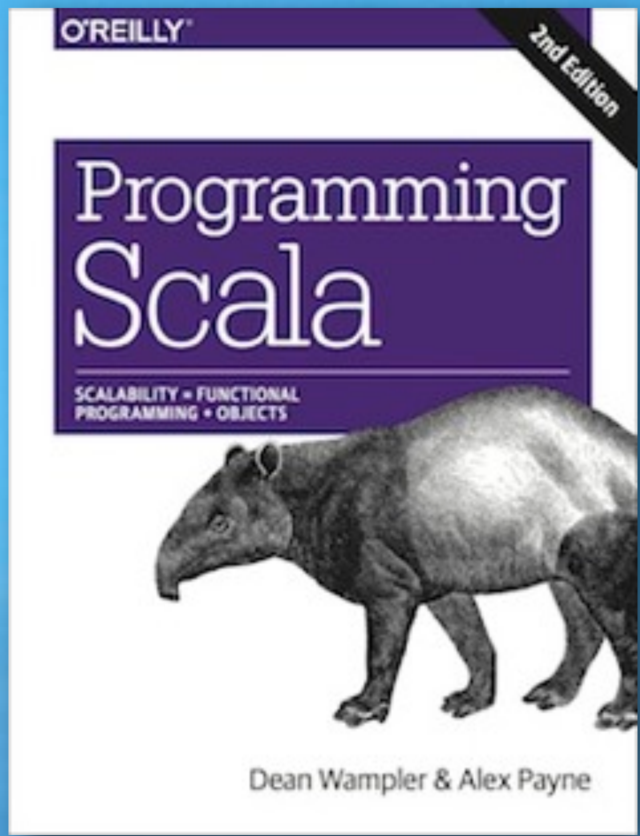
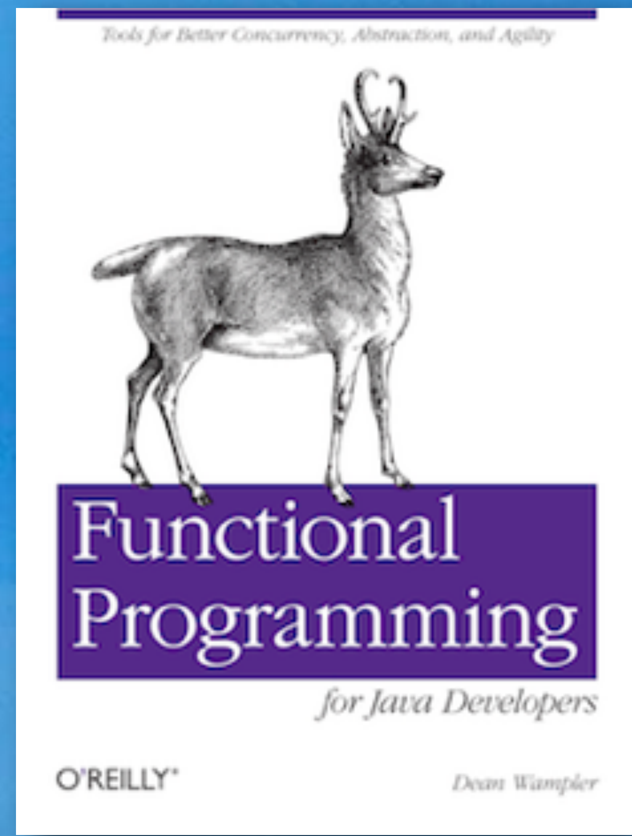
Wednesday, April 23, 14

Image: Detail of the London Eye

Dean Wampler

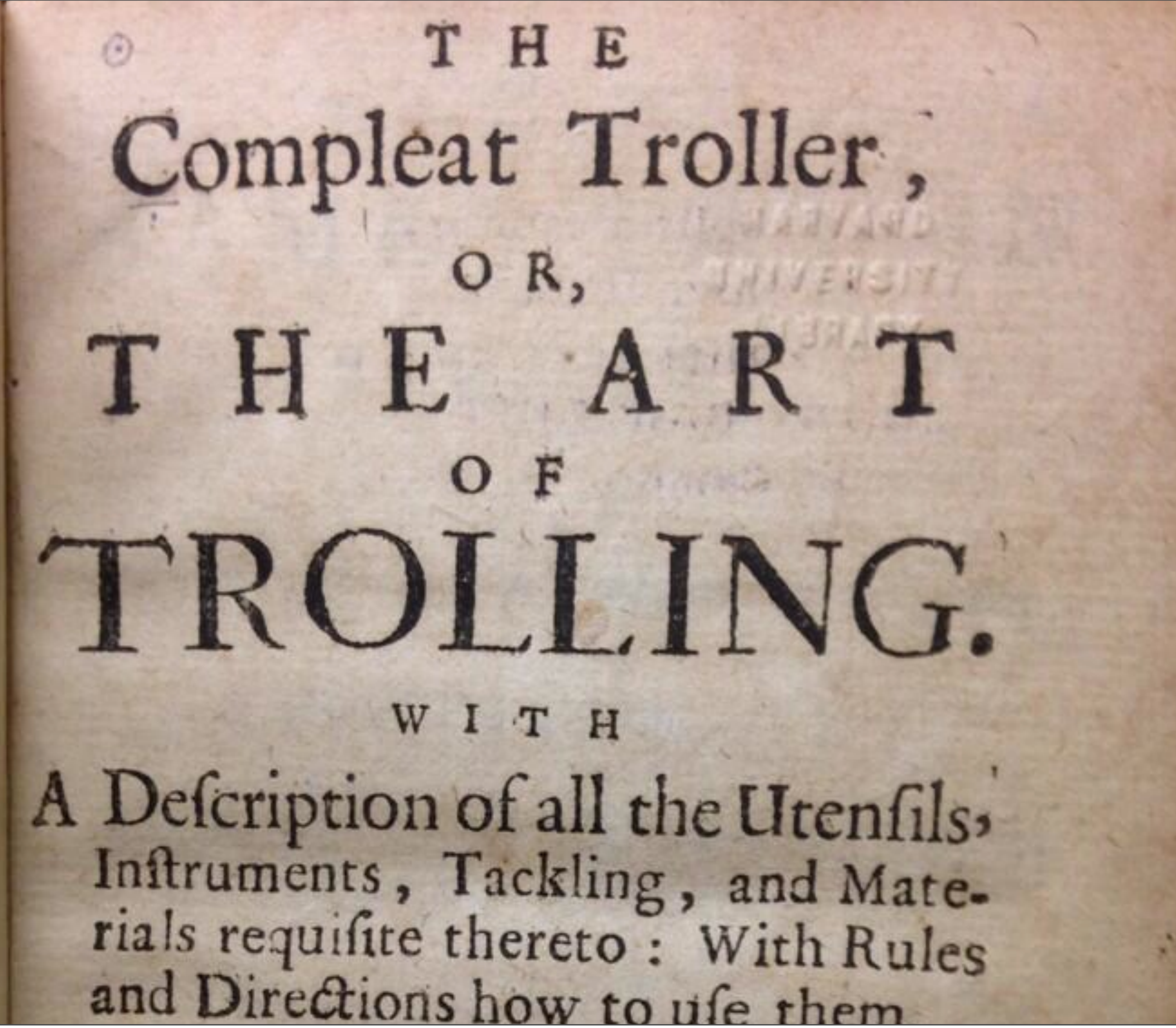




dean.wampler@typesafe.com
polyglotprogramming.com/talks
@deanwampler

About me. You can find this presentation and others on Big Data and Scala at polyglotprogramming.com.
Programming Scala, 2nd Edition is forthcoming.
photo: Dusk at 30,000 ft above the Central Plains of the U.S. on a Winter's Day.

Or this?



THE

Compleat Troller,

OR,

THE ART

OF

TROLLING.

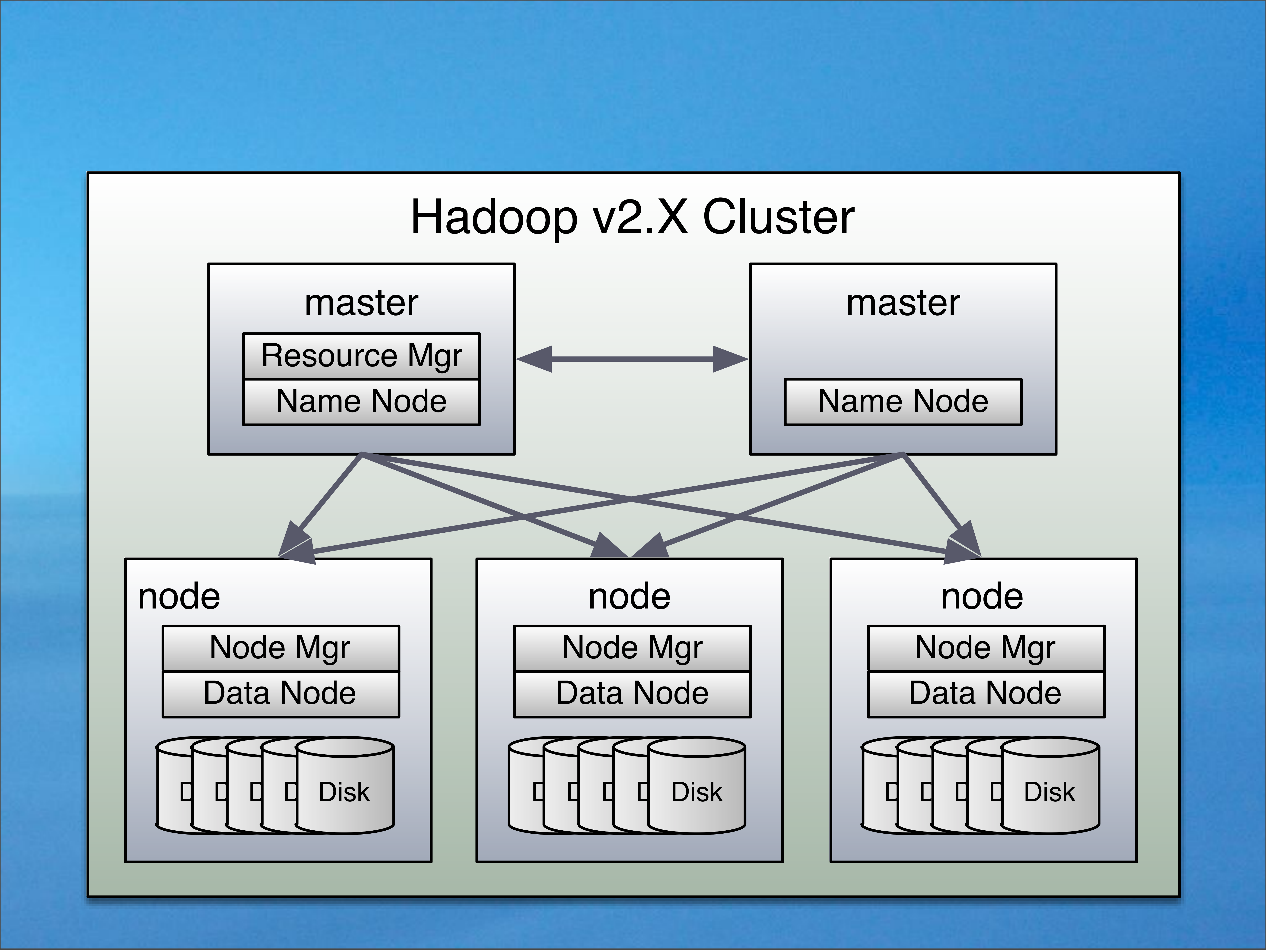
WITH

A Deſcription of all the Utenſils, Inſtruments, Tackling, and Materials requiſite thereto: With Rules and Directions how to uſe them

Wednesday, April 23, 14

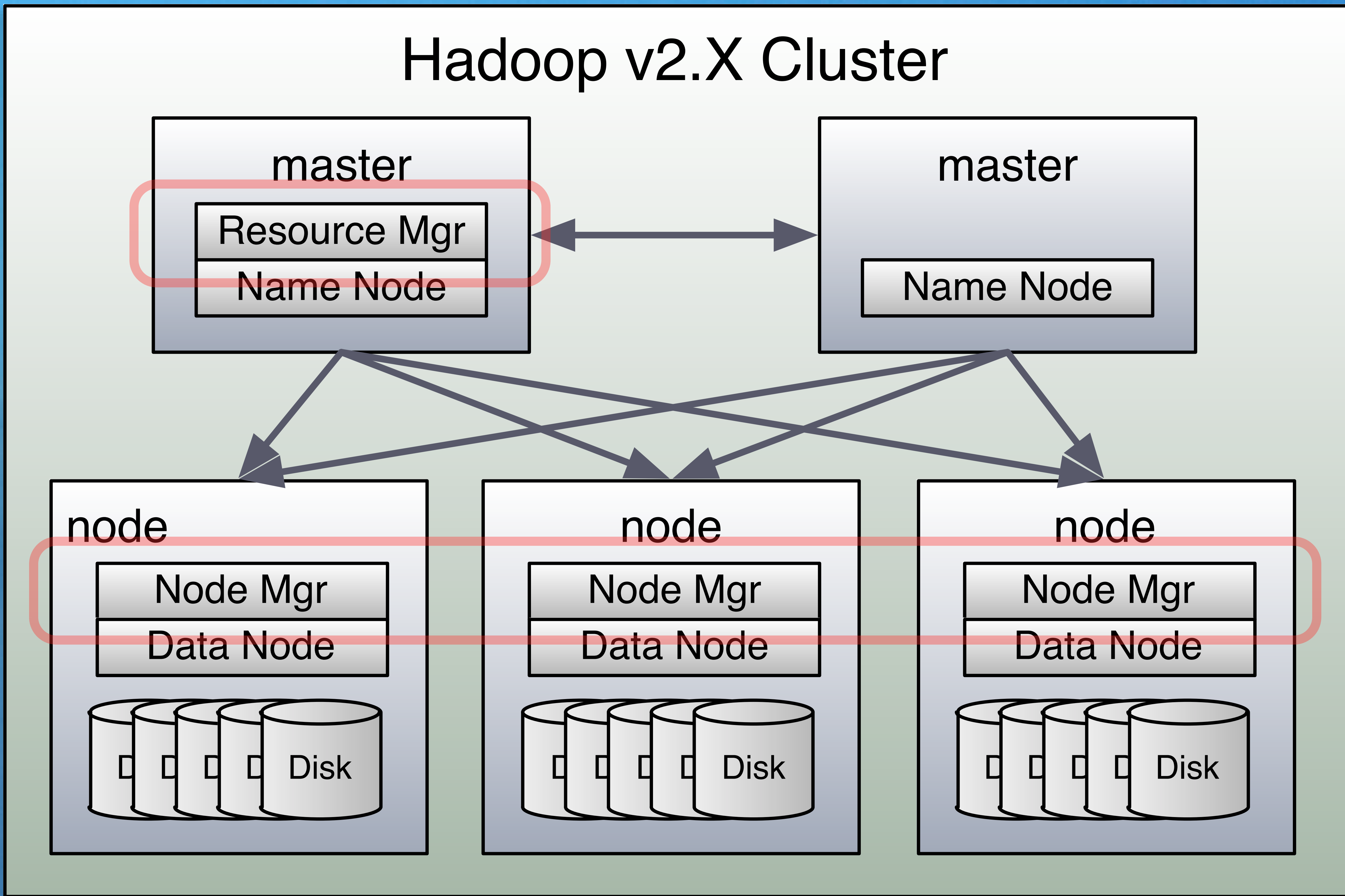
photo: https://twitter.com/john_overholt/status/447431985750106112/photo/1

The state of Hadoop as of last year.
Image: Detail of the London Eye



Schematic view of a Hadoop 2 cluster. For a more precise definition of the services and what they do, see e.g., http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html We aren't interested in great details at this point, but we'll call out a few useful things to know.

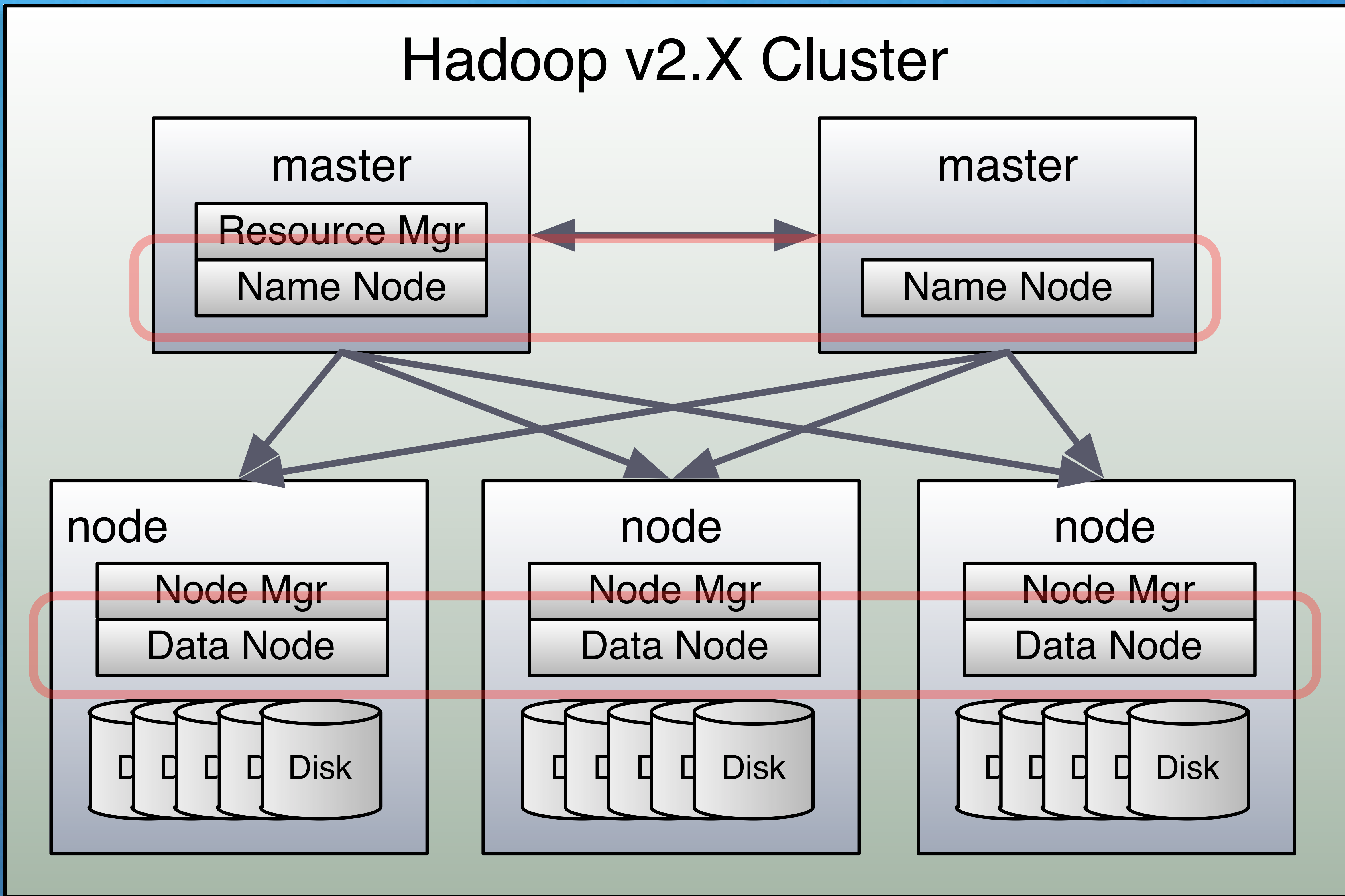
Resource and Node Managers



Wednesday, April 23, 14

Hadoop 2 uses YARN to manage resources via the master Resource Manager, which includes a pluggable job scheduler and an Applications Manager. It coordinates with the Node Manager on each node to schedule jobs and provide resources. Other services involved, including application-specific Containers and Application Masters are not shown.

Name Node and Data Nodes



Wednesday, April 23, 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**

Wednesday, April 23, 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

1 **map** step + 1 **reduce** step
(wash, rinse, repeat)

Wednesday, April 23, 14

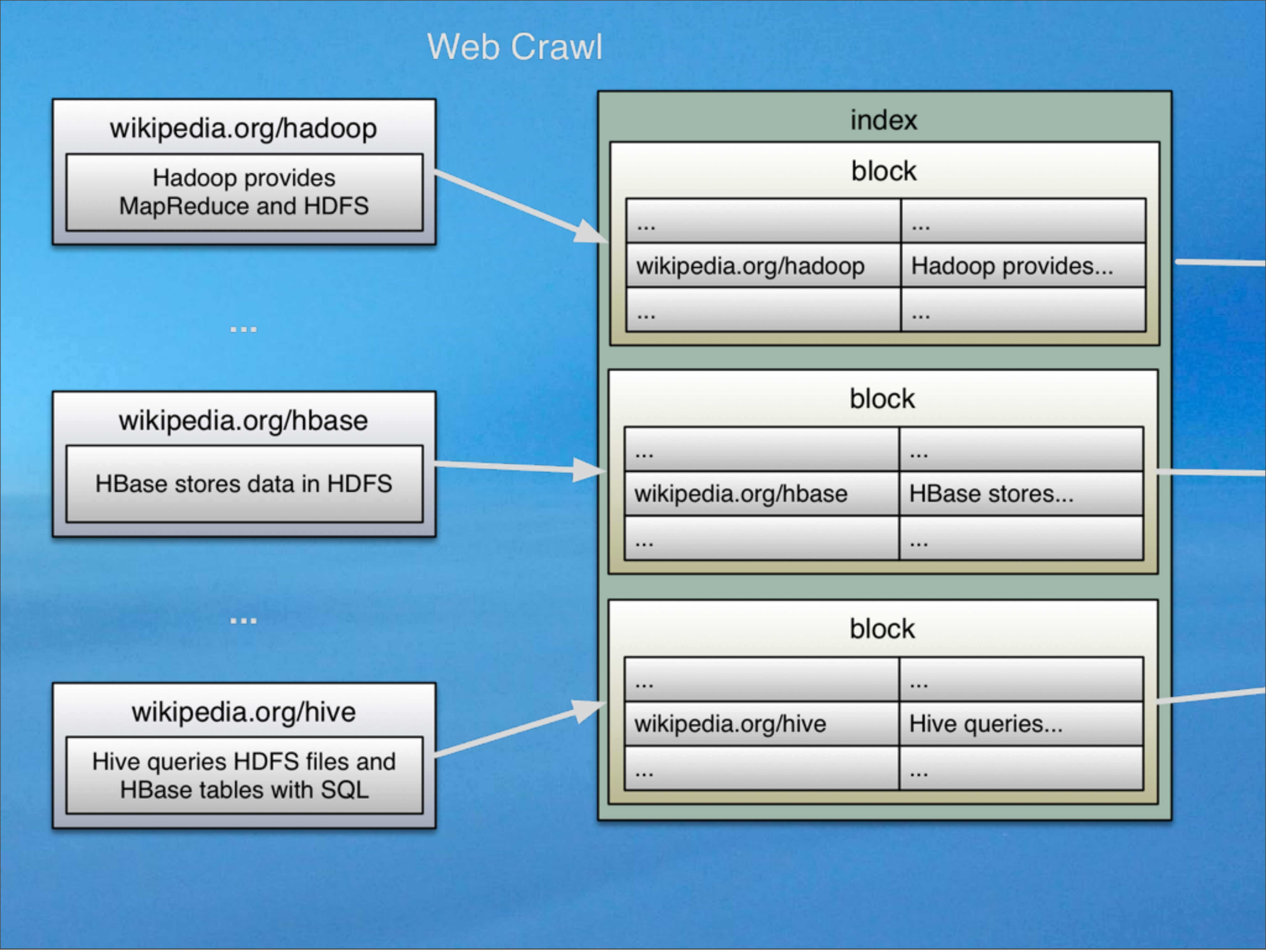
You get 1 map step (although there is limited support for chaining mappers) and 1 reduce step. If you can't implement an algorithm in these two steps, you can chain jobs together, but you'll pay a tax of flushing the entire data set to disk between these jobs.

MapReduce

Example:

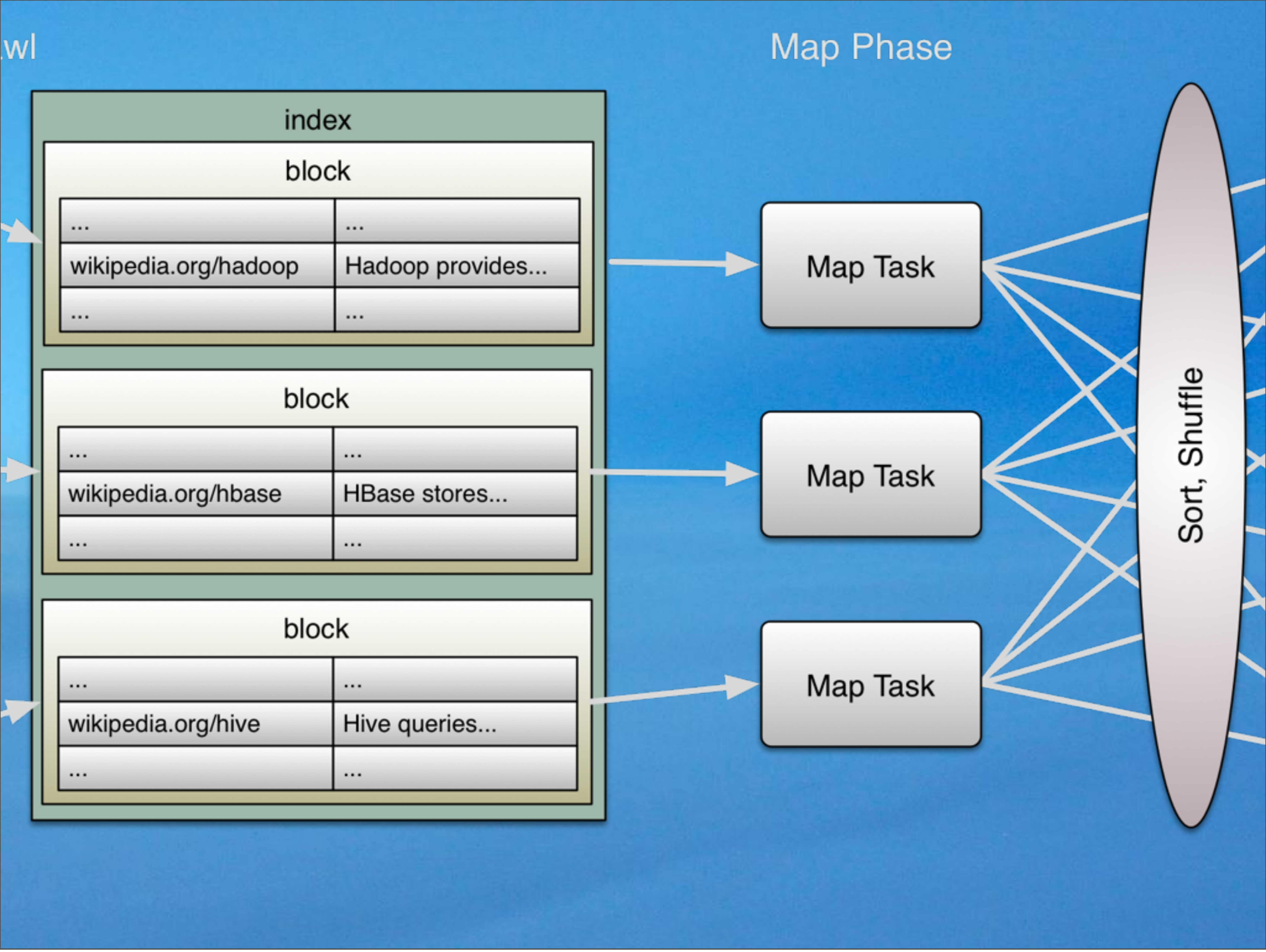
Inverted Index

The inverted index is a classic algorithm needed for building search engines.



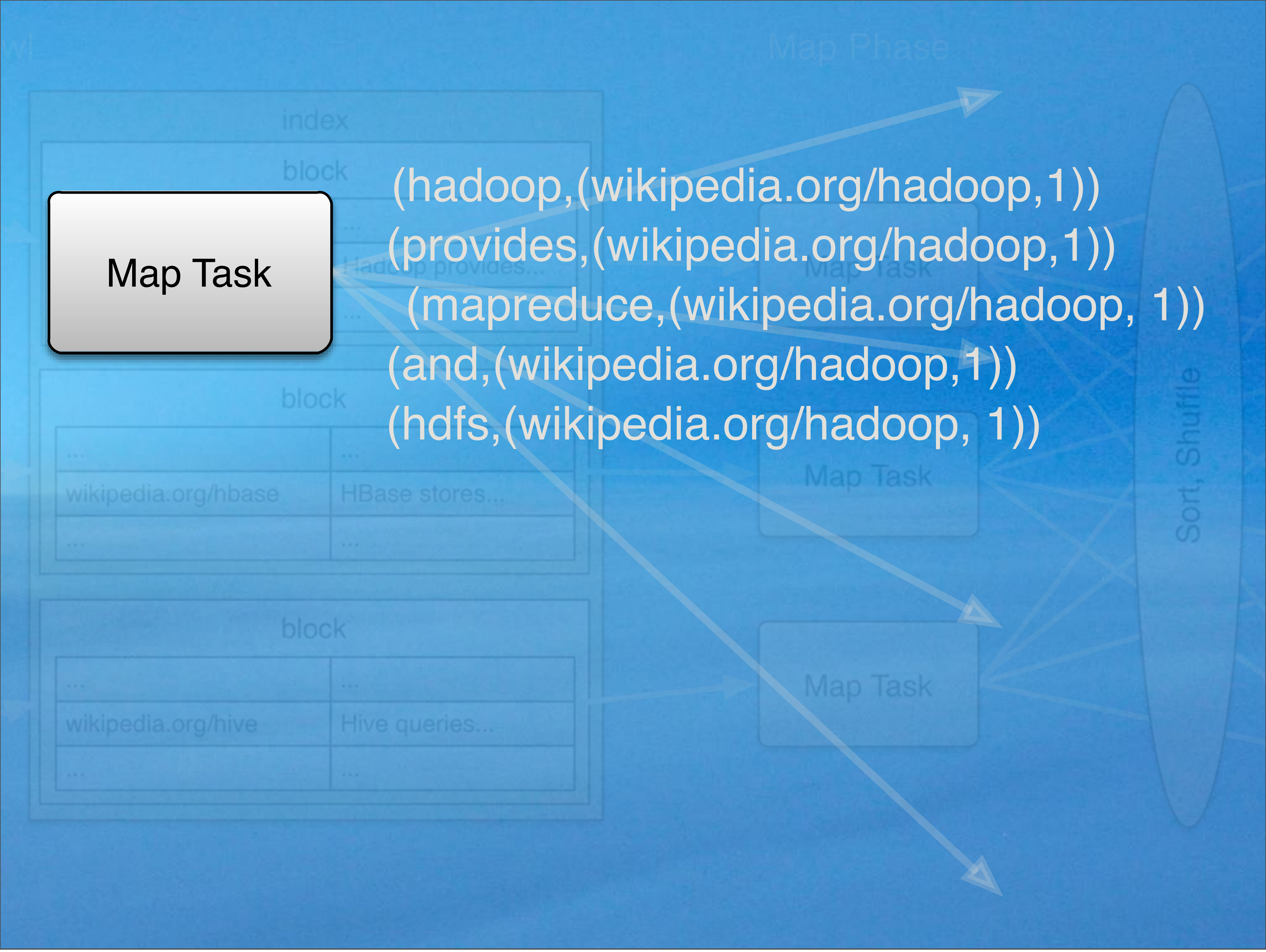
Wednesday, April 23, 14

Before running MapReduce, crawl teh interwebs, find all the pages, and build a data set of URLs -> doc contents, written to flat files in HDFS or one of the more “sophisticated” formats.



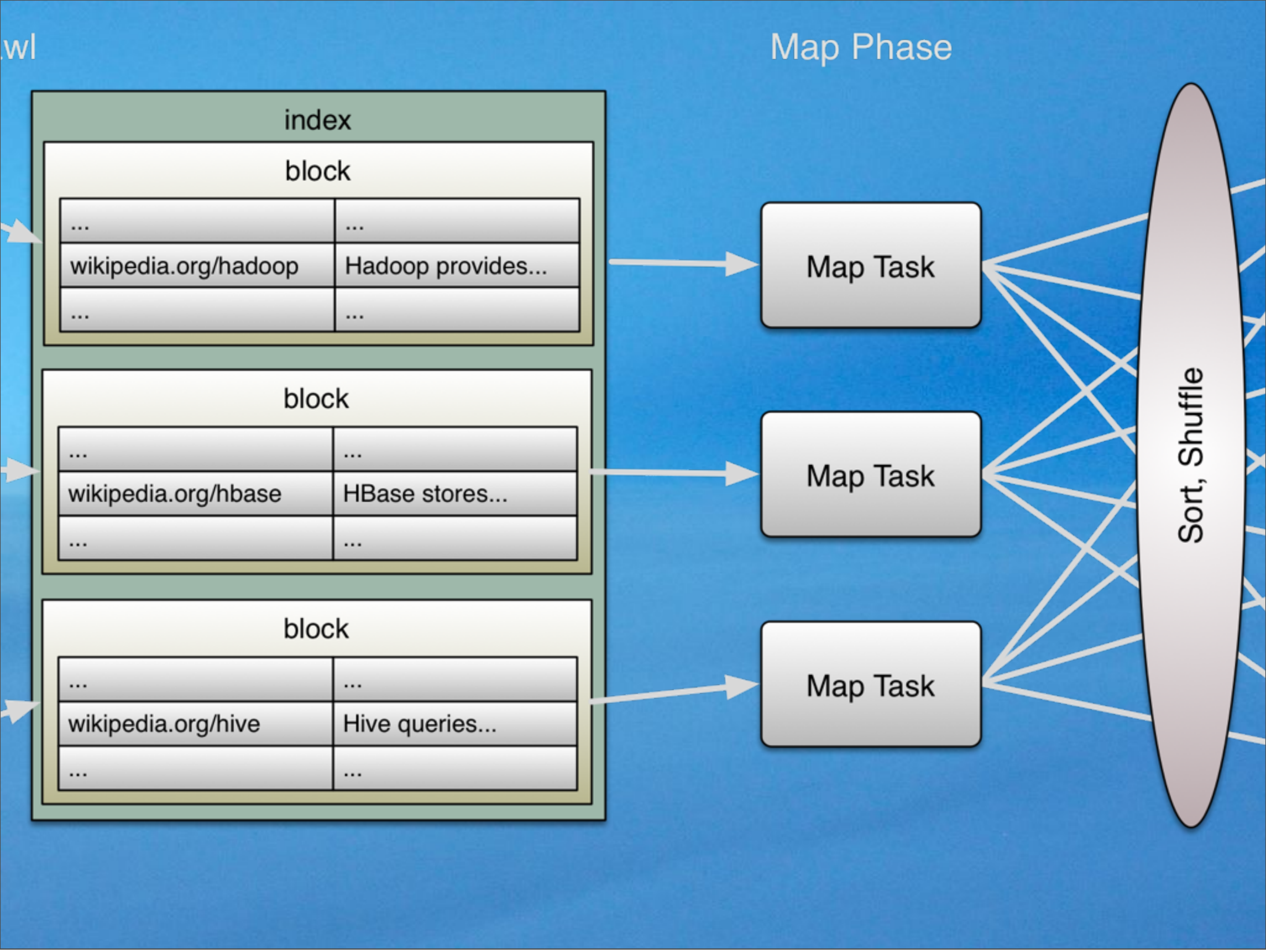
Wednesday, April 23, 14

Now we're running MapReduce. In the map step, a task (JVM process) per file *block* (64MB or larger) reads the rows, tokenizes the text and outputs key-value pairs ("tuples")...

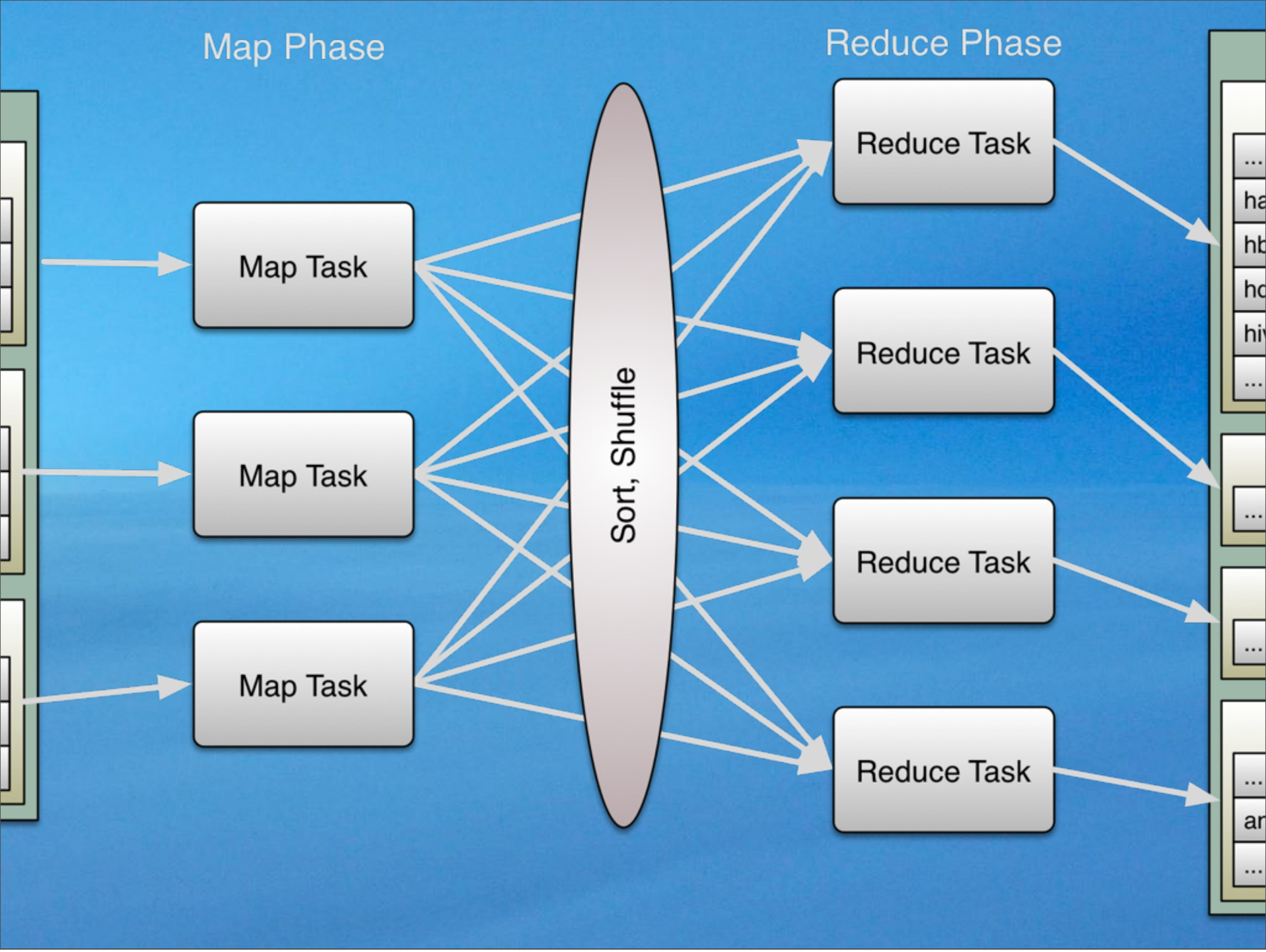


Wednesday, April 23, 14

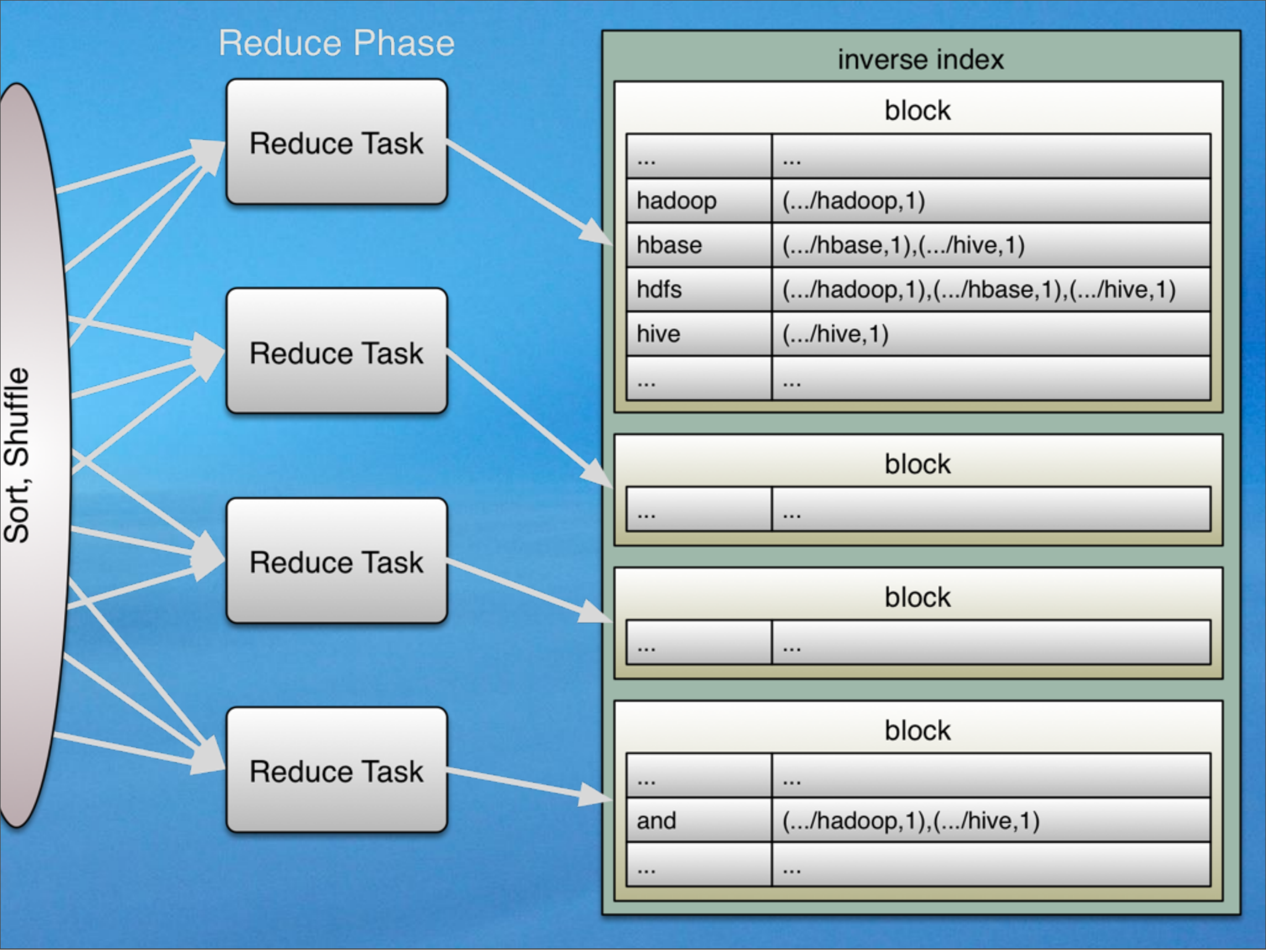
... the keys are each word found and the values are themselves tuples, each URL and the count of the word. In our simplified example, there are typically only single occurrences of each work in each document. The real occurrences are interesting because if a word is mentioned a lot in a document, the chances are higher that you would want to find that document in a search for that word.



.wl
Map Phase
index
block
...
...
wikipedia.org/hadoop
Hadoop provides...
...
...
block
...
...
wikipedia.org/hbase
HBase stores...
...
...
block
...
...
wikipedia.org/hive
Hive queries...
...
...
Map Task
Map Task
Map Task
Sort, Shuffle

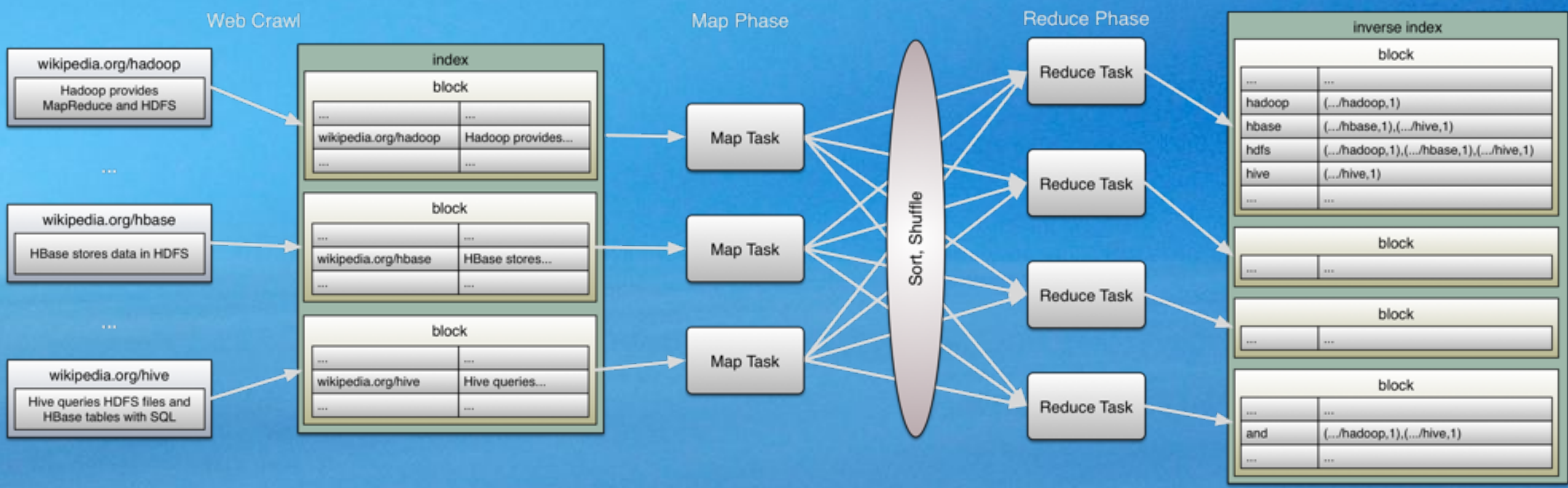


The output tuples are sorted by key locally in each map task, then “shuffled” over the cluster network to reduce tasks (each a JVM process, too), where we want all occurrences of a given key to land on the same reduce task.



Finally, each reducer just aggregates all the values it receives for each key, then writes out new files to HDFS with the words and a list of (URL-count) tuples (pairs).

Altogether...



Wednesday, April 23, 14

Finally, each reducer just aggregates all the values it receives for each key, then writes out new files to HDFS with the words and a list of (URL-count) tuples (pairs).

Wednesday, April 23, 14

This seems okay, right? What's wrong with it?

Awkward

Most algorithms are much harder to implement in this restrictive map-then-reduce model.

Wednesday, April 23, 14

Writing MapReduce jobs requires arcane, specialized skills that few master. For a good overview, see http://lintool.github.io/MapReduceAlgorithms/.

Awkward

Lack of flexibility inhibits optimizations, too.

The inflexible compute model leads to complex code to improve performance where hacks are used to work around the limitations. Hence, optimizations are hard to implement. The Spark team has commented on this, see http://databricks.com/blog/2014/03/26/Spark-SQL-manipulating-structured-data-using-Spark.html

Performance

Full dump to disk between jobs.

Wednesday, April 23, 14

Sequencing jobs wouldn't be so bad if the "system" was smart enough to cache data in memory. Instead, each job dumps everything to disk, then the next job reads it back in again. This makes iterative algorithms particularly painful.

Enter Spark

spark.apache.org

Cluster Computing

Can be run in:

- YARN (Hadoop 2)
- Mesos (Cluster management)
- EC2
- Standalone (“local” mode)

Wednesday, April 23, 14

Once you learn the core set of primitives, it’s easy to compose non-trivial algorithms with little code.

Compute Model

Fine-grained "combinators" for composing algorithms.

Spark

Once you learn the core set of primitives, it's easy to compose non-trivial algorithms with little code.

Compute Model

RDDs:
Resilient,
Distributed
Datasets

Wednesday, April 23, 14

RDDs shard the data over a cluster, like a virtualized, distributed collection (analogous to HDFS). They support intelligent caching, which means no naive flushes of massive datasets to disk. This feature alone allows Spark jobs to run 10-100x faster than comparable MapReduce jobs! The “resilient” part means they will reconstitute shards lost due to process/server crashes.

Compute Model

Written in **Scala**,
with Java and Python APIs.

Wednesday, April 23, 14

Once you learn the core set of primitives, it's easy to compose non-trivial algorithms with little code.

Wednesday, April 23, 14

Let's see an an actual implementation of the inverted index. First, a Hadoop MapReduce (Java) version, adapted from https://developer.yahoo.com/hadoop/tutorial/module4.html#solution It's about 90 lines of code, but I reformatted to fit better.
This is also a slightly simpler version that the one I diagrammed. It doesn't record a count of each word in a document; it just writes (word,doc-title) pairs out of the mappers and the final (word,list) output by the reducers just has a list of documentations, hence repeats.
A second job would be necessary to count the repeats.

```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);
```


I've shortened the original code a bit, e.g., using * import statements instead of separate imports for each class.

I'm not going to explain every line ... nor most lines.

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

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

```java
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...

```
      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> {
```


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 =
```

Wednesday, April 23, 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);
    }
  }
}
```


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.

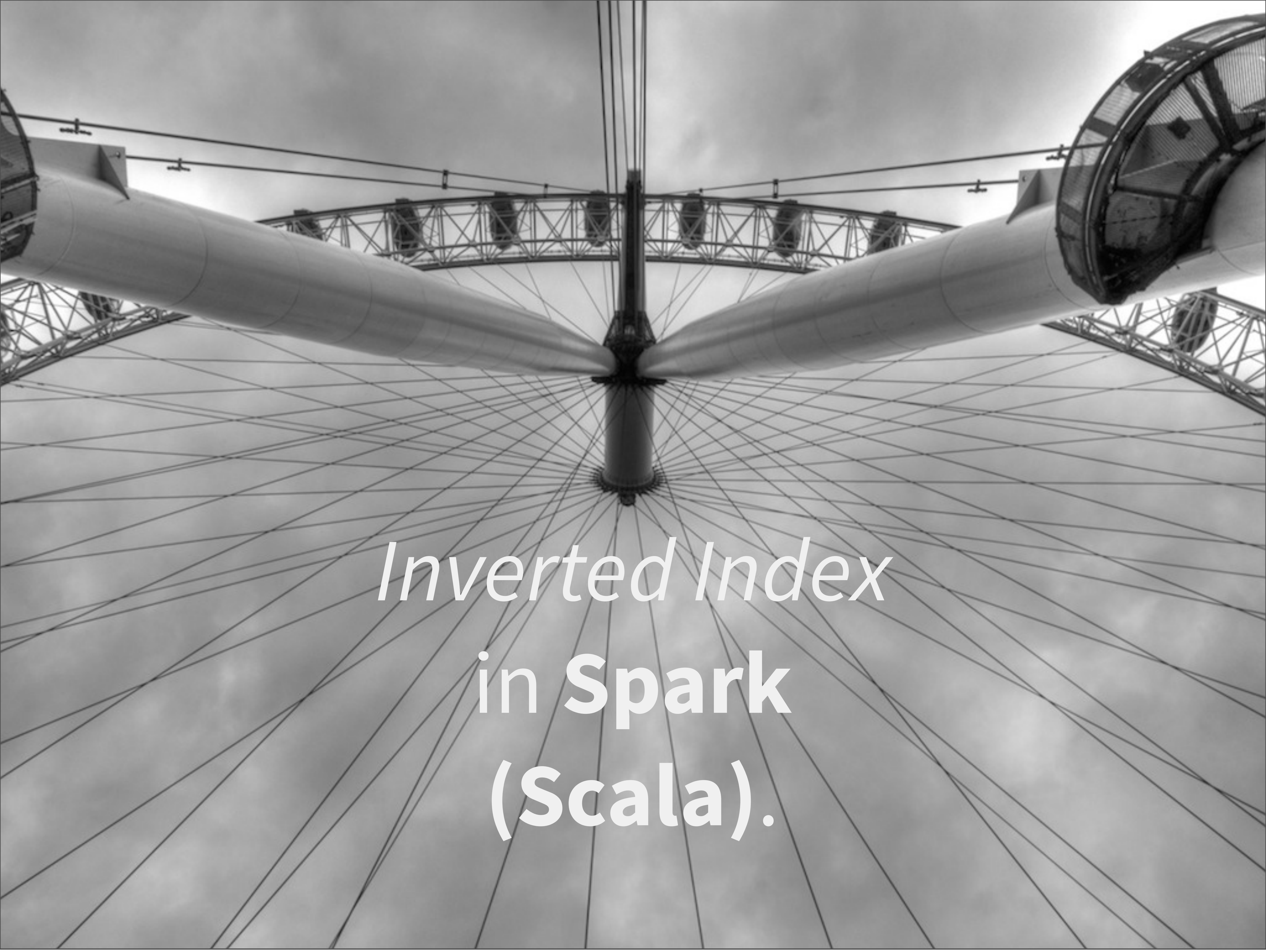
```java
    boolean first = true;
    StringBuilder toReturn =
     new StringBuilder();
    while (values.hasNext()) {
     if (!first)
      toReturn.append(", ");
     first=false;
     toReturn.append(
      values.next().toString());
    }
    output.collect(key,
     new Text(toReturn.toString()));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


EOF

Altogether

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


The whole shebang (6pt. font)



Wednesday, April 23, 14

This code is approximately 45 lines, but it does more than the previous Java example, it implements the original inverted index algorithm I diagrammed where word counts are computed and included in the data.

```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) =>
      text.split("""\W+""").map {
```

Wednesday, April 23, 14

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
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)
}
```


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.

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 {
```

Wednesday, April 23, 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”.

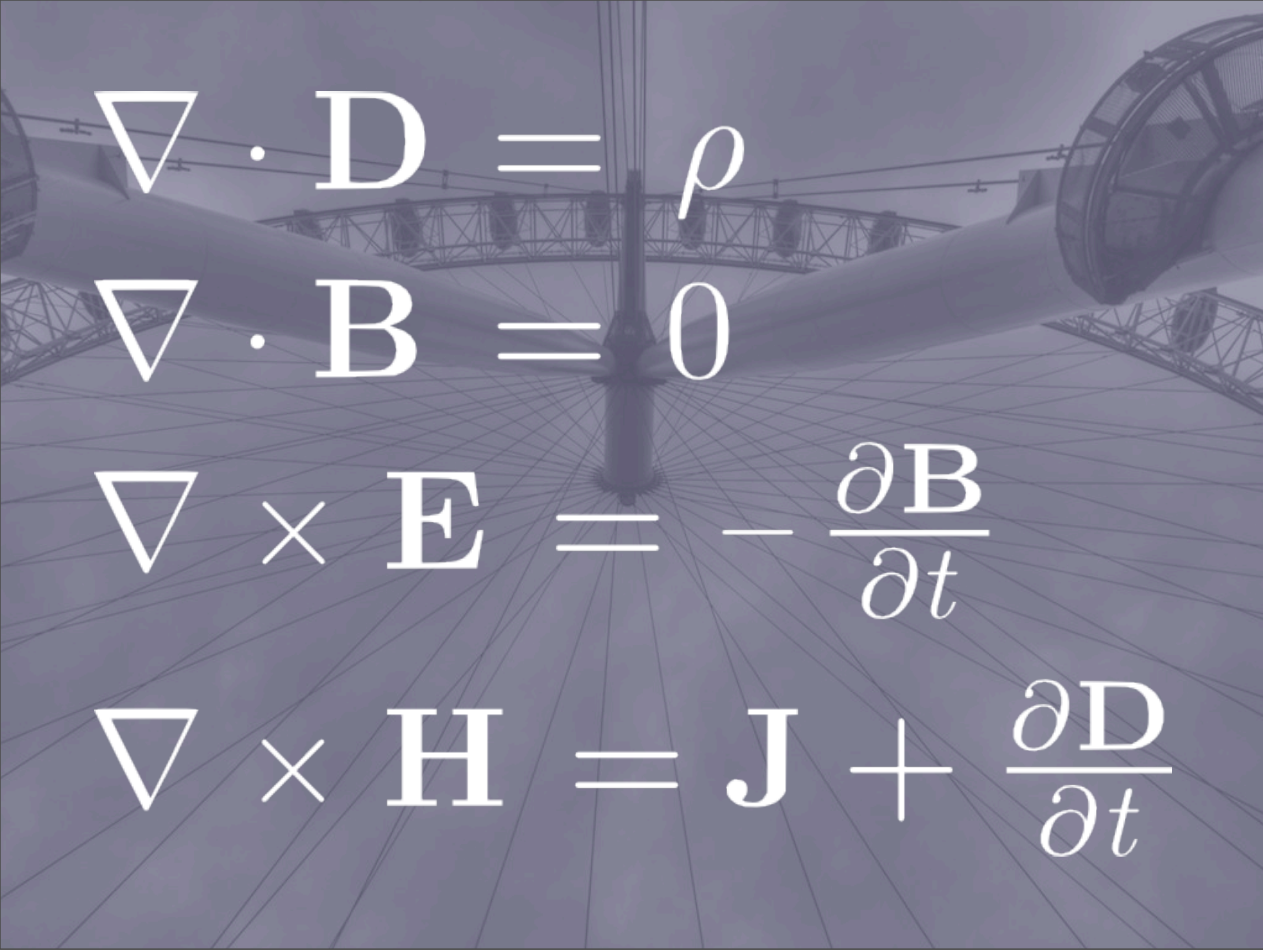
```
}
.flatMap {
  case (path, text) =>
   text.split("""\W+""") map {
    word => (word, path)
   }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
```

Beautiful, no?

Wednesday, April 23, 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.



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(", "))
  }
```

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

Wednesday, April 23, 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 >= 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()
  }
}
```


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.

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
   }
   .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(", "))
   }
   .saveAsTextFile(argz.outpath)

   sc.stop()
 }
}
```


The whole shebang (12pt. font, this time)

```
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
}
```

Powerful Combinators!

Wednesday, April 23, 14

I’ve shortened the original code a bit, e.g., using * import statements instead of separate imports for each class.
I’m not going to explain every line ... nor most lines.
Everything is in one outer class. We start with a main routine that sets up the job. Lotta boilerplate...

Once you learn the core primitives I used, and a few tricks for manipulating the RDD tuples, you can very quickly build complex algorithms for data processing!

The Spark API allowed us to focus almost exclusively on the “domain” of data transformations, while the Java MapReduce version (which does less), forced tedious attention to infrastructure mechanics.

SQL!

Shark:
Hive (SQL query tool)
ported to Spark.

Wednesday, April 23, 14

Use SQL when you can!


Use a SQL query when
you can!!

```
CREATE EXTERNAL TABLE stocks (
  symbol         STRING,
  ymd            STRING,
  price_open     STRING,
  price_close    STRING,
  shares_traded INT)
LOCATION “hdfs://data/stocks”;

-- Year-over-year average closing price.
SELECT year(ymd), avg(price_close)
FROM stocks
WHERE symbol = ‘AAPL’
GROUP BY year(ymd);
```

Hive Query Language.

Hive and the Spark port, Shark, let you use SQL to query and manipulate structured and semistructured data. By default, tab-delimited text files will be assumed for the files found in “data/stocks”. Alternatives can be configured as part of the table metadata.

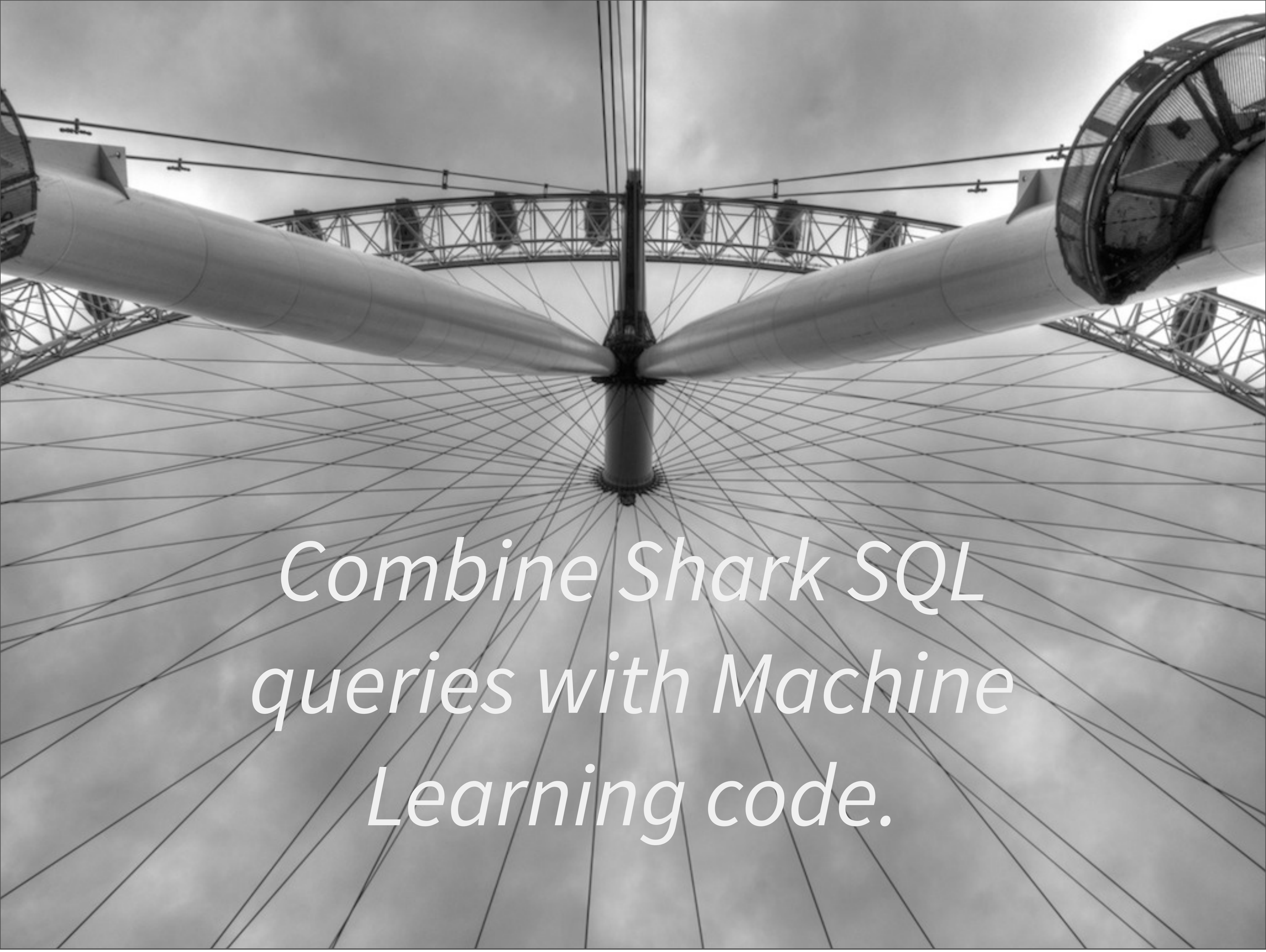
Shark

~10-100x the performance of Hive, due to in-memory caching of RDDs & better Spark abstractions.

Did I mention SQL?

Spark SQL:
Next generation
SQL query tool/API.

A new query optimizer. It will become the basis for Shark internals, replacing messing Hive code that is hard to reason about, extend, and debug.



Wednesday, April 23, 14

We'll use the Spark "MLlib" in the example, then return to it in a moment.

```sql
CREATE TABLE Users(
 userId      INT,
 name        STRING,
 email       STRING,
 age         INT,
 latitude    DOUBLE,
 longitude   DOUBLE,
 subscribed BOOLEAN);

CREATE TABLE Events(
 userId INT,
 action INT);
```

Hive/Shark table definitions (not Scala).

This example adapted from the following blog post announcing Spark SQL:
http://databricks.com/blog/2014/03/26/Spark-SQL-manipulating-structured-data-using-Spark.html

Assume we have these Hive/Shark tables, with data about our users and events that have occurred.

```scala
val trainingDataTable = sql("""
 SELECT e.action, u.age,
        u.latitude, u.longitude
 FROM Users u
 JOIN Events e
 ON u.userId = e.userId""")

val trainingData =
 trainingDataTable map { row =>
  val features =
   Array[Double](row(1), row(2), row(3))
  LabeledPoint(row(0), features)
 }

val model =
 new LogisticRegressionWithSGD()
```

Spark Scala.

Wednesday, April 23, 14

Here is some Spark (Scala) code with an embedded SQL/Shark query that joins the Users and Events tables. The “””...””” string allows embedded line feeds.

The “sql” function returns an RDD, which we then map over to create LabeledPoints, an object used in Spark’s MLlib (machine learning library) for a recommendation engine. The “label” is the kind of event and the user’s age and lat/long coordinates are the “features” used for making recommendations. (E.g., if you’re 25 and near a certain location in the city, you might be interested a nightclub near by...)

```scala
val model =
 new LogisticRegressionWithSGD()
 .run(trainingData)

val allCandidates = sql("""
 SELECT userId, age, latitude, longitude
 FROM Users
 WHERE subscribed = FALSE""")

case class Score(
  userId: Int, score: Double)
val scores = allCandidates map { row =>
 val features =
  Array[Double](row(1), row(2), row(3))
 Score(row(0), model.predict(features))
}
```


Next we train the recommendation engine, using a "logistic regression" fit to the training data, where "stochastic gradient descent" (SGD) is used to train it. (This is a standard tool set for recommendation engines; see for example: http://www.cs.cmu.edu/~wcohen/10-605/assignments/sgd.pdf)

```scala
  .run(trainingData)

val allCandidates = sql("""
 SELECT userId, age, latitude, longitude
 FROM Users
 WHERE subscribed = FALSE""")

case class Score(
  userId: Int, score: Double)
val scores = allCandidates map { row =>
 val features =
  Array[Double](row(1), row(2), row(3))
 Score(row(0), model.predict(features))
}

// Hive table
scores.registerAsTable("Scores")
```


Now run a query against all users who aren't already subscribed to notifications.

```
case class Score(
  userId: Int, score: Double)
val scores = allCandidates map { row =>
 val features =
  Array[Double](row(1), row(2), row(3))
 Score(row(0), model.predict(features))
}

// Hive table
scores.registerAsTable("Scores")

val topCandidates = sql("""
 SELECT u.name, u.email
 FROM Scores s
 JOIN Users u ON s.userId = u.userId
 ORDER BY score DESC
```


Declare a class to hold each user’s “score” as produced by the recommendation engine and map the “all” query results to Scores.
Then “register” the scores RDD as a “Scores” table in Hive’s metadata respository. This is equivalent to running a “CREATE TABLE Scores ...” command at the Hive/Shark prompt!

```
// Hive table
scores.registerAsTable("Scores")

val topCandidates = sql("""
 SELECT u.name, u.email
 FROM Scores s
 JOIN Users u ON s.userId = u.userId
 ORDER BY score DESC
 LIMIT 100""")
```

Wednesday, April 23, 14

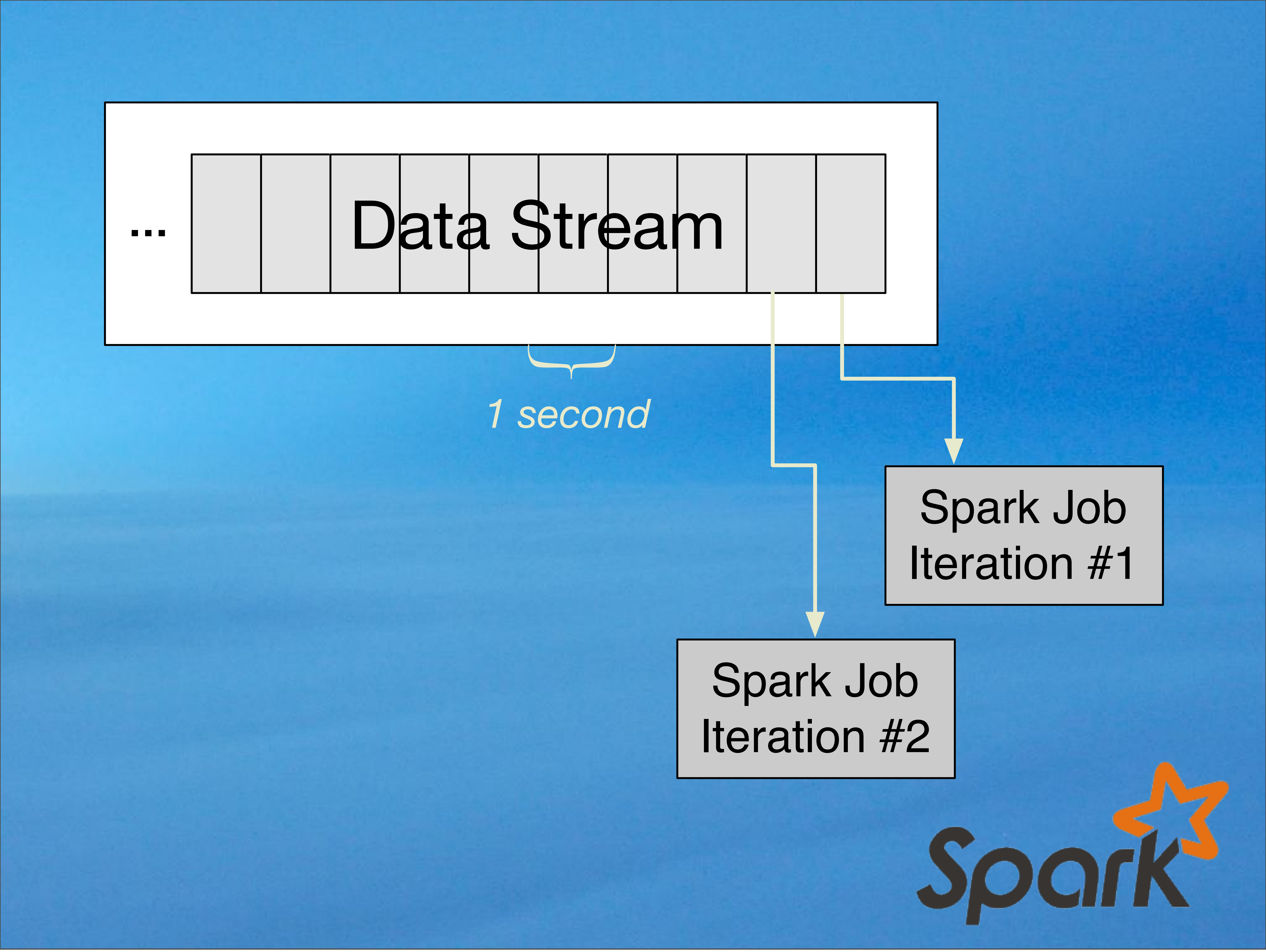
Finally, run a new query to find the people with the highest scores that aren't already subscribing to notifications. You might send them an email next recommending that they subscribe...

Cluster Computing

Spark Streaming:
Use the same abstractions for real-time, event streaming.

Wednesday, April 23, 14

Once you learn the core set of primitives, it's easy to compose non-trivial algorithms with little code.



Wednesday, April 23, 14

You can specify the granularity, such as all events in 1 second windows, then your Spark job is patched each window of data for processing.


Very similar code...

```scala
val sc  = new SparkContext(...)
val ssc = new StreamingContext(
            sc, Seconds(1))

// A DStream that will listen to server:port
val lines =
 ssc.socketTextStream(server, port)

// Word Count...
val words = lines flatMap {
 line => line.split("""\W+""")
}

val pairs = words map (word => (word, 1))
val wordCounts =
 pairs reduceByKey ((n1, n2) => n1 + n2)
```


This example adapted from the following page on the Spark website:
http://spark.apache.org/docs/0.9.0/streaming-programming-guide.html#a-quick-example

We create a StreamingContext that wraps a SparkContext (there are alternative ways to construct it...). It will “clump” the events into 1-second intervals. Next we setup a socket to stream text to us from another server and port (one of several ways to ingest data).

```
  ssc.socketTextStream(server, port)

// Word Count...
val words = lines flatMap {
  line => line.split("""\W+""")
}

val pairs = words map (word => (word, 1))
val wordCounts =
  pairs reduceByKey ((n1, n2) => n1 + n2)

wordCount.print()  // print a few counts...

ssc.start()
ssc.awaitTermination()
```


Now the “word count” happens over each interval (aggregation across intervals is also possible), but otherwise it works like before.

Once we setup the flow, we start it and wait for it to terminate through some means, such as the server socket closing.

Cluster Computing

MLlib:

Machine learning library.

Wednesday, April 23, 14

Spark implements many machine learning algorithms, although a lot more or needed, compared to more mature tools like Mahout and libraries in Python and R.

MLlib

- Linear regression
- Binary classification
- Collaborative filtering
- Clustering
- Others...

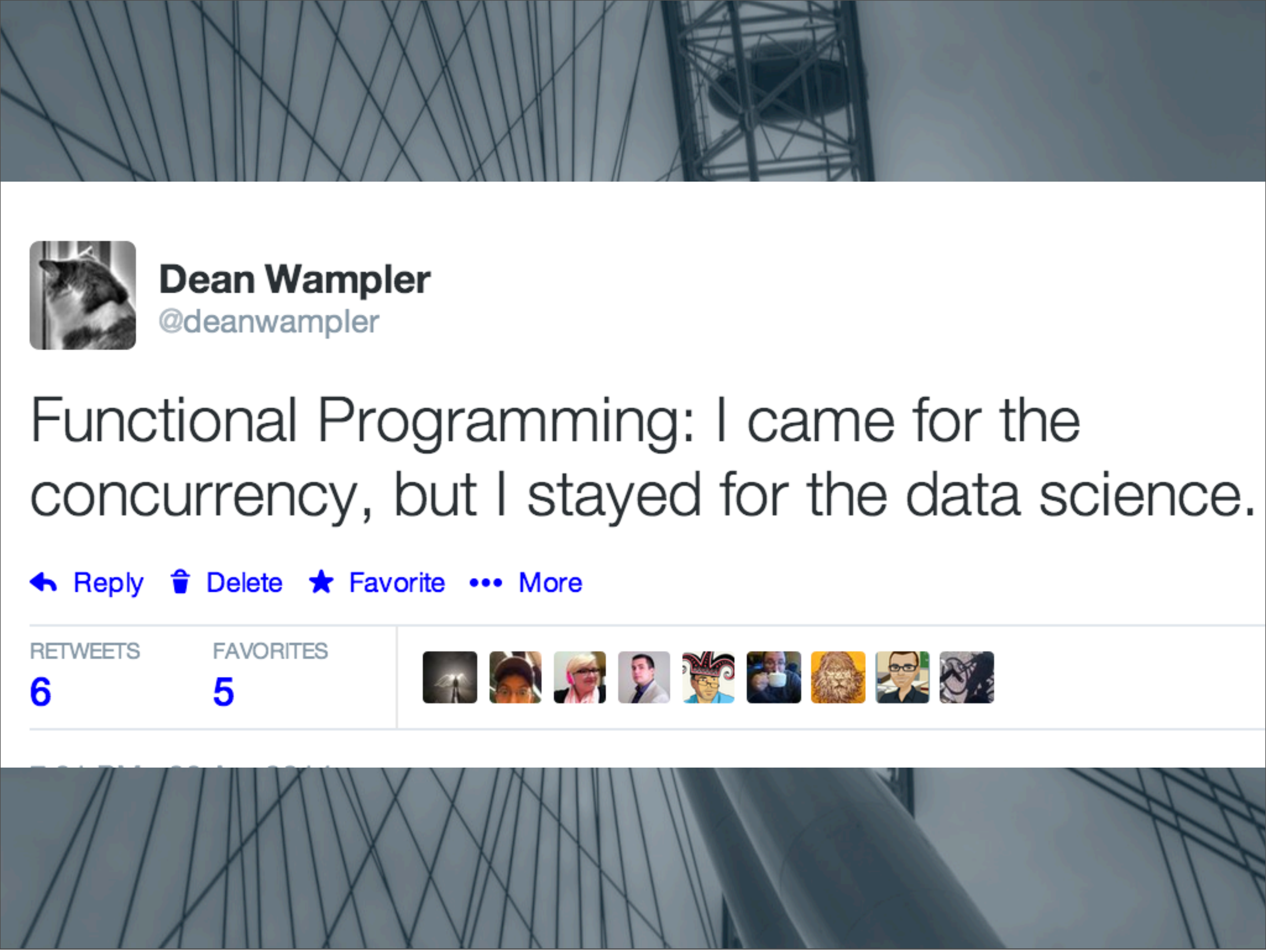
Not as full-featured as more mature toolkits, but the Mahout project has announced they are going to port their algorithms to Spark, which include powerful Mathematics, e.g., Matrix support libraries.

Cluster Computing

GraphX:
Graphical models
and algorithms.

Wednesday, April 23, 14

Some problems are more naturally represented as graphs.
Extends RDDs to support property graphs with directed edges.



Wednesday, April 23, 14

Why is Spark so good (and Java MapReduce so bad)? Because fundamentally, data analytics is Mathematics and programming tools inspired by Mathematics - like Functional Programming - are ideal tools for working with data. This is why Spark code is so concise, yet powerful. This is why it is a great platform for performance optimizations. This is why Spark is a great platform for higher-level tools, like SQL, graphs, etc. Interest in FP started growing ~10 years ago as a tool to attack concurrency. I believe that data is now driving FP adoption even faster. I know many Java shops that switched to Scala when they adopted tools like Spark and Scalding (https://github.com/twitter/scalding).

Spark

A flexible, scalable distributed compute platform with concise, powerful APIs and higher-order tools.

spark.apache.org

Wednesday, April 23, 14

Image: The London Eye on one side of the Thames, Parliament on the other.