

A photograph of the London Eye, showing several of its glass-enclosed pods hanging from a large metal framework against a bright sky.

# Why Spark Is the Next Top (Compute) Model



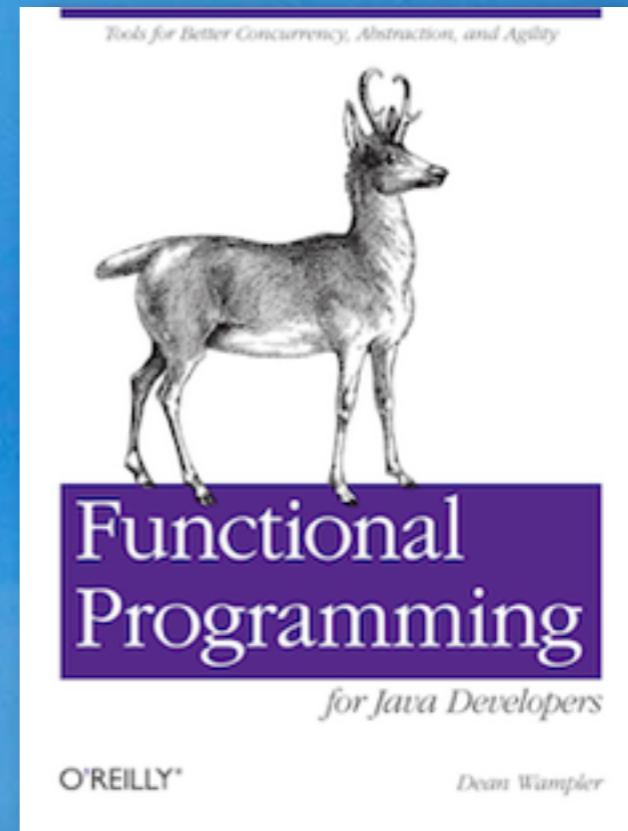
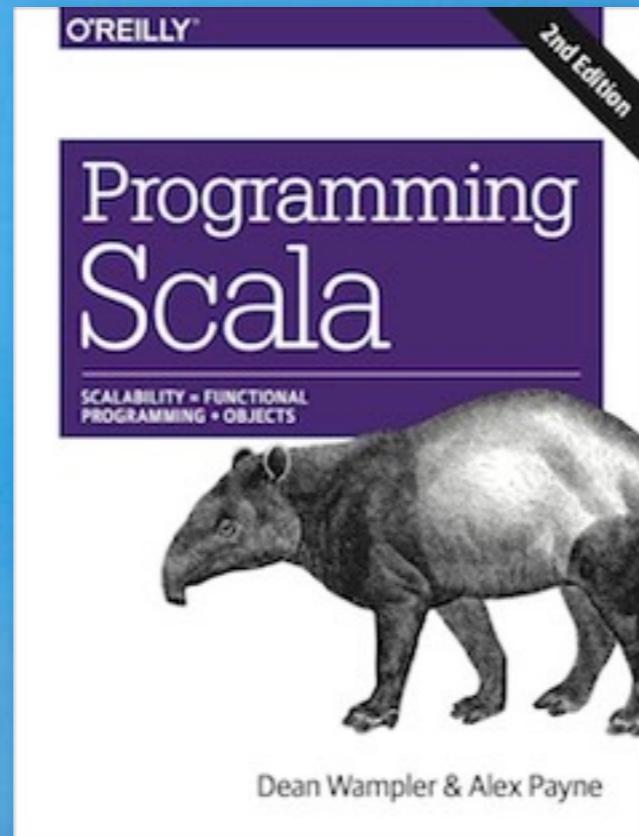
Nike Tech Talks  
Beaverton, OR, Jan. 22, 2015  
@deanwampler



Tuesday, January 20, 15

Copyright (c) 2014-2015, Dean Wampler, All Rights Reserved, unless otherwise noted.  
Image: Detail of the London Eye

# Dean Wampler



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

Tuesday, January 20, 15

About me. You can find this presentation and others on Big Data and Scala at [polyglotprogramming.com](http://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 Mate-  
rials requisite thereto : With Rules  
and Directions how to use them

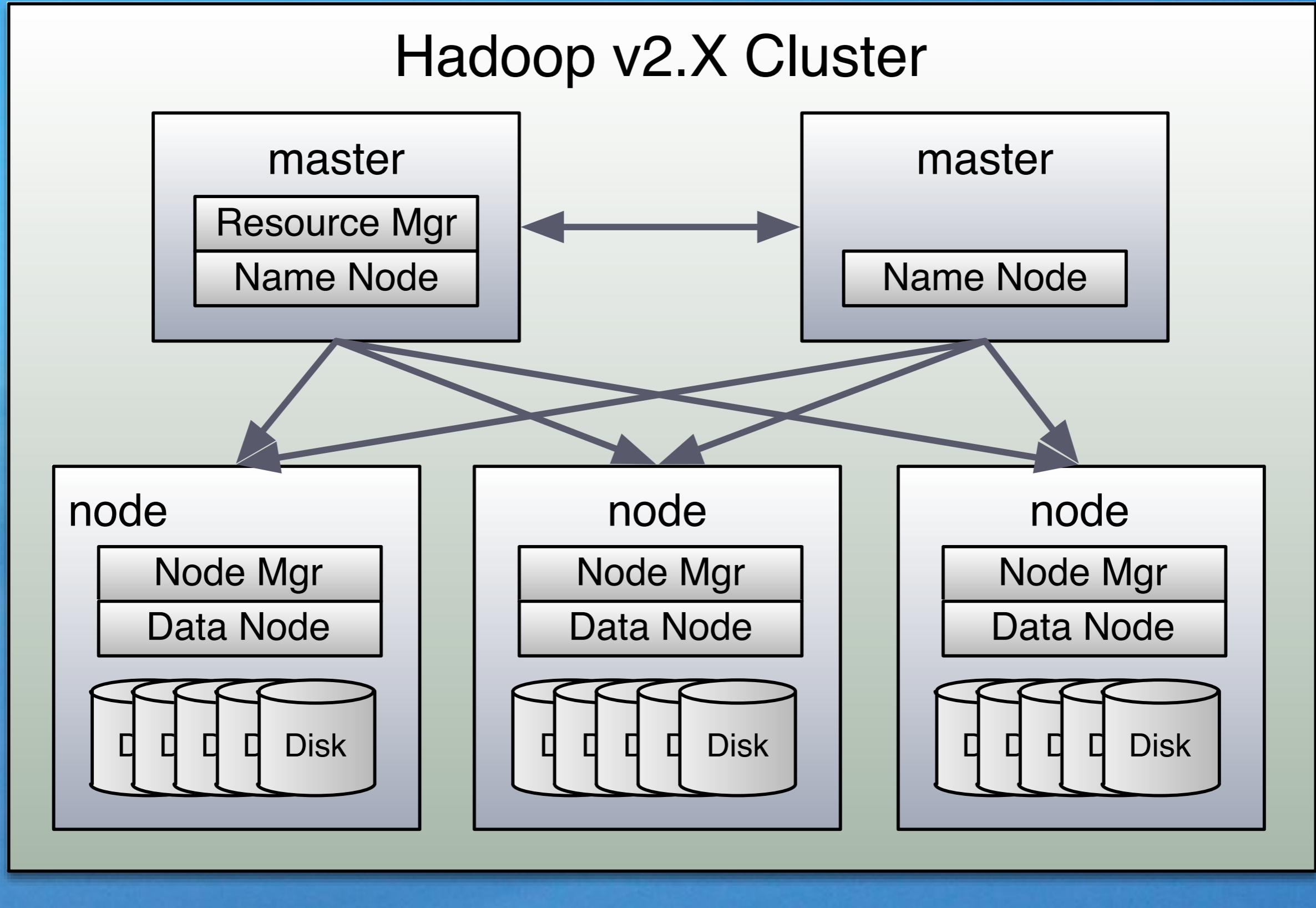
# Hadoop circa 2013



Tuesday, January 20, 15

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

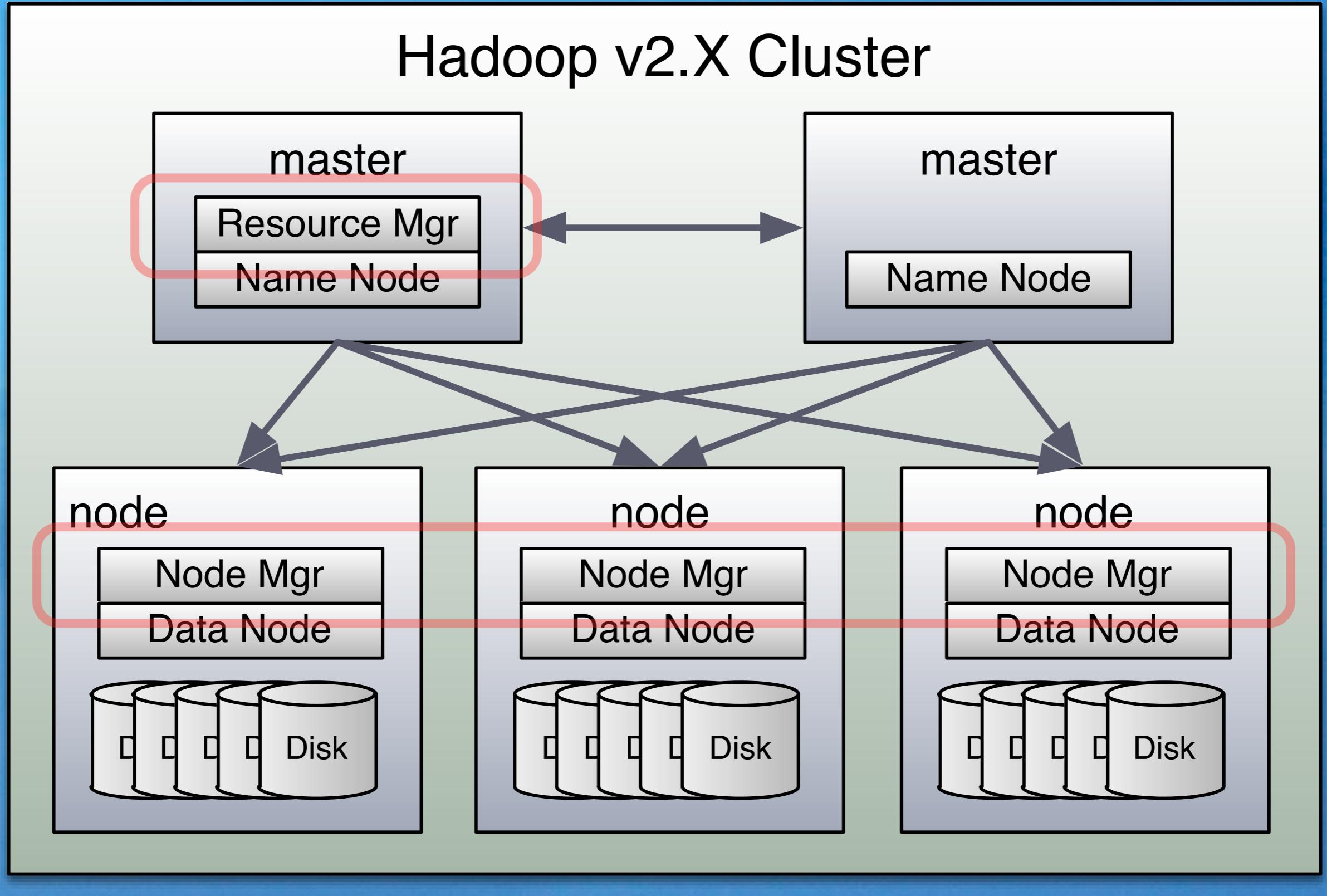
# Hadoop v2.X Cluster



Tuesday, January 20, 15

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.

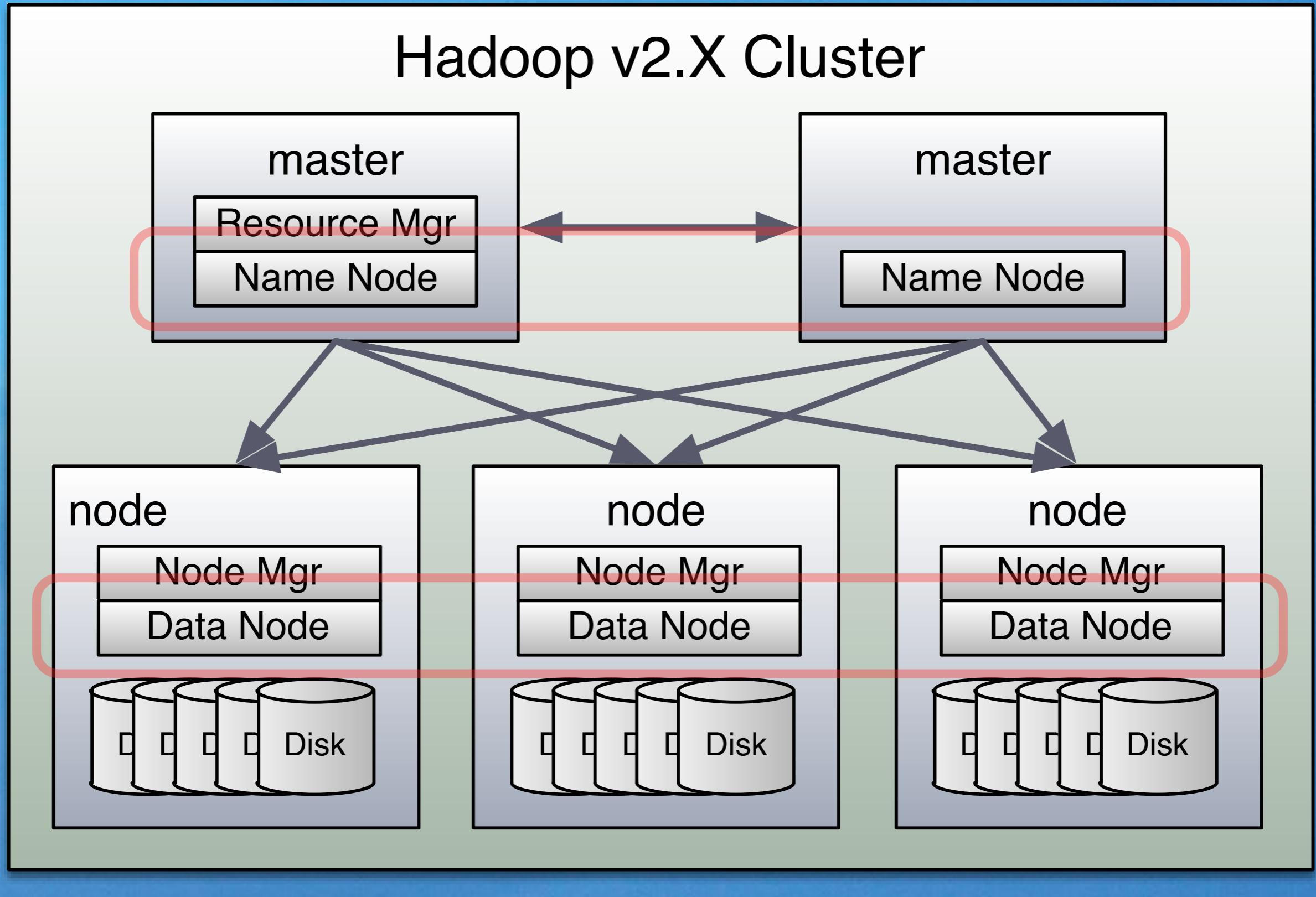
# Resource and Node Managers



Tuesday, January 20, 15

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



Tuesday, January 20, 15

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

Tuesday, January 20, 15

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)**

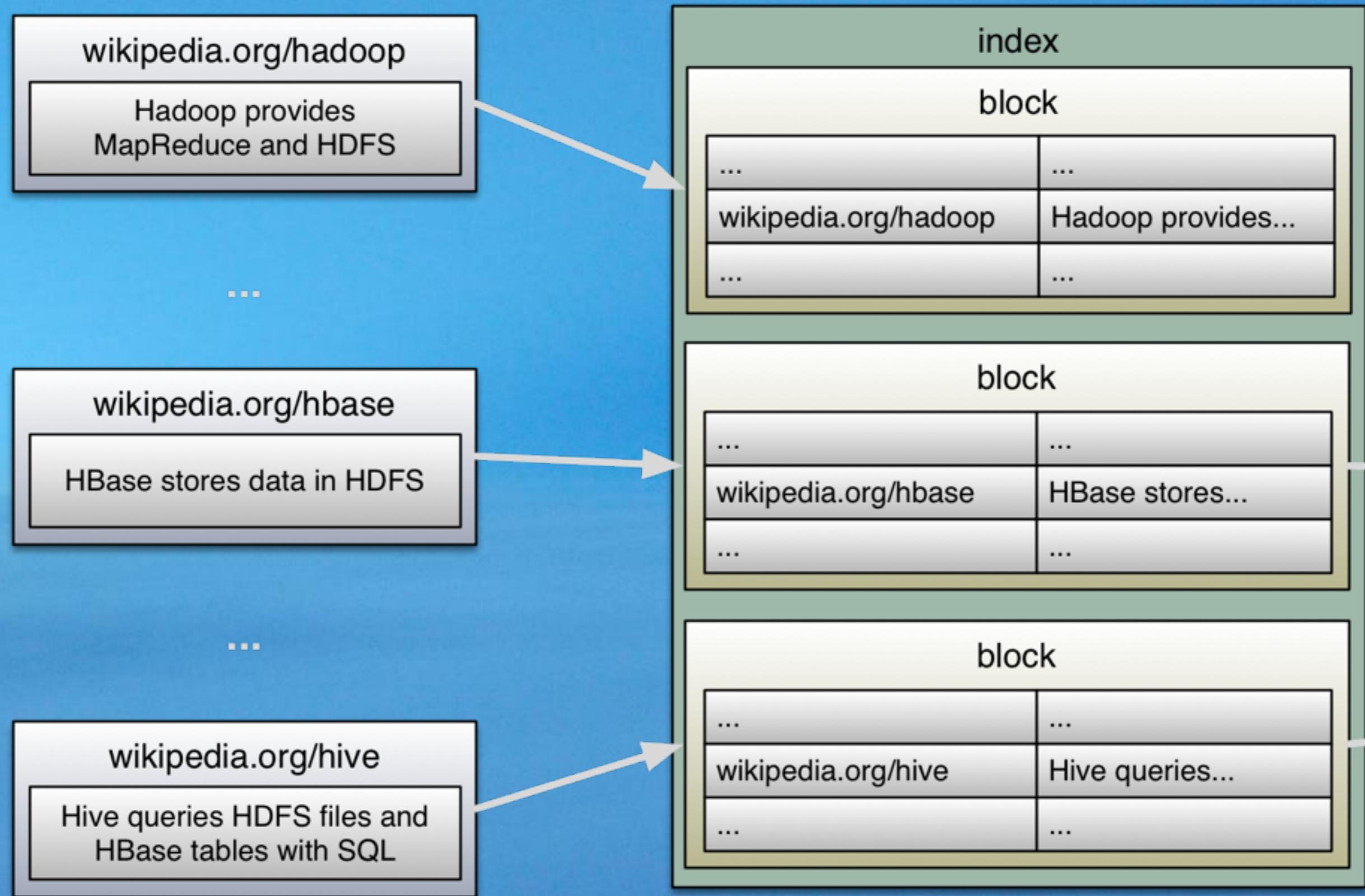
Tuesday, January 20, 15

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

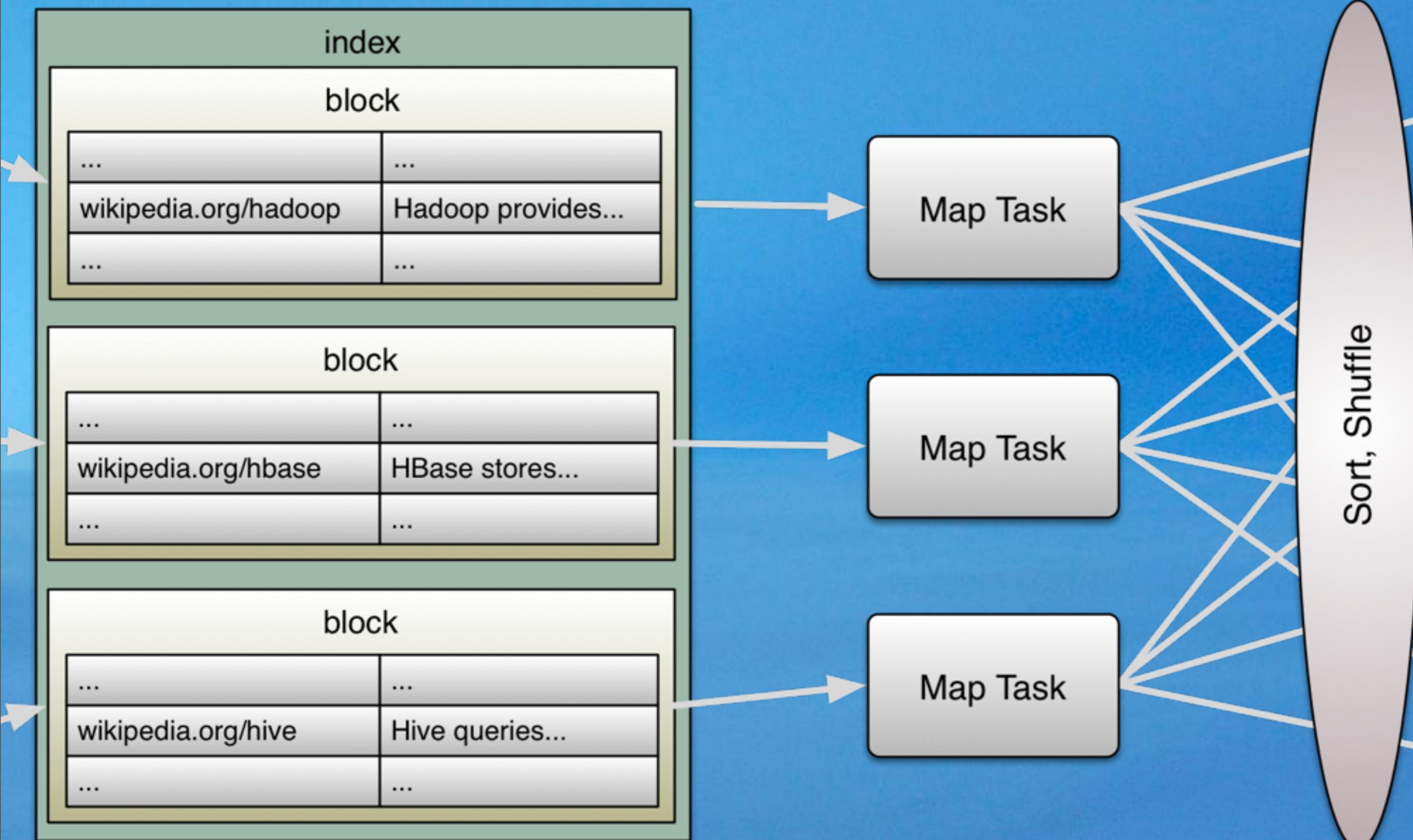
# Web Crawl



Tuesday, January 20, 15

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.

## Map Phase



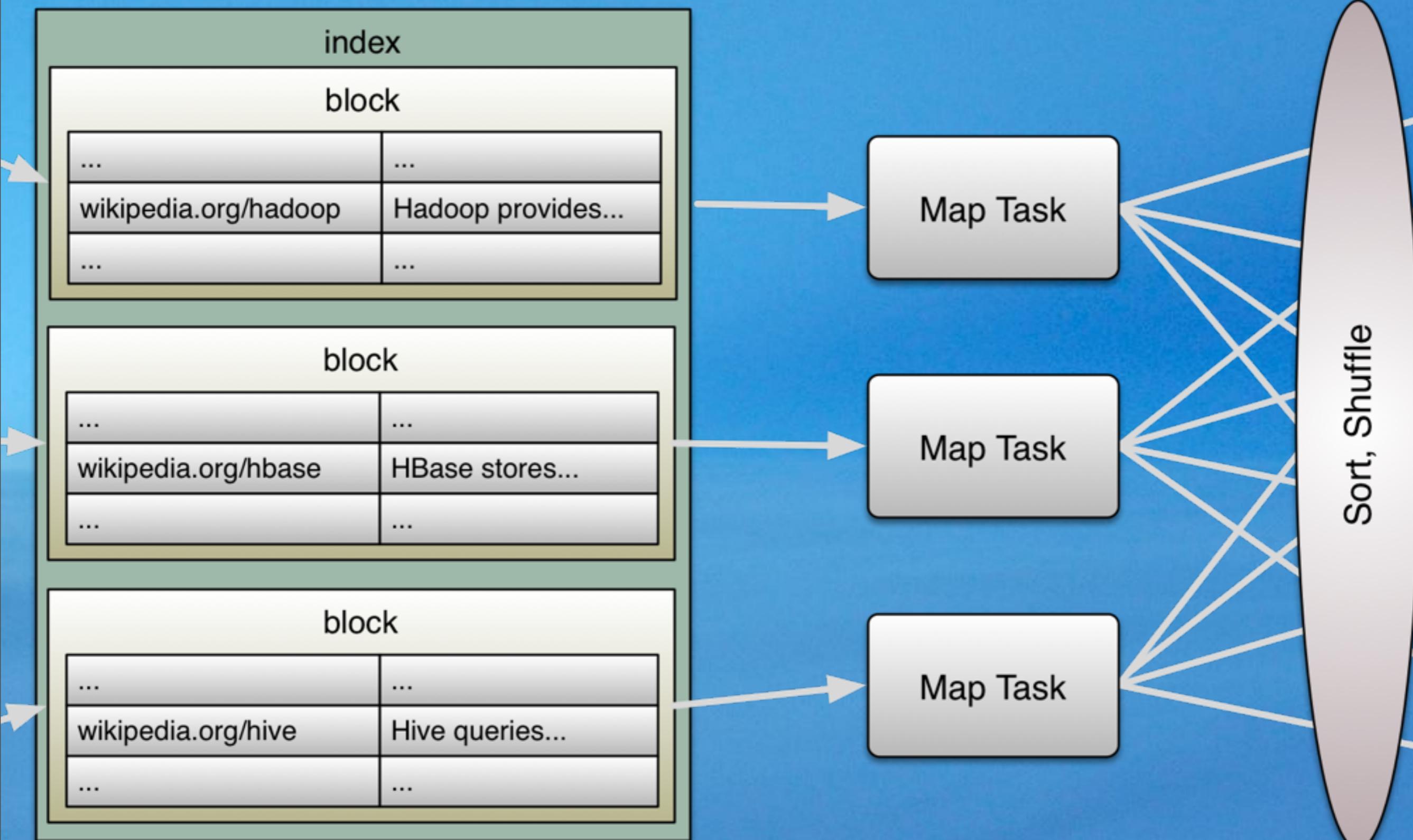
Tuesday, January 20, 15

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")...

## Map Task

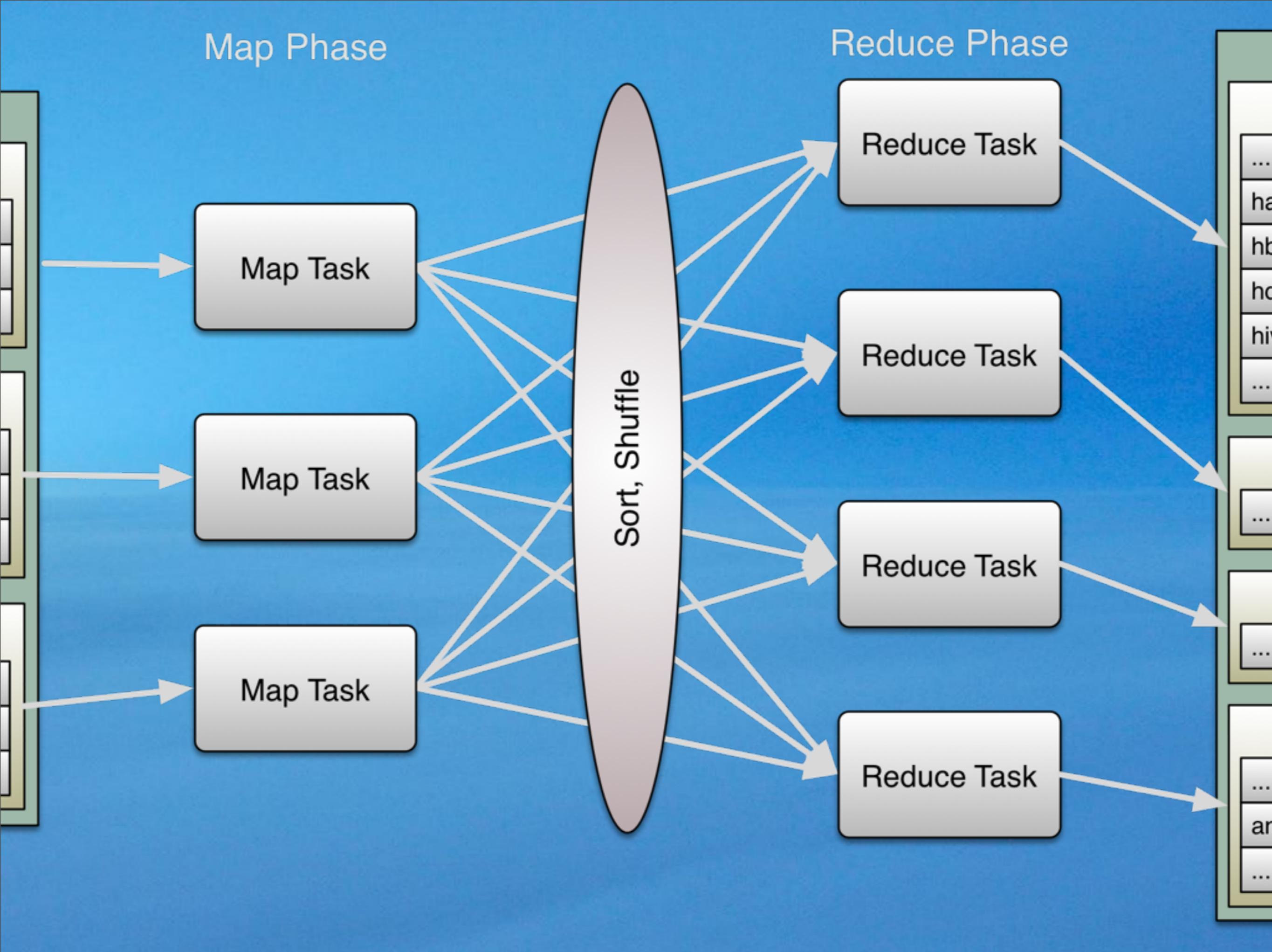
(hadoop,(wikipedia.org/hadoop,1))  
(provides,(wikipedia.org/hadoop,1))  
(mapreduce,(wikipedia.org/hadoop, 1))  
(and,(wikipedia.org/hadoop,1))  
(hdfs,(wikipedia.org/hadoop, 1))

## Map Phase



## Map Phase

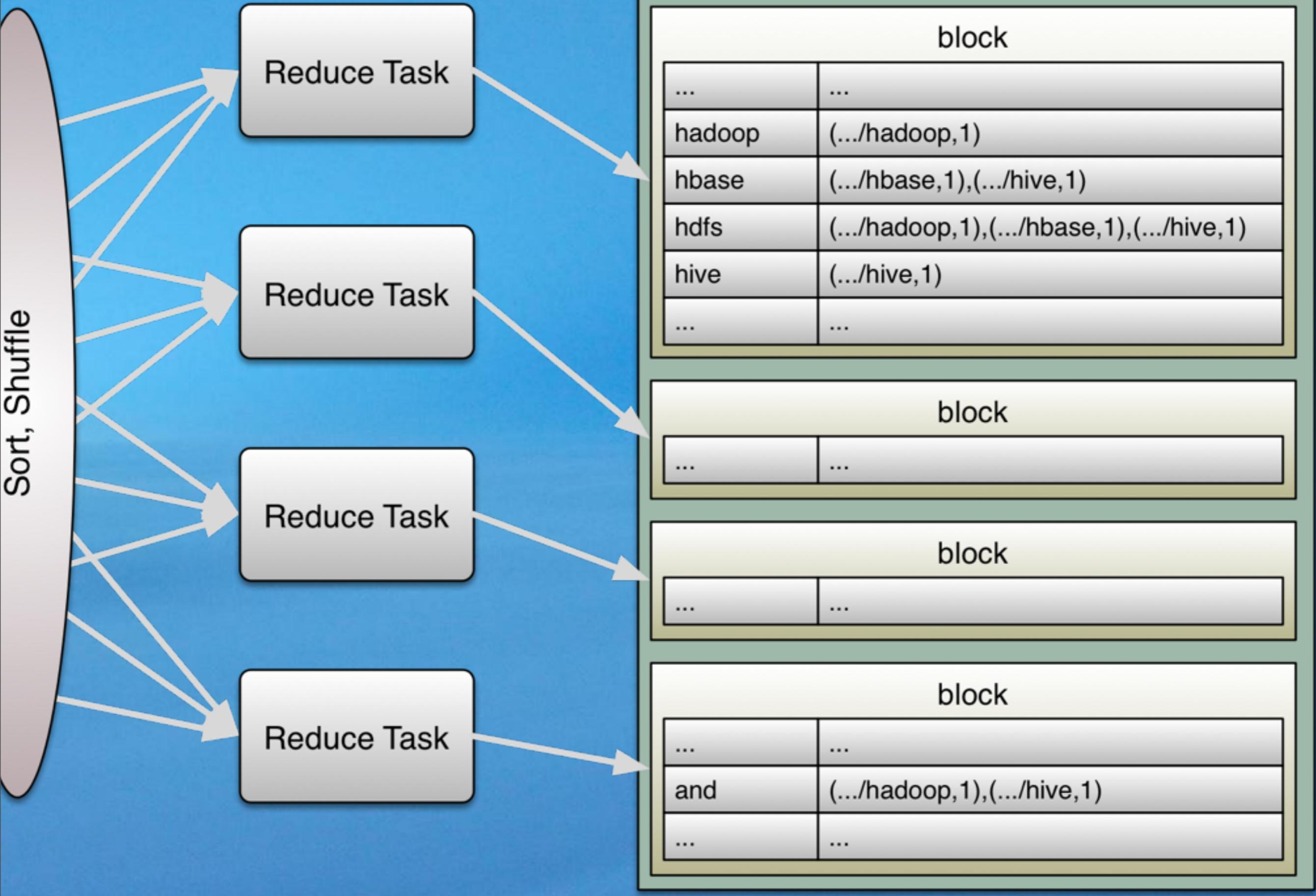
## Reduce Phase



Tuesday, January 20, 15

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.

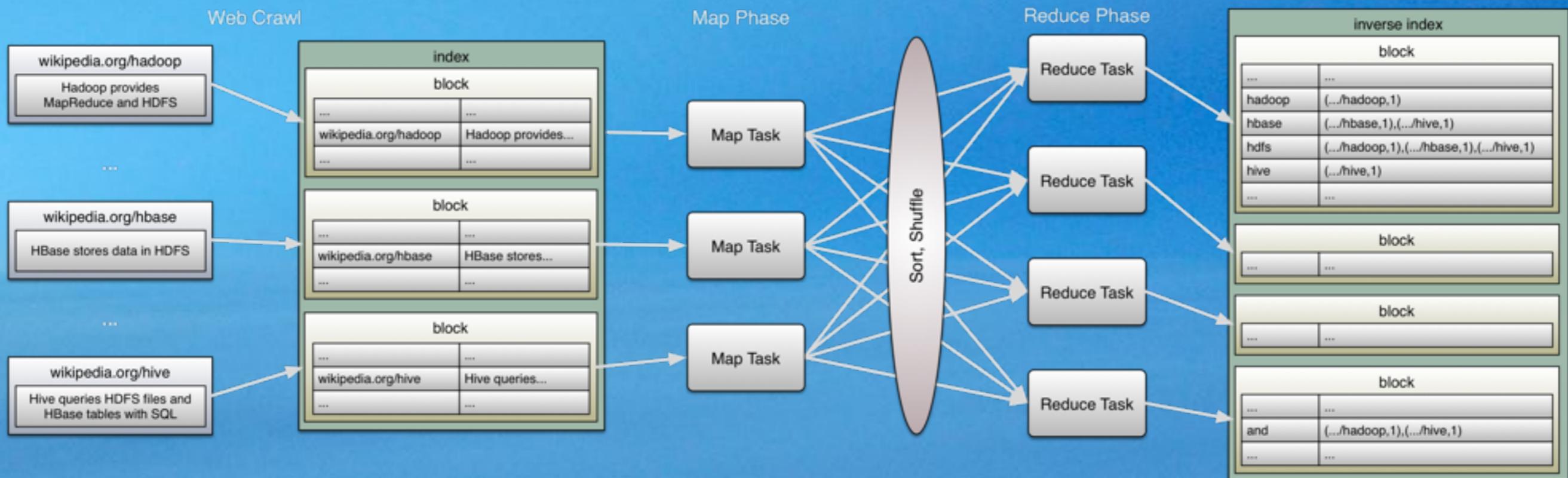
## Reduce Phase



Tuesday, January 20, 15

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...



Tuesday, January 20, 15

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).



What's  
not to like?

Tuesday, January 20, 15

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

# Awkward

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

# Awkward

Lack of flexibility inhibits optimizations, too.

Tuesday, January 20, 15

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.

Tuesday, January 20, 15

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](http://spark.apache.org)

# Cluster Computing

Can be run in:

- YARN (Hadoop 2)
- Mesos (Cluster management)
- EC2
- Standalone mode
- Cassandra
- ...



Tuesday, January 20, 15

If you have a Hadoop cluster, you can run Spark as a seamless compute engine on YARN. (You can also use pre-YARN Hadoop v1 clusters, but there you have manually allocate resources to the embedded Spark cluster vs Hadoop.) Mesos is a general-purpose cluster resource manager that can also be used to manage Hadoop resources. Scripts for running a Spark cluster in EC2 are available. Standalone just means you run Spark's built-in support for clustering (or run locally on a single box - e.g., for development). EC2 deployments are usually standalone... Finally, database vendors like Datastax are integrating Spark.

# Compute Model

Fine-grained “combinators”  
for composing algorithms.



Tuesday, January 20, 15

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

# Compute Model

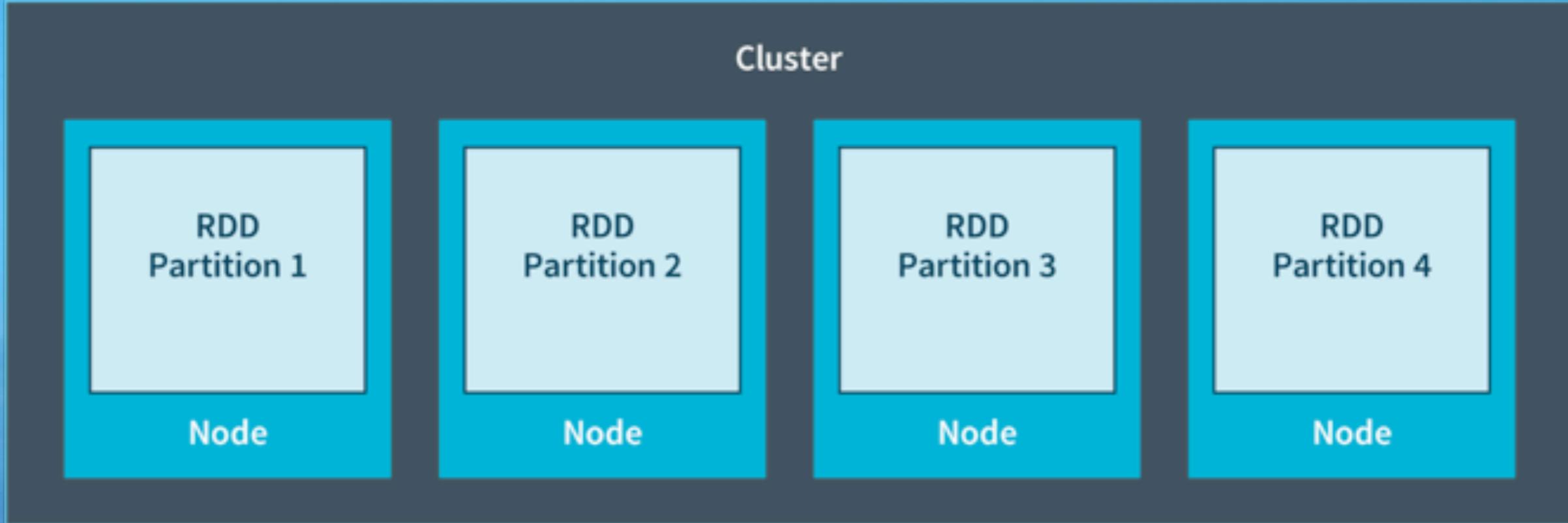
RDDs:  
Resilient,  
Distributed  
Datasets



Tuesday, January 20, 15

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



**Spark**

Tuesday, January 20, 15

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.



Tuesday, January 20, 15

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



# *Inverted Index* in MapReduce (Java).

Tuesday, January 20, 15

Let's see 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 than 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.

```
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);
    }
}
```

Tuesday, January 20, 15

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...

```
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);
```

```
    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> {
```

```
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 =
```

Tuesday, January 20, 15

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
  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);
  }
}
```

Tuesday, January 20, 15

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()));
}
}
```

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

# Altogether

Tuesday, January 20, 15

The whole shebang (6pt. font)



# *Inverted Index* in Spark (Scala).

Tuesday, January 20, 15

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.

```
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, January 20, 15

The InvertedIndex implemented in Spark. This time, we'll also count the occurrences in each document (as I originally described the algorithm) and sort the (url,N) pairs descending by N (count), and ascending by URL.

```
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, January 20, 15

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.

```
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 {
```

Tuesday, January 20, 15

You begin the workflow by declaring a `SparkContext`. We're running in "local" mode, in this case, meaning on a single machine (and using a single core). Normally this argument would be a command-line parameter, so you can develop locally, then submit to a cluster, where "local" would be replaced by the appropriate URI.

```
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 {
```

Tuesday, January 20, 15

The rest of the program is a sequence of function calls, analogous to “pipes” we connect together to construct the data flow. Data will only start “flowing” when we ask for results.

We start by reading one or more text files from the directory “data/crawl”. If running in Hadoop, if there are one or more Hadoop-style “part-NNNNN” files, Spark will process all of them (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
  }
}
```

Tuesday, January 20, 15

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 "fileName: String, text: String".

```
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
  }
}
```

Tuesday, January 20, 15

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. That is, each line (one thing) is converted to a collection of (word,path) pairs (0 to many things), but we don’t want an output collection of nested collections, so flatMap concatenates nested collections into one long “flat” collection of (word,path) pairs.

```

}
.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)
}

```

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

Tuesday, January 20, 15

Next, we map over these pairs and add a single “seed” count of 1. Note the structure of the returned tuple; it’s a two-tuple where the first element is itself a two-tuple holding (word, path). The following special method, reduceByKey is like a groupBy, where it groups over those (word, path) “keys” and uses the function to sum the integers. The popup shows the what the output data looks like.

```

}
.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)
}

```

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

```
    case ((w, p), n) => (w, (p, 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)
    }
    .sortBy {
      case (path, n) => (-n, path)
    }
    (w, seq2.mkString("", ""))
}
• saveAsTextFile("path/to/out")
```

```

    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)
    }
    . sortBy {
      case (path, n) => (-n, path)
    }
    (w, seq2.mkString(", "))
  (word, "(path1, n1), (path2, n2), ...")
}
• saveAsTextFile("/path/to/out")

```

Tuesday, January 20, 15

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 and sorts by count descending, path ascending. (Sorting by path is mostly useful for reproducibility, e.g., in tests!) It outputs the sequence as a final string of comma-separated (path,n) pairs.

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

  sc.stop()
}
}
```

```
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("/path/to/out")

    sc.stop()
  }
}
```

# Altogether

```
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
}
```

*Concise  
Combinators!*

Tuesday, January 20, 15

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...

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

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

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

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

Tuesday, January 20, 15

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%27sEquations.svg>



*The Spark version  
took me ~30 minutes  
to write.*

Tuesday, January 20, 15

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.



*Use a SQL query when  
you can!!*

# Spark SQL!

Mix SQL queries with  
the RDD API.



Tuesday, January 20, 15

Use the best tool for a particular  
problem.

# Spark SQL!

Create, Read, and Delete  
Hive Tables



Tuesday, January 20, 15

Interoperate with Hive, the original Hadoop SQL tool.

# Spark SQL!

Read JSON and  
Infer the Schema



Tuesday, January 20, 15

Read strings that are JSON records, infer the schema on the fly. Also, write RDD records as JSON.

# Spark SQL!

Read and write  
Parquet files



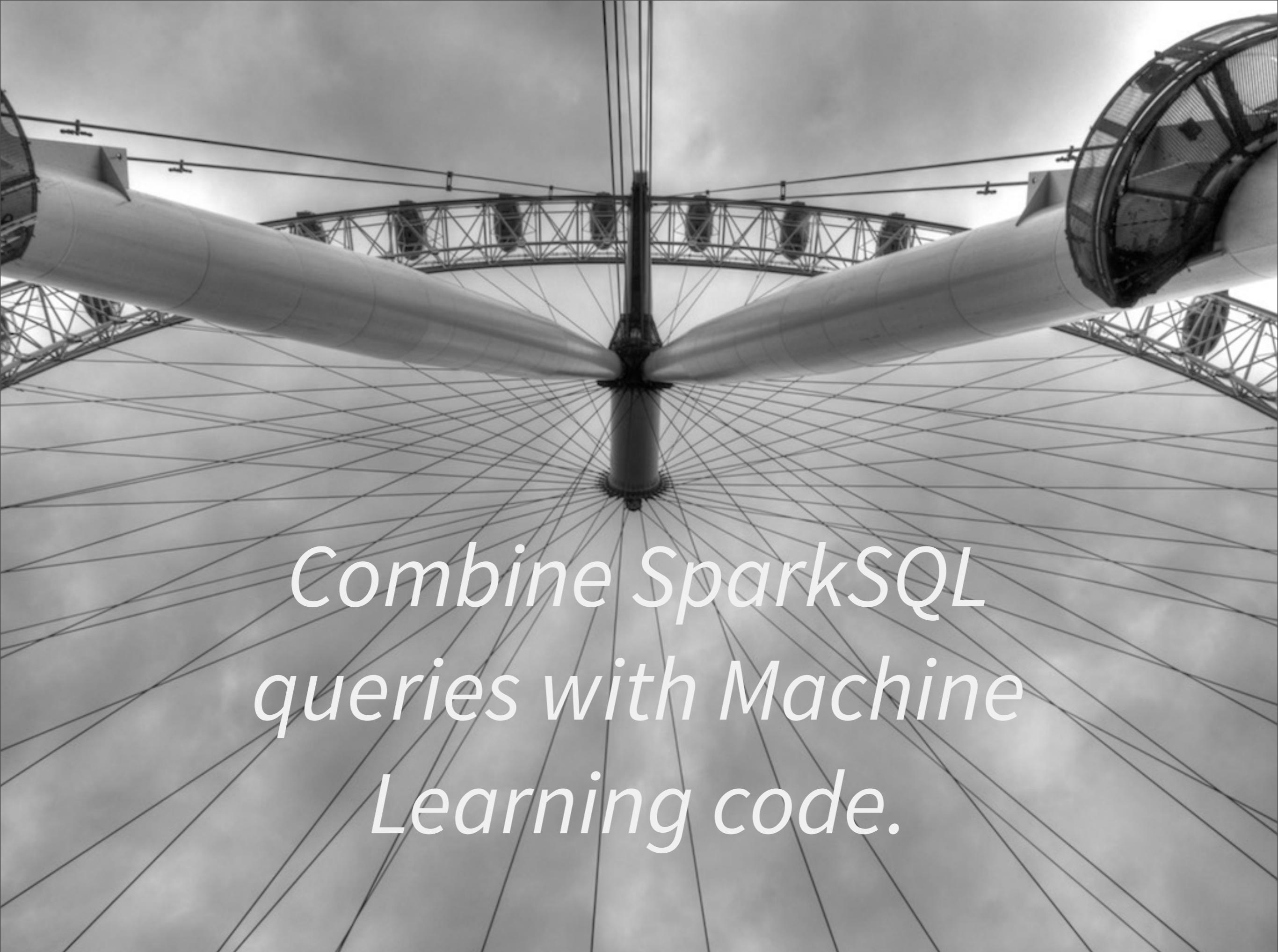
Tuesday, January 20, 15

Parquet is a newer file format developed by Twitter and Cloudera that is becoming very popular. It stores in column order, which is better than row order when you have lots of columns and your queries only need a few of them. Also, columns of the same data types are easier to compress, which Parquet does for you. Finally, Parquet files carry the data schema.

# SparkSQL

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





# *Combine SparkSQL queries with Machine Learning code.*

Tuesday, January 20, 15

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

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

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

Equivalent  
HiveQL Schemas  
definitions.

Tuesday, January 20, 15

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>

Adapted here to use Spark's own SQL, not the integration with Hive. Imagine we have a stream of events from users and the events that have occurred as they used a system.

```
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()  
    .run(trainingData)
```

Tuesday, January 20, 15

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

The “sql” function returns an RDD. If we used the Hive integration and this was a query against a Hive table, we would use the hql(...) function instead.

```
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()  
    .run(trainingData)
```

Tuesday, January 20, 15

We map over the RDD 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...)

```
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))  
}
```

Tuesday, January 20, 15

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>)

```
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))  
}
```

```
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))  
}
```

```
// In-memory table  
scores.registerTempTable("Scores")
```

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

```
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))  
}
```

```
// In-memory table  
scores.registerTempTable("Scores")
```

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

Tuesday, January 20, 15

Then “register” the scores RDD as a “Scores” table in memory. If you use the Hive binding instead, this would be a table in Hive’s metadata storage.

```
// In-memory table  
scores.registerTempTable("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""")
```

Tuesday, January 20, 15

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...

```
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()  
  .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))  
}  
  
// In-memory table  
scores.registerTempTable("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""")
```

# Altogether



# *Event Stream Processing*

# Spark Streaming

Use the same abstractions  
for near real-time,  
event streaming.



Tuesday, January 20, 15

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



**Spark**

Tuesday, January 20, 15

A DStream (discretized stream) wraps the RDDs for each “batch” of events. You can specify the granularity, such as all events in 1 second batches, then your Spark job is passed each batch of data for processing. You can also work with moving windows of batches.



*Very similar code...*

```
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)
```

Tuesday, January 20, 15

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>

```
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)
```

Tuesday, January 20, 15

We create a StreamingContext that wraps a SparkContext (there are alternative ways to construct it...). It will “clump” the events into 1-second intervals.

```
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)
```

```
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()
```

Tuesday, January 20, 15  
Now we “count words”. For each mini-batch (1 second’s worth of data), we split the input text into words (on whitespace, which is too crude). Once we setup the flow, we start it and wait for it to terminate through some means, such as the server socket closing.

```
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()
```

```
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()
```

```
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()
```



# Machine Learning Library

Tuesday, January 20, 15

# MLlib

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



Tuesday, January 20, 15

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.



# *Distributed Graph Computing*

Tuesday, January 20, 15

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

# Graphs

- Examples:
  - Facebook friend networks
  - Twitter followers
  - Page Rank



# Graphs

- Graph algorithms
  - Friends of friends
  - Minimum spanning tree
  - Cliques





# Recap



**Dean Wampler**

@deanwampler

Functional Programming: I came for the concurrency, but I stayed for the data science.

Reply Delete Favorite More

RETWEETS

6

FAVORITES

5



Tuesday, January 20, 15

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

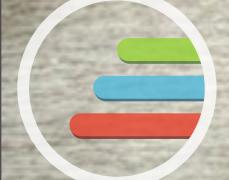
A flexible, scalable distributed  
compute platform with  
concise, powerful APIs and  
higher-order tools.

[spark.apache.org](http://spark.apache.org)



A photograph of the London Eye and the Palace of Westminster (including Big Ben) across the River Thames. The sky is overcast with dramatic clouds.

# Why Spark Is the Next Top (Compute) Model

The Typesafe logo, featuring a circular icon with four horizontal bars in green, blue, and red, followed by the word "Typesafe" in a bold, sans-serif font.

Typesafe

@deanwampler

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

[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)

Tuesday, January 20, 15

Copyright (c) 2014-2015, Dean Wampler, All Rights Reserved, unless otherwise noted.  
Image: The London Eye on one side of the Thames, Parliament on the other.