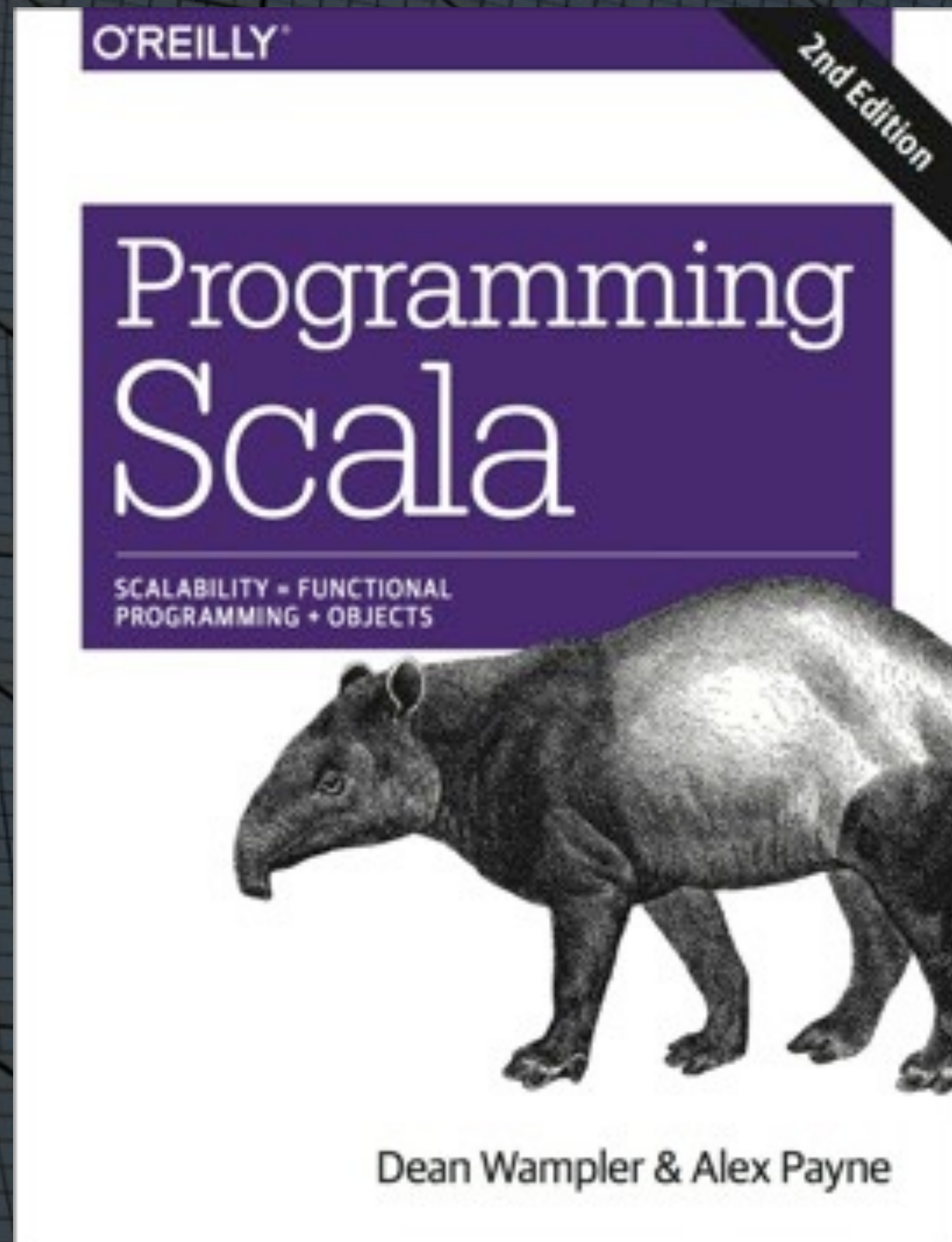
dean.wampler@typesafe.com

Why Scala Is Taking Over the Big Data World

Tuesday, December 9, 14

<plug>

<shameless>

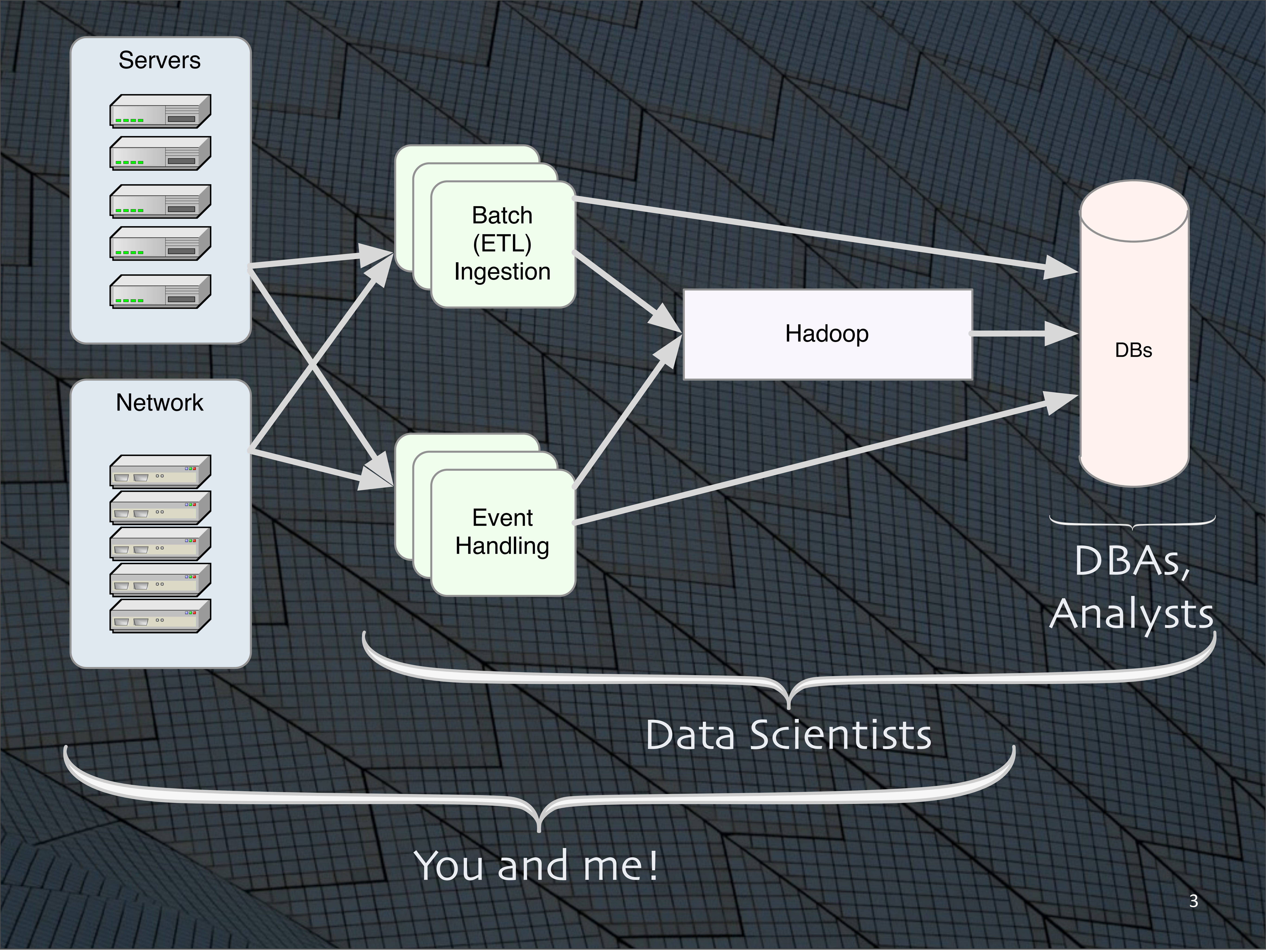


</shameless>

</plug>

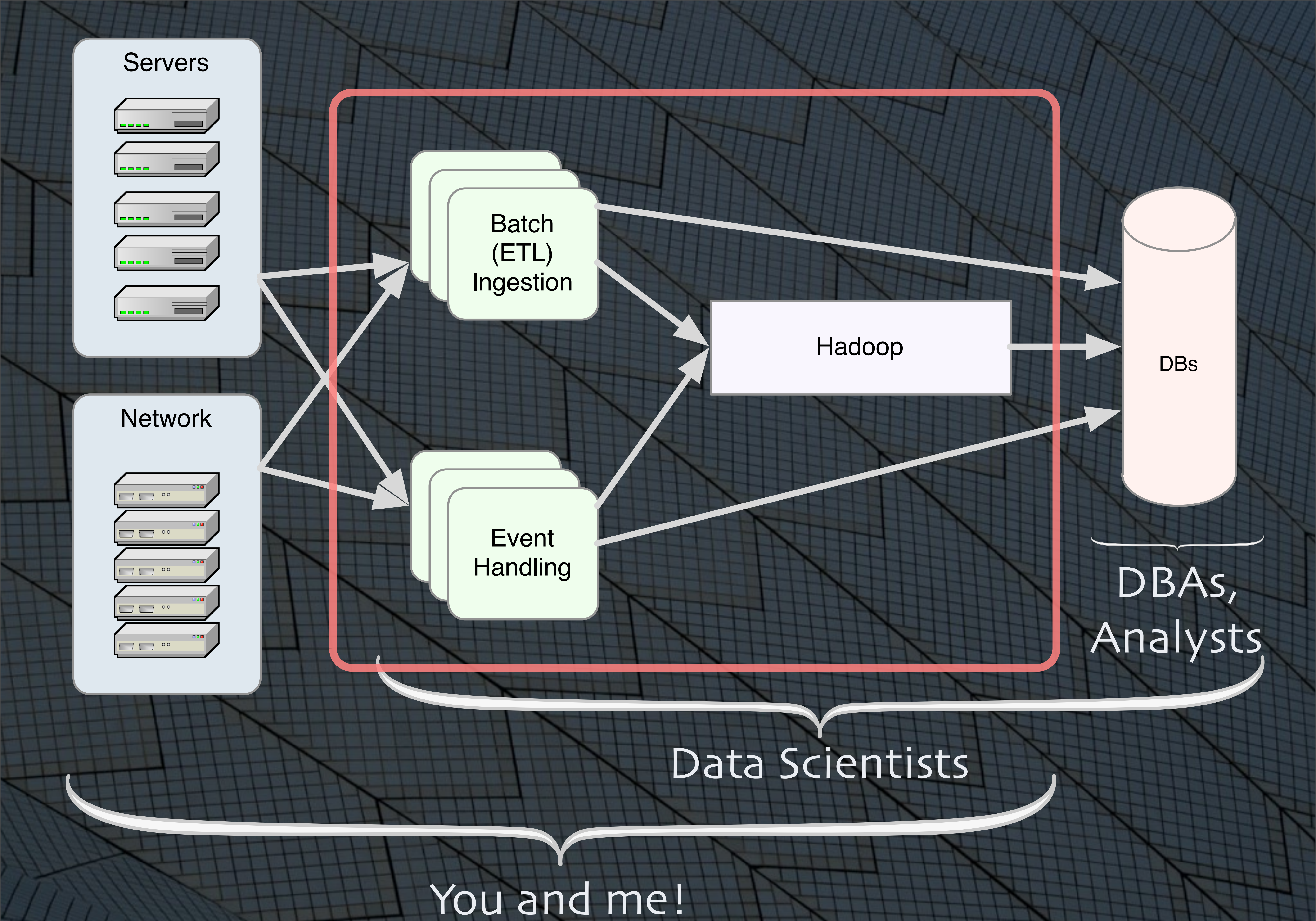
Tuesday, December 9, 14

http://programming-scala.org



Tuesday, December 9, 14

DBAs, traditional data analysts: SQL, SAS. “DBs” could also be distributed file systems.
Data Scientists - Statistics experts. Some programming, especially Python, R, Julia, maybe Matlab, etc.
Developers like us, who figure out the infrastructure (but don’t usually manage it), and write the programs that do batch-oriented ETL (extract, transform, and load), and more real-time event handling.
Often this data gets pumped into Hadoop or other compute engines and data stores, including various SQL and NoSQL DBs, and file systems.



Tuesday, December 9, 14

What we'll focus on.

Tuesday, December 9, 14

Let's put all this into perspective...

http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg

Tuesday, December 9, 14

Let's put all this into perspective, circa 2008...

http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg

Hadoop

Tuesday, December 9, 14

Let's drill down to Hadoop, which first gained widespread awareness in 2008-2009, when Yahoo! announced they were running a 10K core cluster with it, Hadoop became a top-level Apache project, etc.

Hadoop Blog

RSS Video

4000

Scaling Hadoop to 4000 nodes at Yahoo!

By aanand – Tue, Sep 30, 2008 10:04 AM EDT

Recommend 1 Tweet 0

Tue, Sep 30, 2008

We recently ran Hadoop on what we believe is the single largest Hadoop installation, ever:

- 4000 nodes
- 2 quad core Xeons @ 2.5ghz per node
- 4x1TB SATA disks per node
- 8G RAM per node
- 1 gigabit ethernet on each node
- 40 nodes per rack
- 4 gigabit ethernet uplinks from each rack to the core (unfortunately a misconfiguration, we usually do 8 uplinks)
- Red Hat Enterprise Linux AS release 4 (Nahant Update 5)
- Sun Java JDK 1.6.0_05-b13
- So that's well over 30,000 cores with nearly 16PB of raw disk!

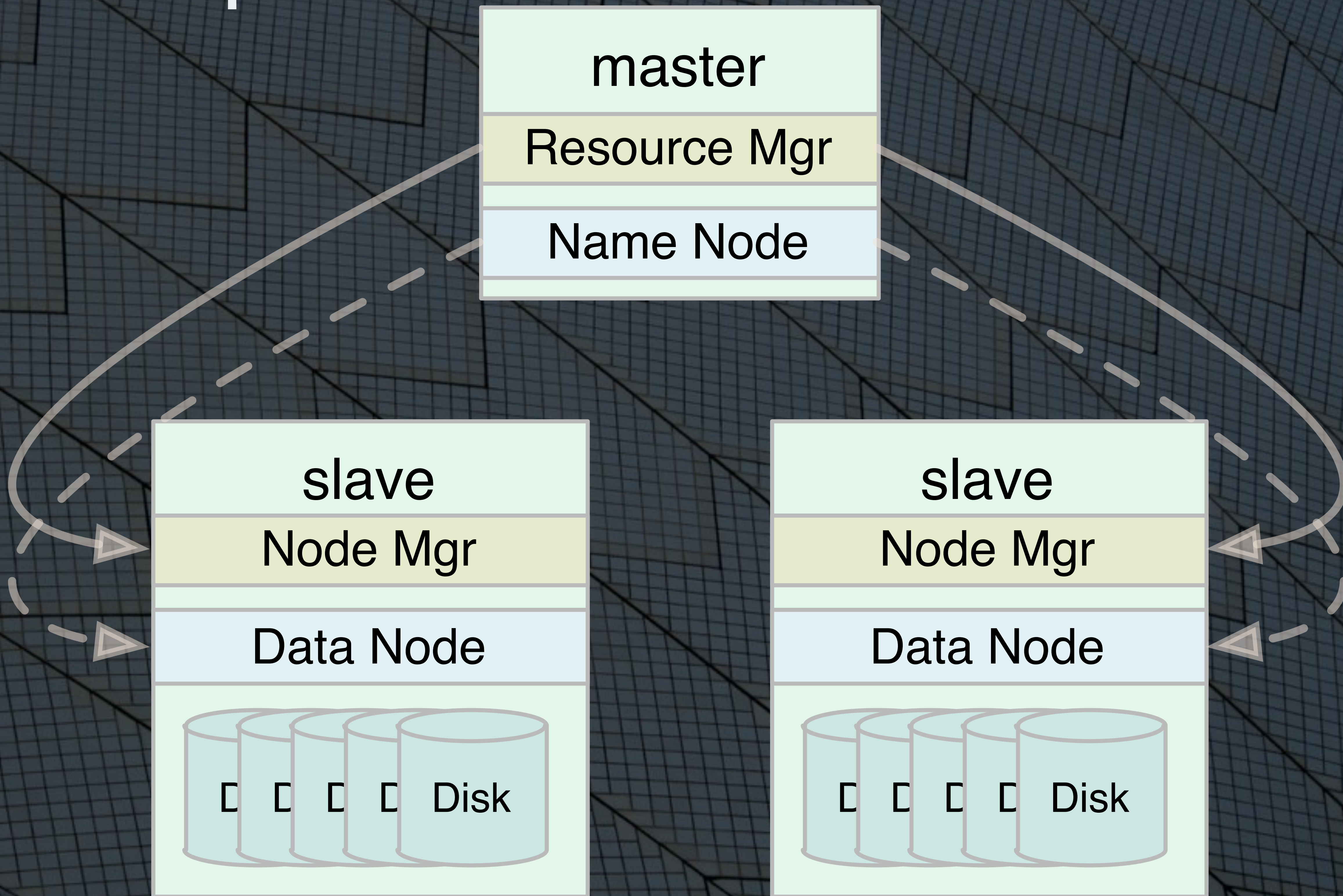
Quant, by today's standards

16PB

Tuesday, December 9, 14

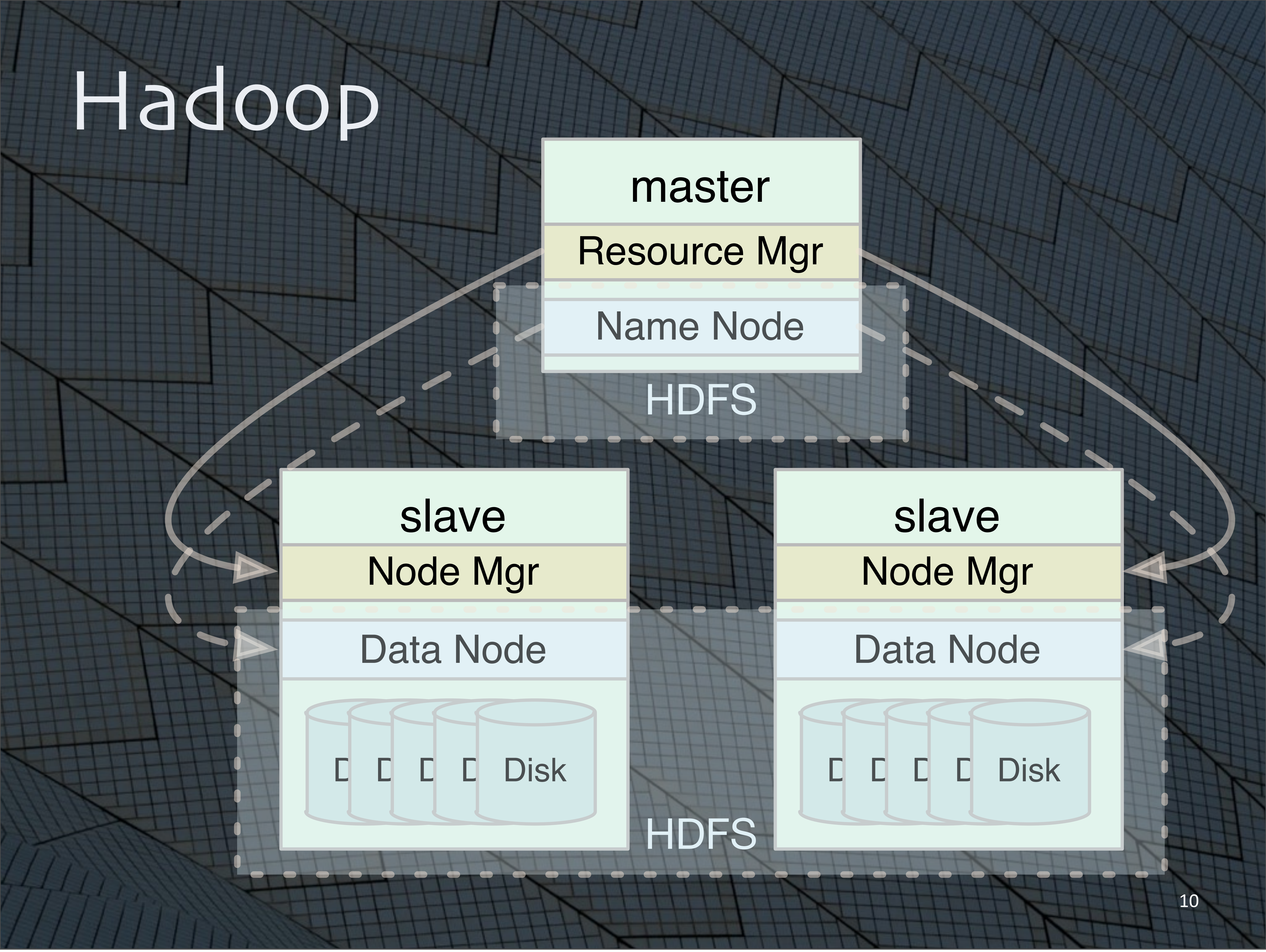
The largest clusters today are roughly 10x in size.

Hadoop



Tuesday, December 9, 14

The schematic view of a Hadoop v2 cluster, with YARN (Yet Another Resource Negotiator) handling resource allocation and job scheduling. (V2 is actually circa 2013, but this detail is unimportant for this discussion). The master services are federated for failover, normally (not shown) and there would usually be more than two slave nodes. Node Managers manage the tasks
The Name Node is the master for the Hadoop Distributed File System. Blocks are managed on each slave by Data Node services.
The Resource Manager decomposes each job in to tasks, which are distributed to slave nodes and managed by the Node Managers. There are other services I’m omitting for simplicity.

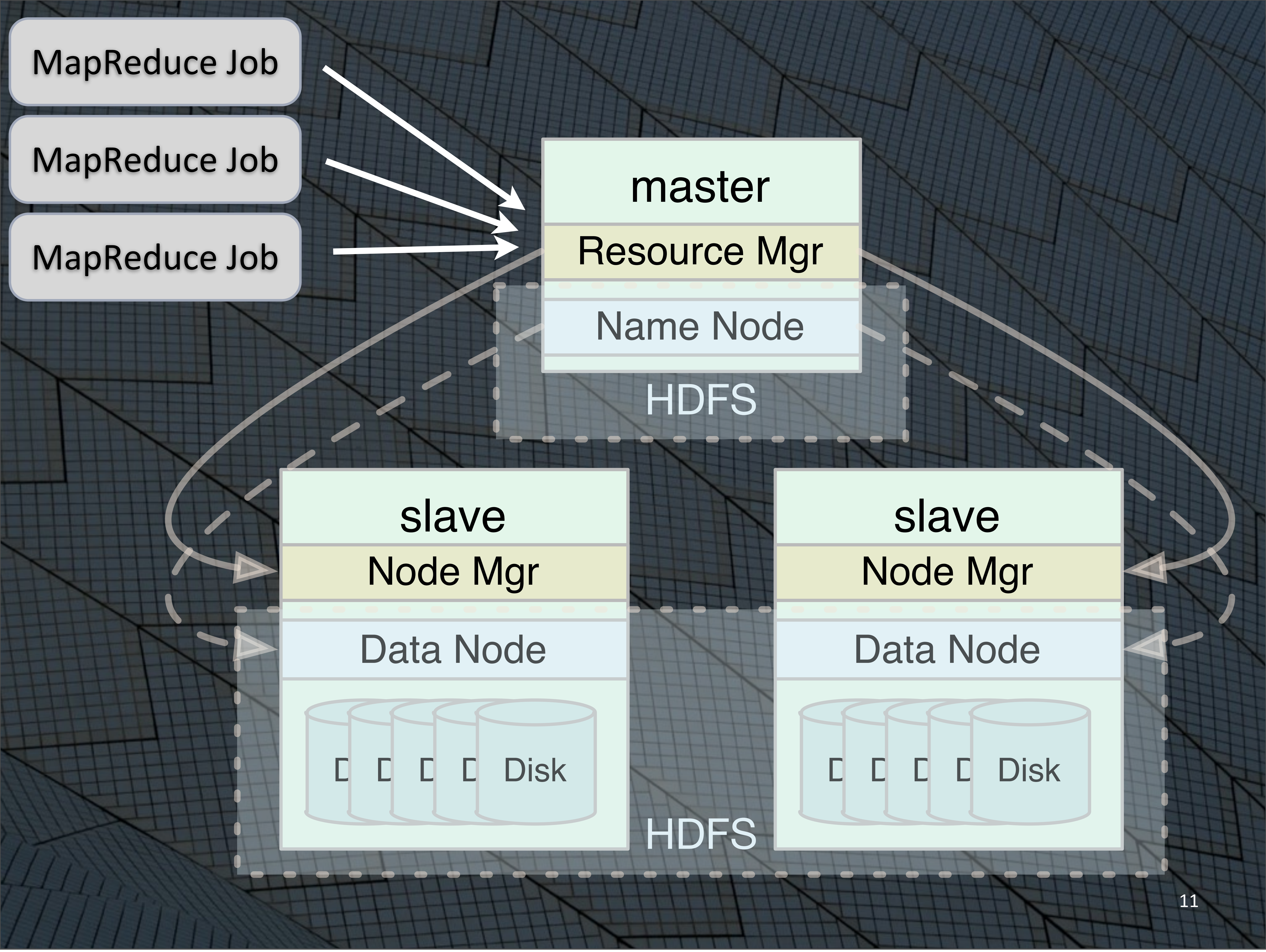


Tuesday, December 9, 14

The schematic view of a Hadoop v2 cluster, with YARN (Yet Another Resource Negotiator) handling resource allocation and job scheduling. (V2 is actually circa 2013, but this detail is unimportant for this discussion). The master services are federated for failover, normally (not shown) and there would usually be more than two slave nodes. Node Managers manage the tasks

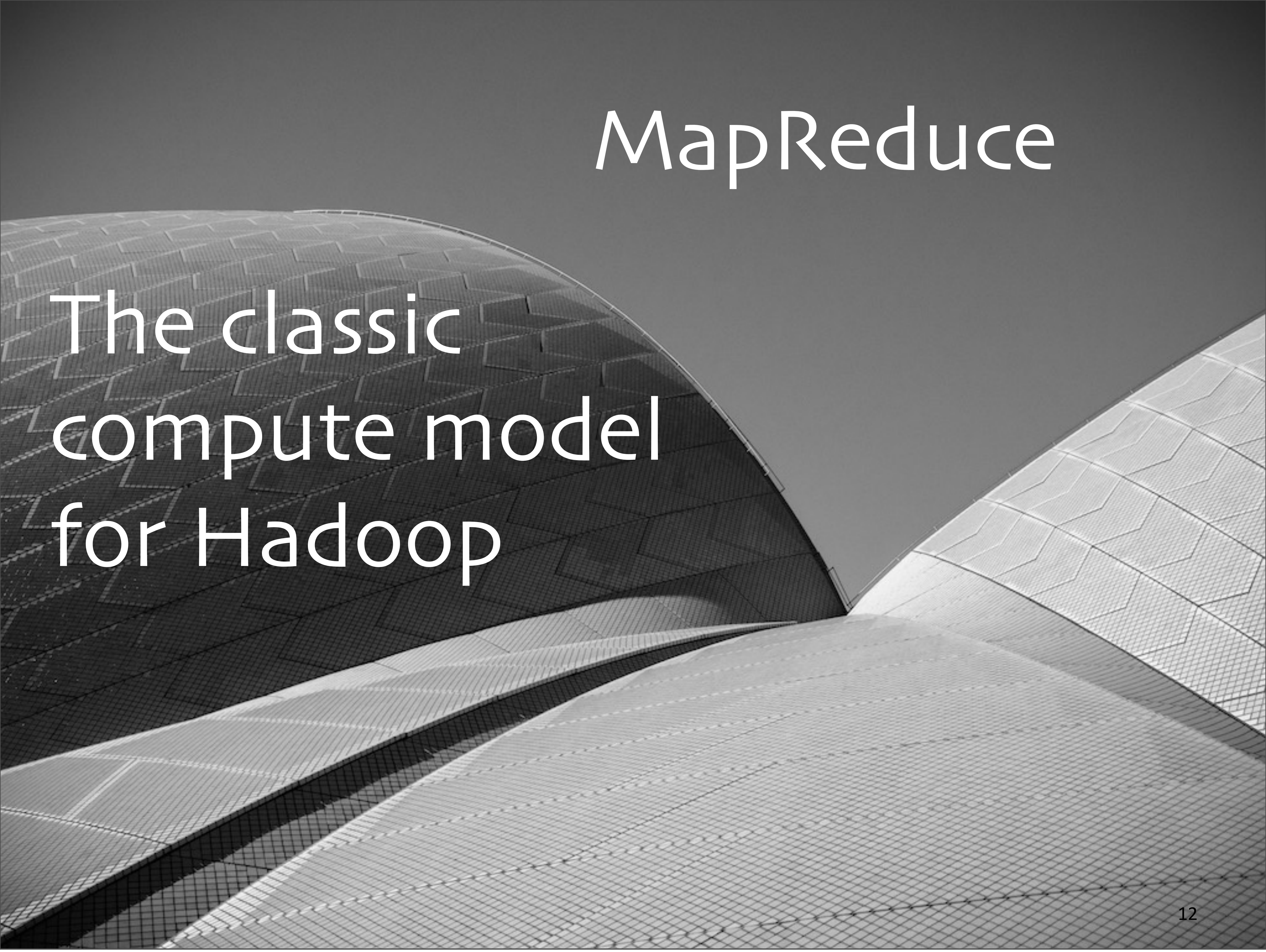
The Name Node is the master for the Hadoop Distributed File System. Blocks are managed on each slave by Data Node services.

The Resource Manager decomposes each job in to tasks, which are distributed to slave nodes and managed by the Node Managers. There are other services I'm omitting for simplicity.



Tuesday, December 9, 14

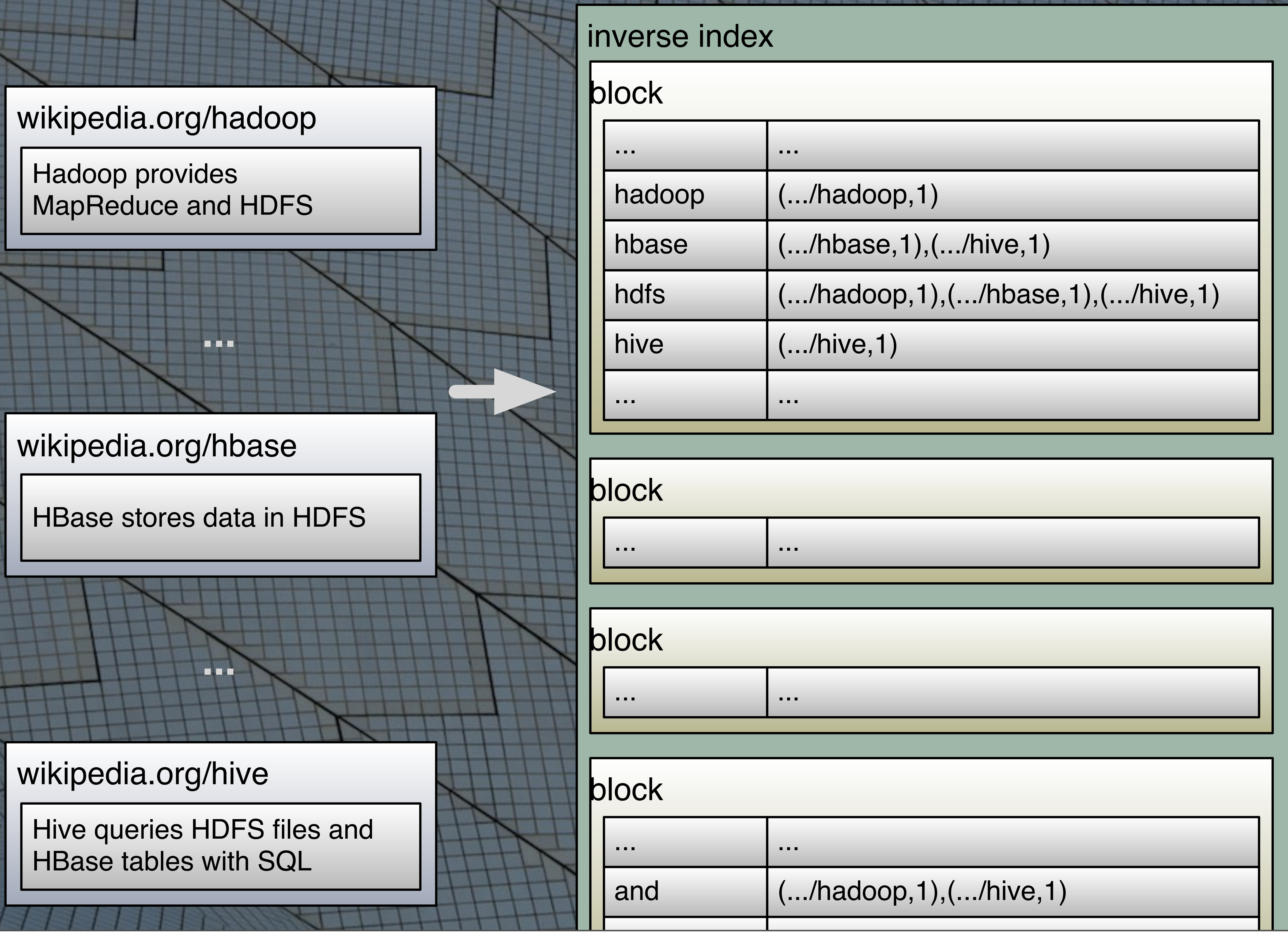
You submit MapReduce jobs to the Resource Manager. Those jobs could be written in the Java API, or higher-level APIs like Cascading, Scalding, Pig, and Hive.



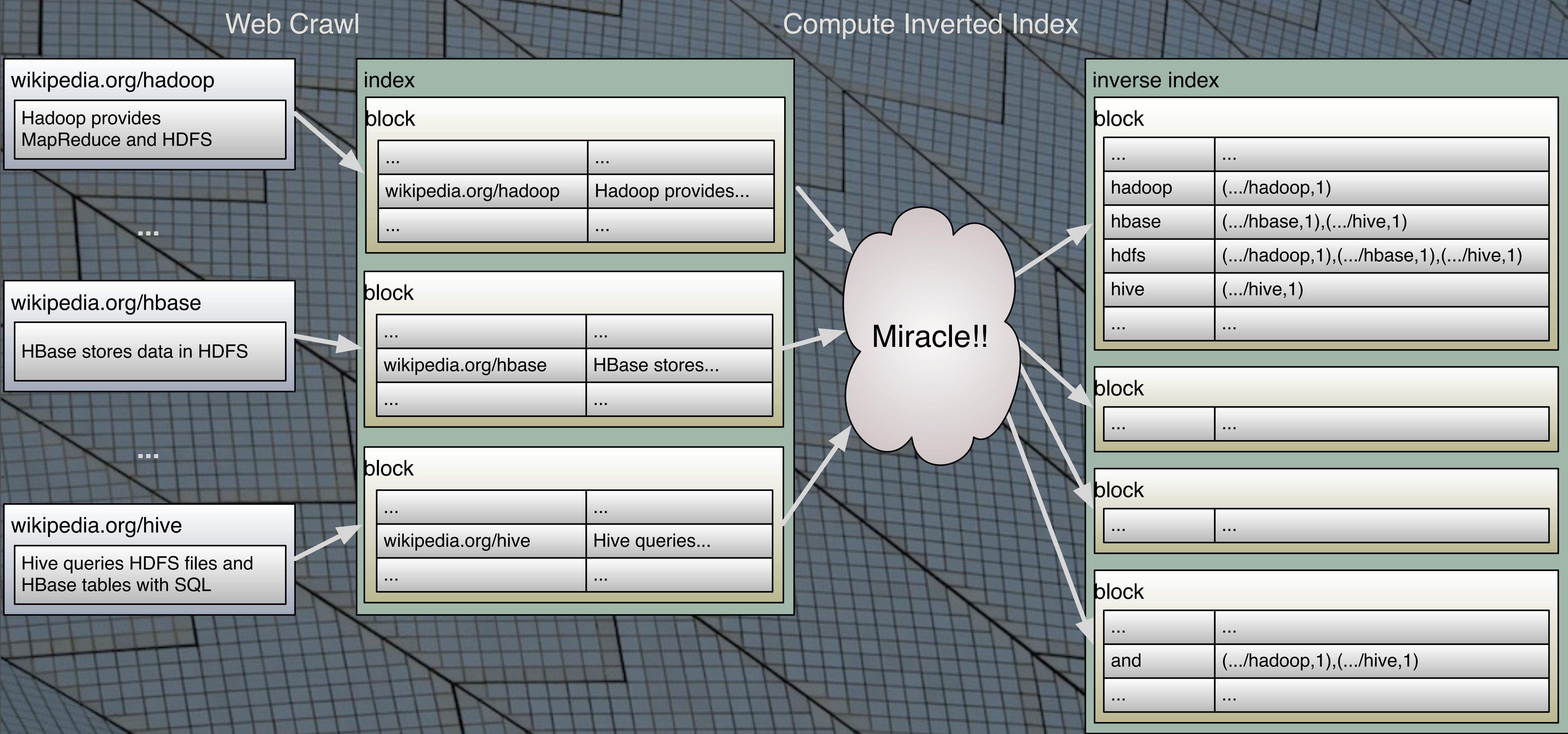
Tuesday, December 9, 14

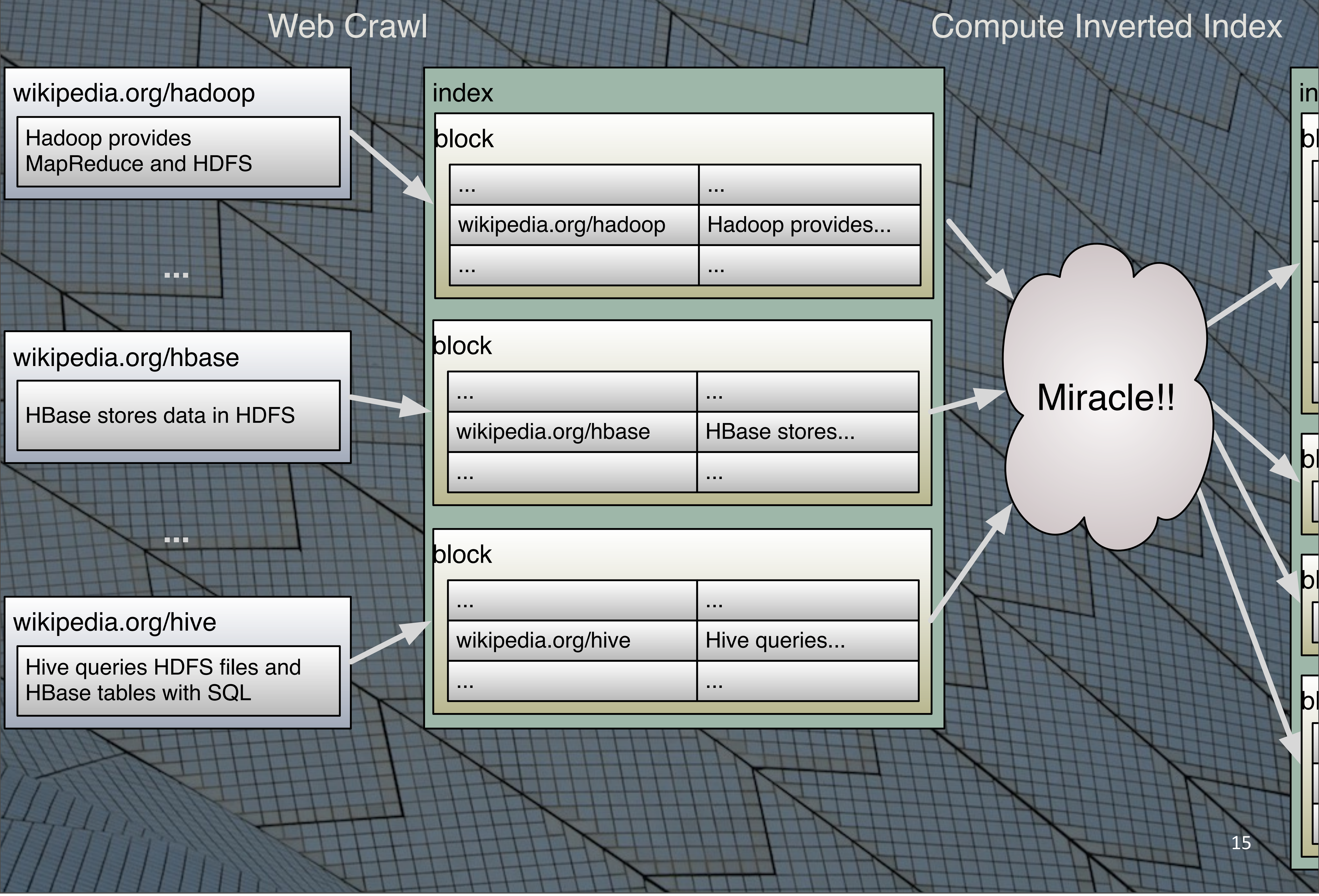
Historically, up to 2013, MapReduce was the officially-supported compute engine for writing all compute jobs.

Example: Inverted Index



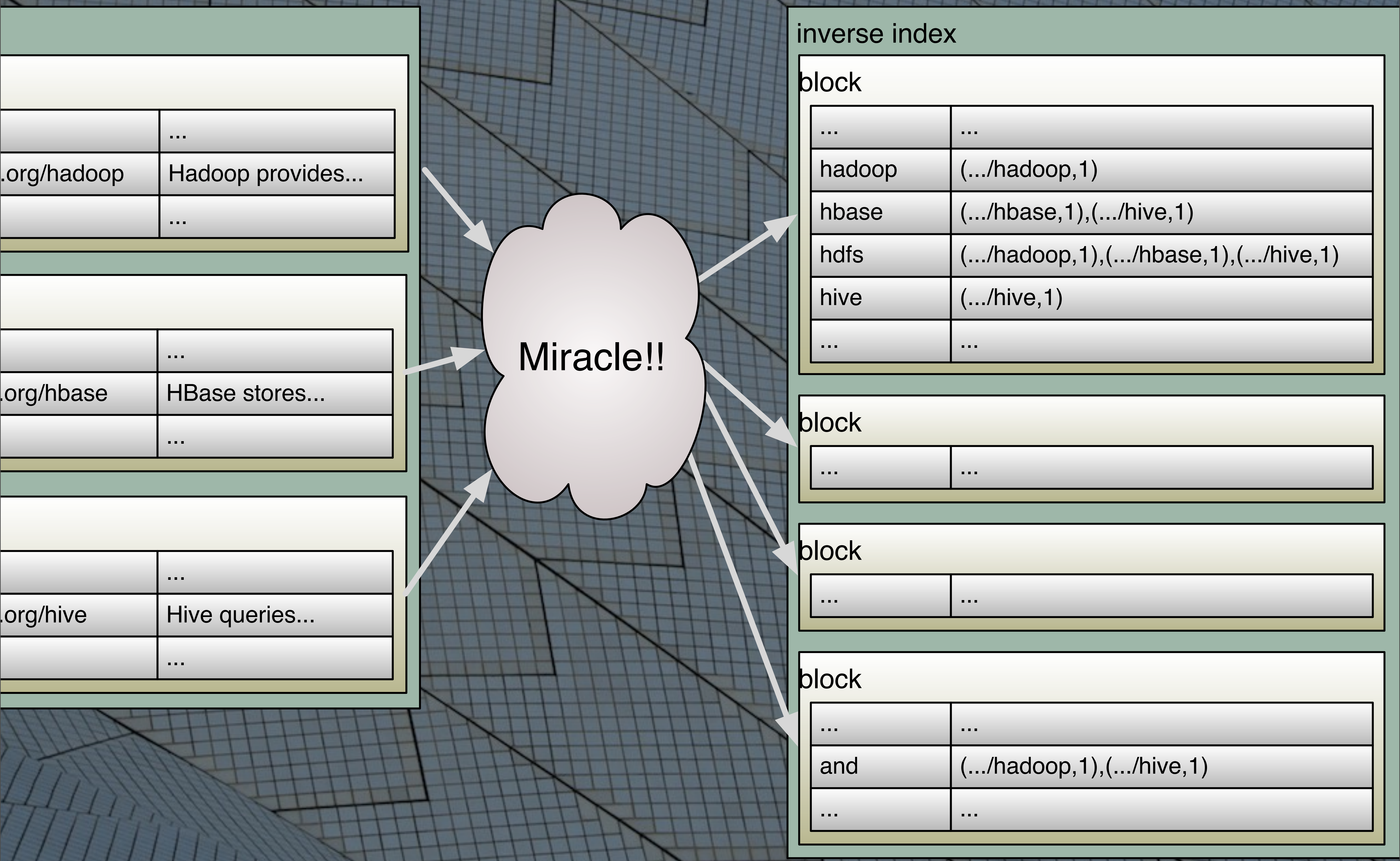
Example: Inverted Index



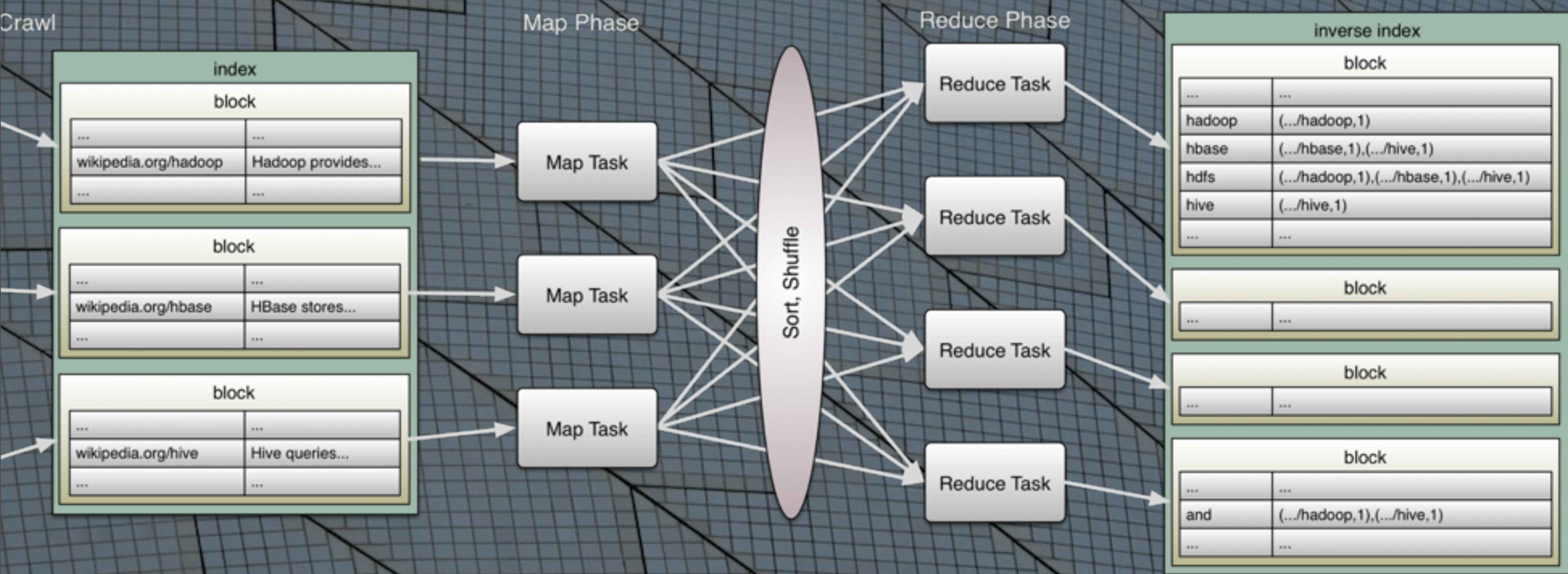


Web Crawl
Compute Inverted Index
wikipedia.org/hadoop
Hadoop provides MapReduce and HDFS
...
wikipedia.org/hbase
HBase stores data in HDFS
...
wikipedia.org/hive
Hive queries HDFS files and HBase tables with SQL
index
block
...
wikipedia.org/hadoop
Hadoop provides...
block
wikipedia.org/hbase
HBase stores...
block
wikipedia.org/hive
Hive queries...
Miracle!!

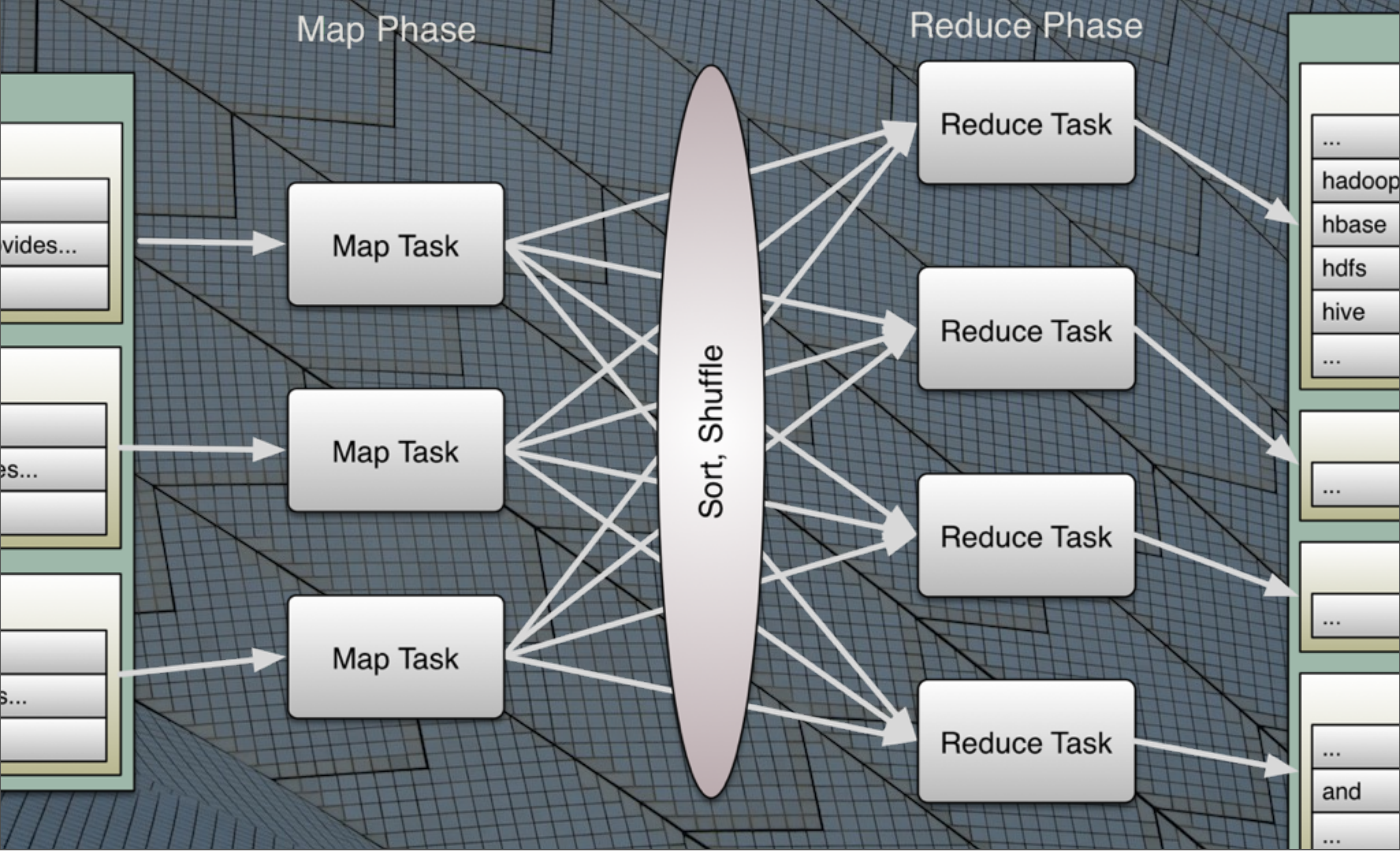
Compute Inverted Index



1 Map step + 1 Reduce step

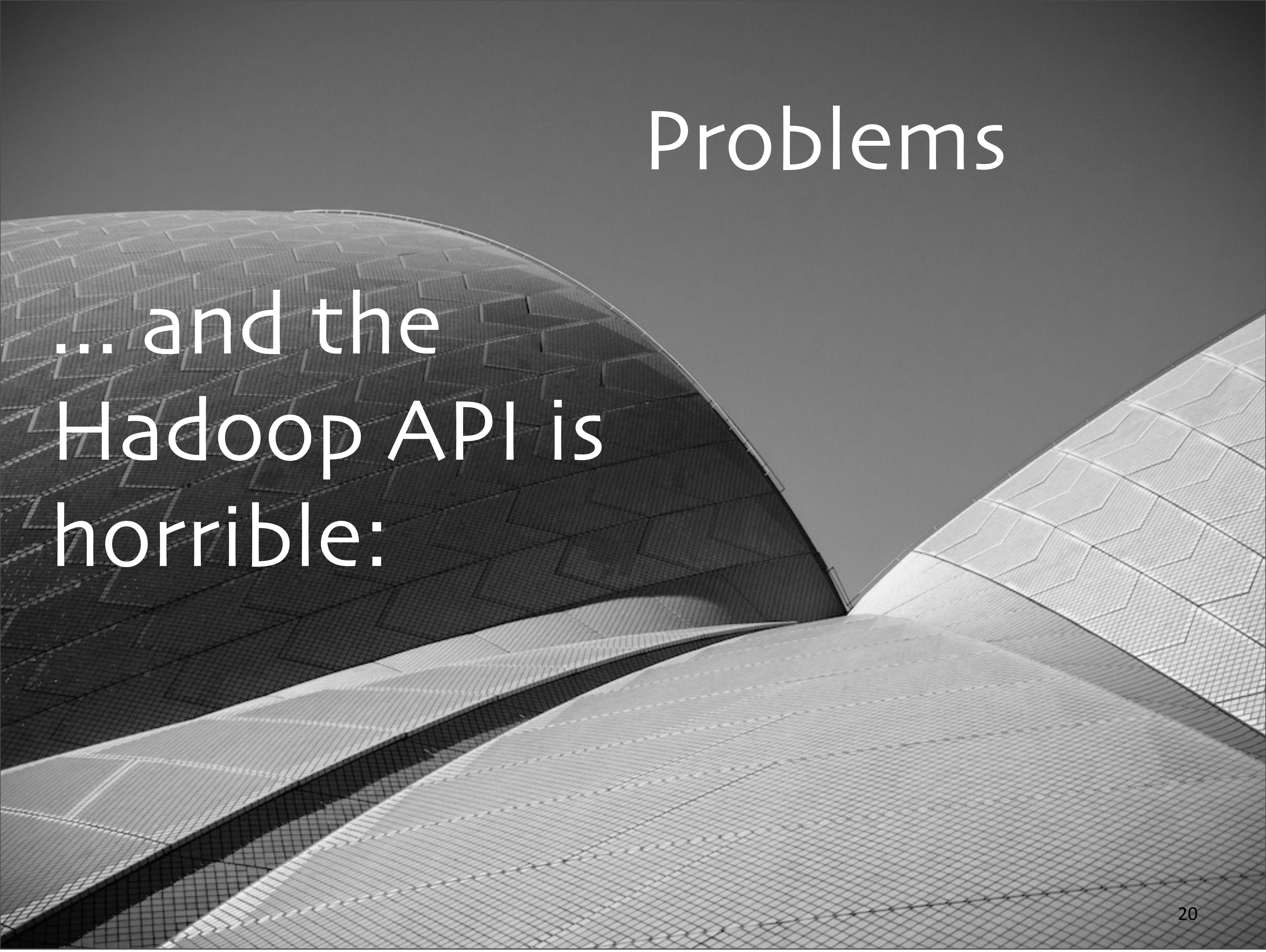


1 Map step + 1 Reduce step



Tuesday, December 9, 14

Nontrivial algorithms are hard to convert to just map and reduce steps, even though you can sequence multiple map+reduce “jobs”. It takes specialized expertise of the tricks of the trade.



Tuesday, December 9, 14

The Hadoop API is very low level and tedious...

```java
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {

  public static void main(String[] args) {
    JobClient client = new JobClient();
    JobConf conf =
     new JobConf(LineIndexer.class);

    conf.setJobName("LineIndexer");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(Text.class);
```


For example, the classic inverted index, used to convert an index of document locations (e.g., URLs) to words into the reverse; an index from words to doc locations. It's the basis of search engines.

I'm not going to explain the details. The point is to notice all the boilerplate that obscures the problem logic.

Everything is in one outer class. We start with a main routine that sets up the job.

I used yellow for method calls, because methods do the real work!! But notice that most of the functions in this code don't really do a whole lot of work for us...

```
JobClient client = new JobClient();
JobConf conf =
 new JobConf(LineIndexer.class);

conf.setJobName("LineIndexer");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
FileInputFormat.addInputPath(conf,
 new Path("input"));
FileOutputFormat.setOutputPath(conf,
 new Path("output"));
conf.setMapperClass(
  LineIndexMapper.class);
conf.setReducerClass(
  LineIndexReducer.class);

client.setConf(conf);
```


Boilerplate

```java
  new Path("output"));
 conf.setMapperClass(
   LineIndexMapper.class);
 conf.setReducerClass(
   LineIndexReducer.class);

 client.setConf(conf);

 try {
  JobClient.runJob(conf);
 } catch (Exception e) {
  e.printStackTrace();
 }
}

public static class LineIndexMapper
 extends MapReduceBase
```


Tuesday, December 9, 14

main ends with a try-catch clause to run the job.

```
public static class LineIndexMapper
 extends MapReduceBase
 implements Mapper<LongWritable, Text,
                   Text, Text> {
 private final static Text word =
  new Text();
 private final static Text location =
  new Text();

 public void map(
  LongWritable key, Text val,
  OutputCollector<Text, Text> output,
  Reporter reporter) throws IOException {

  FileSplit fileSplit =
   (FileSplit)reporter.getInputSplit();
  String fileName =
```


This is the LineIndexMapper class for the mapper. The map method does the real work of tokenization and writing the (word, document-name) tuples.

```
    FileSplit fileSplit =
     (FileSplit)reporter.getInputSplit();
    String fileName =
     fileSplit.getPath().getName();
    location.set(fileName);

    String line = val.toString();
    StringTokenizer itr = new
     StringTokenizer(line.toLowerCase());
    while (itr.hasMoreTokens()) {
     word.set(itr.nextToken());
     output.collect(word, location);
    }
  }
}

public static class LineIndexReducer
```


The rest of the LineIndexMapper class and map method.

```java
public static class LineIndexReducer
  extends MapReduceBase
  implements Reducer<Text, Text,
                     Text, Text> {
  public void reduce(Text key,
    Iterator<Text> values,
    OutputCollector<Text, Text> output,
    Reporter reporter) throws IOException {
    boolean first = true;
    StringBuilder toReturn =
      new StringBuilder();
    while (values.hasNext()) {
      if (!first)
        toReturn.append(", ");
      first=false;
      toReturn.append(
        values.next().toString());
```


The reducer class, LineIndexReducer, with the reduce method that is called for each key and a list of values for that key. The reducer is stupid; it just reformats the values collection into a long string and writes the final (word,list-string) output.

```
  Reporter reporter) throws IOException {
    boolean first = true;
    StringBuilder toReturn =
     new StringBuilder();
    while (values.hasNext()) {
     if (!first)
      toReturn.append(", ");
     first=false;
     toReturn.append(
      values.next().toString());
    }
    output.collect(key,
     new Text(toReturn.toString()));
   }
  }
 }
```

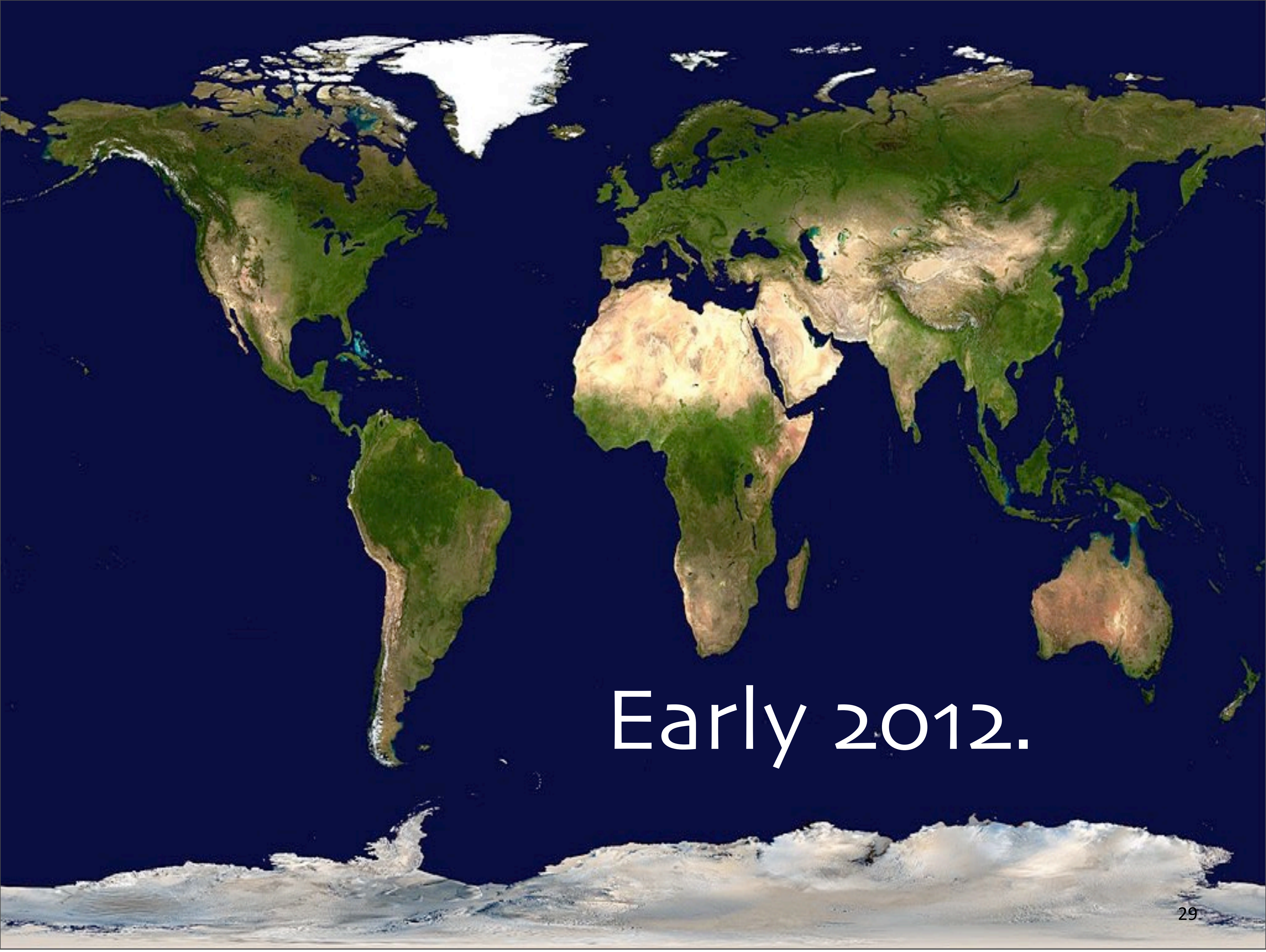

EOF

Altogether

```java
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {

 public static void main(String[] args) {
  JobClient client = new JobClient();
  JobConf conf =
   new JobConf(LineIndexer.class);

  conf.setJobName("LineIndexer");
  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(Text.class);
  FileInputFormat.addInputPath(conf,
   new Path("input"));
  FileOutputFormat.setOutputPath(conf,
   new Path("output"));
  conf.setMapperClass(
    LineIndexMapper.class);
  conf.setReducerClass(
    LineIndexReducer.class);

  client.setConf(conf);

  try {
   JobClient.runJob(conf);
  } catch (Exception e) {
   e.printStackTrace();
  }
 }

 public static class LineIndexMapper
  extends MapReduceBase
  implements Mapper<LongWritable, Text,
                    Text, Text> {
  private final static Text word =
   new Text();
  private final static Text location =
   new Text();

  public void map(
   LongWritable key, Text val,
   OutputCollector<Text, Text> output,
   Reporter reporter) throws IOException {

   FileSplit fileSplit =
    (FileSplit)reporter.getInputSplit();
   String fileName =
    fileSplit.getPath().getName();
   location.set(fileName);

   String line = val.toString();
   StringTokenizer itr = new
    StringTokenizer(line.toLowerCase());
   while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    output.collect(word, location);
   }
  }
 }

 public static class LineIndexReducer
  extends MapReduceBase
  implements Reducer<Text, Text,
                     Text, Text> {
  public void reduce(Text key,
   Iterator<Text> values,
   OutputCollector<Text, Text> output,
   Reporter reporter) throws IOException {
   boolean first = true;
   StringBuilder toReturn =
    new StringBuilder();
   while (values.hasNext()) {
    if (!first)
     toReturn.append(", ");
    first=false;
    toReturn.append(
     values.next().toString());
   }
   output.collect(key,
    new Text(toReturn.toString()));
  }
 }
}
```


Tuesday, December 9, 14

The whole shebang (6pt. font) This would take a few hours to write, test, etc. assuming you already know the API and the idioms for using it.



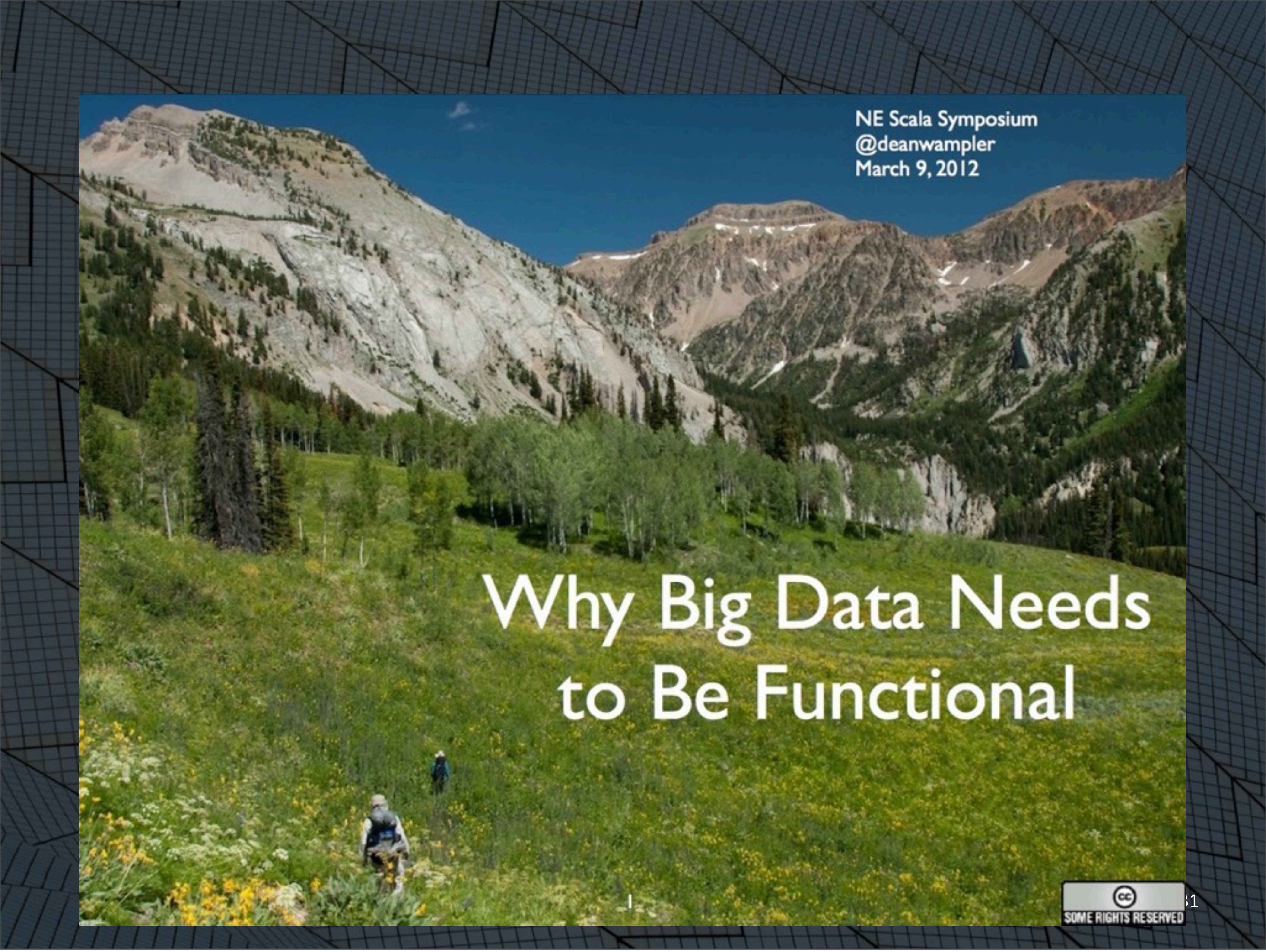
Let's put all this into perspective, circa 2012...

http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg

Dean Wampler

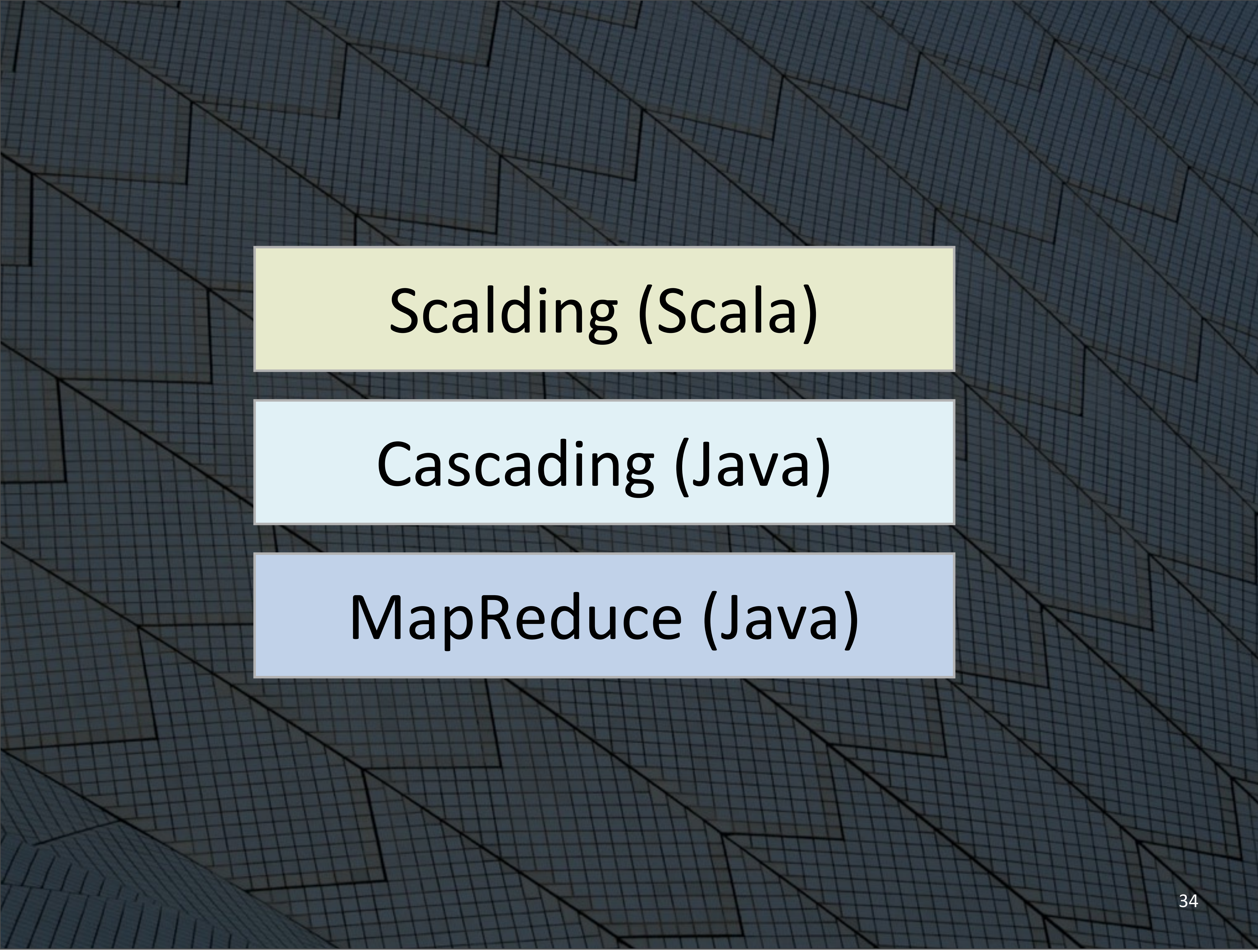
"Trolling the Hadoop community since 2012..."

Tuesday, December 9, 14


NE Scala Symposium
@deanwampler
March 9, 2012
Why Big Data Needs
to Be Functional
1
SOME RIGHTS RESERVED

In which I claimed that:

Twitter wrote a Scala API, https://github.com/twitter/scalding, to hide the mess. Actually, Scalding sits on top of Cascading (http://cascading.org) a higher-level Java API that exposes more sensible “combinators” of operations, but is still somewhat verbose due to the pre-Java 8 conventions it must use. Scalding gives us the full benefits of Scala syntax and functional operations, “combinators”.



Tuesday, December 9, 14

Twitter wrote a Scala API, https://github.com/twitter/scalding, to hide the mess. Actually, Scalding sits on top of Cascading (http://cascading.org) a higher-level Java API that exposes more sensible “combinators” of operations, but is still somewhat verbose due to the pre-Java 8 conventions it must use. Scalding gives us the full benefits of Scala syntax and functional operations, “combinators”.

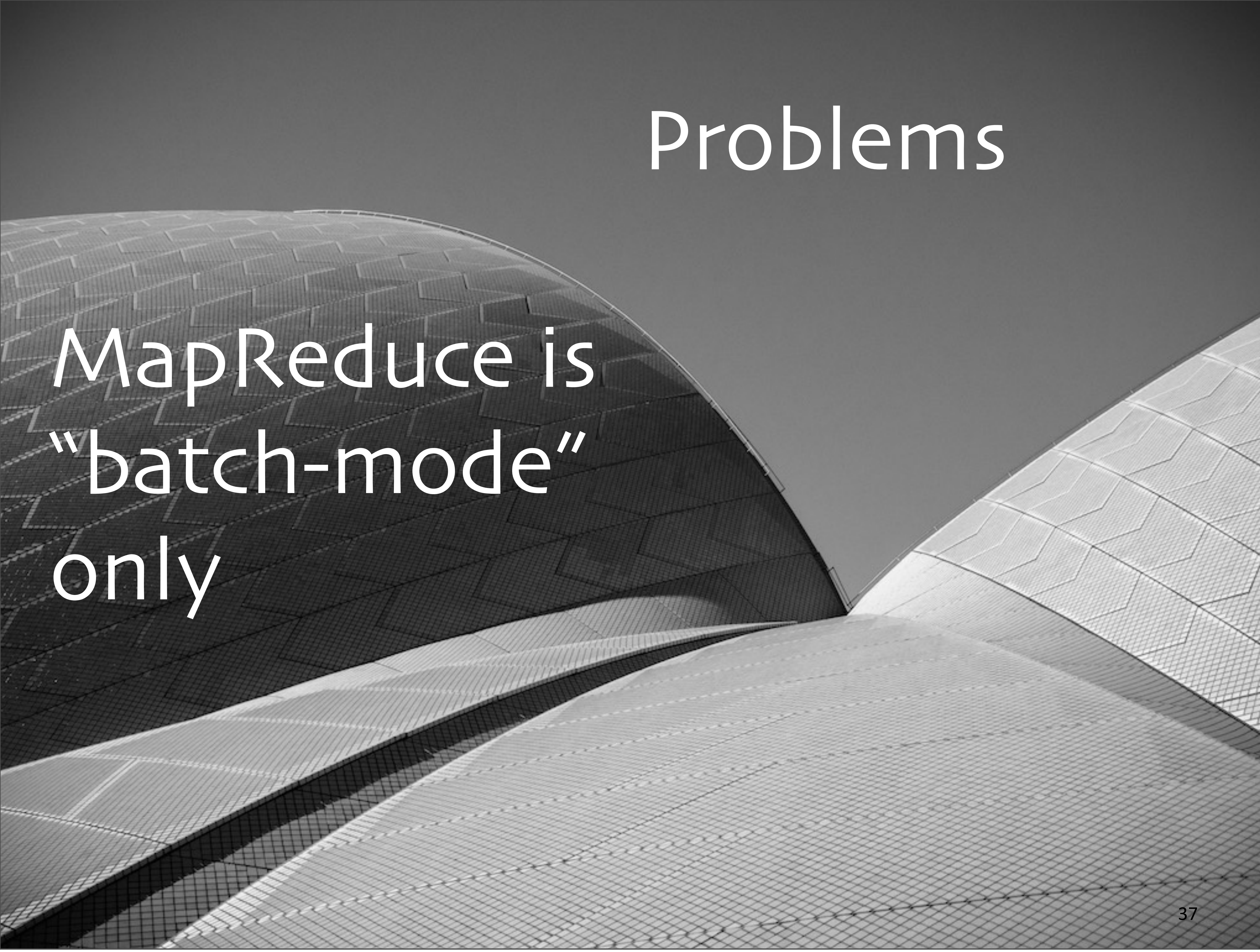
```scala
import com.twitter.scalding._

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

  val texts = Tsv("texts.tsv", ('id, 'text))

  val wordToIds = texts
    .flatMap(('id, 'text) -> ('word, 'id2)) {
      fields: (Long, String) =>
        val (id2, text) =
          text.split("\\s+").map {
            word => (word, id2)
          }
    }
```


Dramatically smaller, succinct code! (https://github.com/echen/rosetta-scone/blob/master/inverted-index/InvertedIndex.scala) Note that this example assumes a slightly different input data format (more than one document per file, with each document id followed by the text all on a single line, tab separated.

```scala
  .flatMap(('id, 'text) -> ('word, 'id2)) {
    fields: (Long, String) =>
     val (id2, text) =
      text.split("\\s+").map {
       word => (word, id2)
      }
   }

  val invertedIndex =
   wordToTweets.groupBy('word) {
    _.toList[Long]('id2 -> 'ids)
   }
  invertedIndex.write(Tsv("output.tsv"))
}
```

That's it!

Dramatically smaller, succinct code! (https://github.com/echen/rosetta-scone/blob/master/inverted-index/InvertedIndex.scala) Note that this example assumes a slightly different input data format (more than one document per file, with each document id followed by the text all on a single line, tab separated.



Tuesday, December 9, 14

You can’t do event-stream processing (a.k.a. “real-time”) with MapReduce, only batch mode processing.

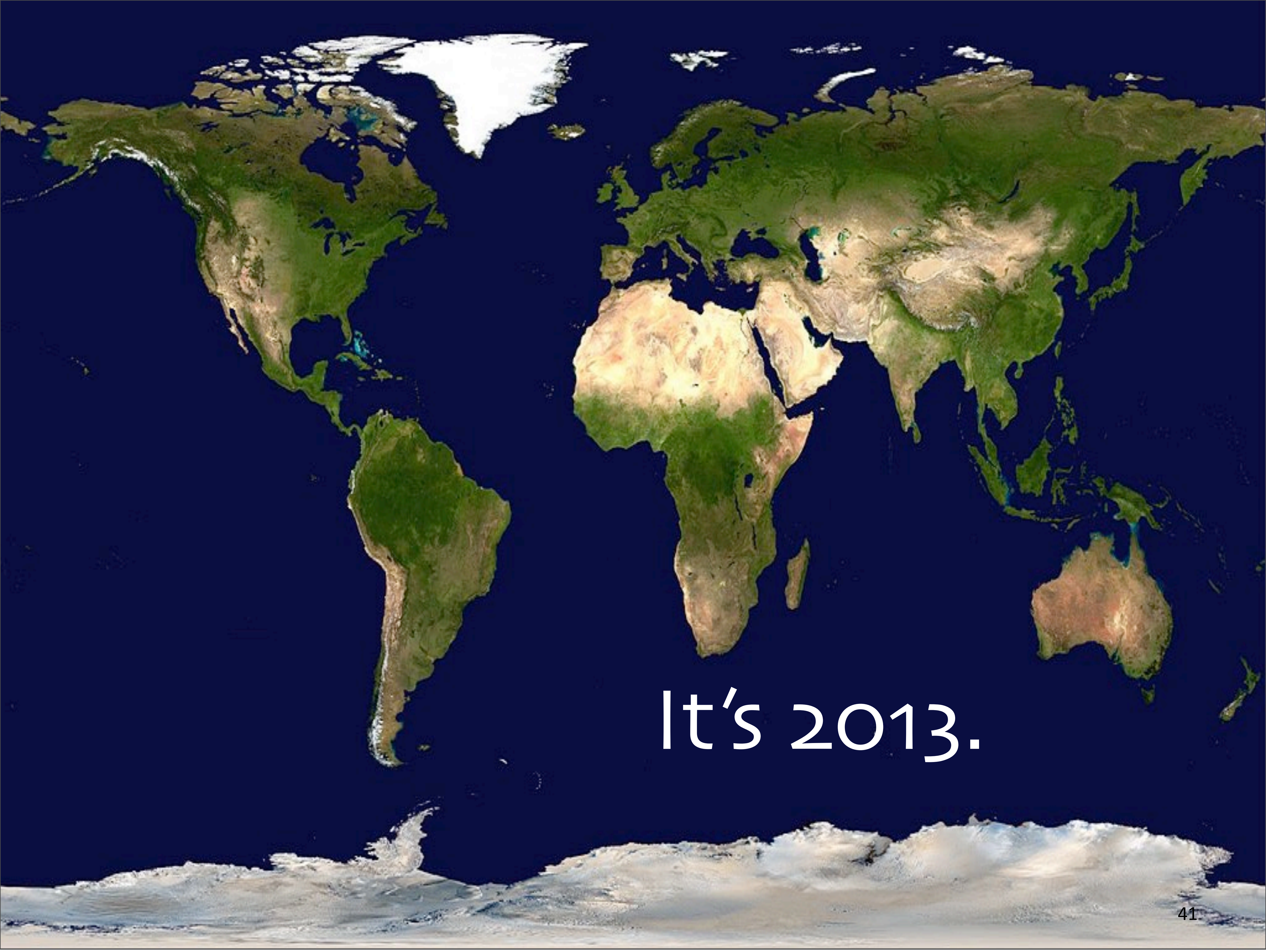
Tuesday, December 9, 14

Storm is a popular framework for scalable, resilient, event-stream processing.

Storm is a popular framework for scalable, resilient, event-stream processing.
Twitter wrote a Scalding-like API called Summingbird (https://github.com/twitter/summingbird) that abstracts over Storm and Scalding, so you can write one program that can run in batch mode or process events.
(For time's sake, I won't show an example.)

While your algorithm may be implemented using a sequence of MR jobs (which takes specialized skills to write...), the runtime system doesn't understand this, so the output of each job is flushed to disk (HDFS), even if it's TBs of data. Then it is read back into memory as soon as the next job in the sequence starts!

This problem plagues Scalding (and Cascading), too, since they run on top of MapReduce (although Cascading is being ported to Spark, which we'll discuss next). However, as of mid-2014, Cascading is being ported to a new, faster runtime called Apache Tez, and it might be ported to Spark, which we'll discuss. Twitter is working on its own optimizations within Scalding. So the perf. issues should go away by the end of 2014.



Tuesday, December 9, 14

Let's put all this into perspective, circa 2013...

http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg

Tuesday, December 9, 14

The Hadoop community has realized over the last several years that a replacement for MapReduce is needed. While MR has served the community well, it's a decade old and shows clear limitations and problems, as we've seen. In late 2013, Cloudera, the largest Hadoop vendor officially embraced Spark as the replacement. Most of the other Hadoop vendors followed.

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
   .map { line =>
     val array = line.split("\t", 2)
     (array(0), array(1))
   }
   .flatMap {
     case (path, text) =>
```


This implementation is more sophisticated than the Scalding example. It also computes the count/document of each word. Hence, there are more steps (some of which could be merged).

It starts with imports, then declares a singleton object (a first-class concept in Scala), with a main routine (as in Java).

The methods are colored yellow again. Note this time how dense with meaning they are this time.

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
   .map { line =>
     val array = line.split("\t", 2)
     (array(0), array(1))
   }
   .flatMap {
     case (path, text) =>
```


This implementation is more sophisticated than the Scalding example. It also computes the count/document of each word. Hence, there are more steps (some of which could be merged).

It starts with imports, then declares a singleton object (a first-class concept in Scala), with a main routine (as in Java).

The methods are colored yellow again. Note this time how dense with meaning they are this time.

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
   .map { line =>
     val array = line.split("\t", 2)
     (array(0), array(1))
   }
   .flatMap {
     case (path, text) =>
```


You being the workflow by declaring a SparkContext (in “local” mode, in this case). The rest of the program is a sequence of function calls, analogous to “pipes” we connect together to perform the data flow.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
   .map { line =>
     val array = line.split("\t", 2)
     (array(0), array(1))
   }
   .flatMap {
     case (path, text) =>
```


Tuesday, December 9, 14

Next we read one or more text files. If “data/crawl” has 1 or more Hadoop-style “part-NNNNN” files, Spark will process all of them (in parallel if running a distributed configuration; they will be processed synchronously in local mode).

```
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
   text.split("""\W+""") map {
    word => (word, path)
   }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
```


Now we begin a sequence of transformations on the input data.

First, we map over each line, a string, to extract the original document id (i.e., file name, UUID), followed by the text in the document, all on one line. We assume tab is the separator. “(array(0), array(1))” returns a two-element “tuple”. Think of the output RDD has having a schema “String fileName, String text”.

flatMap maps over each of these 2-element tuples. We split the text into words on non-alphanumeric characters, then output collections of word (our ultimate, final “key”) and the path. Each line is converted to a collection of (word,path) pairs, so flatMap converts the collection of collections into one long “flat” collection of (word,path) pairs.

```scala
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
   text.split("""\W+""") map {
    word => (word, path)
   }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
```


Next, flatMap maps over each of these 2-element tuples. We split the text into words on non-alphanumeric characters, then output collections of word (our ultimate, final "key") and the path. Each line is converted to a collection of (word,path) pairs, so flatMap converts the collection of collections into one long "flat" collection of (word,path) pairs.

```scala
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
   text.split("""\W+""") map {
    word => (word, path)
   }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
```


Then we map over these pairs and add a single “seed” count of 1.

```
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.groupBy {
  case ((w, p), n) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }.sortBy {
      case (path, n) => (-n, path)
    }
    (w, seq2.mkString(", "))
}
.saveAsTextFile(argz.outpath)
```

((word1, path1), n1)
((word2, path2), n2)
...

reduceByKey does an implicit “group by” to bring together all occurrences of the same (word, path) and then sums up their counts.

```
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.groupBy {
  case ((w, p), n) => w
}
.map {
  ca
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }.sortBy {
      case (path, n) => (-n, path)
    }
    (w, seq2.mkString(", "))
}
.saveAsTextFile(argz.outpath)
```

(word, Seq((word, (path1, n1)), (word, (path2, n2)), ...))

...

Now we do an explicit group by to bring all the same words together. The output will be (word, Seq((word, (path1, n1)), (word, (path2, n2)), ...)), where Seq is a Scala abstraction for sequences, e.g., Lists, Vectors, etc.

```scala
      case ((w, p), n) => w
    }
    .map {
      case (w, seq) =>
        val seq2 = seq map {
          case (_, (p, n)) => (p, n)
        }.sortBy {
          case (path, n) => (-n, path)
        }
        (w, seq2.mkString(", "))
    }
    .saveAsT
    
    sc.stop()
  }
}
```

(word, “(path4, n4), (path3, n3), (path2, n2), ...”)
...

Back to the code, the last map step looks complicated, but all it does is sort the sequence by the count descending (because you would want the first elements in the list to be the documents that mention the word most frequently), then it converts the sequence into a string.

```
      case ((w, p), n) => w
    }
    .map {
      case (w, seq) =>
        val seq2 = seq map {
          case (_, (p, n)) => (p, n)
        }.sortBy {
          case (path, n) => (-n, path)
        }
        (w, seq2.mkString(", "))
    }
    .saveAsTextFile(argz.outpath)

    sc.stop()
  }
}
```


Tuesday, December 9, 14

We finish by saving the output as text file(s) and stopping the workflow.

Altogether

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
   .map { line =>
     val array = line.split("\t", 2)
     (array(0), array(1))
   }
   .flatMap {
     case (path, text) =>
      text.split("""\W+""") map {
       word => (word, path)
      }
   }
   .map {
     case (w, p) => ((w, p), 1)
   }
   .reduceByKey {
     case (n1, n2) => n1 + n2
   }
   .groupBy {
     case (w, (p, n)) => w
   }
   .map {
     case (w, seq) =>
       val seq2 = seq map {
         case (_, (p, n)) => (p, n)
       }
       (w, seq2.mkString(", "))
   }
   .saveAsTextFile(argz.outpath)

   sc.stop()
  }
}
```


Tuesday, December 9, 14

The whole shebang (14pt. font, this time)

```
  text.split("""\W+""") map {
    word => (word, path)
  }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.groupBy {
  case (w, (p, n)) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }
```

Powerful, beautiful combinators

Stop for a second and admire the simplicity and elegance of this code, even if you don't understand the details. This is what coding should be, IMHO, very concise, to the point, elegant to read. Hence, a highly-productive way to work!!

$$\nabla \cdot \mathbf{D} = \rho$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

Another example of a beautiful and profound DSL, in this case from the world of my first profession, Physics: Maxwell’s equations that unified Electricity and Magnetism: http://upload.wikimedia.org/wikipedia/commons/c/c4/Maxwell'sEquations.svg

Spark also has a streaming mode for "mini-batch" event handling.

(We'll come back to it...)

Spark Streaming operates on data "slices" of granularity as small as 0.5-1 second. Not quite the same as single event handling, but possibly all that's needed for ~90% (?) of scenarios.

Tuesday, December 9, 14

Let’s recap why is Scala taking over the big data world.

Elegant DSLs

```
...
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
...
```


You would be hard pressed to find a more concise DSL for big data!!

The JVM

Tuesday, December 9, 14

You have the rich Java ecosystem at your fingertips.

Functional Combinators

SQL Analogs

```
CREATE TABLE inverted_index (
  word    CHARACTER(64),
  id1     INTEGER,
  count1  INTEGER,
  id2     INTEGER,
  count2  INTEGER);
```

```
val inverted_index:
Stream[(String,Int,Int,Int,Int)]
```


You have functional “combinators”, side-effect free functions that combine/compose together to create complex algorithms with minimal effort.
For simplicity, assume we only keep the two documents where the word appears most frequently, along with the counts in each doc and we’ll assume integer ids for the documents..
We’ll model the same data set in Scala with a Stream, because we’re going to process it in “batch”.

Functional Combinators

```
SELECT * FROM inverted_index
WHERE word LIKE 'sc%';
```

Restrict

```
inverted_index.filter {
  case (word, _) =>
    word startsWith "sc"
}
```


Tuesday, December 9, 14

The equivalent of a SELECT ... WHERE query in both SQL and Scala.

Functional Combinators

```
SELECT word FROM inverted_index;
```

Projection

```
inverted_index.map {
  case (word, _, _, _, _) =>
    word
}
```


Projecting just the fields we want.

Functional Combinators

```
SELECT count1, COUNT(*) AS size
FROM inverted_index
GROUP BY count1
ORDER BY size DESC;
```

Group By and Order By

```
inverted_index.groupBy {
  case (_, _, count1, _, _) => count1
} map {
  case (count1, words) =>(count1,words.size)
} sortBy {
 case (count, size) => -size
}
```

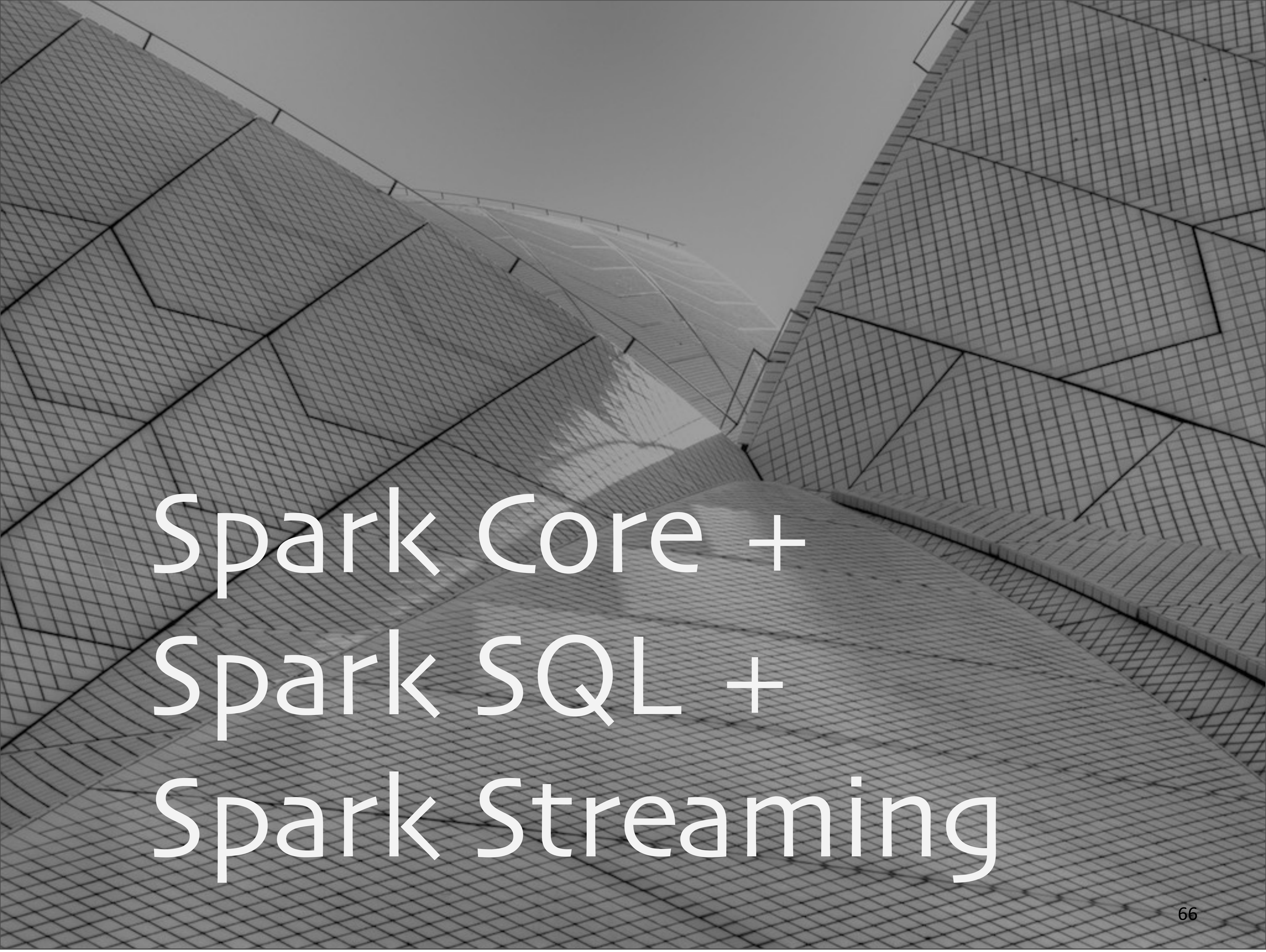

Tuesday, December 9, 14

Group By: group by the frequency of occurrence for the first document, then order by the group size, descending.

Unification?

Tuesday, December 9, 14

Can we unify SQL and Spark?



Tuesday, December 9, 14

https://spark.apache.org/sql/
https://spark.apache.org/streaming/

```scala
val sparkContext =
  new SparkContext("local[*]", "Much Wow!")
val streamingContext =
  new StreamingContext(
    sparkContext, Seconds(60))
val sqlContext =
  new SQLContext(sparkContext)
import sqlContext._

case class Flight(
  number:      Int,
  carrier:     String,
  origin:      String,
  destination: String,
  ...)
```


Tuesday, December 9, 14

```scala
val sparkContext =
  new SparkContext("local", "connections")
val streamingContext =
  new StreamingContext(
    sparkContext, Seconds(60))
val sqlContext =
  new SQLContext(sparkContext)
import sqlContext._

case class Flight(
  number:      Int,
  carrier:     String,
  origin:      String,
  destination: String,
  ...)
```


Create the SparkContext that manages for the driver program, followed by context object for streaming and another for the SQL extensions.
Note that the latter two take the SparkContext as an argument. The StreamingContext is constructed with an argument for the size of each batch of events to capture, every 60 seconds here.

```scala
val sparkContext =
  new SparkContext("local", "connections")
val streamingContext =
  new StreamingContext(
    sparkContext, Seconds(60))
val sqlContext =
  new SQLContext(sparkContext)
import sqlContext._

case class Flight(
  number:      Int,
  carrier:     String,
  origin:      String,
  destination: String,
  ...)
```


Tuesday, December 9, 14

For some APIs, Spark uses the idiom of importing the members of an instance.

```
import sqlContext._

case class Flight(
  number:      Int,
  carrier:     String,
  origin:      String,
  destination: String,
  ...)

object Flight {
  def parse(str: String): Option[Flight]=
    {...}
}

val server = ...  // IP address or name
val port = ...    // integer
val dStream =
  streamingContext.socketTextStream(
```


Tuesday, December 9, 14

Define a case class to represent a schema, and a companion object to define a parse method for parsing a string into an instance of the class. Return an option in case a string can't be parsed.
In this case, we'll simulate a data stream of data about airline flights, where the records contain only the flight number, carrier, and the origin and destination airports, and other data we'll ignore for this example, like times.

```
}

val server = ...   // IP address or name
val port = ...     // integer
val dStream =
  streamingContext.socketTextStream(
    server, port)

val flights = for {
  line <- dStream
  flight <- Flight.parse(line)
} yield flight

flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
```


Read the data stream from a socket originating at a given server:port address.

```scala
}

val server = ...   // IP address or name
val port = ...     // integer
val dStream =
  streamingContext.socketTextStream(
    server, port)

val flights = for {
  line <- dStream
  flight <- Flight.parse(line)
} yield flight

flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
```


Tuesday, December 9, 14

Parse the text stream into Flight records.

```scala
flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
      destination, COUNT(*)
    FROM flights
    GROUP BY carrier, origin, destination
    ORDER BY c4 DESC
    LIMIT 20""").foreach(println)
}

streamingContext.start()
streamingContext.awaitTermination()
streamingContext.stop()
```


A DStream is a collection of RDDs, so for each RDD (effectively, during each batch interval), invoke the anonymous function, which takes as arguments the RDD and current timestamp (epoch milliseconds), then we register the RDD as a “SQL” table named “flights” and run a query over it that groups by the carrier, origin, and destination, selects for those fields, plus the hard-coded timestamp (i.e., “hardcoded” for each batch interval), and the count of records in the group. Also order by the count descending, and return only the first 20 records.

```
flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
      destination, COUNT(*)
    FROM flights
    GROUP BY carrier, origin, destination
    ORDER BY c4 DESC
    LIMIT 20""").foreach(println)
}

streamingContext.start()
streamingContext.awaitTermination()
streamingContext.stop()
```


Start processing the stream, await termination, then close it all down.

Tuesday, December 9, 14

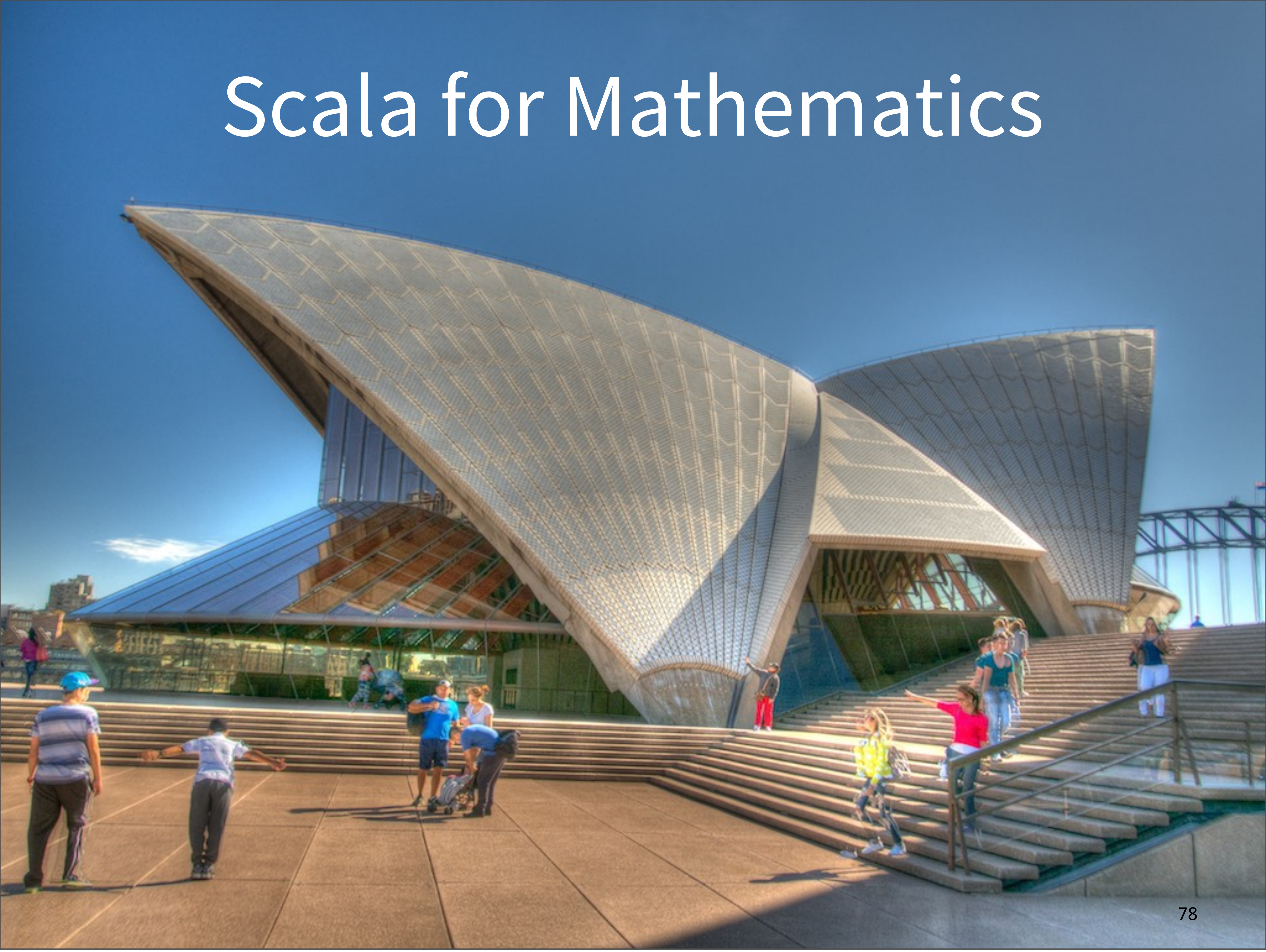
The title of this talk is in the present tense (present participle to be precise?), but has Scala already won? Is the game over?

Tuesday, December 9, 14

See also the bonus slides that follow.

Bonus Slides

Scala for Mathematics



Spire and Algebird. ScalaZ also has some of these data structures and algorithms.

Tuesday, December 9, 14

https://github.com/twitter/algebird

Algebraic types like Monoids, which generalize addition.

- A set of elements.
- An associative binary operation.
- An identity element.

For big data, you're often willing to trade 100% accuracy for much faster approximations. Algebird implements many well-known approx. algorithms, all of which can be modeled generically using Monoids or similar data structures.

Efficient approximation algorithms.

– "Add All the Things", infoq.com/presentations/abstract-algebra-analytics

Tuesday, December 9, 14

For big data, you're often willing to trade 100% accuracy for much faster approximations. Algebird implements many well-known approx. algorithms, all of which can be modeled generically using Monoids or similar data structures.

Hash, don’t Sample!

-- Twitter

Tuesday, December 9, 14

Sampling was a common way to deal with excessive amounts of data. The new mantra, exemplified by this catch phrase from Twitter’s data teams is to use approximation algorithms where the data is usually hashed into space-efficient data structures. You make a space vs. accuracy trade off. Often, approximate answers are good enough.

Tuesday, December 9, 14

https://github.com/non/spire

- Types: Complex, Quaternion, Rational, Real, Interval, ...
- Algebraic types: Semigroups, Monoids, Groups, Rings, Fields, Vector Spaces, ...
- Trigonometric Functions.
- ...

It exploits macros with generics to implement efficient computation without requiring custom types for particular primitives.

Tuesday, December 9, 14

What could be improved?

Tuesday, December 9, 14

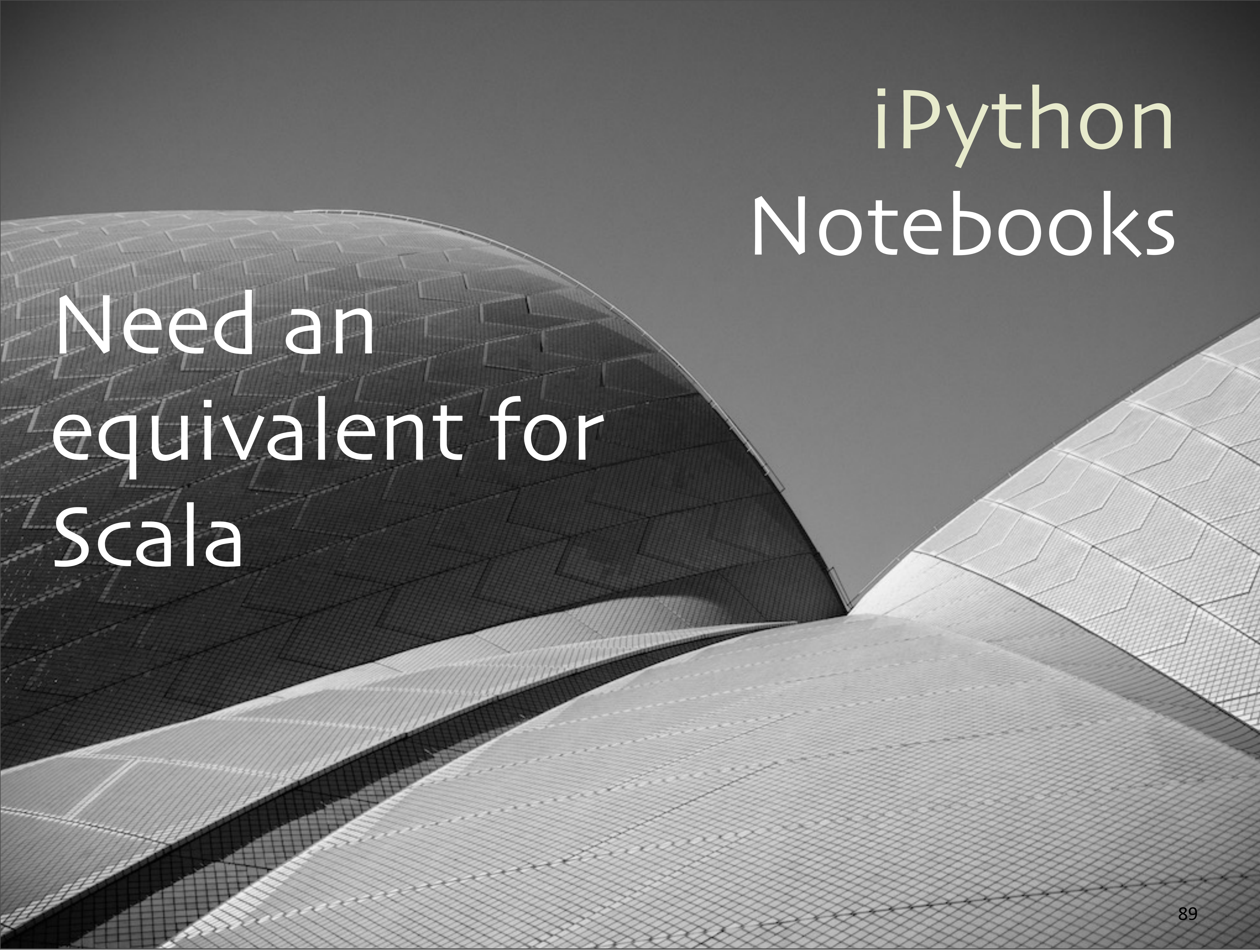
Spark and other tools use Tuples or Case classes to define schemas. In 2.11, case classes are no longer limited to 22 fields, but it’s not always convenient to define a case class when a tuple would do.

Tuesday, December 9, 14

Do we want to create an instance of an object for each record? We already have support for data formats like Parquet that implement columnar storage. Can our Scala APIs transparently use arrays of primitives for columns, for better performance? Spark has a support for Parquet which does this. Can we do it and should we do it for all data sets?

Tuesday, December 9, 14

Speaking of primitives, we need to solve the remaining issues with specialization of collections for primitives. Miniboxing??



Tuesday, December 9, 14

iPython Notebooks are very popular with data scientists, because they integrate data visualization, etc. with code.
github.com/Bridgewater/scala-notebook is a start.