

Data Science at Scale with Spark

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

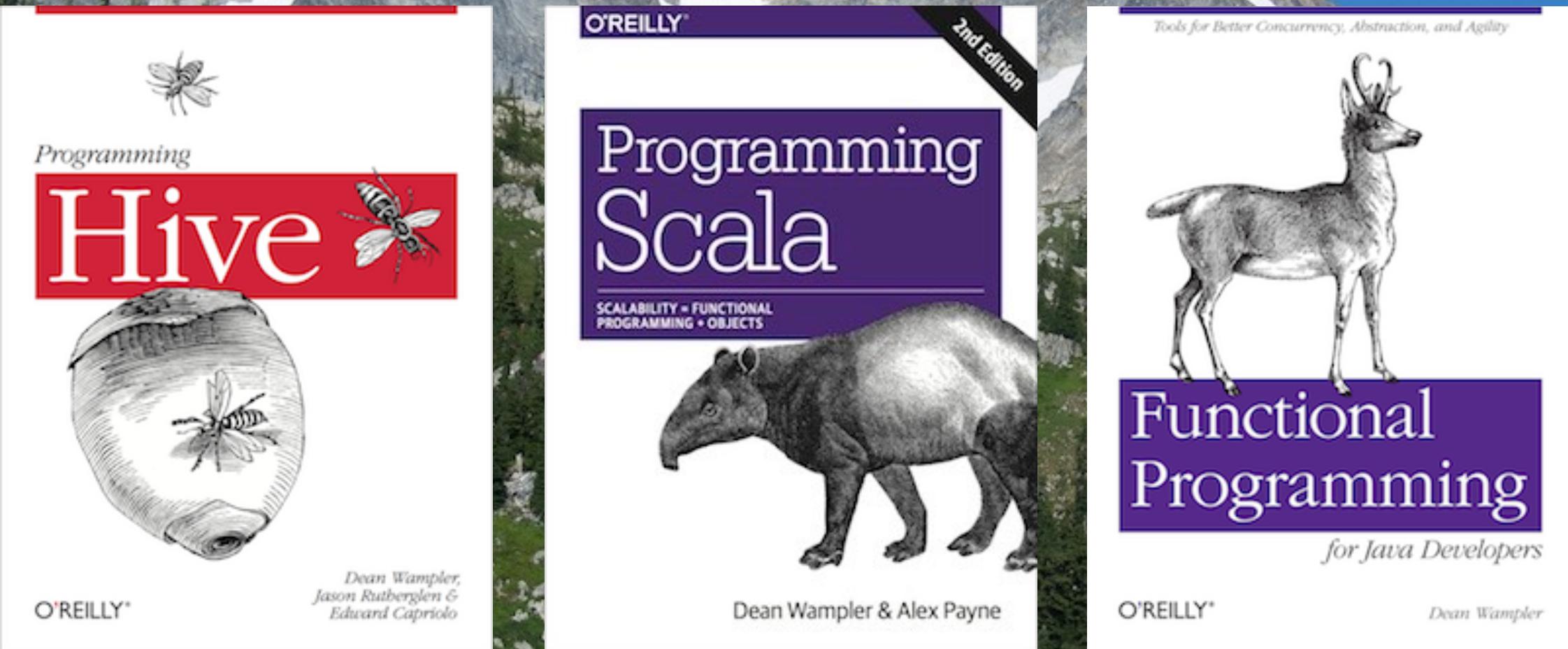


Thursday, November 12, 15

Photos Copyright © Dean Wampler, 2011-2015, except where noted. Some Rights Reserved. (Most are from the North Cascades, Washington State, August 2013.)
The content is free to reuse, but attribution is requested.
<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>



<shameless>
<plug>



</plug>
</shameless>

Thursday, November 12, 15

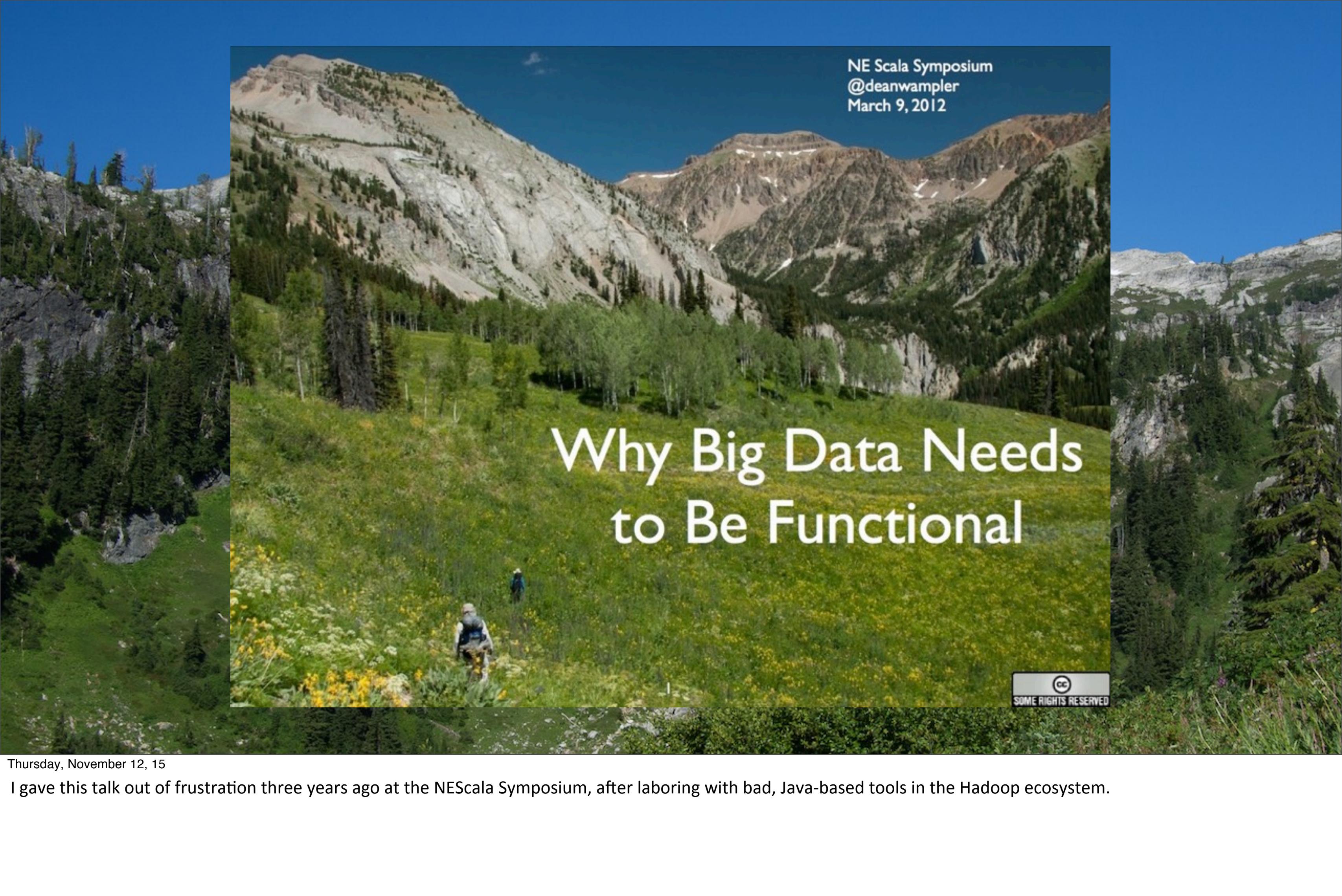
Every developer talk should have some XML!!



*“Trolling the Hadoop
community since 2012...”*

Thursday, November 12, 15

My linkedin profile since 2012??



NE Scala Symposium
@deanwampler
March 9, 2012

Why Big Data Needs to Be Functional



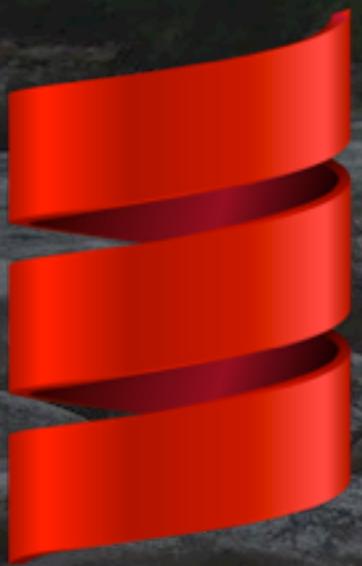
Thursday, November 12, 15

I gave this talk out of frustration three years ago at the NEScala Symposium, after laboring with bad, Java-based tools in the Hadoop ecosystem.



Why the JVM?

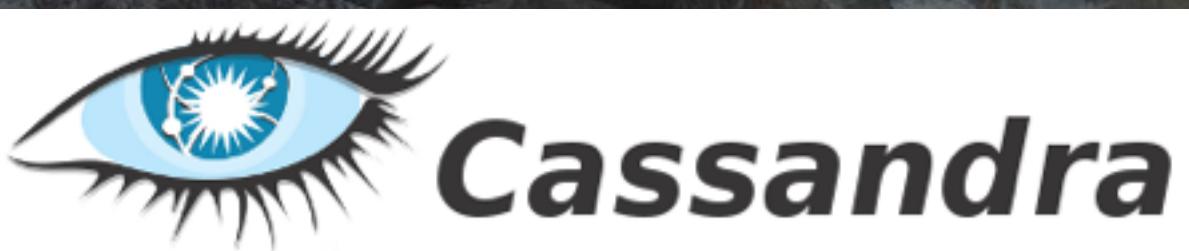
The JVM



Algebroid
Spire
...



Big Data Tools



samza

A photograph of a person walking away from the camera on a narrow dirt path through a dense forest. The forest floor is covered in thick green moss and fallen tree branches. The trees are tall and have dark, textured bark. The lighting is soft, suggesting a shaded environment.

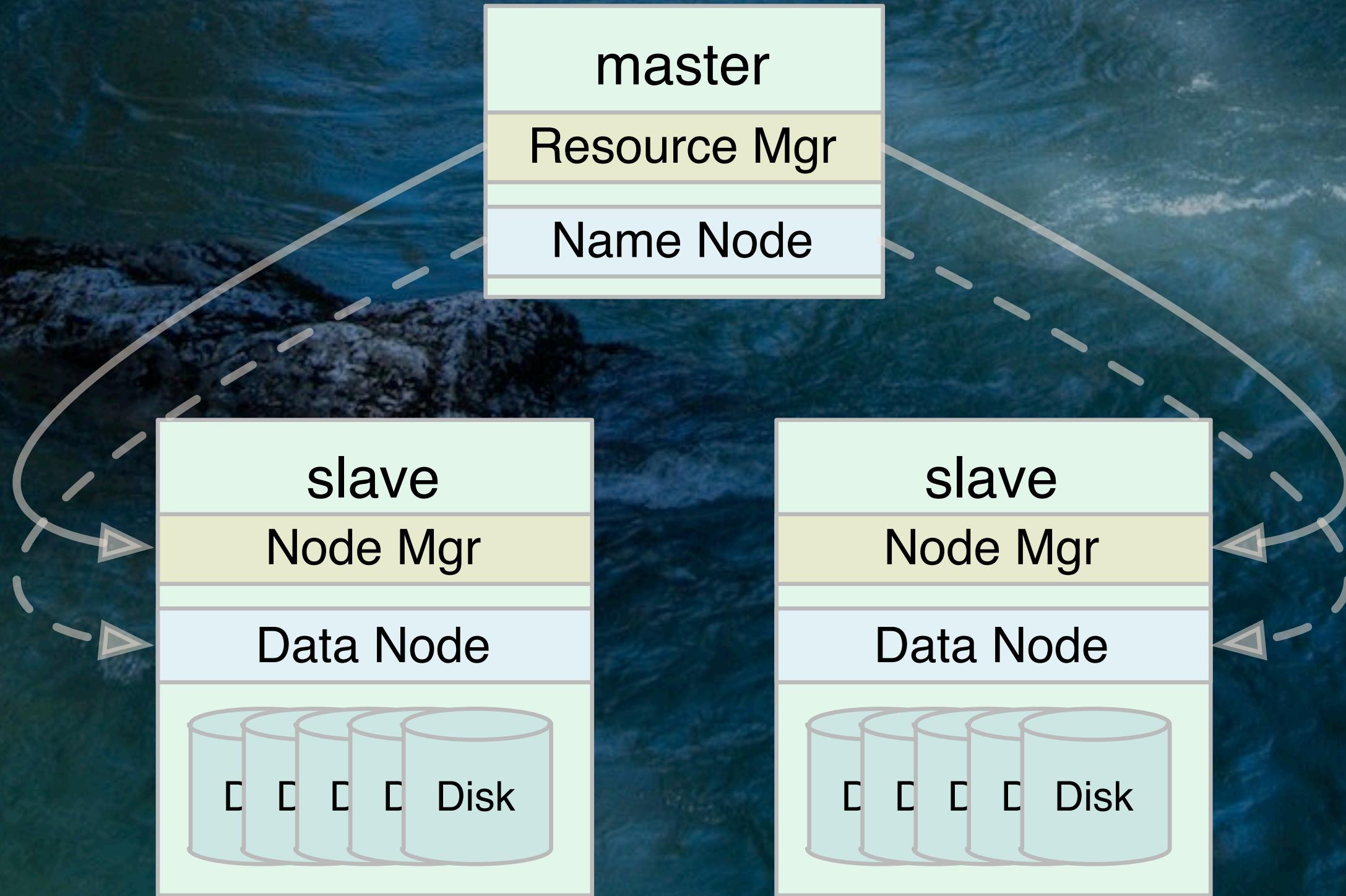
Hadoop

Hadoop

Thursday, November 12, 15

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

Hadoop

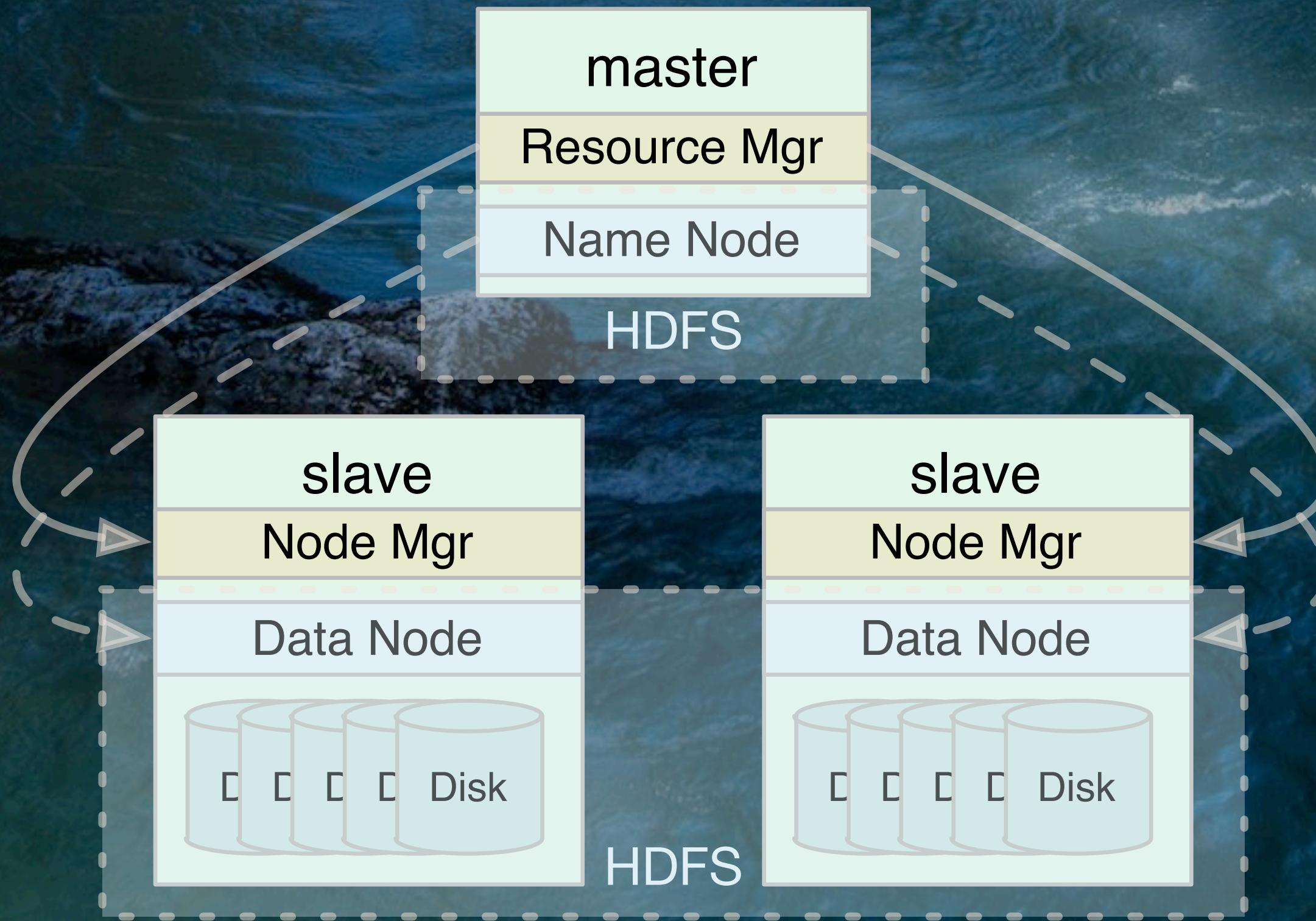


9

Thursday, November 12, 15

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

Hadoop



10

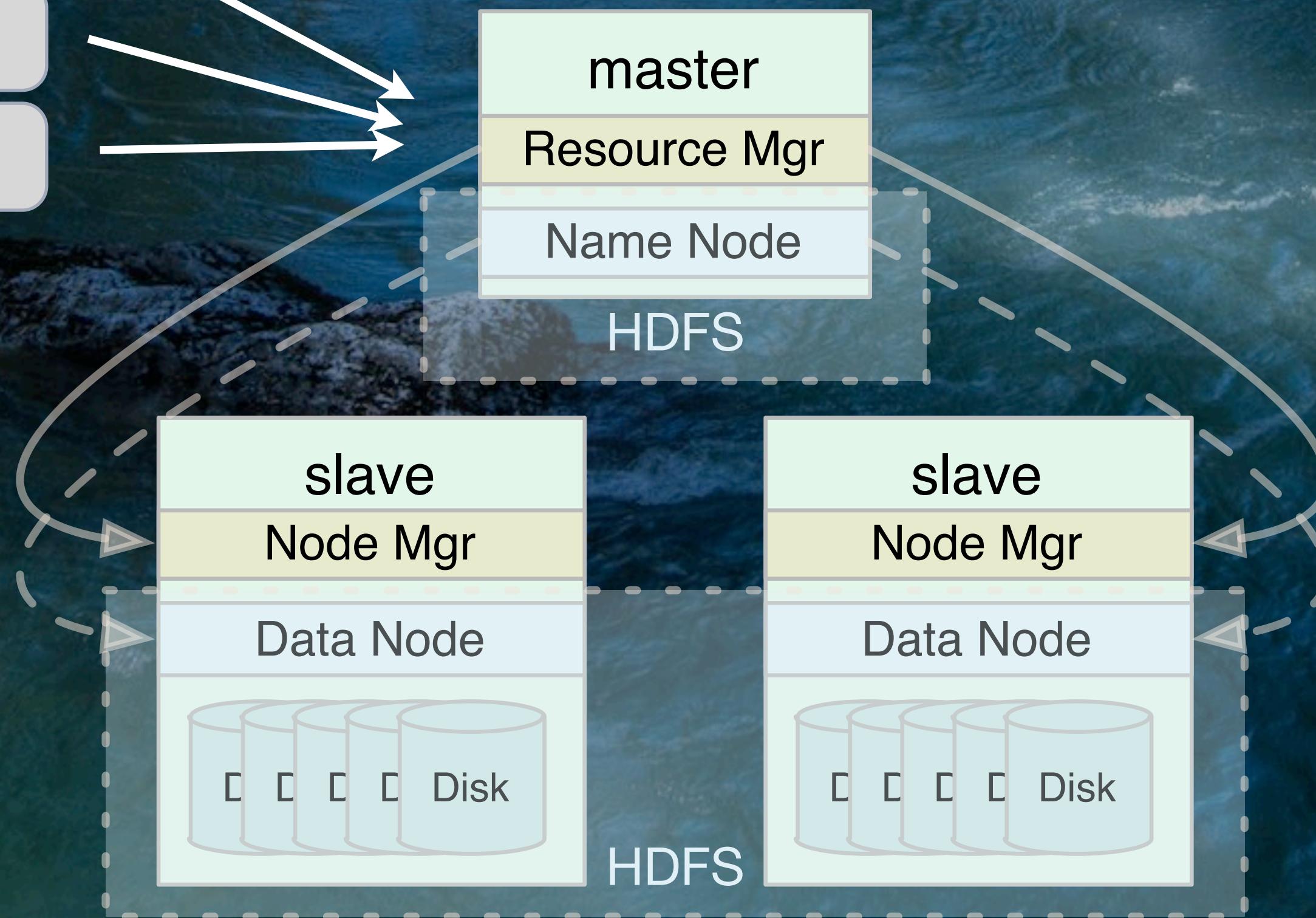
Thursday, November 12, 15

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

MapReduce Job

MapReduce Job

MapReduce Job



11

Thursday, November 12, 15

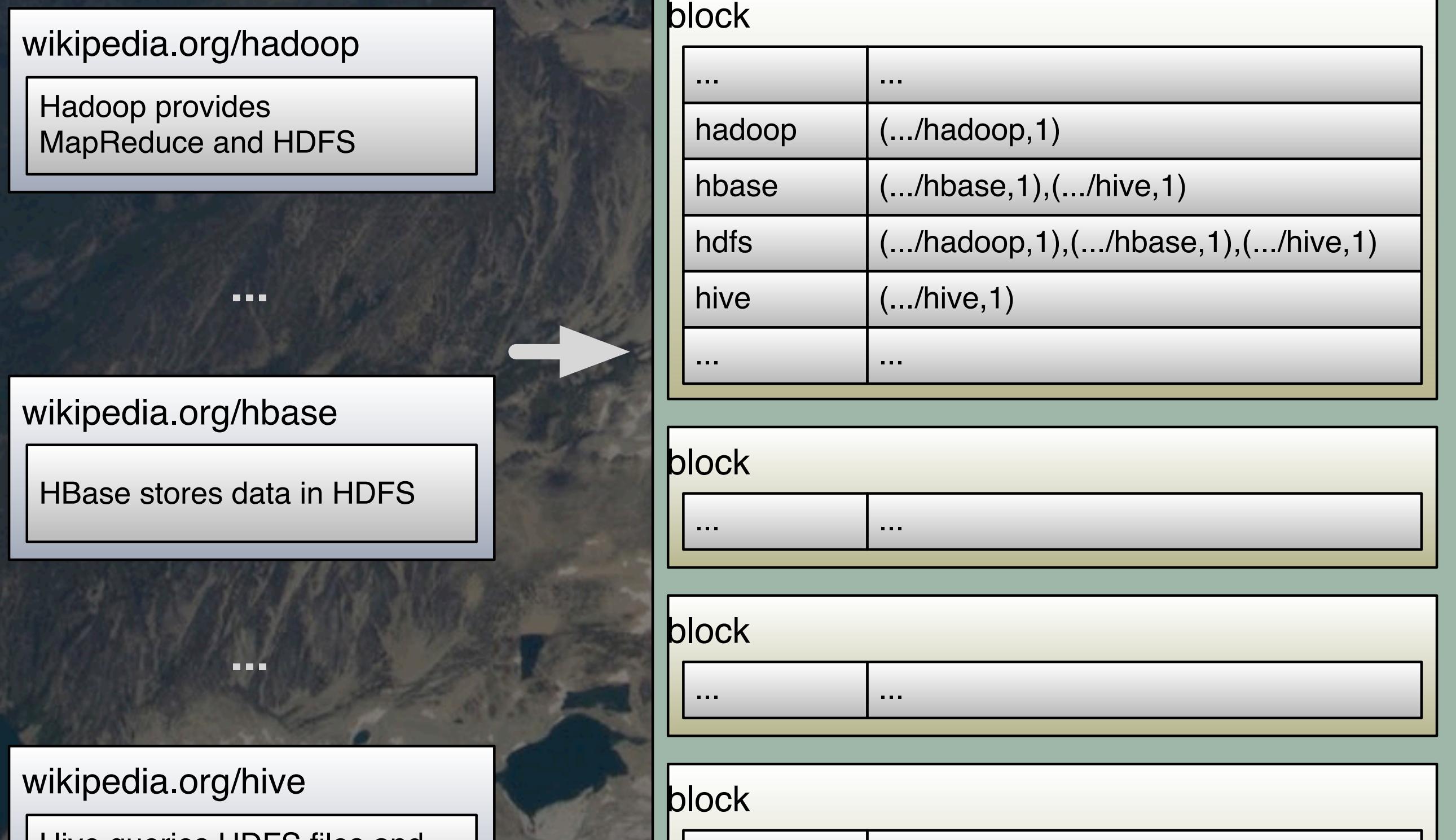
You submit MapReduce jobs to the Resource Manager. Those jobs could be written in the Java API, or higher-level APIs like Cascading, Scalding, Pig, and Hive.

MapReduce

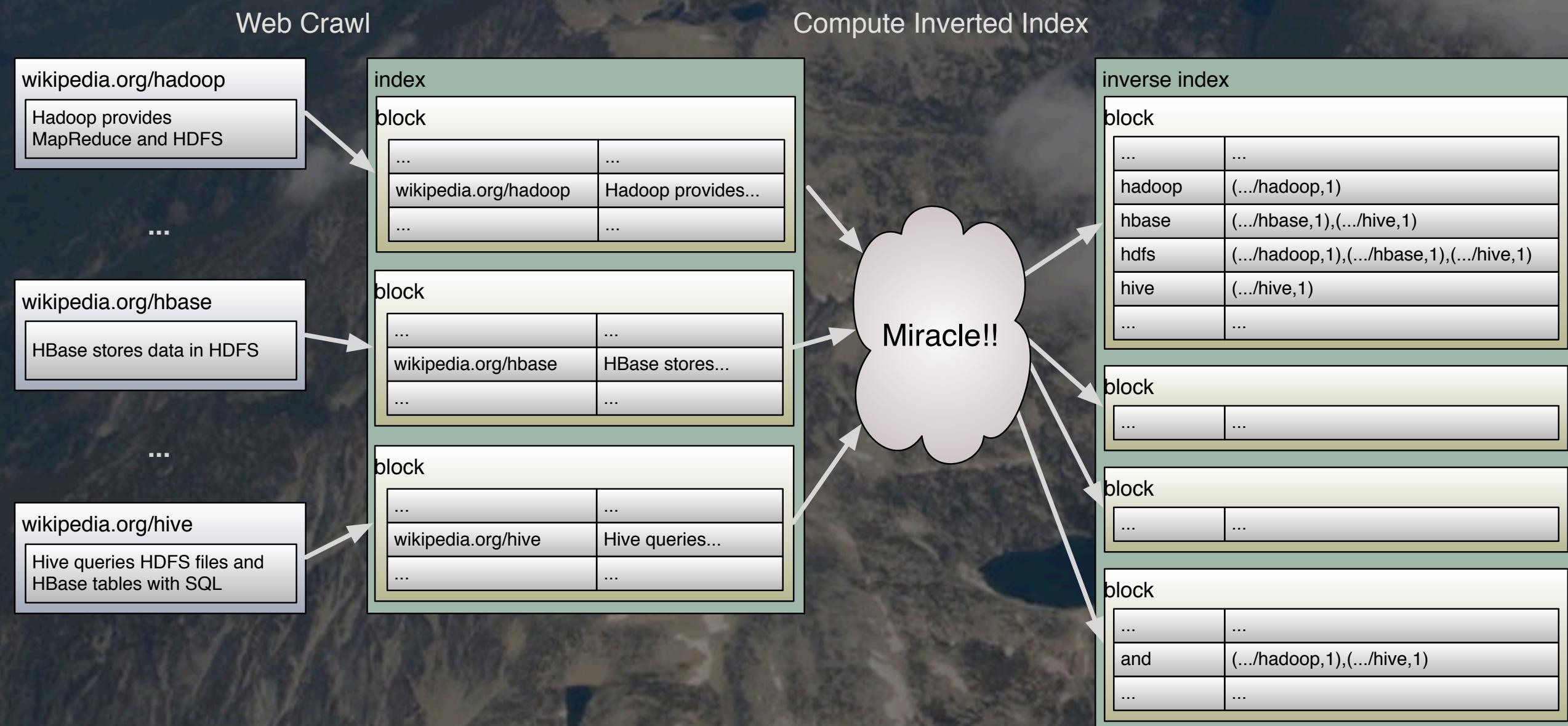
Thursday, November 12, 15

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

Example: Inverted Index

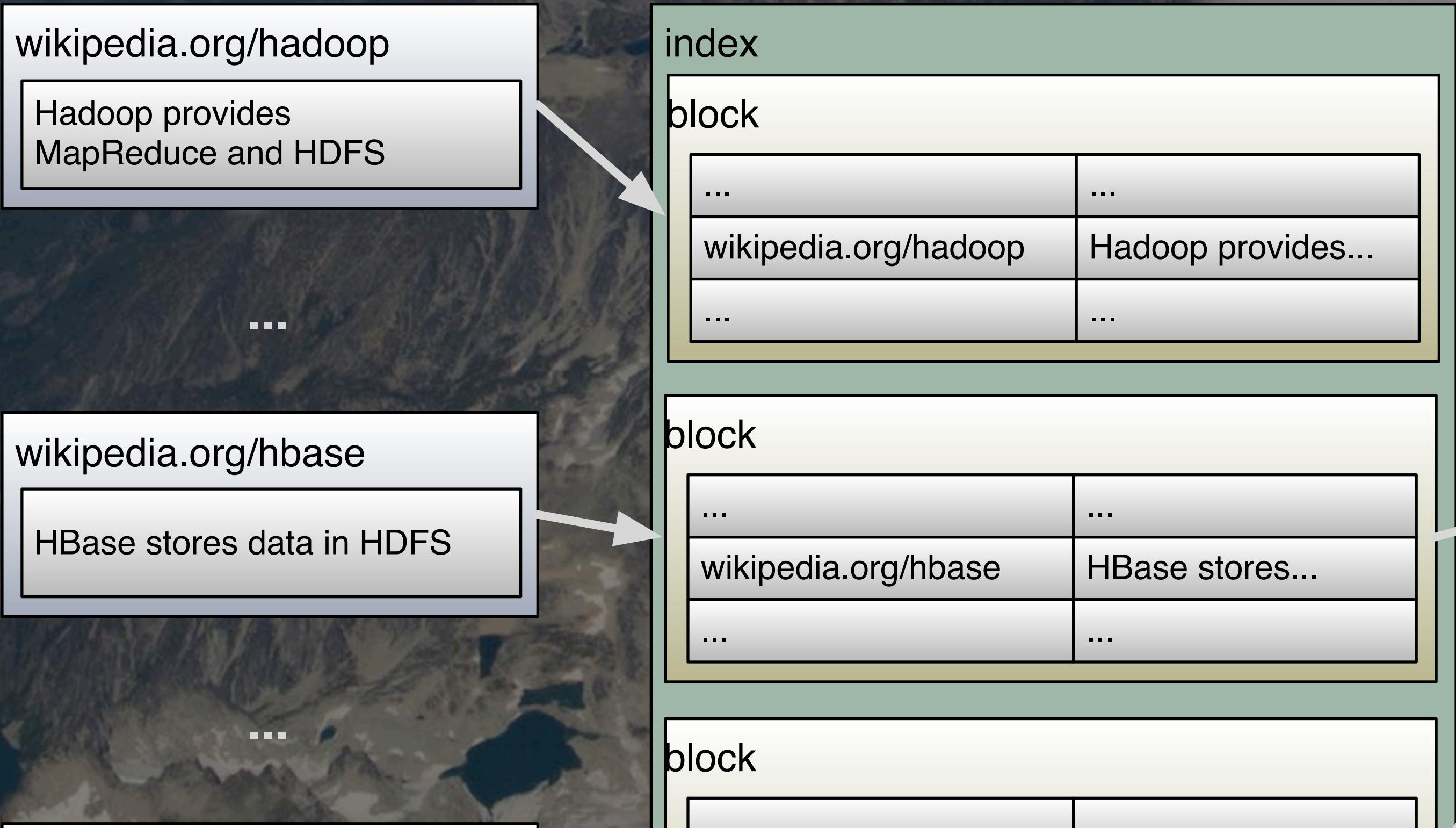


Example: Inverted Index



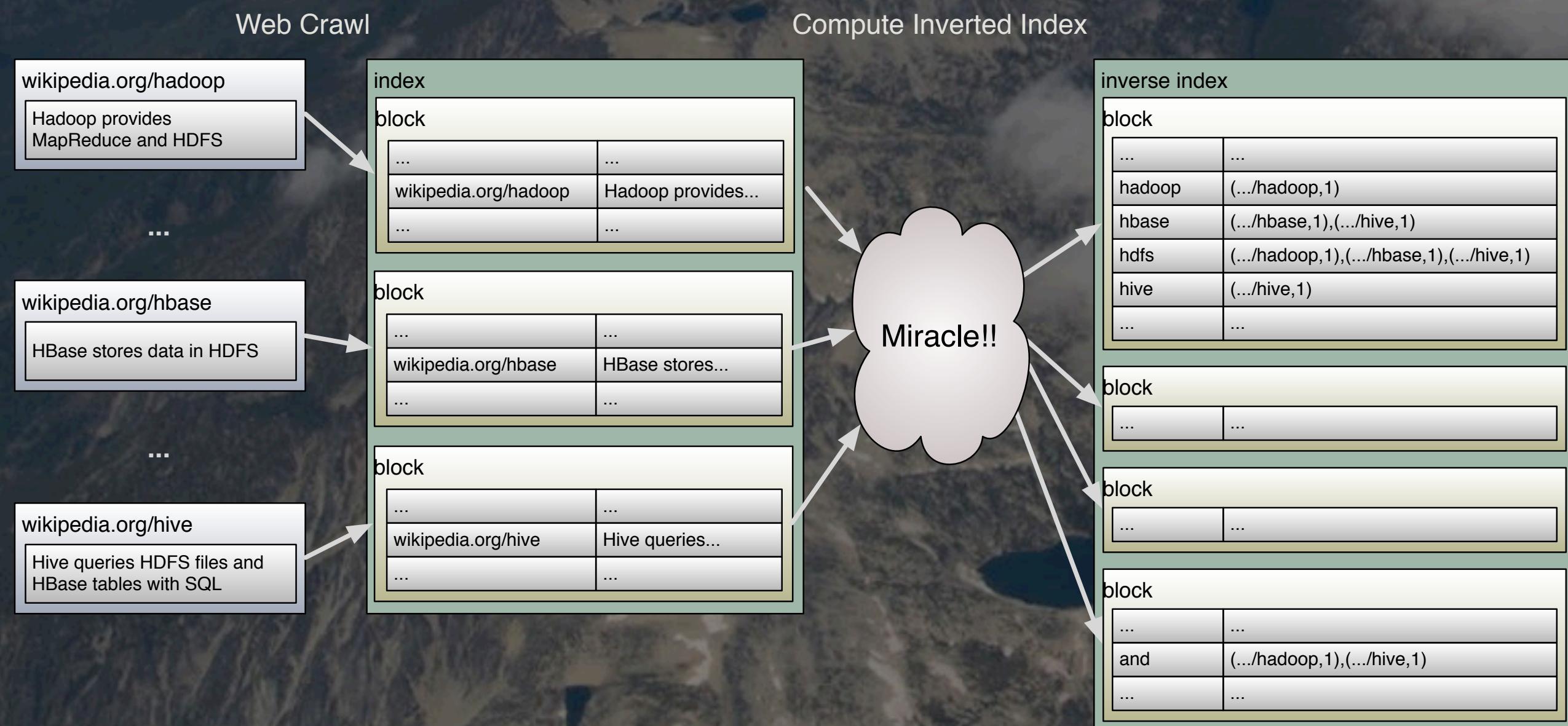
Web Crawl

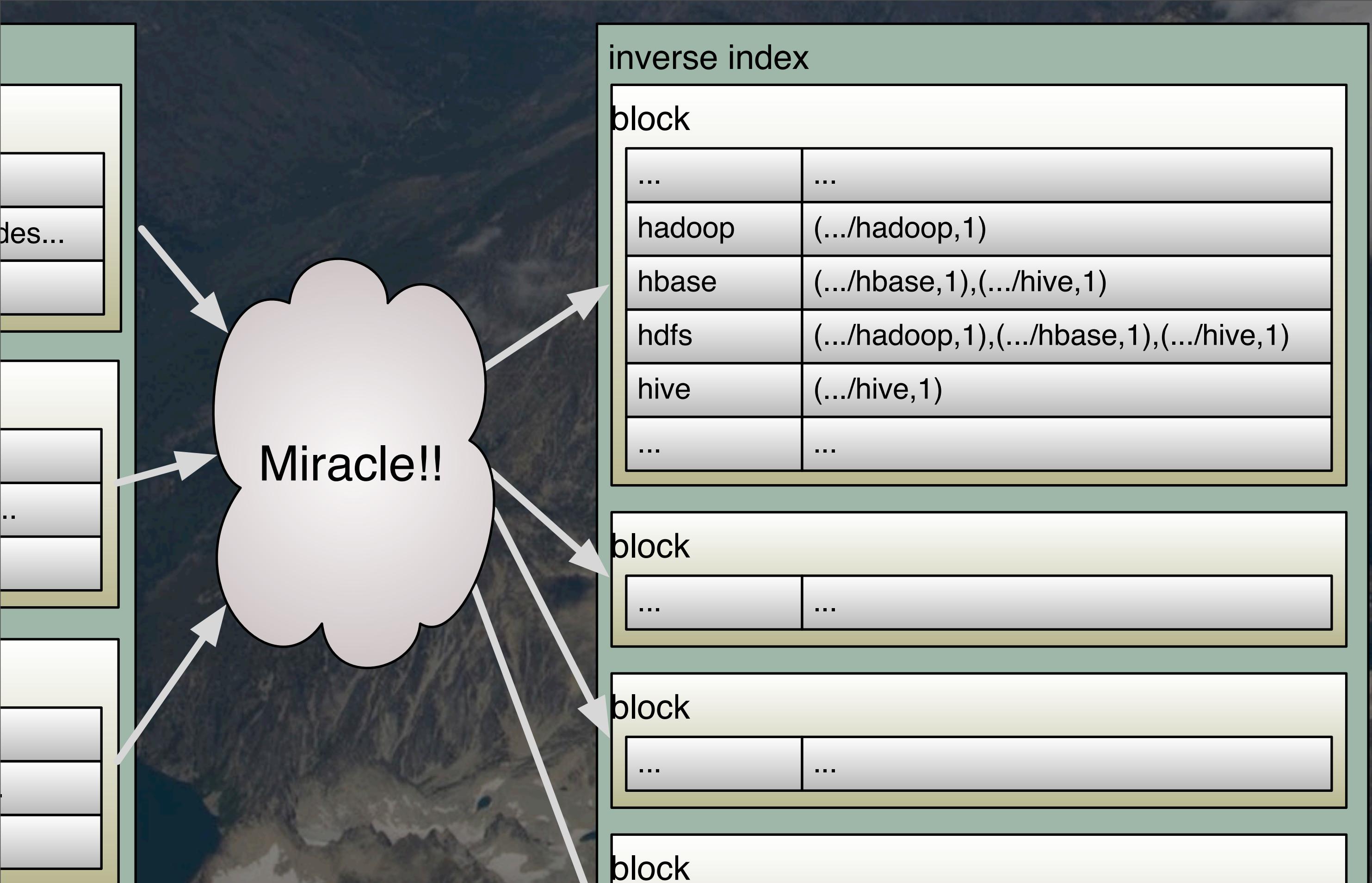
Comput



Thursday, November 12, 15

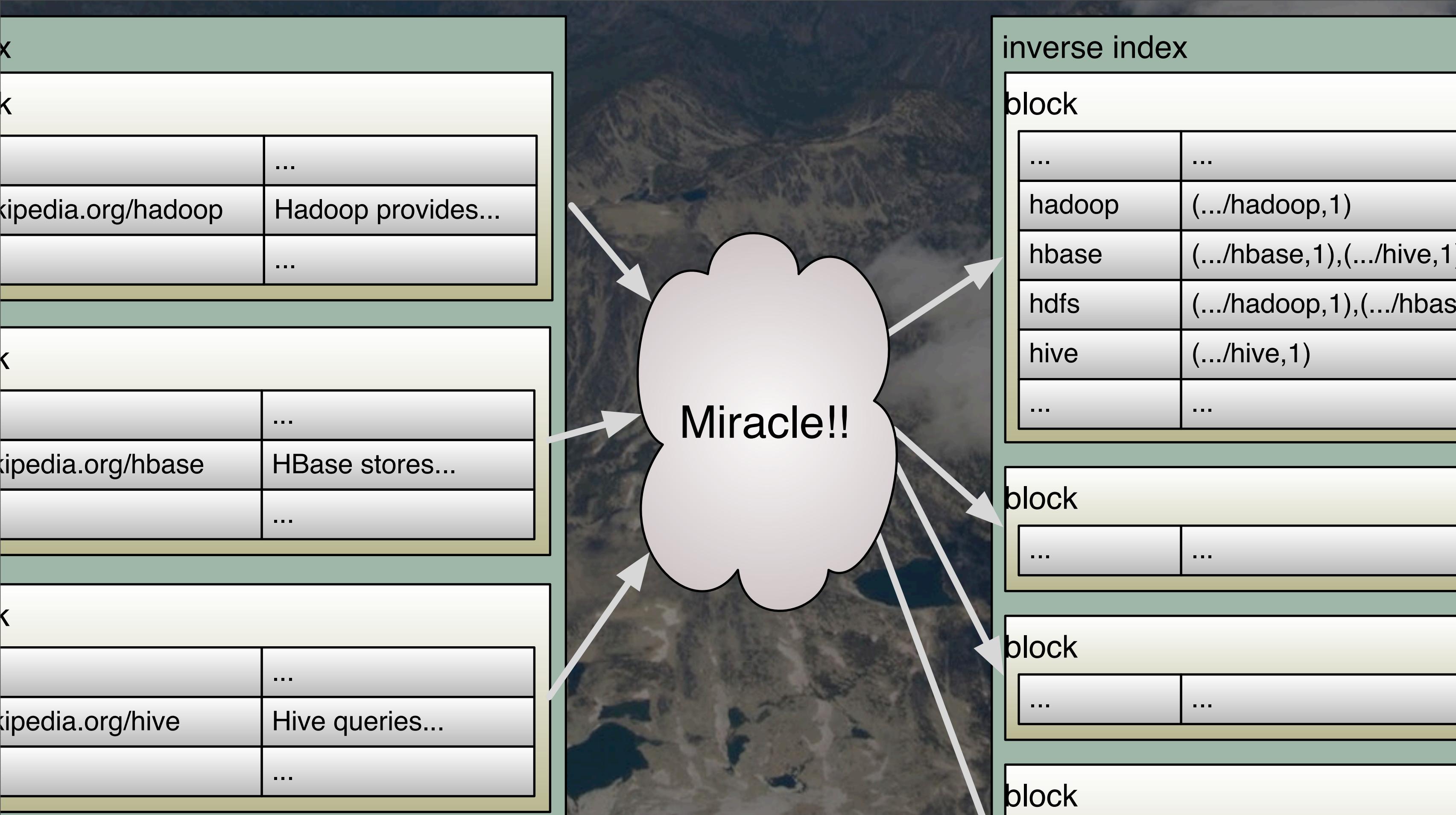
Zoom into details. The initial web crawl produces this two-field data set, with the document id (e.g., the URL, and the contents of the document, possibly cleaned up first, e.g., removing HTML tags).





Thursday, November 12, 15

Zoom into details. This is the output we expect, a two-column dataset with word keys and a list of tuples with the doc id and count for that document.



Thursday, November 12, 15

I won't explain how the "miracle" is implemented in MapReduce, for time's sake, but it's covered in the bonus slides.

Problems

Hard to implement
algorithms...

19

Thursday, November 12, 15

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. Developers need a lot more “canned” primitive operations with which to construct data flows.

Another problem is that many algorithms, especially graph traversal and machine learning algos, which are naturally iterative, simply can’t be implemented using MR due to the performance overhead. People “cheated”; used MR as the framework (“main”) for running code, then hacked iteration internally.

Java MapReduce Inverted Index

```
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"));
    }
}
```

20

Thursday, November 12, 15

This is an implementation of the “inverted index” with MapReduce. Actually, it omits some useful features, such as ordering the list of documents for each word by the count descending, as you would want. This program takes a few hours to write, test, etc. assuming you already know the API and the idioms for using it, because of all the low-level, tedious details. It’s a relatively simple algorithm, so imagine doing something more complicated.

```
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(LineIndexMapper.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()) {
```

22

Thursday, November 12, 15

Here's the mapper class, again with lots of boilerplate and MapReduce-specific types and idiosyncrasies.

```
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);
    }
}
```

Actual business logic.

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

Actual business logic.



Higher Level Tools?

25

Thursday, November 12, 15

Well, can we implement higher level tools?



```
CREATE TABLE students (
    name STRING, age INT, gpa FLOAT);
LOAD DATA ...;
...
SELECT name FROM students;
```



```
A = LOAD 'students' USING PigStorage()  
    AS (name:chararray, age:int, gpa:float);  
B = FOREACH A GENERATE name;  
DUMP B;
```

Trivial Pig example. It's basically the same as the Hive example.

Pig is a dataflow language that's more expressive than SQL, but not Turing complete. So, you have to know how to write UDFs for it, but at least you can use several supported languages.



Cascading (Java)

MapReduce

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

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

29

Thursday, November 12, 15

Trivial Pig example. It's basically the same as the Hive example.

Problems

Only “Batch mode”;
What about streaming?

30

Thursday, November 12, 15

Another MapReduce problem: event stream processing is increasingly important, both because some systems have tight SLAs and because there is a competitive advantage to minimizing the time between data arriving and information being extracted from it, even when otherwise a batch-mode analysis would suffice. MapReduce doesn't support it and neither can Scalding or Cascading, since they are based on MR (although MR is being replaced with alternatives as we speak...).

Problems

Performance needs
to be better

31

Thursday, November 12, 15

Another MapReduce problem: performance is not good.

Spark



32

Thursday, November 12, 15

Spark is a wholesale replacement for MapReduce that leverages lessons learned from MapReduce. The Hadoop community realized that a replacement for MR was 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 have followed suit.

Productivity?

Very concise, elegant, functional APIs.

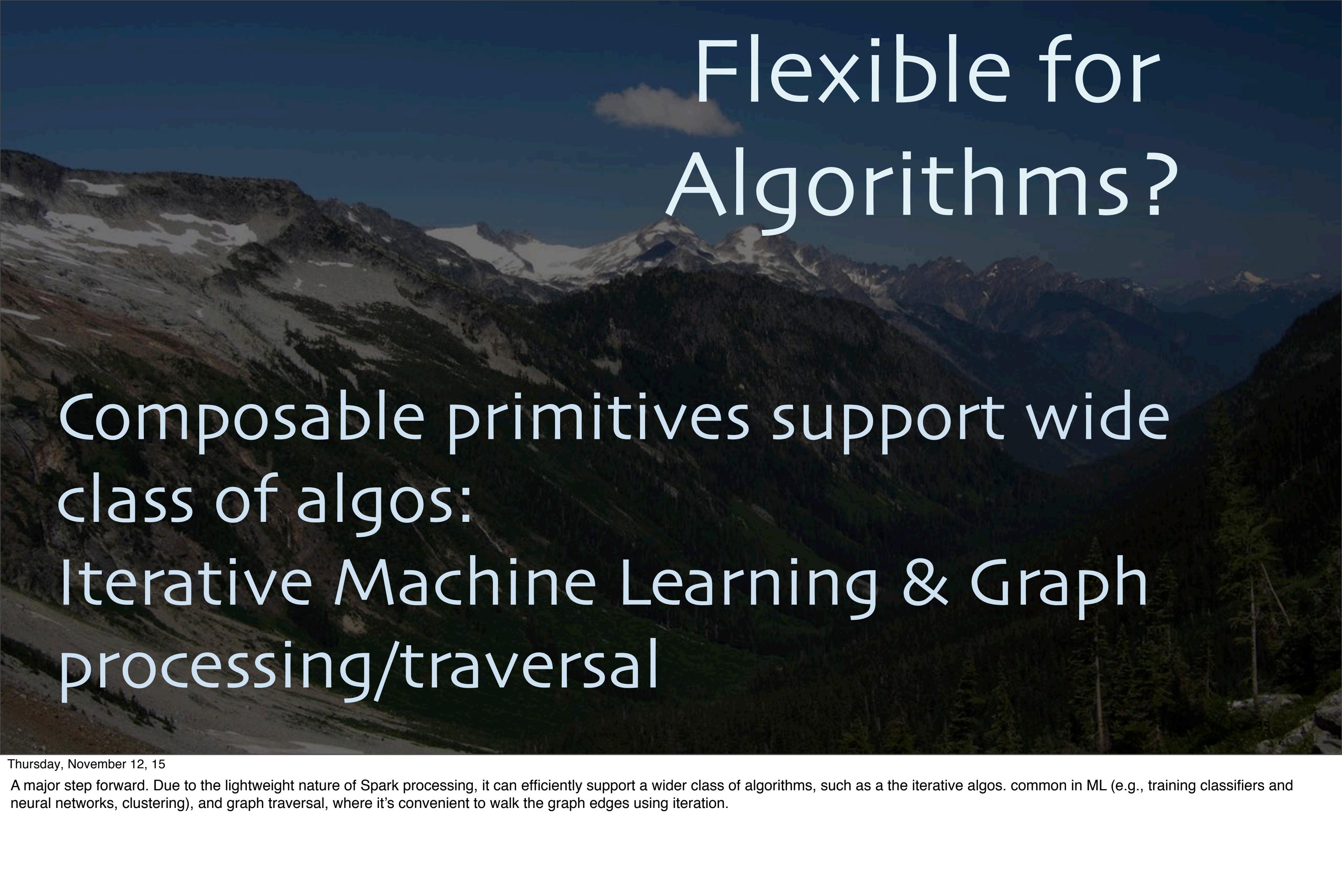
- Python, R
- Scala, Java
- ... and SQL!

Productivity?

Interactive shell (REPL)

- Scala, Python, R, and SQL

Notebooks (iPython/Jupyter-like)



Flexible for Algorithms?

Composable primitives support wide class of algos:
Iterative Machine Learning & Graph processing/traversal

35

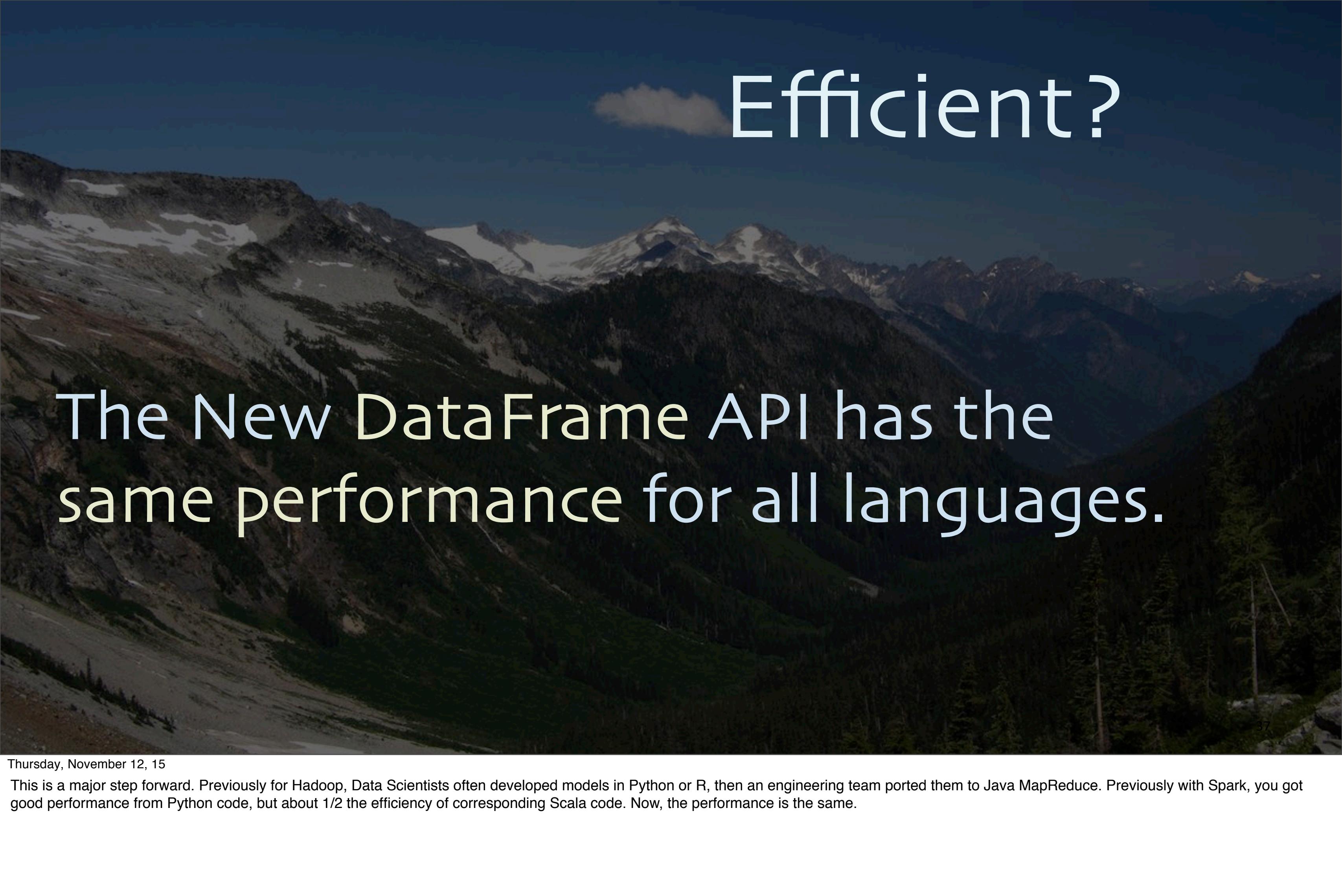
Thursday, November 12, 15

A major step forward. Due to the lightweight nature of Spark processing, it can efficiently support a wider class of algorithms, such as the iterative algos. common in ML (e.g., training classifiers and neural networks, clustering), and graph traversal, where it's convenient to walk the graph edges using iteration.

Efficient?

Builds a dataflow DAG:

- Combines steps into “stages”
- Can cache intermediate data

The background of the slide is a photograph of a mountain range. In the foreground, there's a steep slope covered in brownish vegetation and patches of snow. The middle ground shows more forested slopes leading up to a range of mountains with prominent peaks. One peak is heavily covered in snow, while others have patches of snow at their summits. The sky is a clear, pale blue with a single, wispy white cloud.

Efficient?

The New DataFrame API has the same performance for all languages.

37

Thursday, November 12, 15

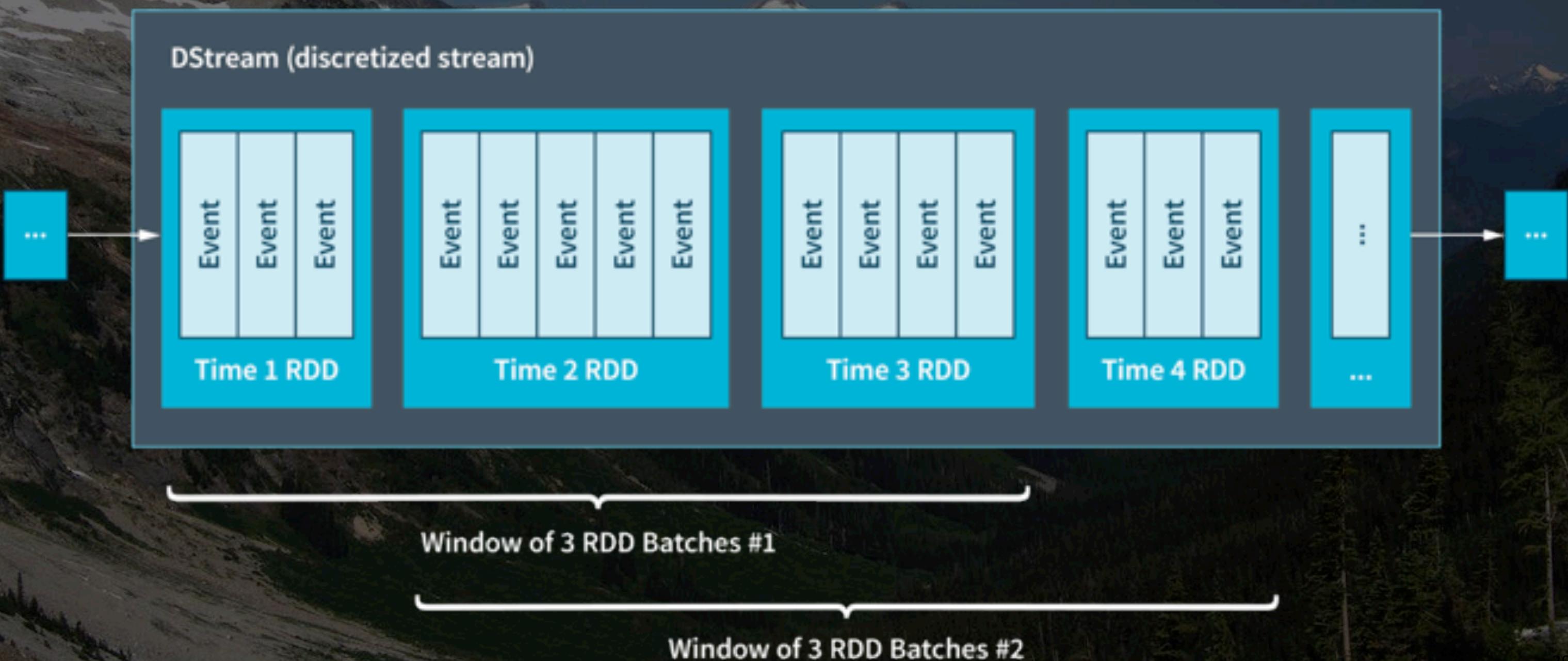
This is a major step forward. Previously for Hadoop, Data Scientists often developed models in Python or R, then an engineering team ported them to Java MapReduce. Previously with Spark, you got good performance from Python code, but about 1/2 the efficiency of corresponding Scala code. Now, the performance is the same.

Batch + Streaming?

Streams - “mini batch” processing:

- Reuses “batch” code
- Adds “window” functions

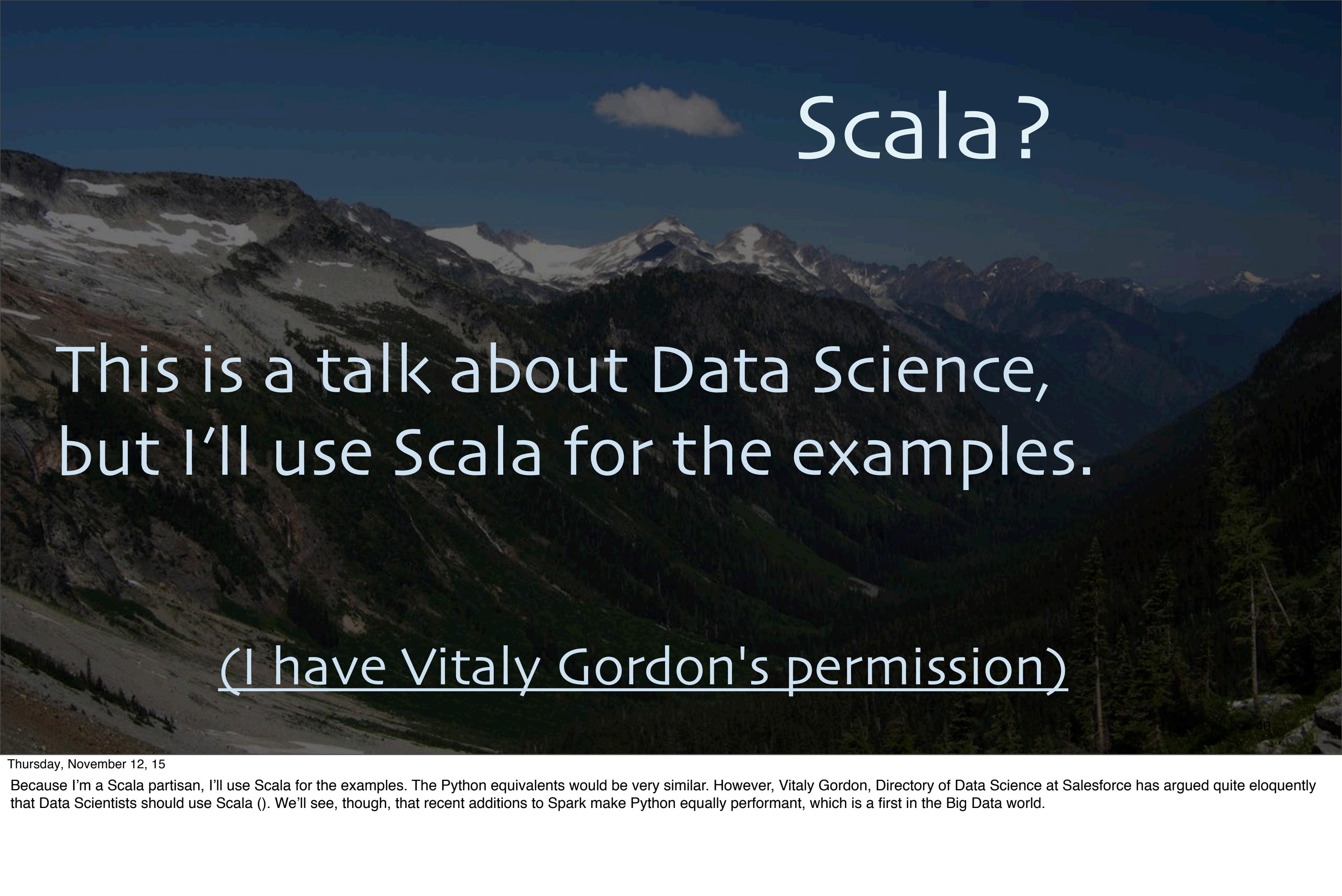
RDDs & DStreams



39

Thursday, November 12, 15

The core concept is a Resilient Distributed Dataset, a partitioned collection. They are resilient because if one partition is lost, Spark knows the lineage and can reconstruct it. For streaming, one of these is created per batch iteration, with a DStream (discretized stream) holding all of them, which supports window functions.



Scala?

This is a talk about Data Science,
but I'll use Scala for the examples.

(I have Vitaly Gordon's permission)

40

Thursday, November 12, 15

Because I'm a Scala partisan, I'll use Scala for the examples. The Python equivalents would be very similar. However, Vitaly Gordon, Directory of Data Science at Salesforce has argued quite eloquently that Data Scientists should use Scala (). We'll see, though, that recent additions to Spark make Python equally performant, which is a first in the Big Data world.

```
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) =>
```

41

Thursday, November 12, 15

This implementation is more sophisticated than the MR and Scalding example. It also computes the count/document of each word. Hence, there are more steps. 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.

```
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) =>
```

42

Thursday, November 12, 15

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.

```
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) =>
```

43

Thursday, November 12, 15

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

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

44

Thursday, November 12, 15

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

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

```
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 {
```

```

    }
}

.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  (n1, n2) => n1 + n2
}
.map {
  case ((word, path), n) => (word, (path, n))
}
.groupByKey
.mapValues { iter =>
  iter.toSeq.sortBy {
    case (path, n) => (-n, path)
  }.mkString(", ")
}

```

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

```

    }
}

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

.map {
  case ((word, path), n) => (word, (path, n))
}
.groupByKey
.mapValues { iter =>
  iter.toSeq.sortBy {
    case (path, n) => (-n, path)
  }.mkString(", ")
}

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

```

```

    .map {
      case ((word, path), n) => (word, (path, n))
    }
    .groupByKey
    .mapValues { iter =>
      iter.toSeq.sortBy { case (path, n) => (-n, path) }
        (word, Seq((path1, n1), (path2, n2), (path3, n3), ...))
    }
    .mkString(", ")
  }
  .saveAsTextFile("output/inverted-index")
}

sc.stop()
}
}

```

```

.map {
  case ((word, path), n) => (word, (path, n))
}
.groupByKey
.mapValues { iter =>
  iter.toSeq.sortBy {
    case (path, n) => (-n, path)
  }.mkString(", ")
}
.saveAsTextFile("output/inverted-index")

sc.stop()
}
}

```

(word, "(path4, 80), (path19, 51), (path8, 12), ...")
...

```
.map {  
    case ((word, path), n) => (word, (path, n))  
}  
.groupByKey  
.mapValues { iter =>  
    iter.toSeq.sortBy {  
        case (path, n) => (-n, path)  
    }.mkString(", ")  
}  
.saveAsTextFile("output/inverted-index")  
  
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 {
        (n1, n2) => n1 + n2
      }
      .map {
        case ((word, path), n) => (word, (path, n))
      }
      .groupByKey
      .mapValues { iter =>
        iter.toSeq.sortBy {
          case (path, n) => (-n, path)
        }.mkString(", ")
      }
      .saveAsTextFile(argz.outpath)

    sc.stop()
  }
}
```

Altogether

```
word -> (word, path)
}
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  (n1, n2) => n1 + n2
}
.map {
  case ((word, path), n) => (word, (path, n))
}
.groupByKey
.mapValues { iter =>
  iter.toSeq.sortBy {
    case (path, n) => (-n, path)
  }.mkString(" ")
}
```

Powerful,
beautiful
combinators

52

Thursday, November 12, 15

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



The larger ecosystem

53

Thursday, November 12, 15

Okay, I love me some Scala, but what about SQL? What about other models, like graphs?

SQL Revisited



54

Spark SQL

- Integrates with Hive
- Use Spark's own SQL dialect
- Use the new DataFrame API

Spark SQL

- Integrates with Hive
- Use Spark's own SQL dialect
- Use the new DataFrame API

```
import org.apache.spark.sql.hive._

val sc = new SparkContext(...)
val sqlc = new HiveContext(sc)

sqlc.sql(
"CREATE TABLE wc (word STRING, count INT)").show()

sqlc.sql("""
LOAD DATA LOCAL INPATH '/path/to/wc.txt'
INTO TABLE wc""").show()

sqlc.sql("""
SELECT * FROM wc
ORDER BY count DESC""").show()
```

```
import org.apache.spark.sql.hive._

val sc = new SparkContext(...)
val sqlc = new HiveContext(sc)

sqlc.sql(
"CREATE TABLE wc (word STRING, count INT)").show()

sqlc.sql("""
LOAD DATA LOCAL INPATH '/path/to/wc.txt'
INTO TABLE wc""").show()

sqlc.sql("""
SELECT * FROM wc
ORDER BY count DESC""").show()
```

```
import org.apache.spark.sql.hive._

val sc = new SparkContext(...)
val sqlc = new HiveContext(sc)

sqlc.sql(
"CREATE TABLE wc (word STRING, count INT)").show()

sqlc.sql("""
LOAD DATA LOCAL INPATH '/path/to/wc.txt'
INTO TABLE wc""").show()

sqlc.sql("""
SELECT * FROM wc
ORDER BY count DESC""").show()
```

1. Create a Hive table for Word Count data.
2. Load text data into the table. (We'll assume the file format is already in Hive's preferred format for text, but if not, that's easy to fix...)
3. Sort by count descending and "show" the first 20 records (show is normally used only in interactive sessions).

- Prefer Python??

- Just replace:

```
import org.apache.spark.sql.hive._
```

- With this:

```
from pyspark.sql import HiveContext
```

- and delete the vals.

- Prefer Just HiveQL??
 - Use spark-sql shell script:

```
CREATE TABLE wc (word STRING, count INT);
```

```
LOAD DATA LOCAL INPATH '/path/to/wc.txt'  
INTO TABLE wc;
```

```
SELECT * FROM wc  
ORDER BY count DESC;
```

Spark SQL

- Integrates with Hive
- Use Spark's own SQL dialect
- Use the new DataFrame API

```
import org.apache.spark.sql._

val sc = new SparkContext(...)
val sqlc = new SQLContext(sc)

val df = sqlc.load("/path/to/wc.parquet")

val ordered_df = df.orderBy($"count".desc)
ordered_df.show()
ordered_df.cache()

val long_words =
  ordered_df.filter($"word".length > 20)
long_words.save("/path/to/long_words.parquet")
```

63

Thursday, November 12, 15

Rather than look at Spark's dialect of SQL, let's look at DataFrames, a new feature that's built on the SQL engine (e.g., the query optimizer, called Catalyst). They are inspired by Python Pandas' and R's concepts of data frames.

I won't discuss the Python differences, but the code is close to this, as before.

```
import org.apache.spark.sql._

val sc = new SparkContext(...)
val sqlc = new SQLContext(sc)

val df = sqlc.load("/path/to/wc.parquet")

val ordered_df = df.orderBy($"count".desc)
ordered_df.show()
ordered_df.cache()

val long_words =
  ordered_df.filter($"word".length > 20)
long_words.save("/path/to/long_words.parquet")
```

64

Thursday, November 12, 15

Note that we use SQLContext, not HiveContext this time, but in fact either works, because HiveContext is a subclass of SQLContext. So, if you need to work with Hive tables and SparkSQL DataFrames, that's fine.

```
import org.apache.spark.sql._

val sc = new SparkContext(...)
val sqlc = new SQLContext(sc)

val df = sqlc.load("/path/to/wc.parquet")

val ordered_df = df.orderBy($"count".desc)
ordered_df.show()
ordered_df.cache()

val long_words =
  ordered_df.filter($"word".length > 20)
long_words.save("/path/to/long_words.parquet")
```

```
import org.apache.spark.sql._

val sc = new SparkContext(...)
val sqlc = new SQLContext(sc)

val df = sqlc.load("/path/to/wc.parquet")

val ordered_df = df.orderBy($"count".desc)
ordered_df.show()
ordered_df.cache()

val long_words =
  ordered_df.filter($"word".length > 20)
long_words.save("/path/to/long_words.parquet")
```

66

Thursday, November 12, 15

Order by the counts descending. The idiomatic `$"count".desc` is one of several ways to specify the name of the column of interest and, in this case, specify descending sort.
Show prints out 20 records with column headers. Used mostly for interactive sessions.
The cache call tells Spark to save this data set because we'll reuse it over and over. Otherwise, Spark will go back through the ancestor DAG of RDDs/DataFrames to recompute it each time.

```
import org.apache.spark.sql._

val sc = new SparkContext(...)
val sqlc = new SQLContext(sc)

val df = sqlc.load("/path/to/wc.parquet")

val ordered_df = df.orderBy($"count".desc)
ordered_df.show()
ordered_df.cache()

val long_words =
  ordered_df.filter($"word".length > 20)
long_words.save("/path/to/long_words.parquet")
```

67

Thursday, November 12, 15

Find all the words of length > 20 .

Save to the file system as Parquet files.

Machine Learning

MLlib



Thursday, November 12, 15

A big attraction of Big Data is the hope that Machine Learning will extract \$\$\$ from data. Spark's features make scalable ML libraries possible, and MLlib is a growing collection of them.

Streaming KMeans Example



Thursday, November 12, 15

<https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/mllib/StreamingKMeansExample.scala> and <https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/StreamingKMeans.scala> Since both streaming and ML are hot, let's use them together. Spark has 3 built-in libraries for streaming ML. The others are for linear and logistic regression.

Compute clusters iteratively in a dataset as it streams into the system. On a second stream, use those clusters to make predictions.

```
import ...spark.mllib.clustering.StreamingKMeans
import ...spark.mllib.linalg.Vectors
import ...spark.mllib.regression.LabeledPoint
import ...spark.streaming.{  
    Seconds, StreamingContext}  
  
val sc = new SparkContext(...)  
val ssc = new StreamingContext(sc, Seconds(10))  
  
val trainingData = ssc.textFileStream(...)  
.map(Vectors.parse)  
val testData = ssc.textFileStream(...)  
.map(LabeledPoint.parse)  
  
val model = new StreamingKMeans()  
    .setK(10)  
    .setPartitions(10)
```

70

Thursday, November 12, 15

Many details omitted for brevity. See the Spark distributions examples for the full source listing:

<https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/mllib/StreamingKMeansExample.scala>

```
import ...spark.mllib.clustering.StreamingKMeans
import ...spark.mllib.linalg.Vectors
import ...spark.mllib.regression.LabeledPoint
import ...spark.streaming.{  
    Seconds, StreamingContext}
```

```
val sc = new SparkContext(...)  
val ssc = new StreamingContext(sc, Seconds(10))  
  
val trainingData = ssc.textFileStream(...)  
.map(Vectors.parse)  
val testData = ssc.textFileStream(...)  
.map(LabeledPoint.parse)  
  
val model = new StreamingKMeans()  
    .setK(3).setPartitions(1)
```

```
import ...spark.mllib.clustering.StreamingKMeans
import ...spark.mllib.linalg.Vectors
import ...spark.mllib.regression.LabeledPoint
import ...spark.streaming.{  
    Seconds, StreamingContext}
```

```
val sc = new SparkContext(...)  
val ssc = new StreamingContext(sc, Seconds(10))
```

```
val trainingData = ssc.textFileStream(...)  
.map(Vectors.parse)
```

```
val testData = ssc.textFileStream(...)  
.map(LabeledPoint.parse)
```

```
val model = new StreamingKMeans()  
    .setK(3).setPartitions(3)
```

72

```
import ...spark.mllib.clustering.StreamingKMeans
import ...spark.mllib.linalg.Vectors
import ...spark.mllib.regression.LabeledPoint
import ...spark.streaming.{  
    Seconds, StreamingContext}
```

```
val sc = new SparkContext(...)  
val ssc = new StreamingContext(sc, Seconds(10))
```

```
val trainingData = ssc.textFileStream(...)  
.map(Vectors.parse)  
val testData = ssc.textFileStream(...)  
.map(LabeledPoint.parse)
```

```
val model = new StreamingKMeans()  
    .setK(3).setPartitions(1)
```

73

Thursday, November 12, 15

Set up streams that watch for new text files (with one “record” per line) in the elided (...) directories. One will be for training data and the other for test data (for which we’ll make predictions). As input lines are ingested, they parsed into an MLlib Vector for the training data (doesn’t have a label and Vector is not to be confused with Scala’s Vector type). The test data is labeled.

```
.map(LabeledPoint.parse)
```

```
val model = new StreamingKMeans()  
.setK(K_CLUSTERS)  
.setDecayFactor(1.0)  
.setRandomCenters(N_FEATURES, 0.0)
```

```
val f: LabeledPoint => (Double, Vector) =  
lp => (lp.label, lp.features)
```

```
model.trainOn(trainingData)  
model.predictOnValues(testData.map(f)).print()
```

```
ssc.start()
```

```
ssc.awaitTermination()
```

74

Thursday, November 12, 15

Set up the streaming K-Means model with the number of centroids given by K_CLUSTERS, the decay factor of 1.0 means remember all data seen when computing new centroids (0.0 would mean forget all past data and use only this batch's data). Finally, initialize centroids randomly for feature vectors of N_FEATURES size. The 0.0 is a weight factor used to detect when a small cluster should be dropped.

```
.map(LabeledPoint.parse)
```

```
val model = new StreamingKMeans()  
.setK(K_CLUSTERS)  
.setDecayFactor(1.0)  
.setRandomCenters(N_FEATURES, 0.0)
```

```
val f: LabeledPoint => (Double, Vector) =  
lp => (lp.label, lp.features)
```

```
model.trainOn(trainingData)  
model.predictOnValues(testData.map(f)).print()
```

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

75

Thursday, November 12, 15

This is a Scala named function (as opposed to a method or an anonymous function). It takes in a labeled point and returns a tuple of the label and the vector of features.

```
.map(LabeledPoint.parse)
```

```
val model = new StreamingKMeans()  
.setK(K_CLUSTERS)  
.setDecayFactor(1.0)  
.setRandomCenters(N_FEATURES, 0.0)
```

```
val f: LabeledPoint => (Double, Vector) =  
lp => (lp.label, lp.features)
```

```
model.trainOn(trainingData)  
model.predictOnValues(testData.map(f)).print()
```

```
ssc.start()
```

```
ssc.awaitTermination()
```

76

Thursday, November 12, 15

As the training data arrives on one stream, incrementally train the model.

As the test data comes in on the other stream, map it to the expected (label, features) format then predict the label using the model. The final print() is a debug statement that dumps the first 10 or so results during each batch.

```
.map(LabeledPoint.parse)
```

```
val model = new StreamingKMeans()  
.setK(K_CLUSTERS)  
.setDecayFactor(1.0)  
.setRandomCenters(N_FEATURES, 0.0)
```

```
val f: LabeledPoint => (Double, Vector) =  
lp => (lp.label, lp.features)
```

```
model.trainOn(trainingData)  
model.predictOnValues(testData.map(f)).print()
```

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

77



Graph Processing

GraphX

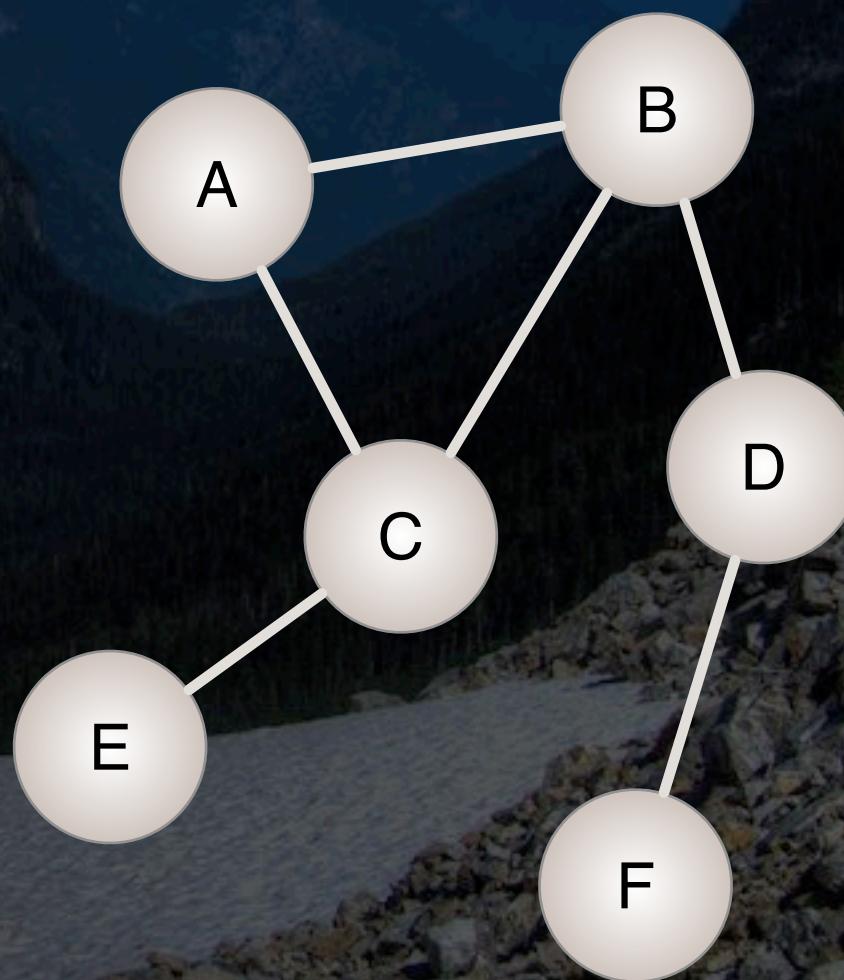
78

Thursday, November 12, 15

Spark's overall efficiently makes it possible to represent "networked" data as a graph structure and use various graph algorithms on it.

GraphX

- Social networks
- Epidemics
- Teh Interwebs
- ...



79

Thursday, November 12, 15

Examples. While modeling many real-world systems is natural with graphs, efficient graph processing over a distributed system has been a challenge, leading people to use ad-hoc implementations for particular problems. Spark is making it possible to implement distributed graphs with reasonable efficiency, so that graph abstractions and algorithms are easier to expose to end users.

```
import scala.collection.mutable
import org.apache.spark._
import ...spark.storage.StorageLevel
import ...spark.graphx._
import ...spark.graphx.lib._
import ...spark.graphx.PartitionStrategy._

val nEdgePartitions = 20
val partitionStrategy =
  PartitionStrategy.CanonicalRandomVertexCut
val edgeStorageLevel = StorageLevel.MEMORY_ONLY
val vertexStorageLevel = StorageLevel.MEMORY_ONLY
val tolerance = 0.001F
val input = "..."
```

```
import scala.collection.mutable  
import org.apache.spark._  
import ...spark.storage.StorageLevel  
import ...spark.graphx._  
import ...spark.graphx.lib._  
import ...spark.graphx.PartitionStrategy._
```

```
val nEdgePartitions = 20  
val partitionStrategy =  
  PartitionStrategy.CanonicalRandomVertexCut  
val edgeStorageLevel = StorageLevel.MEMORY_ONLY  
val vertexStorageLevel = StorageLevel.MEMORY_ONLY  
val tolerance = 0.001F  
val input = "..."
```

```
import scala.collection.mutable
import org.apache.spark._
import ...spark.storage.StorageLevel
import ...spark.graphx._
import ...spark.graphx.lib._
import ...spark.graphx.PartitionStrategy._
```

```
val nEdgePartitions = 20
val partitionStrategy =
  PartitionStrategy.CanonicalRandomVertexCut
val edgeStorageLevel = StorageLevel.MEMORY_ONLY
val vertexStorageLevel = StorageLevel.MEMORY_ONLY
val tolerance = 0.001F
val input = "..."
```

Thursday, November 12, 15

Most of these values would/could be command-line options: the number of partitions to split the graph over, the strategy, how to cache edges and vertices (which are RDDs under the hood; other options include spilling to disk), the tolerance for convergence of PageRank, and the input data location.
Graph partitioning duplicates nodes several times across the cluster, rather than edges. There are several built-in PartitionStrategy values.

```
val tolerance = 0.001  
val input = "..."
```

```
val sc = new SparkContext(...)  
  
val unpartitionedGraph = GraphLoader.edgeListFile(  
    sc, input, numEdgePartitions,  
    edgeStorageLevel, vertexStorageLevel).cache
```

```
val graph = partitionStrategy.foldLeft(  
    unpartitionedGraph)(_.partitionBy(_))  
println("# vertices = " + graph.vertices.count)  
println("# edges = " + graph.edges.count)
```

```
val pr = PageRank.runUntilConvergence(  
    graph, tolerance).vertices.cache()
```

```
val tolerance = 0.001  
val input = "..."
```

```
val sc = new SparkContext(...)
```

```
val unpartitionedGraph = GraphLoader.edgeListFile(  
  sc, input, numEdgePartitions,  
  edgeStorageLevel, vertexStorageLevel).cache
```

```
val graph = partitionStrategy.foldLeft(  
  unpartitionedGraph)(_.partitionBy(_))  
println("# vertices = " + graph.vertices.count)  
println("# edges = " + graph.edges.count)
```

```
val pr = PageRank.runUntilConvergence(  
  graph, tolerance).vertices.cache()
```

```
unpartitionedGraph) (_.partitionBy(_))
println("# vertices = " + graph.vertices.count)
println("# edges = " + graph.edges.count)
```

```
val pr = PageRank.runUntilConvergence(
    graph, tolerance).vertices.cache()
```

```
println("Total rank: " + pr.map(_.value).reduce(_ + _))
```

```
pr.map {
    case (id, r) => id + "\t" + r
}.saveAsTextFile(...)
```

```
sc.stop()
```

```
unpartitionedGraph) (_.partitionBy(_))
println("# vertices = " + graph.vertices.count)
println("# edges = " + graph.edges.count)

val pr = PageRank.runUntilConvergence(
    graph, tolerance).vertices.cache()

println("Total rank: " + pr.map(_.value).reduce(_ + _))

pr.map {
    case (id, r) => id + "\t" + r
}.saveAsTextFile(...)

sc.stop()
```

Thank You!



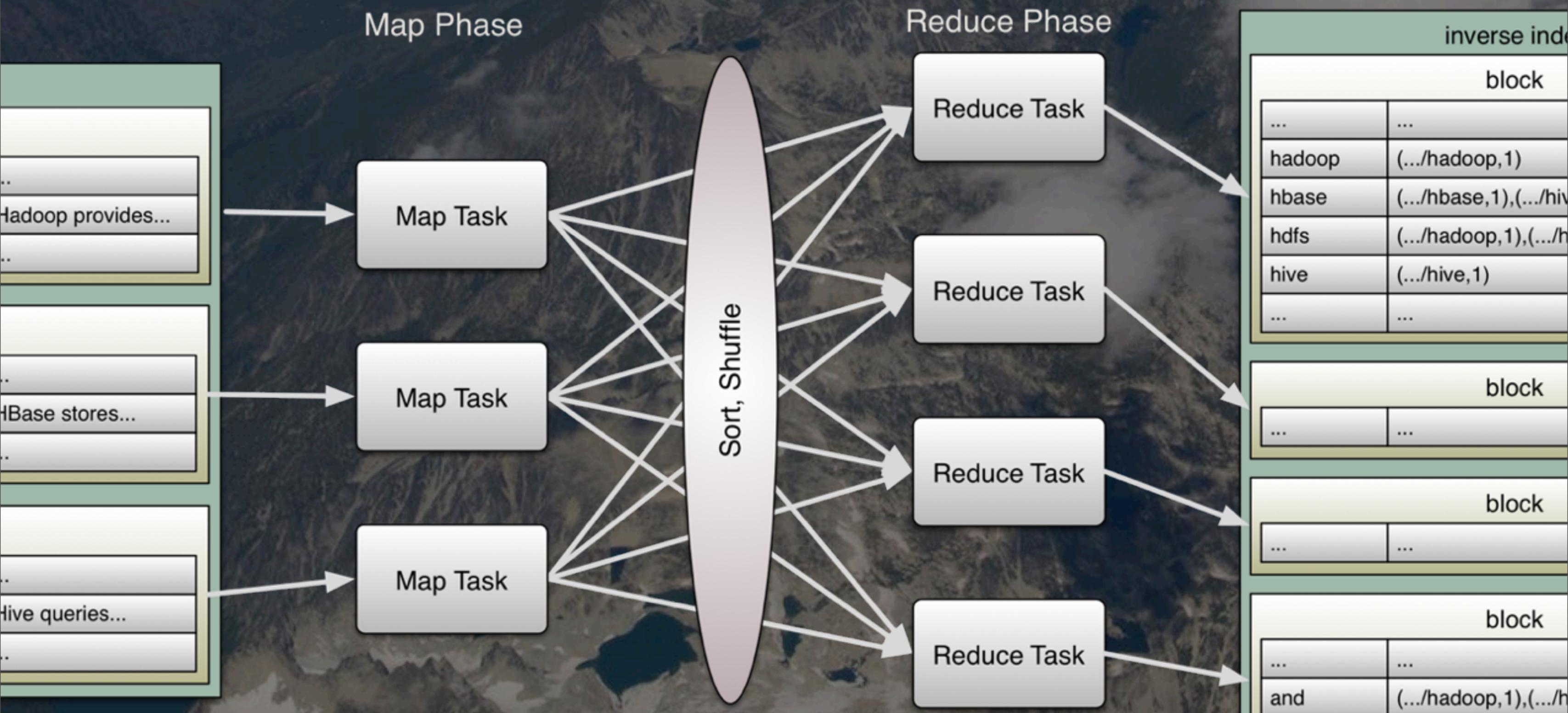
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

Bonus Slides: Details of MapReduce implementation for the Inverted Index



88

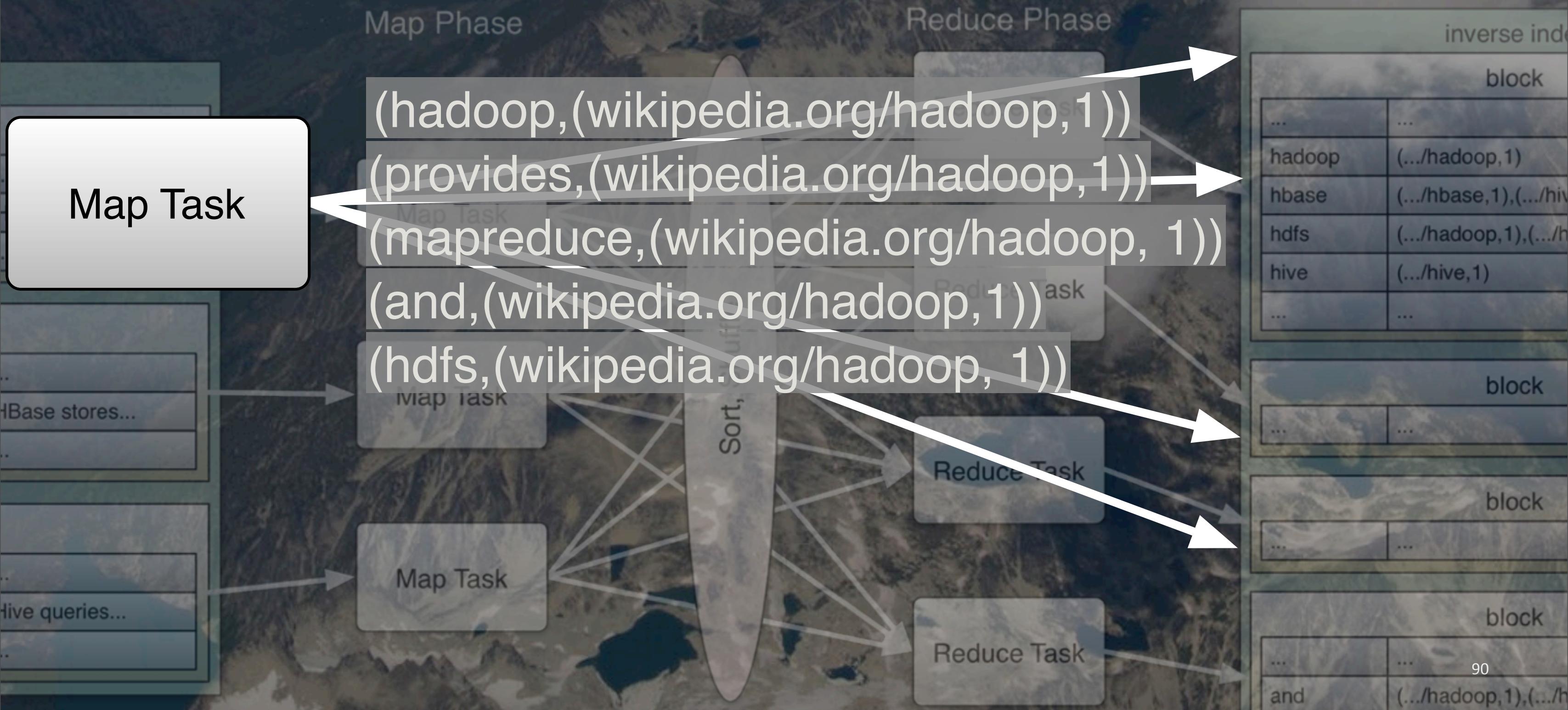
1 Map step + 1 Reduce step



Thursday, November 12, 15

A one-pass MapReduce job can do this calculation. We'll discuss the details.

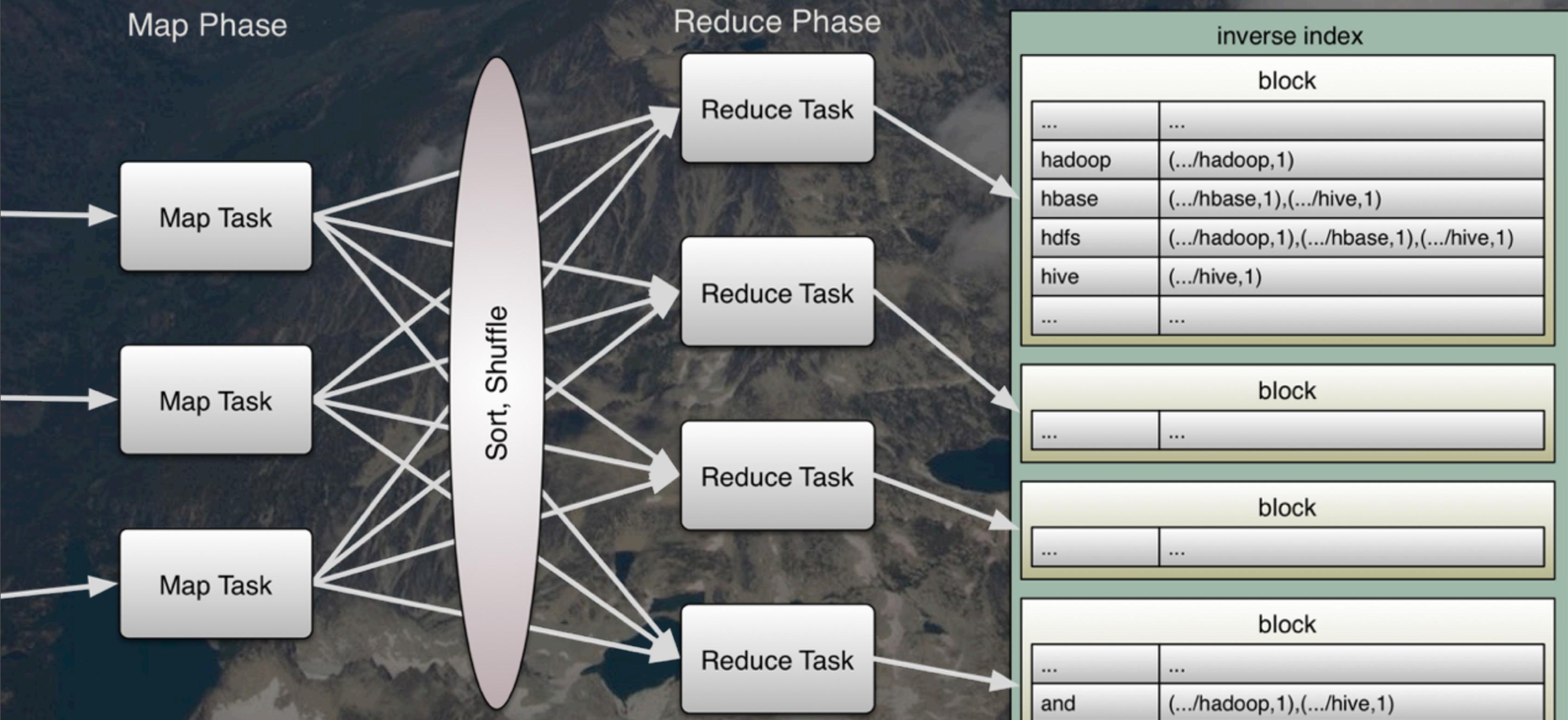
1 Map step + 1 Reduce step



Thursday, November 12, 15

Each map task parses the records. It tokenizes the contents and write new key-value pairs (shown as tuples here), with the word as the key, and the rest, shown here as a second element that is itself a tuple, which holds the document id and the count.

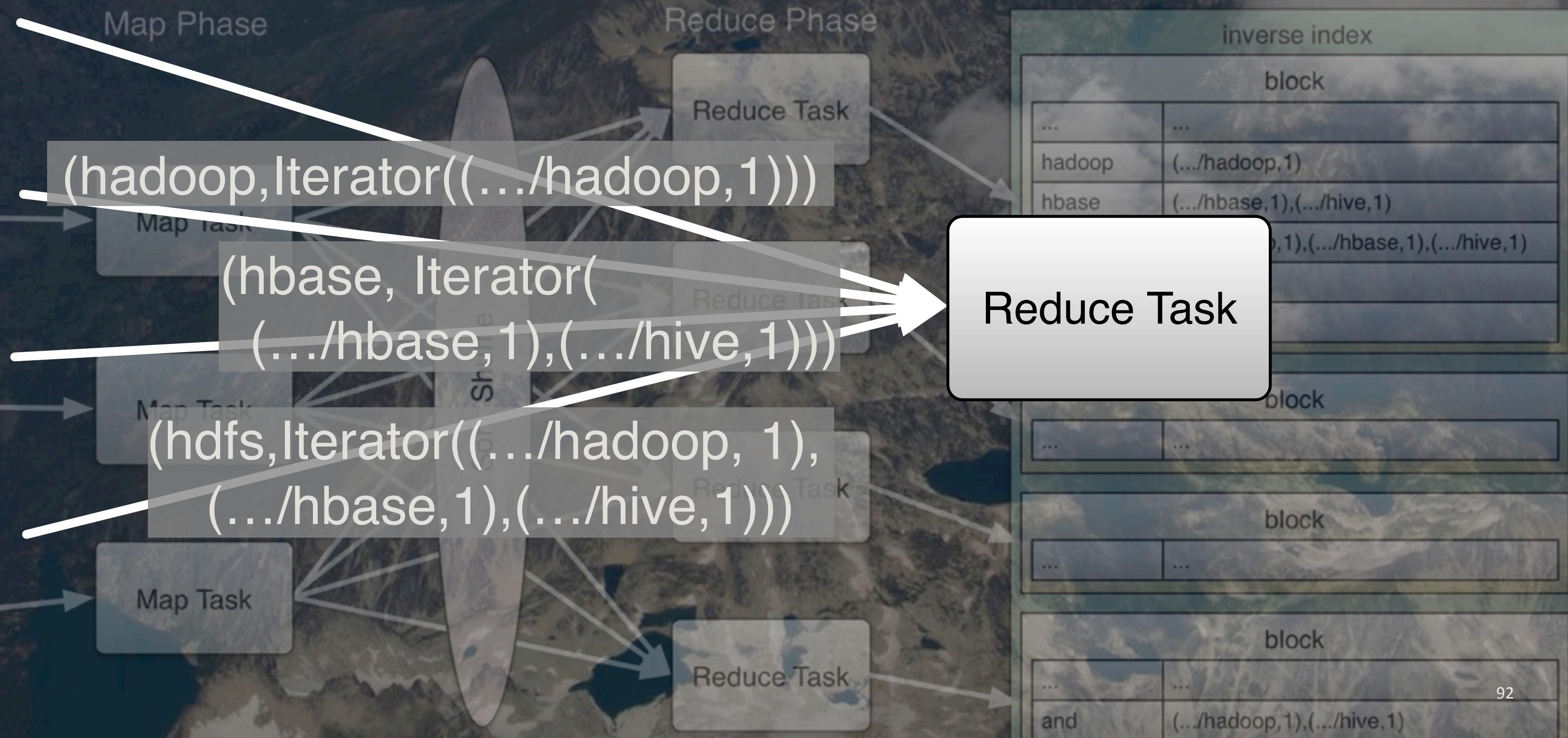
1 Map step + 1 Reduce step



Thursday, November 12, 15

The output key,value pairs are sorted by key within each task and then “shuffled” across the network so that all occurrences of the same key arrives at the same reducer, which will gather together all the results for a given set of keys.

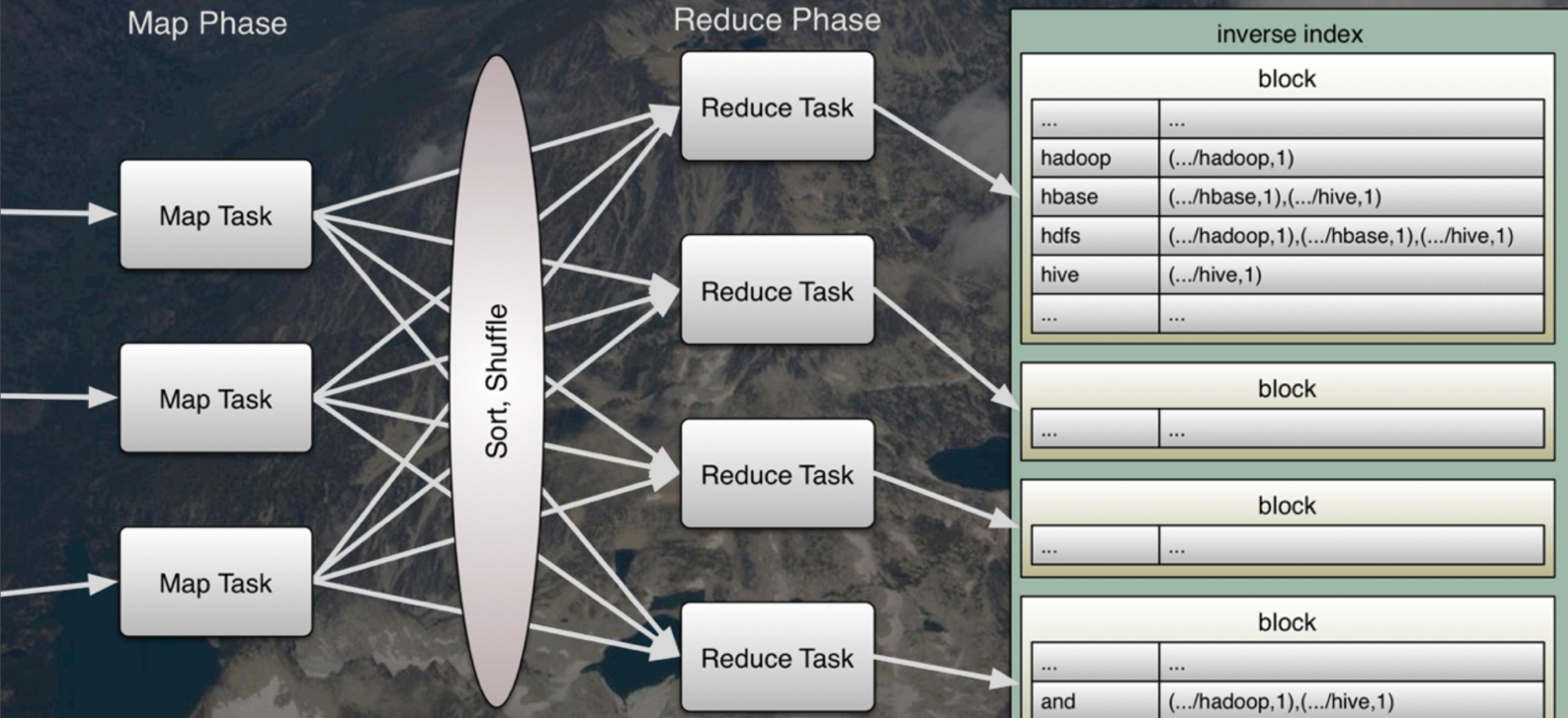
1 Map step + 1 Reduce step



Thursday, November 12, 15

The Reduce input is the key and an iterator through all the (id,count) values for that key.

1 Map step + 1 Reduce step



Thursday, November 12, 15

The output key,value pairs are sorted by key within each task and then “shuffled” across the network so that all occurrences of the same key arrives at the same reducer, which will gather together all the results for a given set of keys.