

# ScalaDays 2014



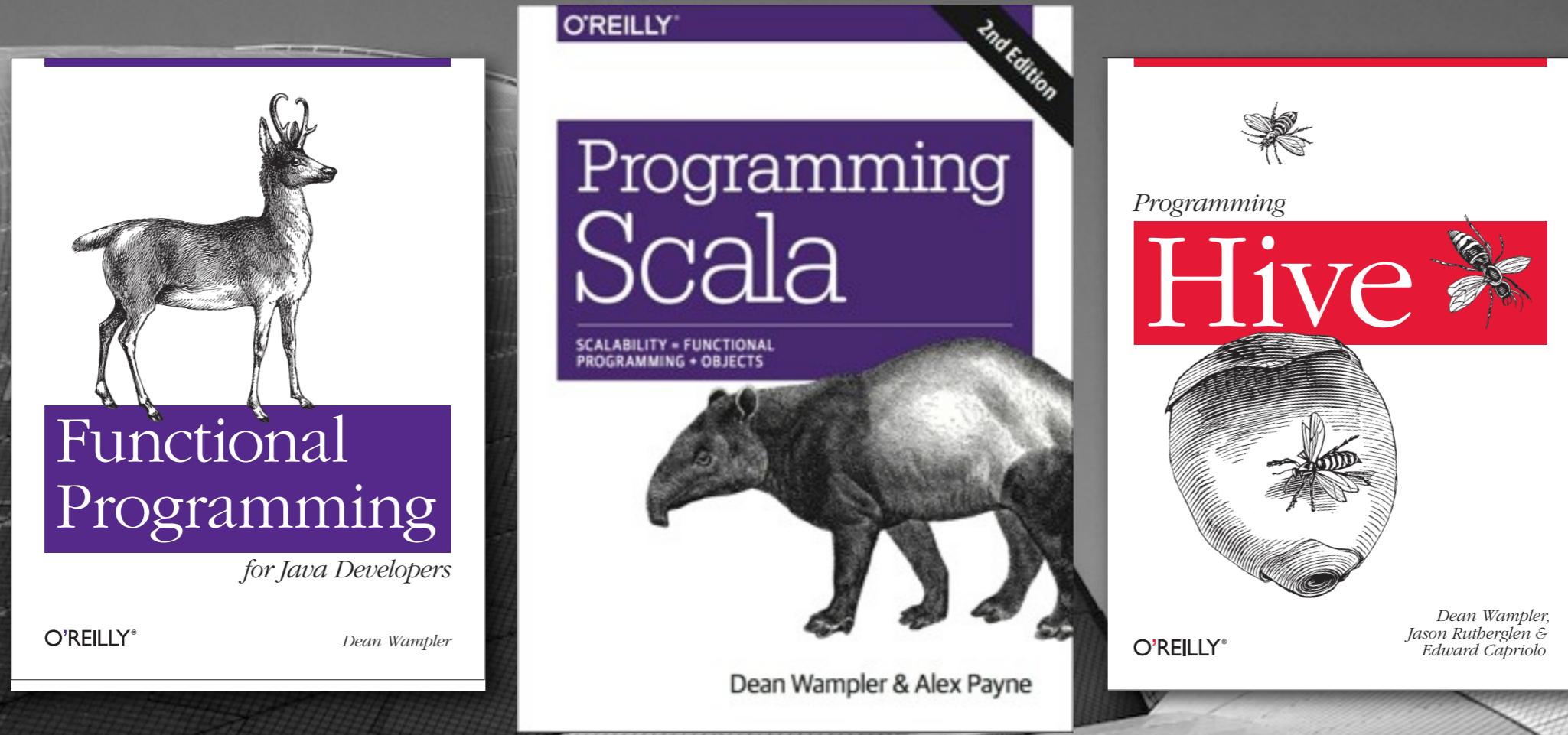
# Why Scala Is Taking Over the Big Data World



Tuesday, June 17, 14

Photos Copyright © Dean Wampler, 2011-2014, Some Rights Reserved.  
The content is free to reuse, but attribution is requested.  
<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

# Dean Wampler



[dean.wampler@typesafe.com](mailto:dean.wampler@typesafe.com)  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)  
[@deanwampler](https://twitter.com/deanwampler)



Tuesday, June 17, 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](http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg)



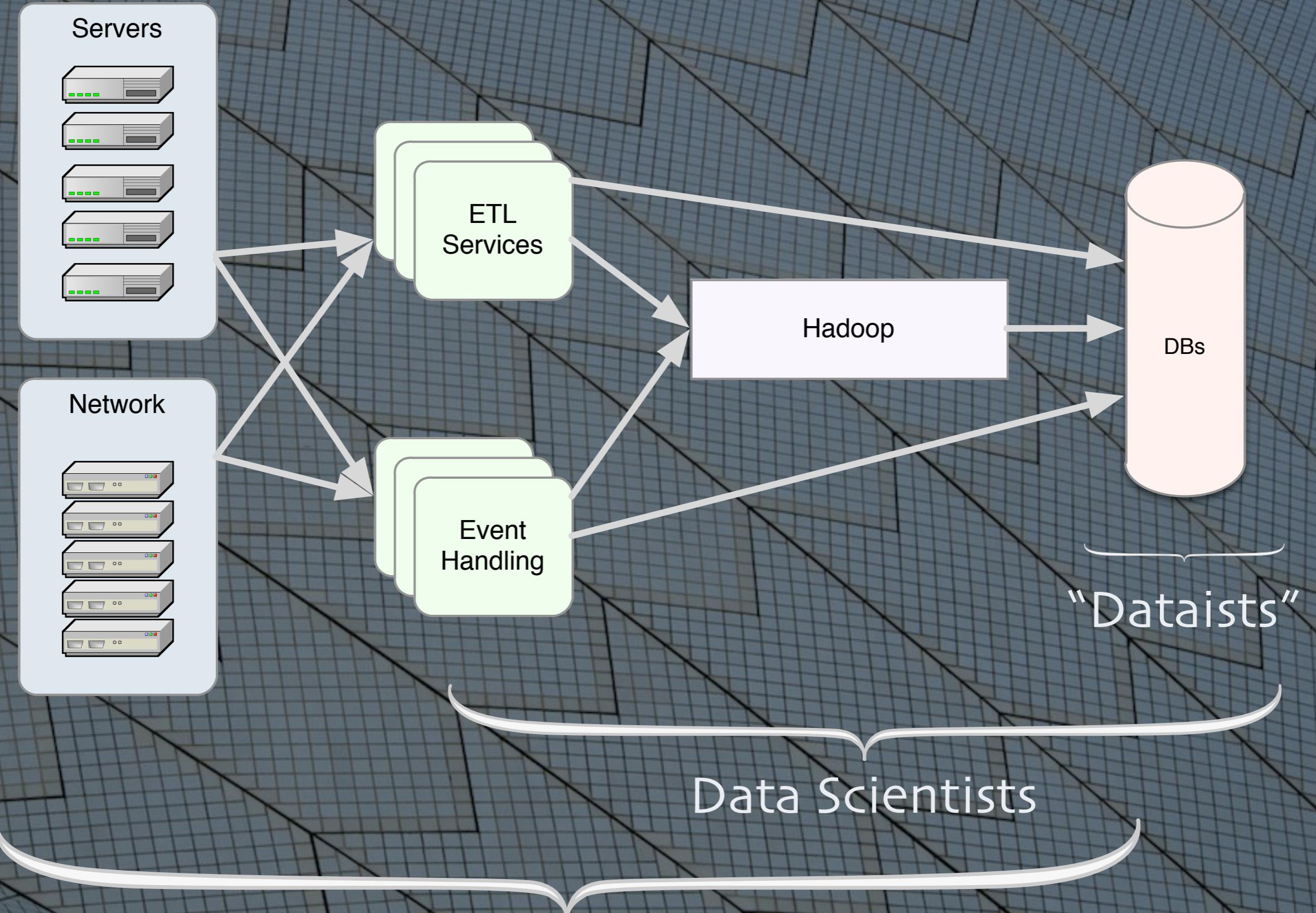
... It's 2013...

4

Tuesday, June 17, 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](http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg)



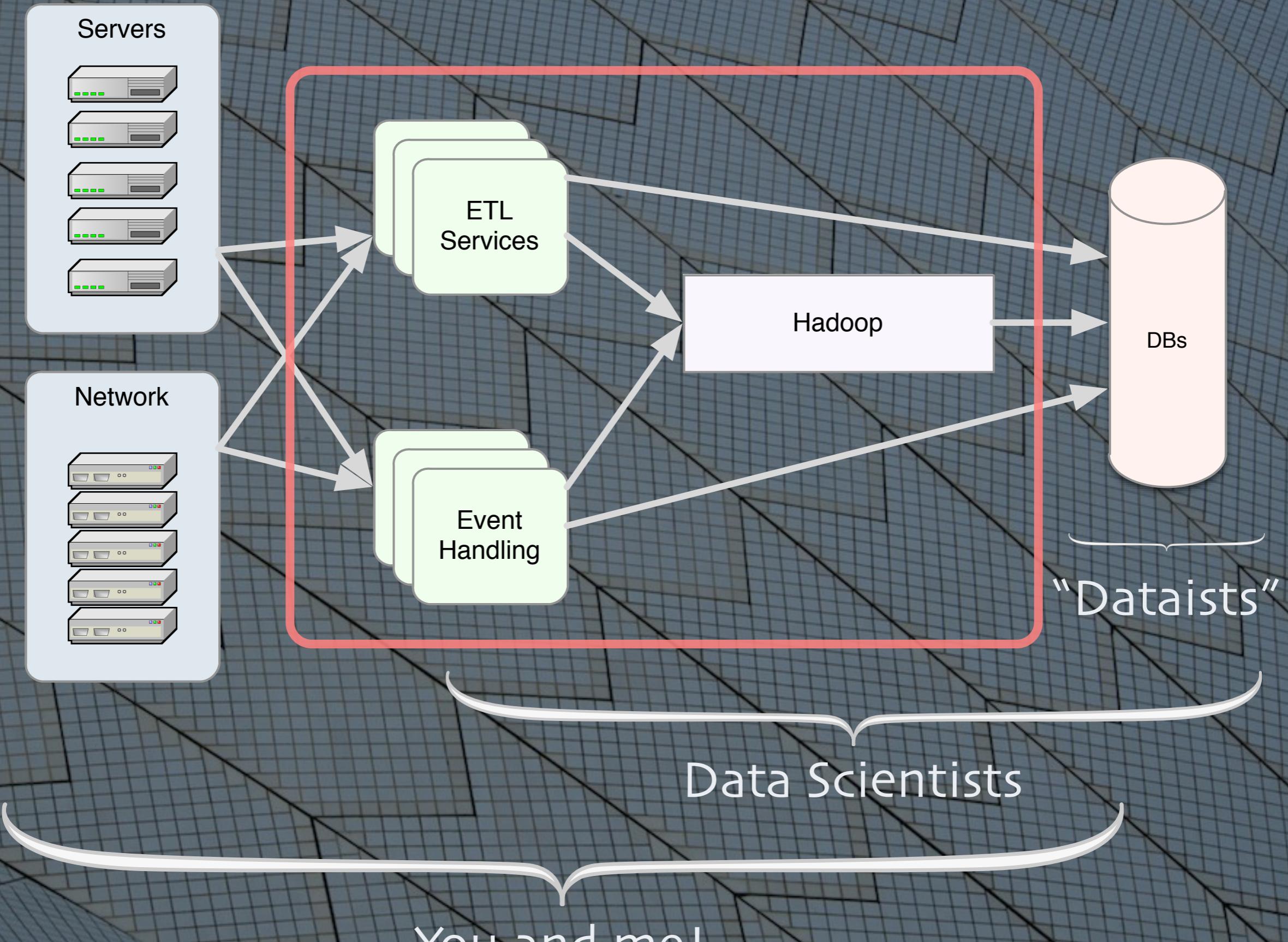
5

Tuesday, June 17, 14

“Dataists” - Data Architects/DBAs: SQL, SAS

Data Scientists - Statistics experts. Some programming, especially Python, R, Julia, maybe Matlab, etc.

Developers like us!



# Hadoop

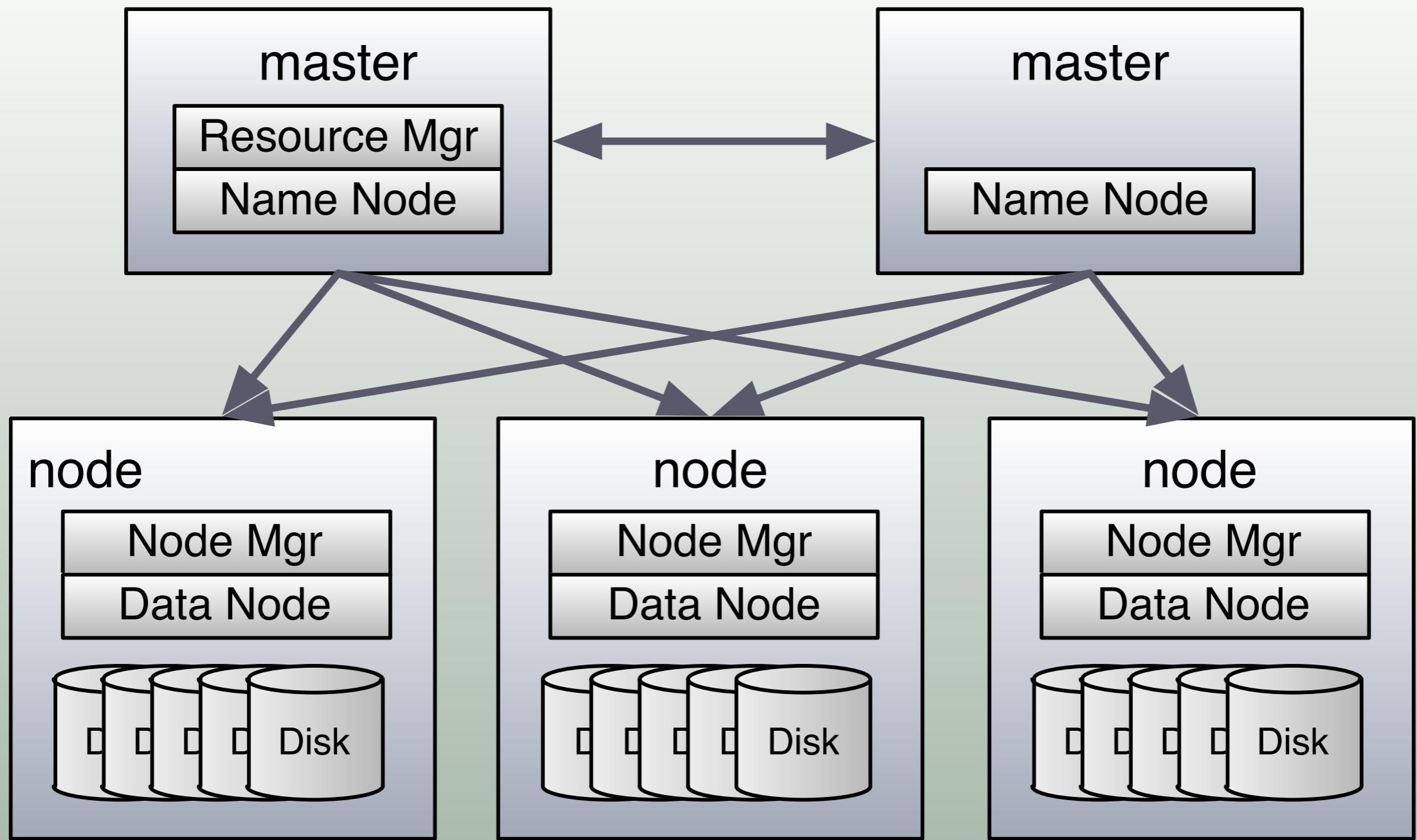


7

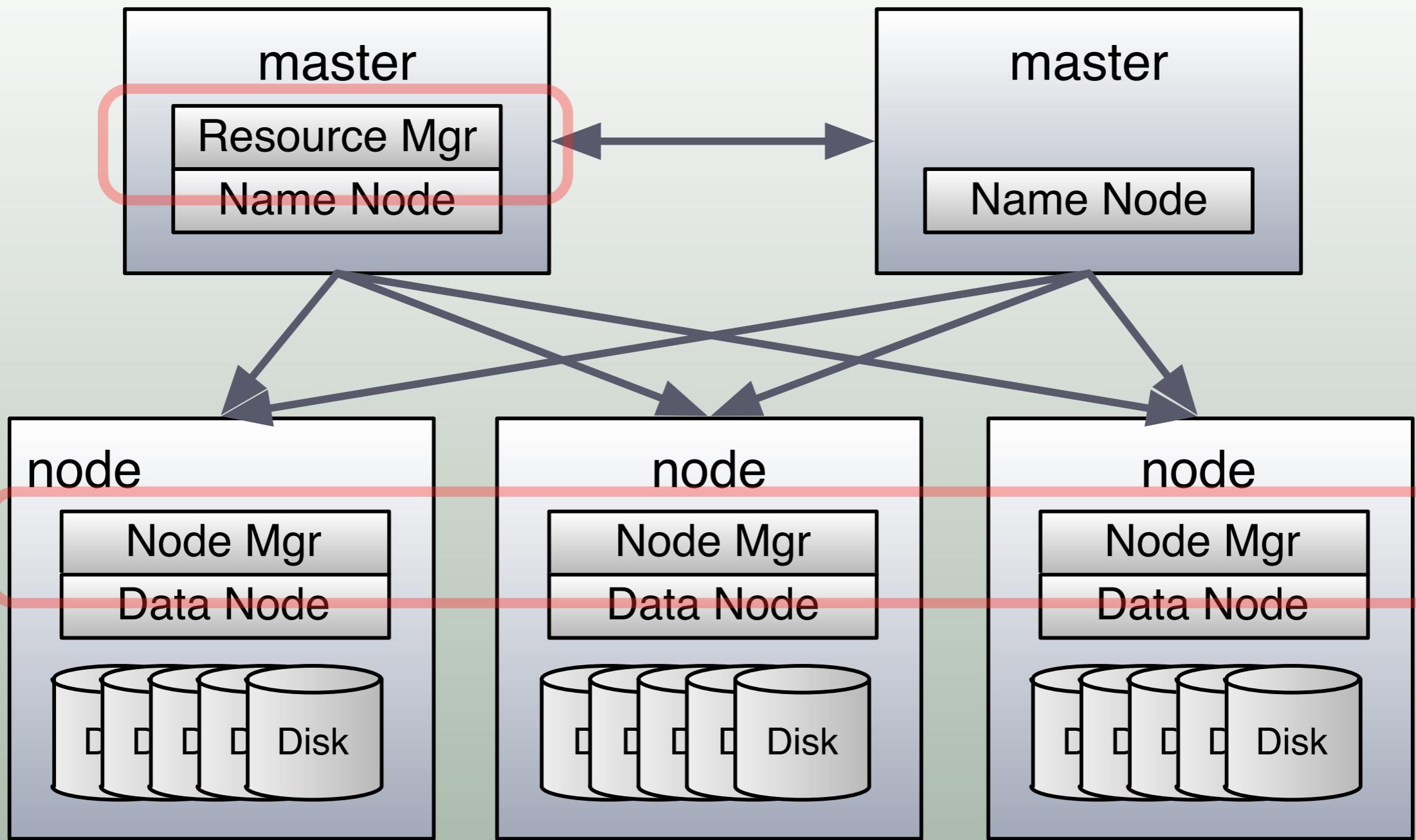
Tuesday, June 17, 14

Let's drill down to Hadoop, circa 2013.

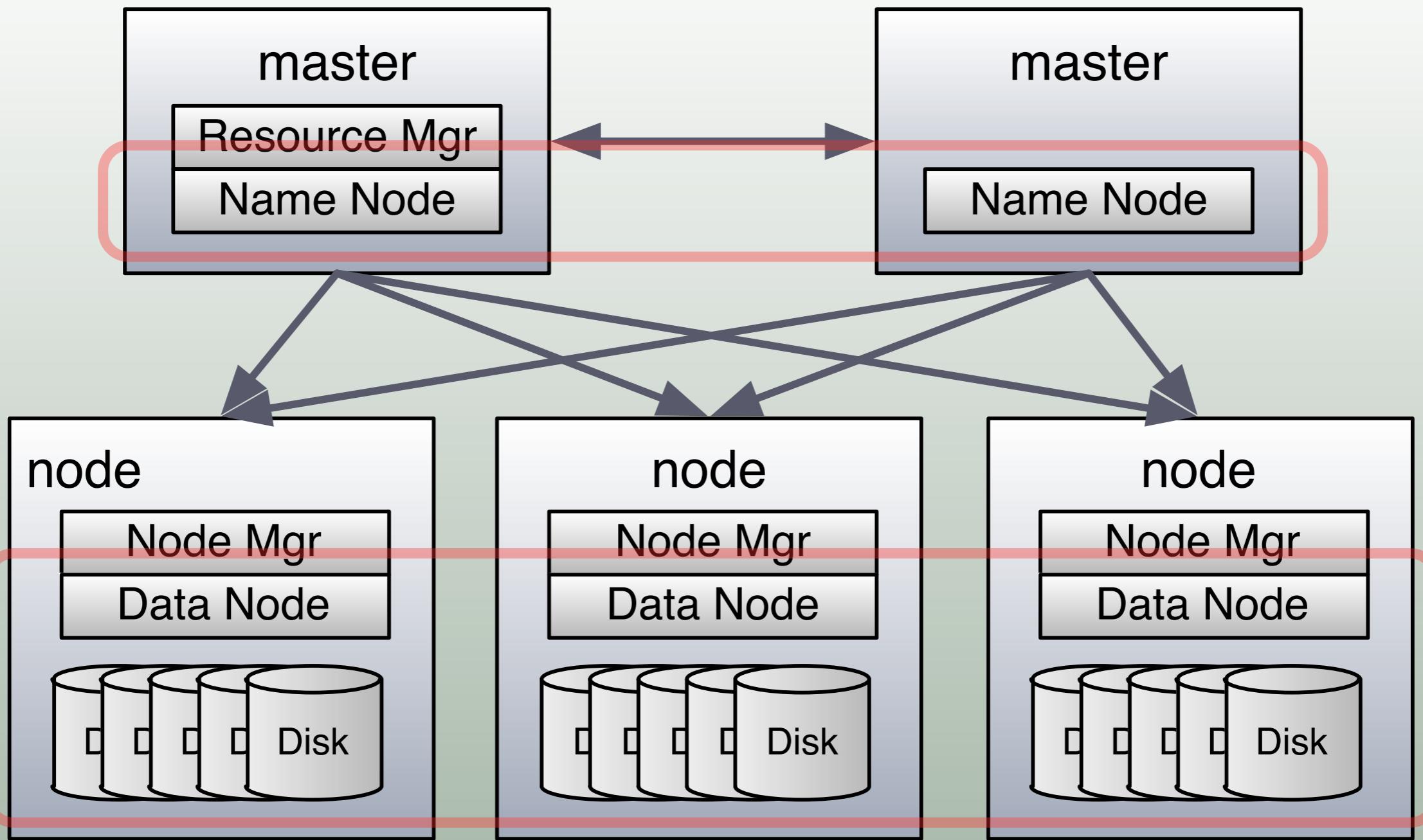
# Hadoop v2.X Cluster



# Hadoop v2.X Cluster



# Hadoop v2.X Cluster



10

Tuesday, June 17, 14

Hadoop 2 clusters federate the Name node services that manage the file system, HDFS. They provide horizontal scalability of file-system operations and resiliency when service instances fail. The data node services manage individual blocks for files.

# MapReduce

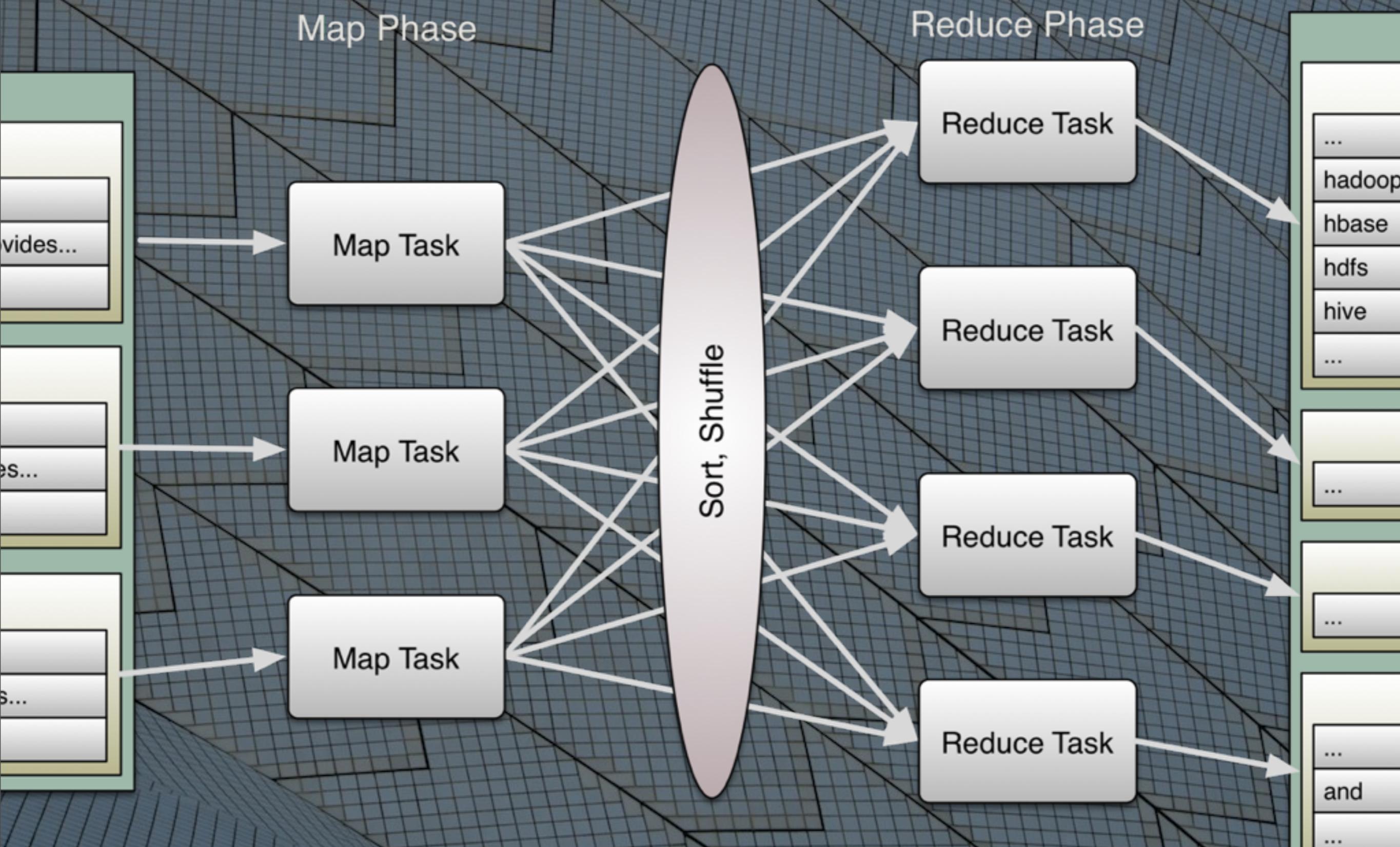
The classic  
compute model  
for Hadoop

11

Tuesday, June 17, 14

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

# 1 Map step + 1 Reduce step



# Problems

Hard to  
implement  
algorithms...

13

Tuesday, June 17, 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.

# Problems

... and the  
Hadoop API is  
horrible:

```
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);
```

15

Tuesday, June 17, 14

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);
```

16

```
    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> {
```

17

```
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 -
```

18

Tuesday, June 17, 14

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);  
}  
}  
}
```

19

```
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, toReturn);  
    }  
}
```

20

Tuesday, June 17, 14

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.

```
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()));
}
```

# Altogether

```
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()));
        }
    }
}
```

# Solution

Use Scalding

23

Tuesday, June 17, 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.

```
import com.twitter.scalding._

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

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

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

24

Tuesday, June 17, 14

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).

```
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) =
        fields.text.split("\\s+").map {
          word => (word, id2)
        }
    }
}
```

Das ist alles!

25

# Problems

MapReduce is  
“batch-mode”  
only



Twitter wrote  
Summingbird for  
Storm + Scalding

Storm!

27

Tuesday, June 17, 14

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.)

# Problems

Flush to disk,  
then reread  
between jobs

Wasteful

28

Tuesday, June 17, 14

While your algorithm may be implemented using a sequence of MR jobs, 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).

# Solution

Use Spark

(Late 2013)

29

Tuesday, June 17, 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.

```
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(s =>
```

Tuesday, June 17, 14

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.

```
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)  
}
```

Tuesday, June 17, 14

You begin 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.

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).

```
.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
}
.map {
```

Tuesday, June 17, 14

sc.textFile returns an RDD with a string for each line of input text. So, the first thing we do is map over these strings to extract the original document id (i.e., file name), 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 {
    case (path, text) =>
        text.split("""\W+""") map {
            word => (word, path)
        }
}

.map {
    case (w, p) => ((w, p), 1)
}

.reduceByKey {
    case (n1, n2) => n1 + n2
}

.map {
    case ((w, p), n) => (w, (p, n))
}

.groupBy {
```

Powerful

combinators

Tuesday, June 17, 14

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.

Then we map over these pairs and add a single count of 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
}

.map {
    case ((w, p), n) => (w, (p, n))
}

.groupBy {
```

Beautiful,  
no?

Tuesday, June 17, 14

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.

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

$$\nabla \cdot D = \rho$$

$$\nabla \cdot B = 0$$

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

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

Tuesday, June 17, 14

Another example of a beautiful and profound DSL, in this case from the world of Physics: Maxwell's equations: <http://upload.wikimedia.org/wikipedia/commons/c/c4/Maxwell'sEquations.svg>

```
}

.reduceByKey {
    case (n1, n2) => n1 + n2
}

.map {
    case ((w, p), n) => (w, (p, n))
}

(groupBy {
    case (w, (p, n)) => w ...
})  
(...)

.map {
    case (w, seq) =>
        val seq2 = seq map {
            case (_, (p, n)) => (p, n)
        }
        (w, seq2.mkString("", ""))
}
```

Tuesday, June 17, 14

reduceByKey does an implicit “group by” to bring together all occurrences of the same (word, path) and then sums up their counts.  
Note the input to the next map is now ((word, path), n), where n is now  $\geq 1$ . We transform these tuples into the form we actually want, (word, (path, n)).

```
}

  .groupBy {
    case (w, (p, n)) => w
  } (word, seq((word, (path1, n1)), (word, (path2, n2)), ...))
  .map {
    case (w, seq) =>
      val seq2 = seq map {
        case (_, (p, n)) => (p, n)
      }
      (w, seq2.mkString("", ""))
  } (word, "(path1, n1), (path2, n2), ...")
  .saveAsTextFile(argz.outpath)

  sc.stop()
}
}
```

Tuesday, June 17, 14

Now we do an explicit group by to bring all the same words together. The output will be (word, (word, (path1, n1)), (word, (path2, n2)), ...). The last map removes the redundant “word” values in the sequences of the previous output. It outputs the sequence as a final string of comma-separated (path,n) pairs.

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

```
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
      }
      .map {
        case ((w, p), n) => (w, (p, n))
      }
      .groupByKey {
        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()
  }
}
```

# Altogether



Spark also has a streaming mode for “semi-real time” event handling.

39

Tuesday, June 17, 14

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.

See also today's talk by  
@michaelarmbrust

[scaladays.org/#schedule/  
Catalyst--A-Functional-  
Query-Optimizer-for-Spark-  
and-Shark](https://scaladays.org/#schedule/Catalyst--A-Functional-Query-Optimizer-for-Spark-and-Shark)

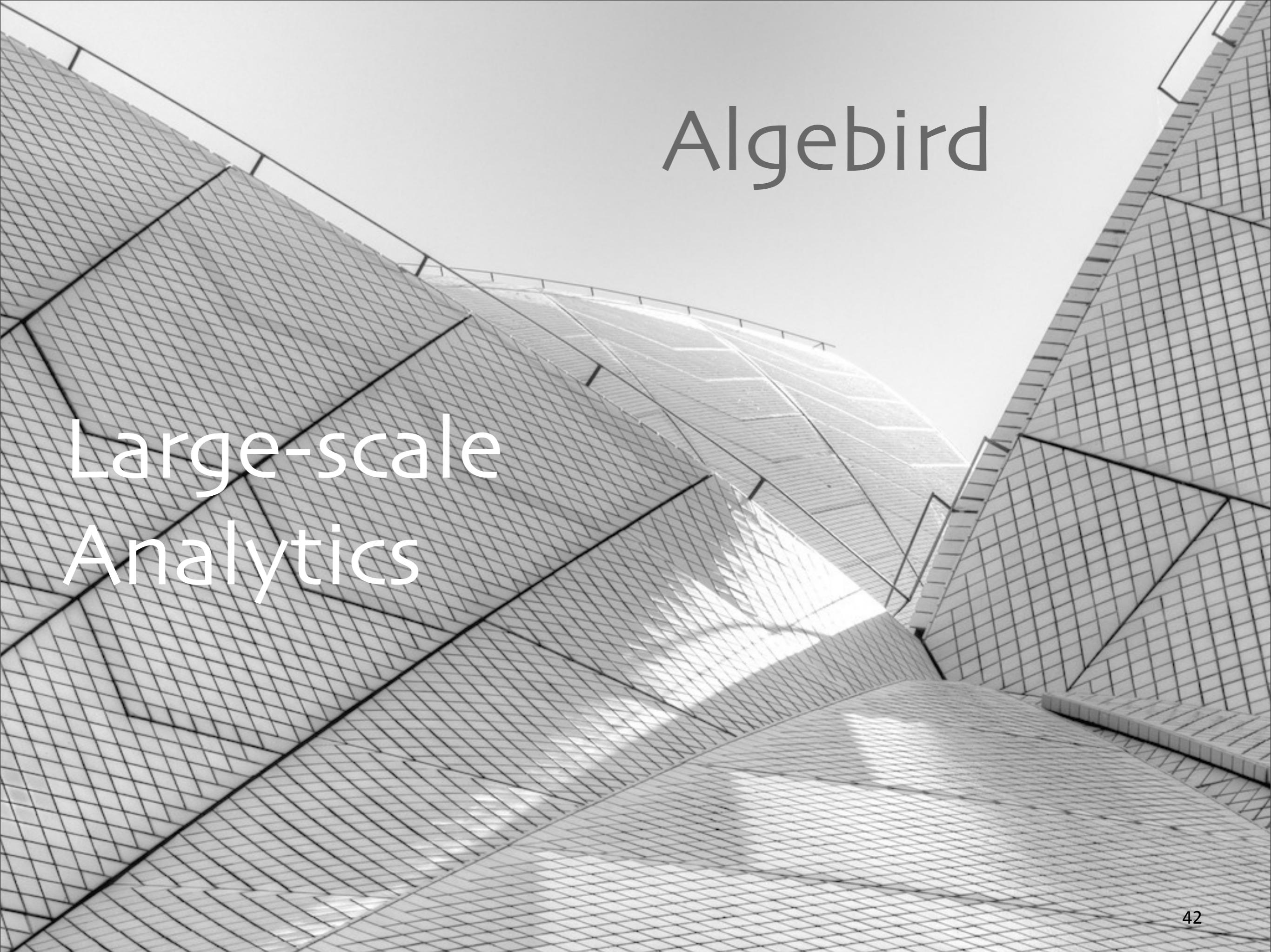
# Scala for Mathematics



41

Tuesday, June 17, 14

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



# Algebird

## Large-scale Analytics

42

- Algebraic types like **Monoids**:
  - A set of elements.
  - An associative binary operation (think “addition”).
  - An identity element.

43

Tuesday, June 17, 14

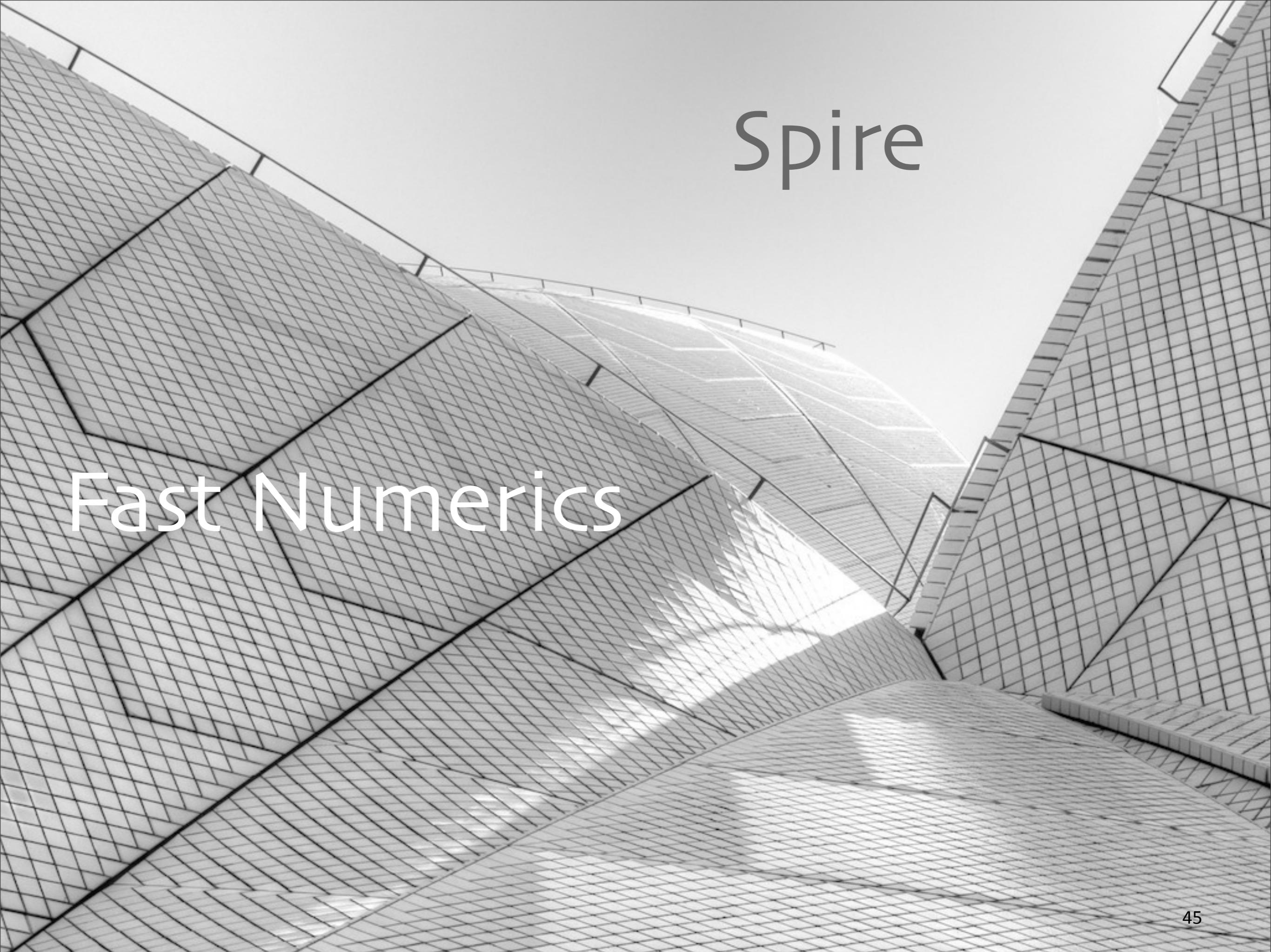
For big data, you’re often willing to trade 100% accuracy for much faster approximations. Algebroid implements many well-known approx. algorithms, all of which can be modeled generically using Monoids or similar data structures.

- Efficient approximation algorithms.
  - “Add All the Things”,  
[infoq.com/presentations/  
abstract-algebra-analytics](http://infoq.com/presentations/abstract-algebra-analytics)

44

Tuesday, June 17, 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.



Spire

Fast Numerics

45

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



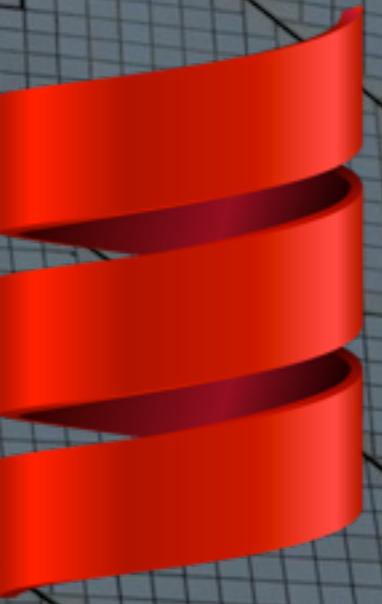
# Winning

47

# 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  
}  
...  
...
```

# The JVM



# Functional Combinators

SQL  
Analog

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

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

50

Tuesday, June 17, 14

For simplicity, assume we only keep the two documents where the word appears most frequently, along with the counts in each doc.  
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 = 'Scala';
```

Restrict

```
inverted_index.filter {  
  case (word, count) =>  
    word == "Scala"  
}
```

# Functional Combinators

```
SELECT word FROM inverted_index;
```

## Projection

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

# Functional Combinators

```
SELECT count, size(word) AS size  
FROM inverted_index  
GROUP BY count1  
ORDER BY size DESC;
```

## Group By and Order By

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

53

# What to Fix?



Tuesday, June 17, 14

What could be improved?

- More hardening of Spark and alternatives.
- Need an “iPython” for Scala.
  - [github.com/Bridgewater/scala-notebook](https://github.com/Bridgewater/scala-notebook)
- Need data visualization tools.

- How to enforce Schema at scale?
  - See what Spark SQL is doing.
- Better primitive operation support for data crunching.
- Better JVM optimizations.

56

Tuesday, June 17, 14

Is it too expensive to create a tuple or case class for each record? It's fine to use these convenient constructs to define record schemas, but at scale, we want to work with arrays of primitives for optimal performance.

Speaking of which, we could use unsigned types and we would like to see the problem of collection specialization solved for good (mini-boxing?), if possible.  
At the JVM level, value types (on stack rather than on heap) and unsigned primitives would help.

[dean.wampler@typesafe.com](mailto:dean.wampler@typesafe.com)  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)  
[@deanwampler](https://twitter.com/deanwampler)

