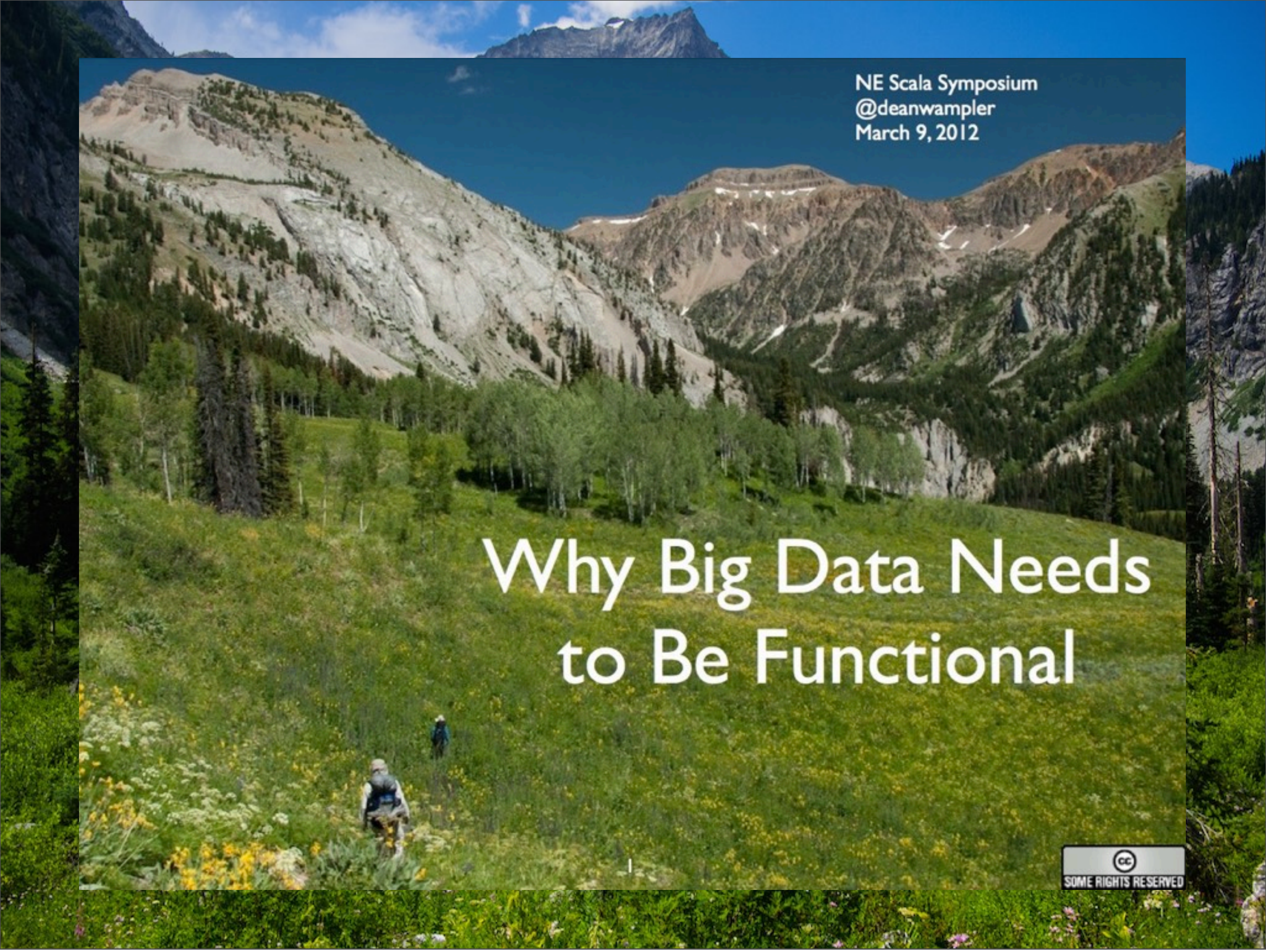
Tuesday, May 12, 15

Tuesday, May 12, 15

Every developer talk should have some XML!!

Tuesday, May 12, 15

My linkedin profile since 2012??



Tuesday, May 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

Algebird

Spire

...

Big Data Tools

Spark

Scalding

Apache Kafka

hadoop

Cassandra

STORM

The Apache Software Foundation

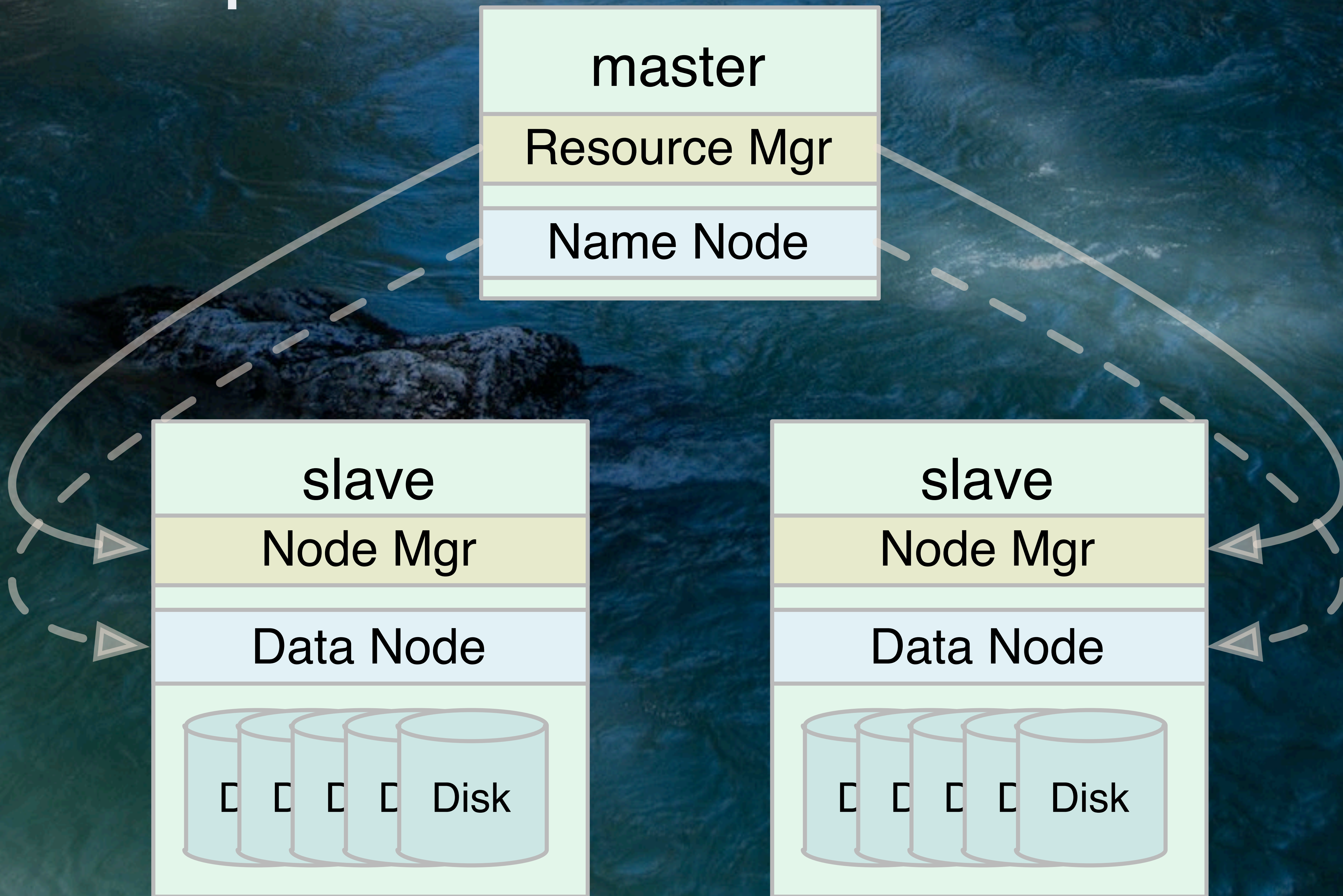
Apache Solr

samza

Tuesday, May 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



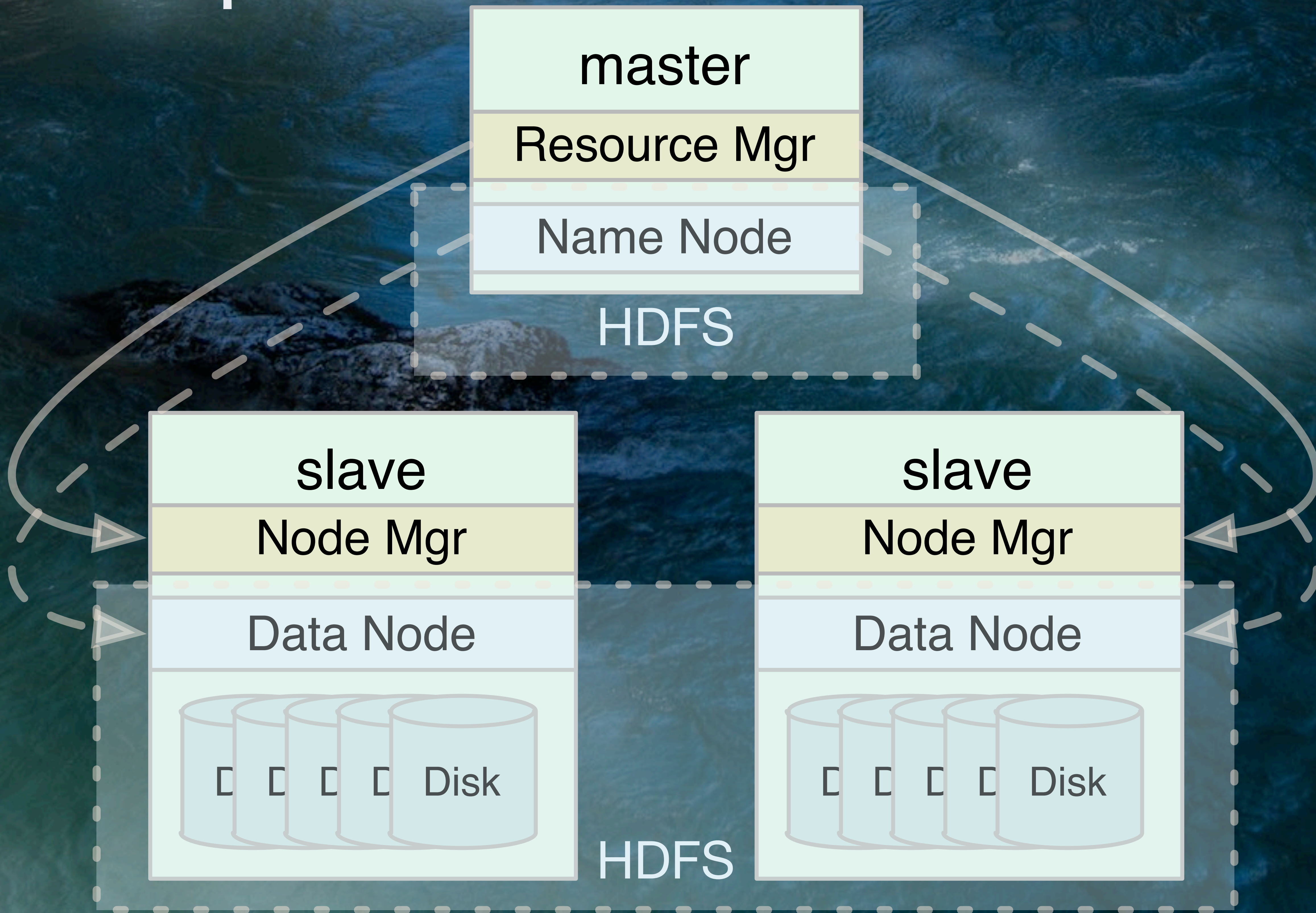
Tuesday, May 12, 15

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

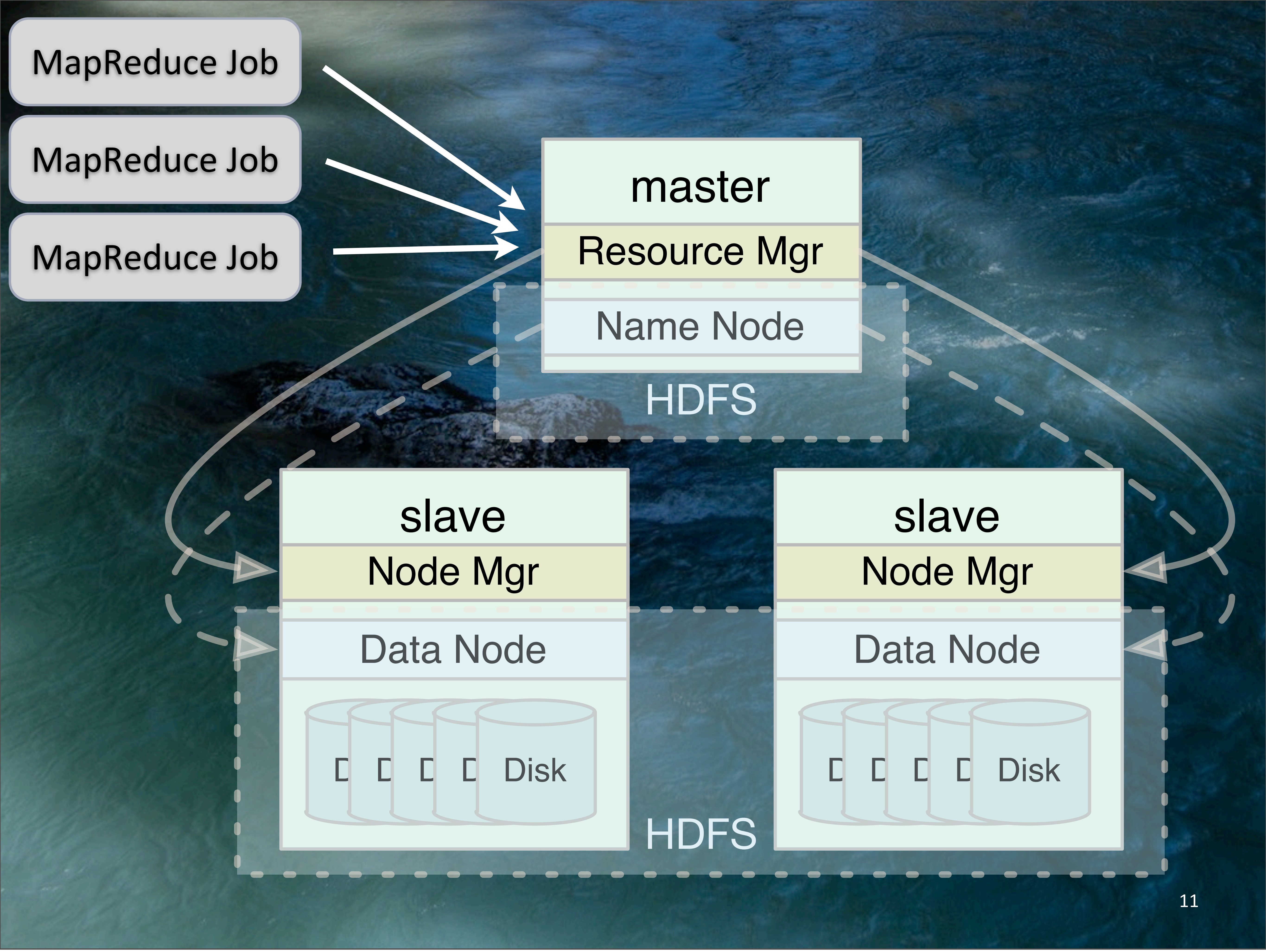


Tuesday, May 12, 15

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.

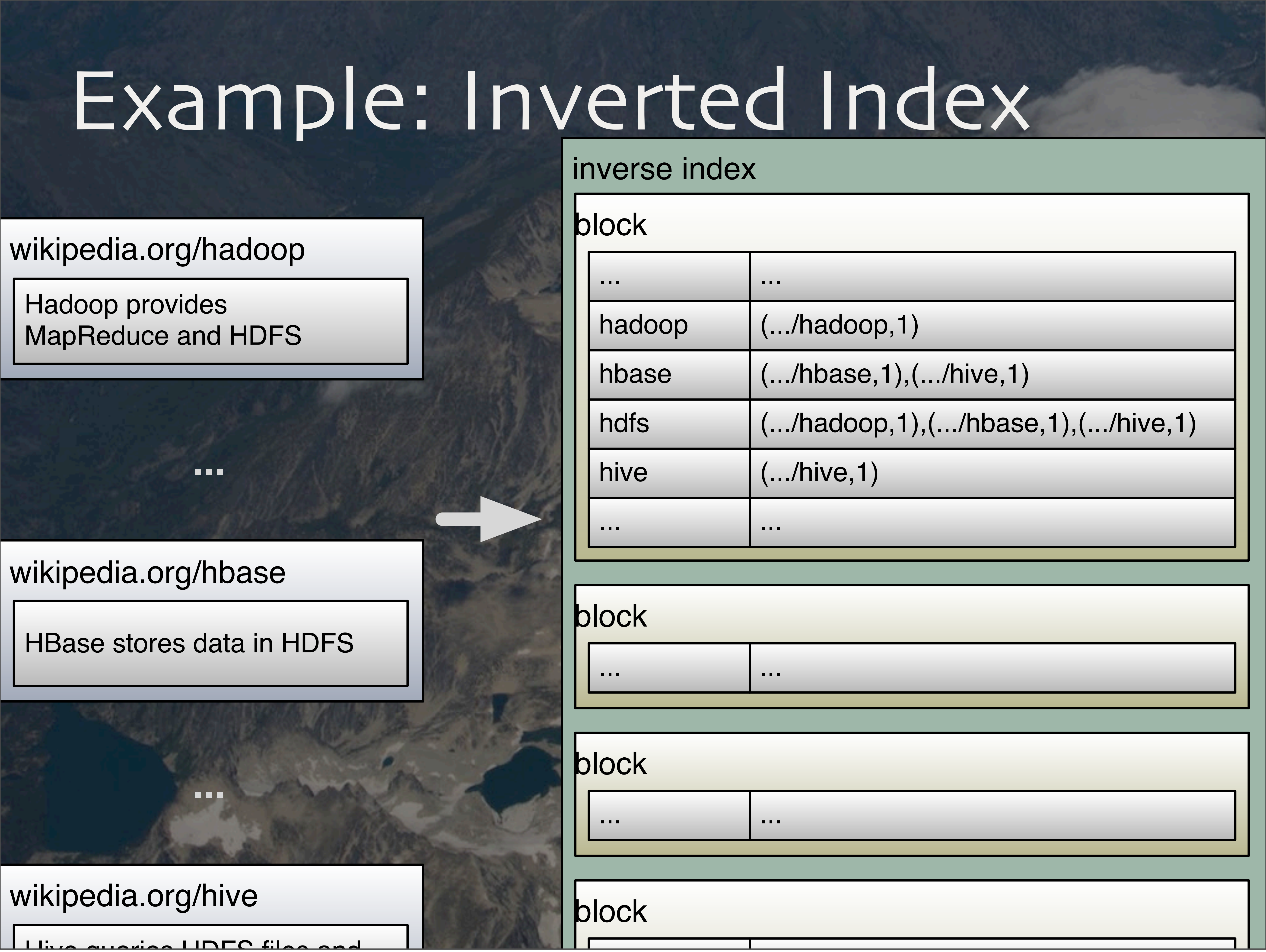
The Resource Manager decomposes each job in to tasks, which are distributed to slave nodes and managed by the Node Managers. There are other services I'm omitting for simplicity.



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

Tuesday, May 12, 15

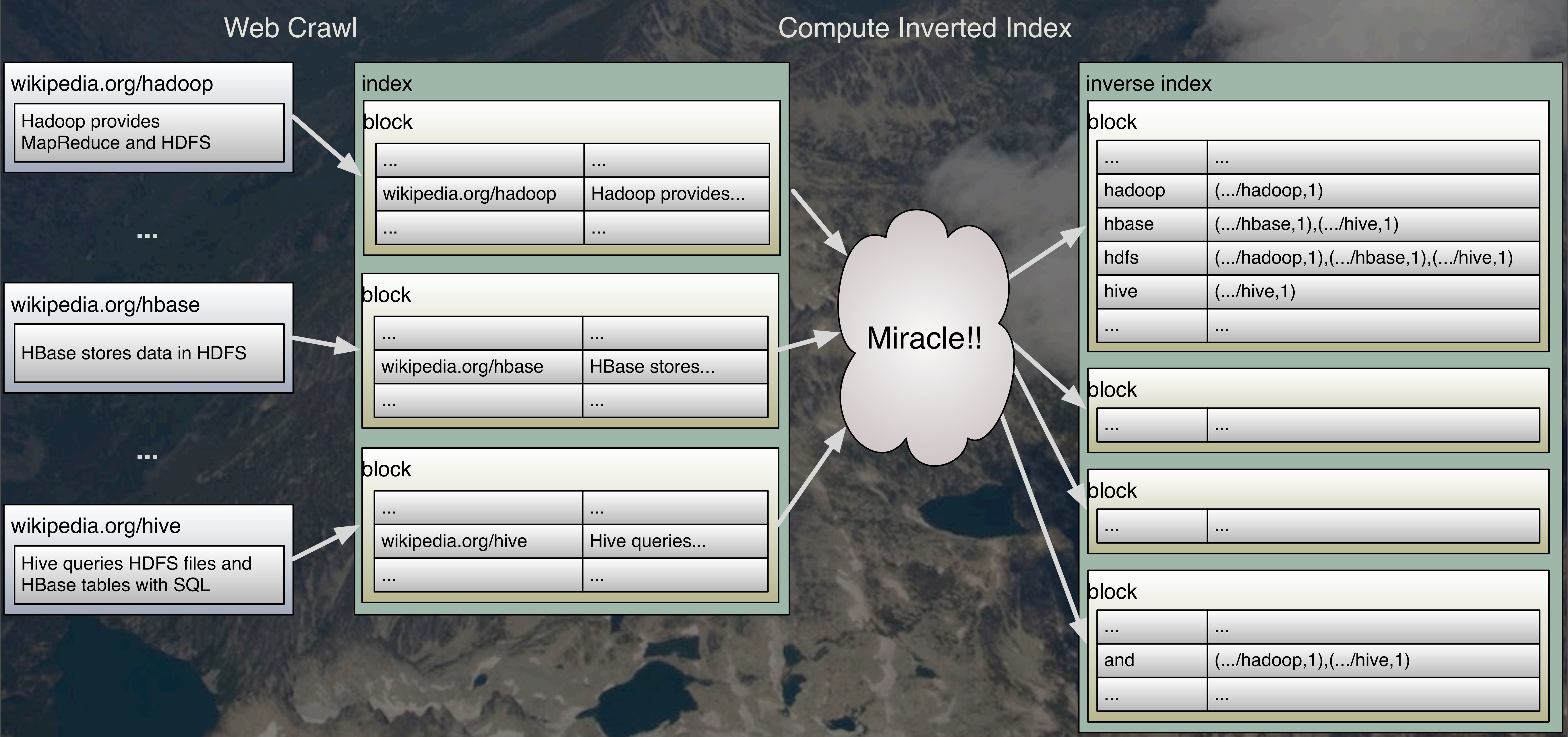
Historically, up to 2013, MapReduce was the officially-supported compute engine for writing all compute jobs.



Tuesday, May 12, 15

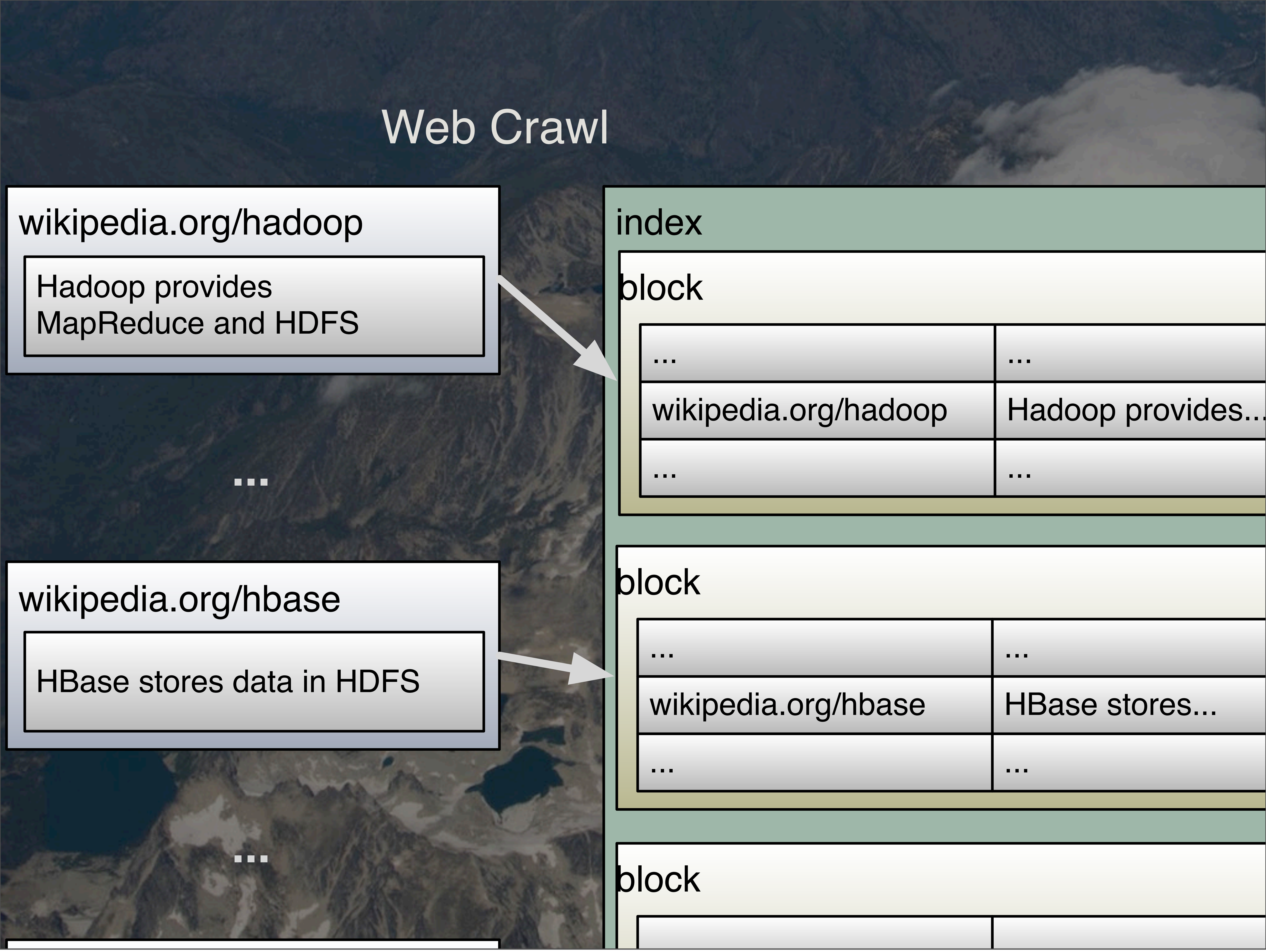
We want to crawl the Internet (or any corpus of docs), parse the contents and create an “inverse” index of the words in the contents to the doc id (e.g., URL) and count the number of occurrences per doc, since you will want to search for docs that use a particular term a lot.

Example: Inverted Index

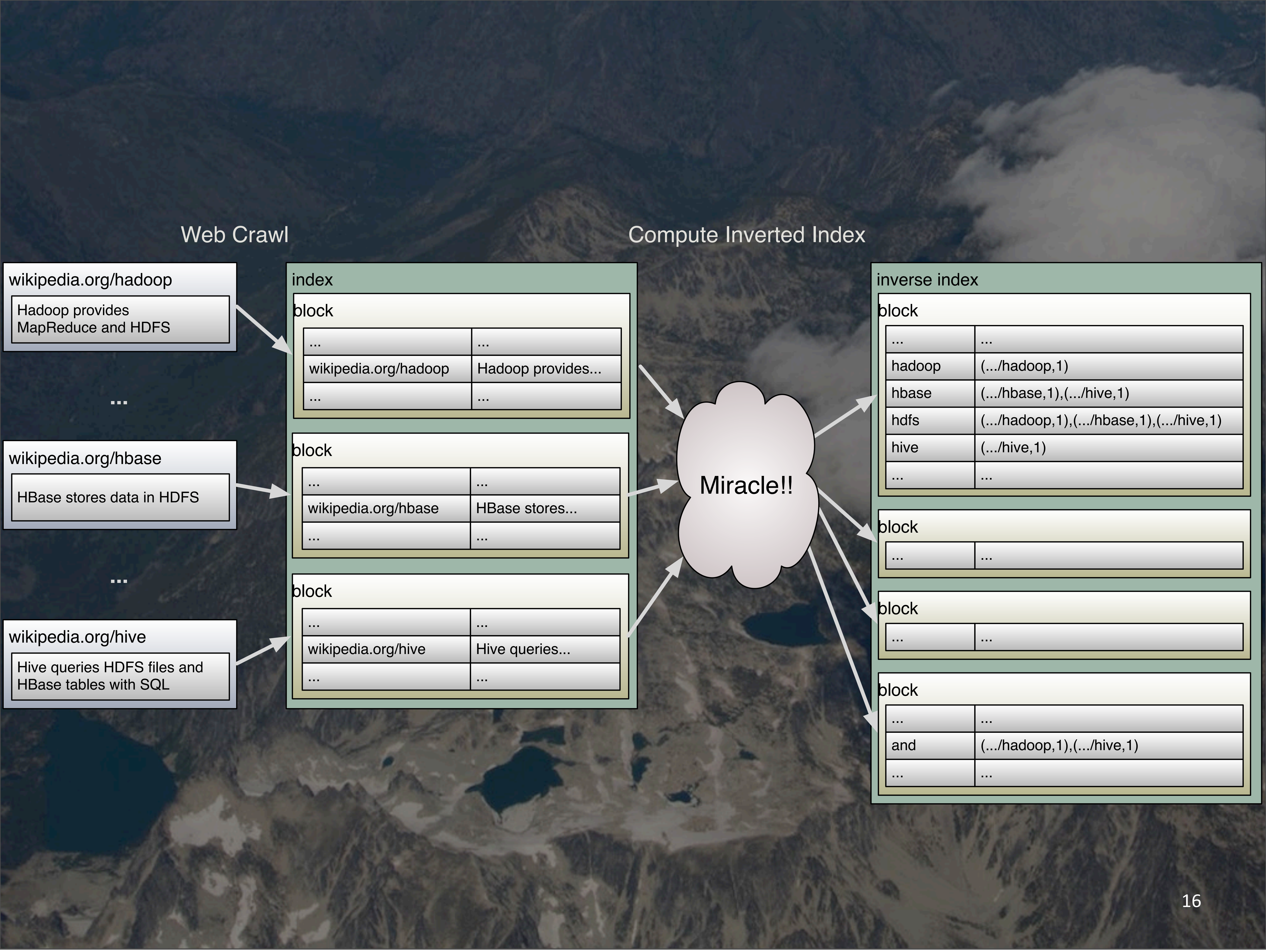


Tuesday, May 12, 15

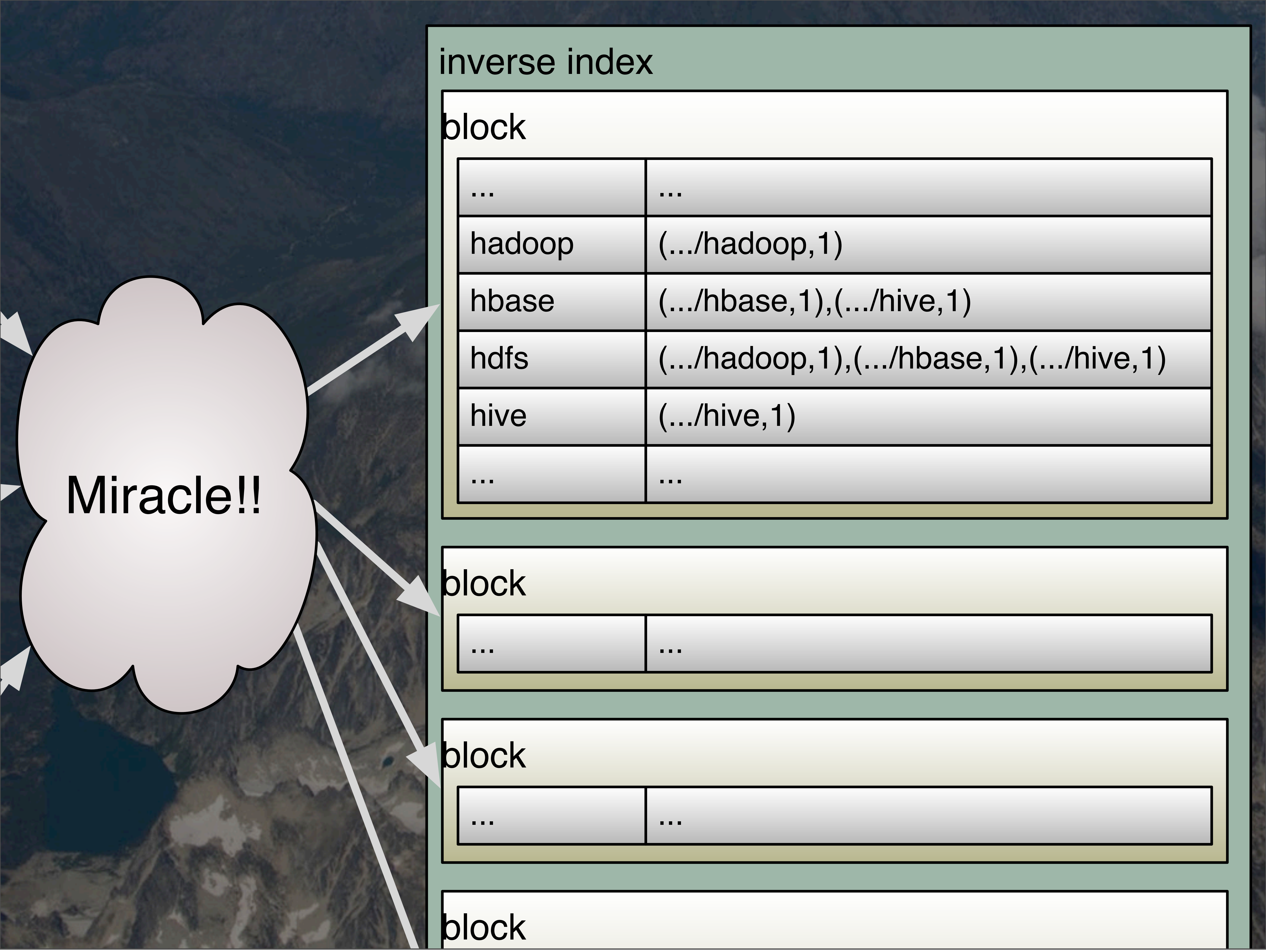
It’s done in two stages. First web crawlers generate a data set with two two-field records, containing each document id (e.g., the URL). Then that data set is read in batch (such as a MapReduce job) that “miraculously” creates the inverted index.



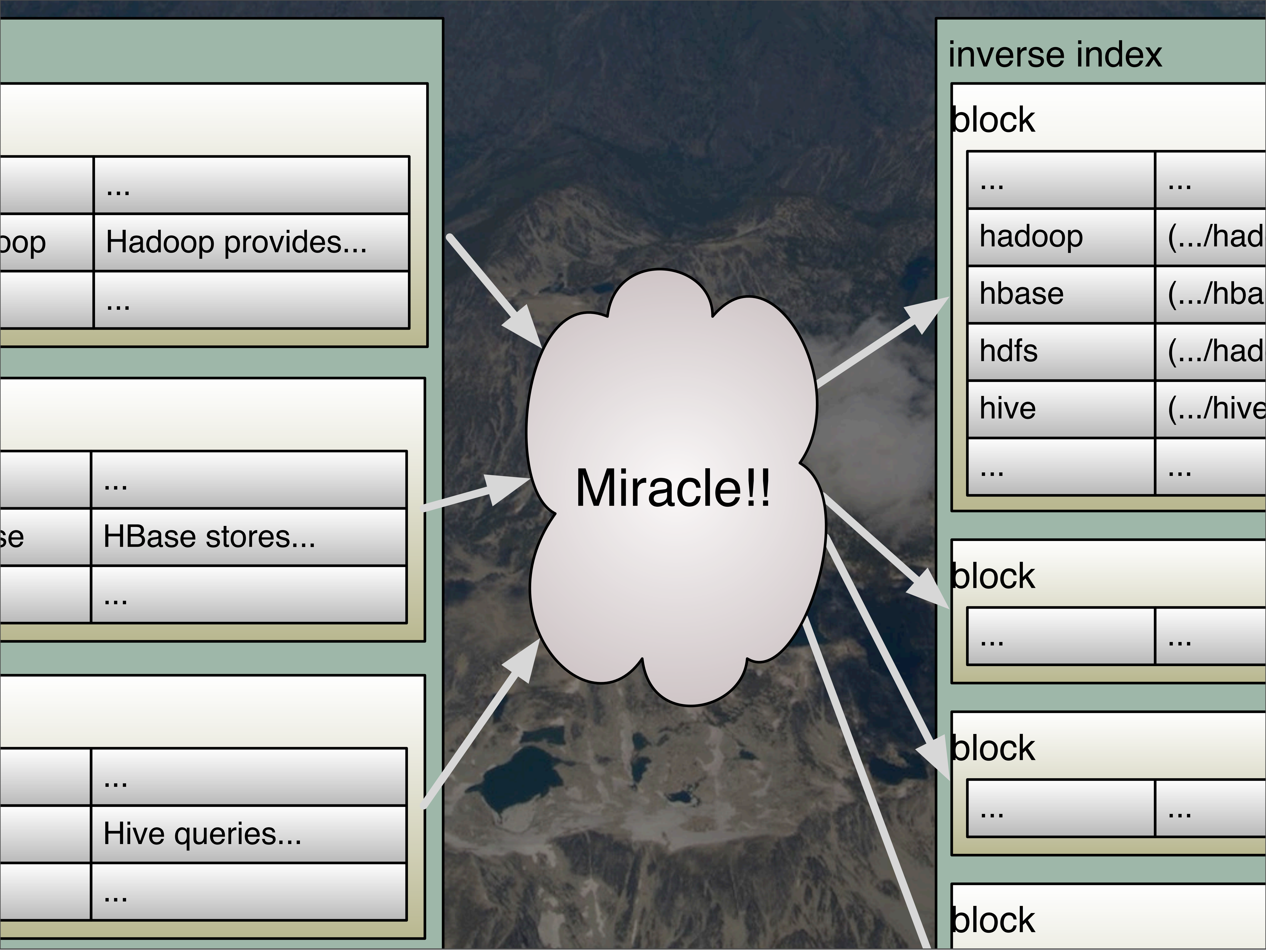
Zoom into details. The initial web crawl produces this two-field data set, with the document id (e.g., the URL, and the contents of the document, possibly cleaned up first, e.g., removing HTML tags).



Tuesday, May 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.



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

Tuesday, May 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
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()));
  }
 }
}
```

Java MapReduce Implementation

This is an implementation with MapReduce (6pt. font) Actually, it omits ordering the (docid, count) tuples by count descending, as you would want. This would take a few hours to write, test, etc. assuming you already know the API and the idioms for using it. It’s a relatively simple algorithm, so imagine doing something more complicated.

Tuesday, May 12, 15

Well, can we implement higher level tools?

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

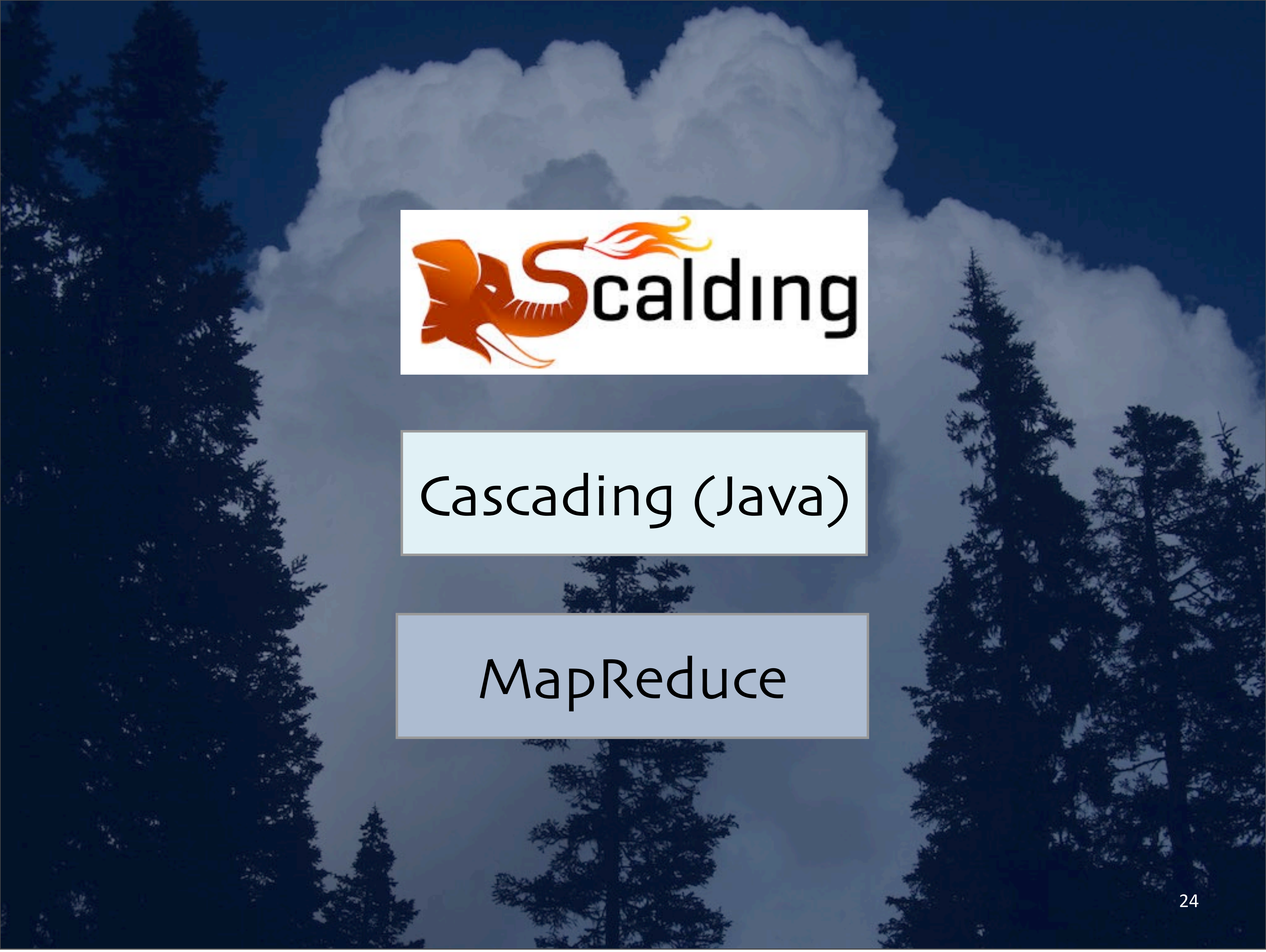

Tuesday, May 12, 15

The first SQL on Hadoop. It’s purely for querying, not CRUD (although you can create new tables - files really) with a query.
Using SQL is great for many people, but extending Hive requires coding Java UDFs (user-defined functions) to an API that isn’t always easy.

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


Tuesday, May 12, 15

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.



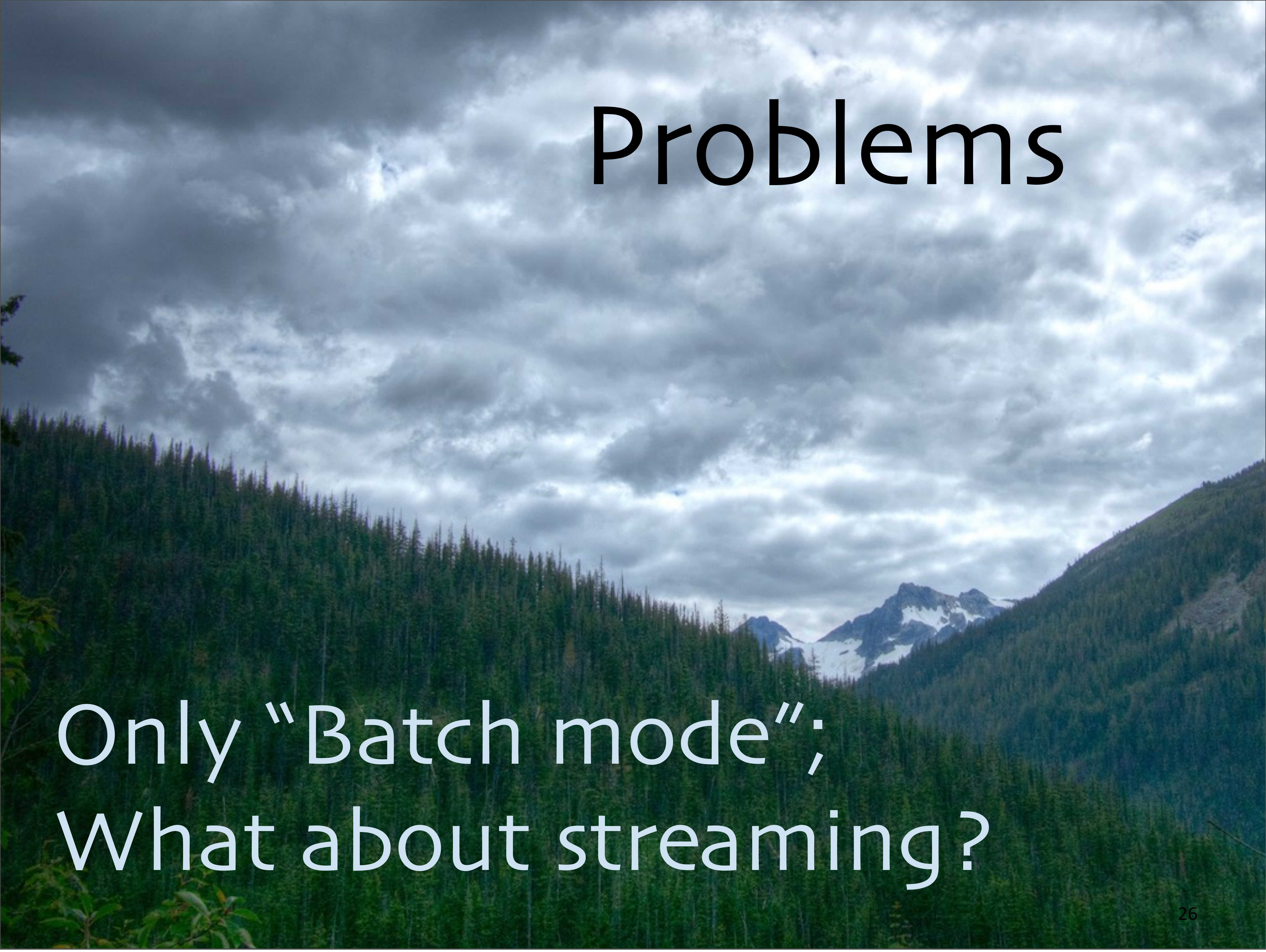
Tuesday, May 12, 15

Scalding was the first Scala “DSL” for Hadoop, providing a Turing complete, elegant API for developer productivity.

```
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"))
}
```


Tuesday, May 12, 15

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

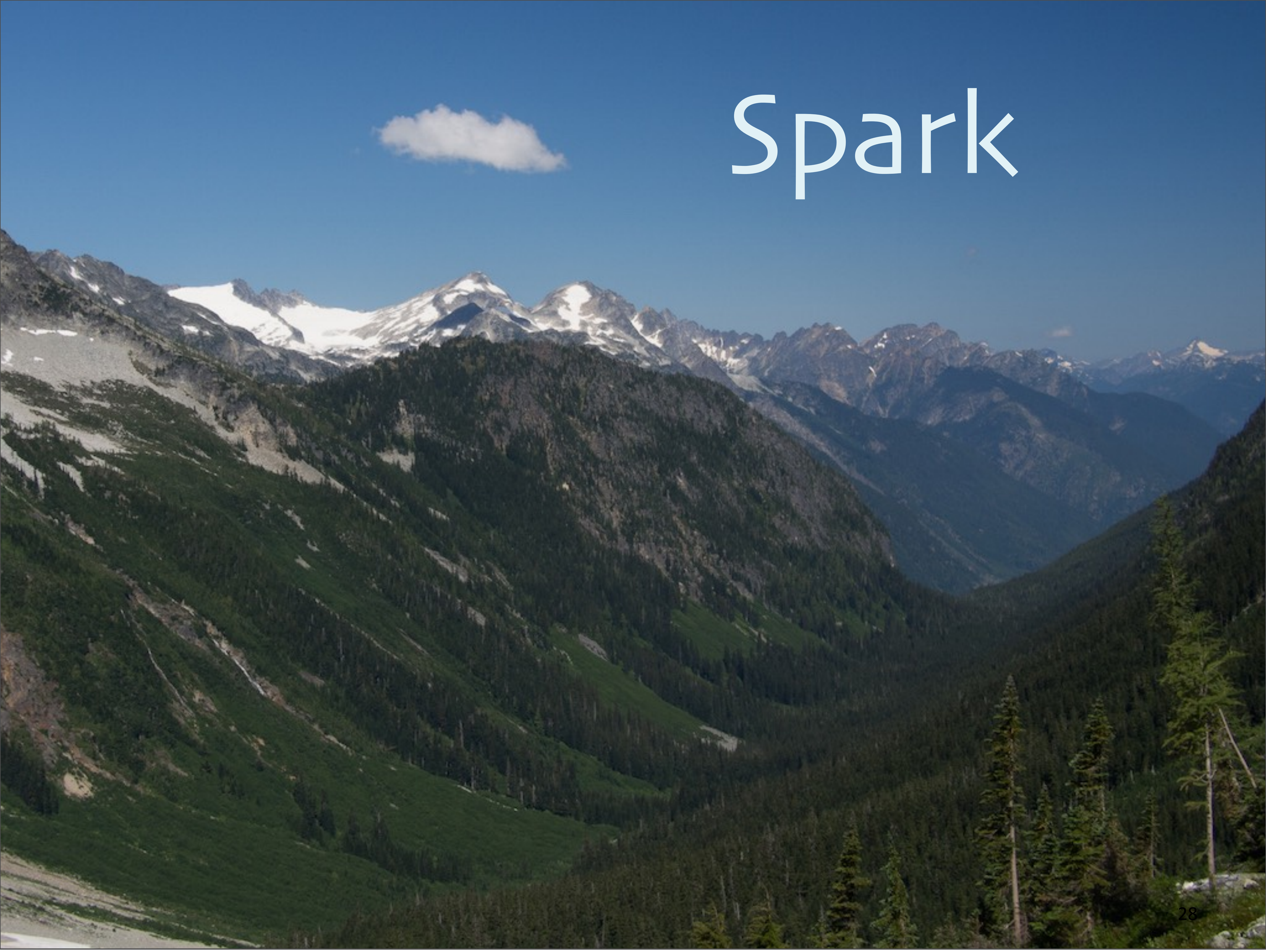


Tuesday, May 12, 15

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...).

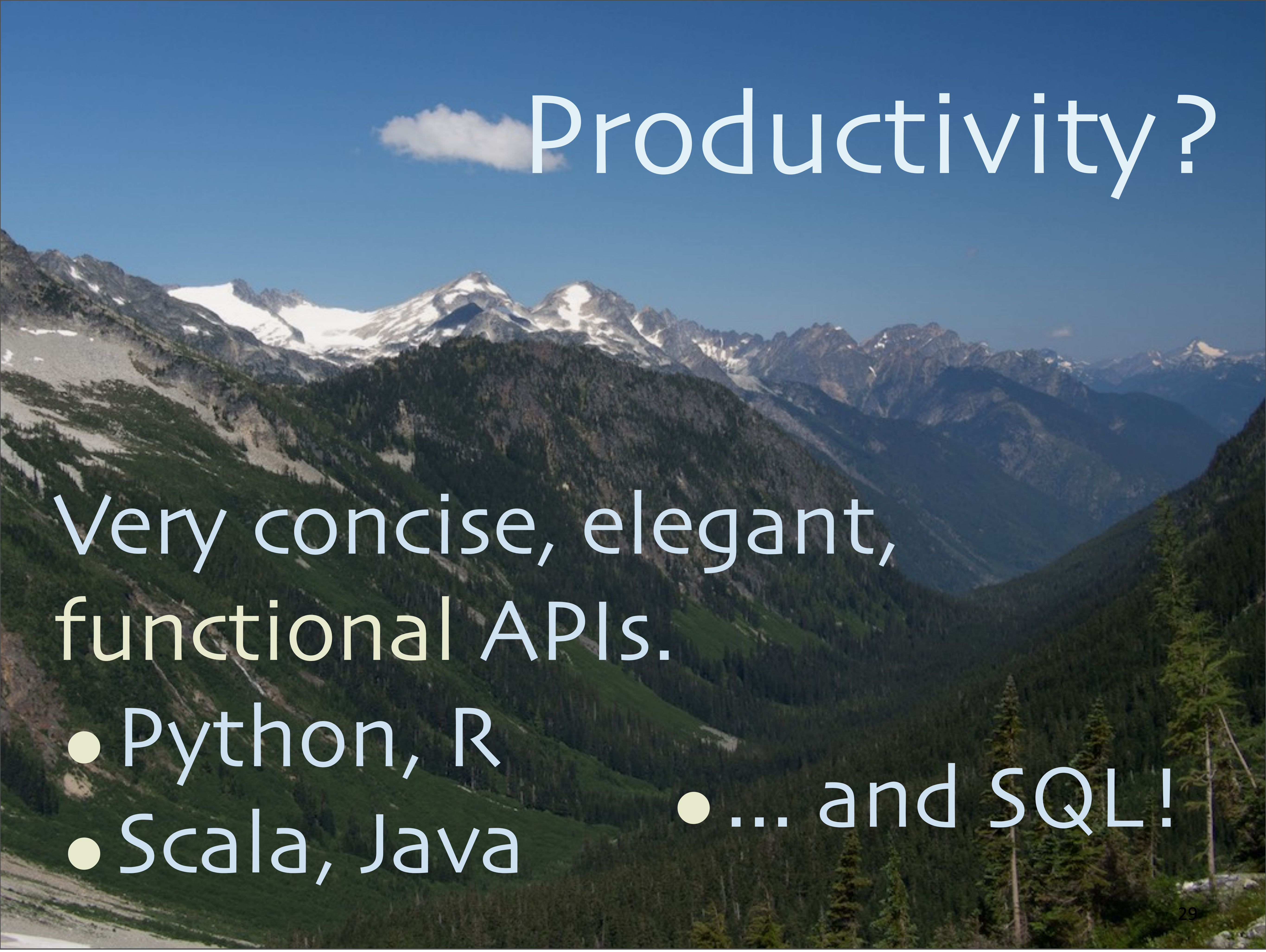
Tuesday, May 12, 15

Another MapReduce problem: performance is not good.



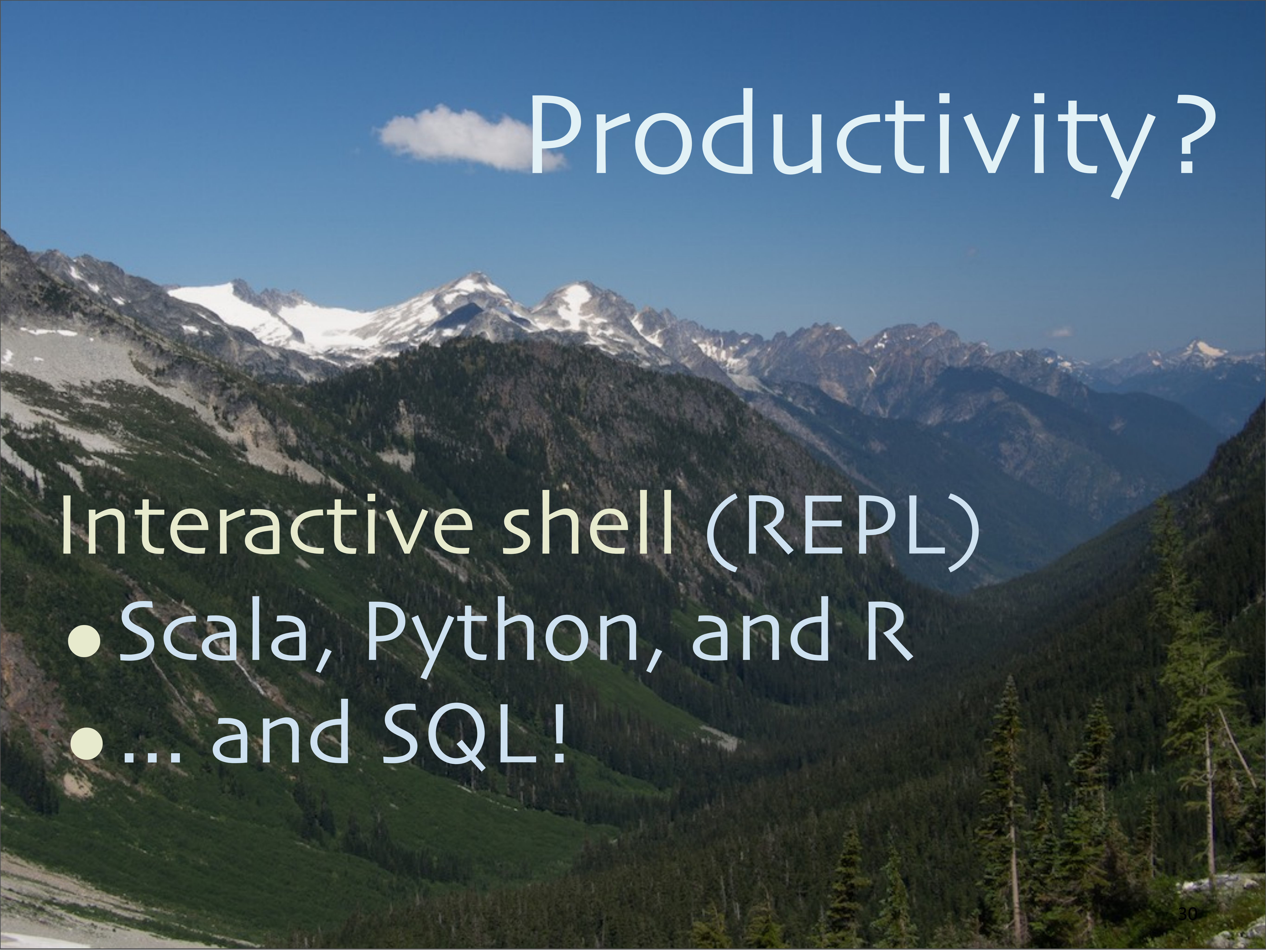
Tuesday, May 12, 15

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.



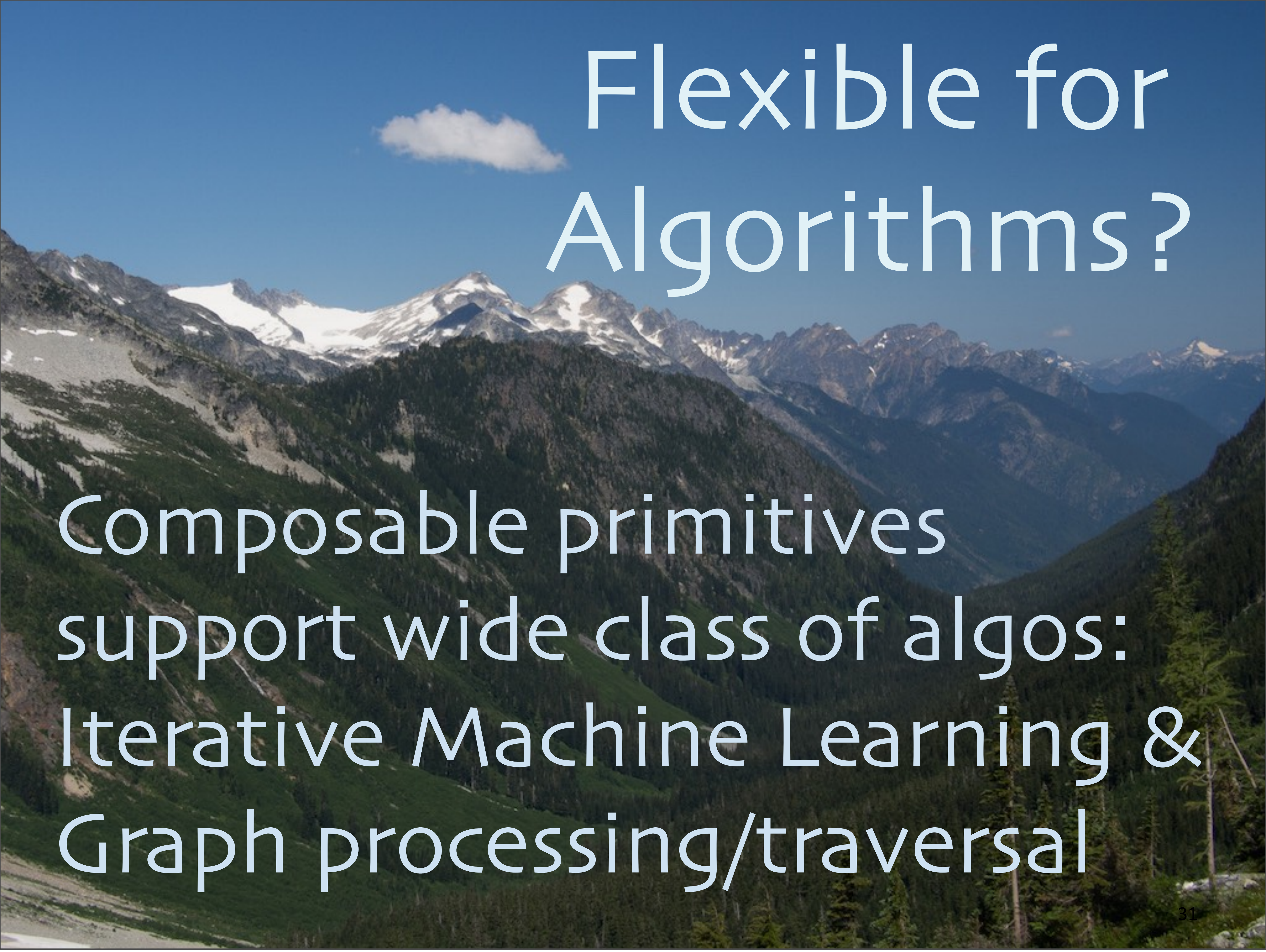
Tuesday, May 12, 15

We'll see by example shortly why this true.
While Spark was written in Scala, it has a Java and Python API, too, and an R API is almost released.



Tuesday, May 12, 15

This is especially useful for the SQL queries we'll discuss, but also handy once you know the API for experimenting with data and/or algorithms.

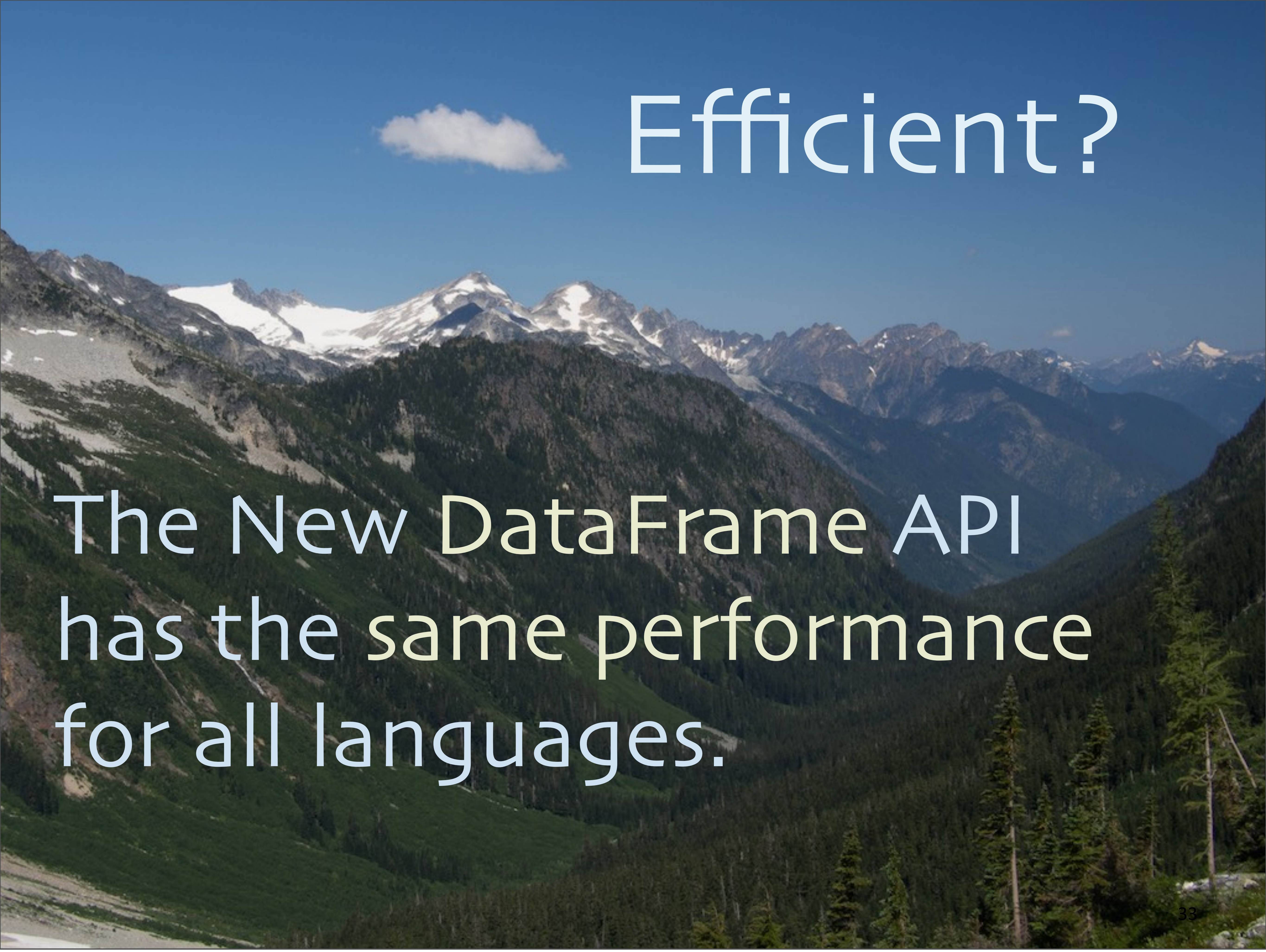


Tuesday, May 12, 15

A major step forward. Due to the lightweight nature of Spark processing, it can efficiently support a wider class of algorithms, such as a 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.

Tuesday, May 12, 15

How is Spark more efficient? As we'll see, Spark programs are actually "lazy" dataflows definitions that are only evaluated on demand. Because Spark has this directed acyclic graph of steps, it knows what data to attempt to cache in memory between steps (with programmable tweaks) and it can combine many logical steps into one "stage" of computation, for efficient execution while still providing an intuitive API experience.



Tuesday, May 12, 15

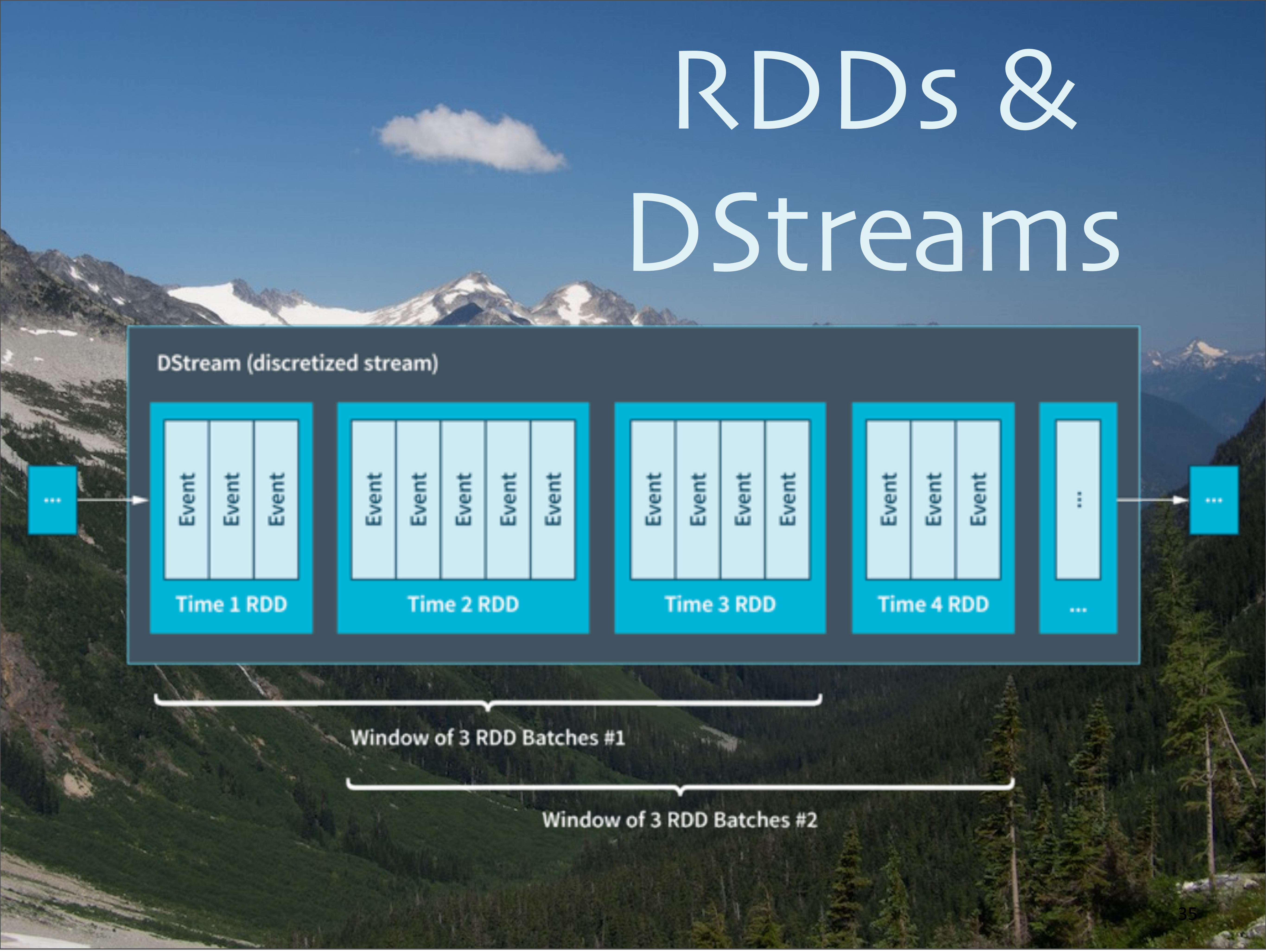
This is a major step forward. Previously for Hadoop, Data Scientists often developed models in Python or R, then an engineering team ported them to Java MapReduce. Previously with Spark, you got good performance from Python code, but about 1/2 the efficiency of corresponding Scala code. Now, the performance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

Batch + Streaming?

Streams - “mini batch” processing:

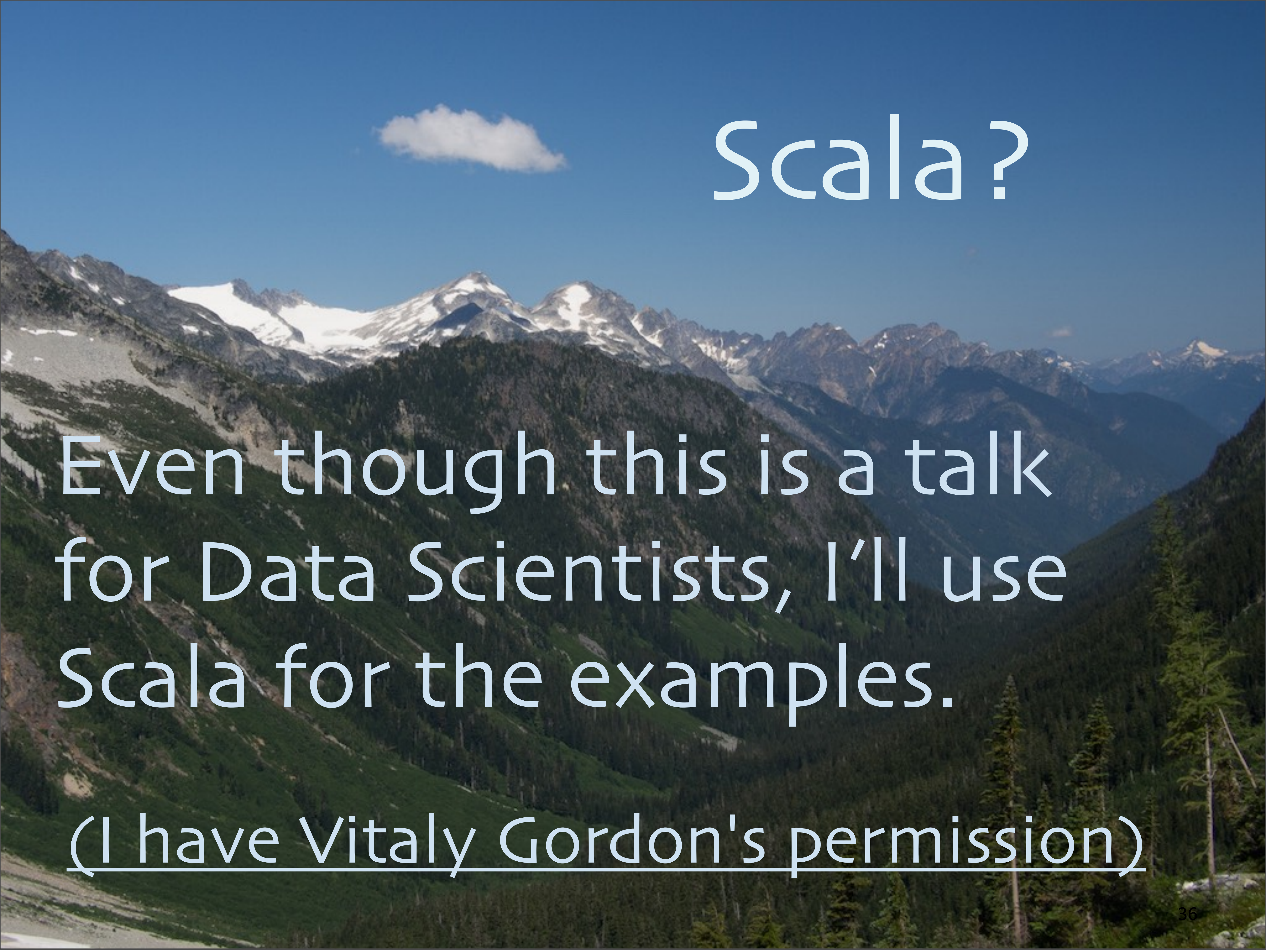
- Reuse “batch” code
- Adds “window” functions

Spark also started life as a batch-mode system, but Spark’s dataflow stages and in-memory, distributed collections (RDDs - resilient, distributed datasets) are lightweight enough that streams of data can be timesliced (down to ~1 second) and processed in small RDDs, in a “mini-batch” style. This gracefully reuses all the same RDD logic, including your code written for RDDs, while also adding useful extensions like functions applied over moving windows of these batches.



Tuesday, May 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.



Tuesday, May 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.

```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) =>
```


Tuesday, May 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.

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

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

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

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


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

```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) =>
```


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).

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


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

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

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

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


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

```scala
}
.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)
```

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

Tuesday, May 12, 15

Then we map over these pairs and add a single "seed" count of 1, then use "reduceByKey", which does an implicit "group by" to bring together all occurrences of the same (word, path) and then sums up their counts. (It's much more efficient than groupBy, because it avoids creating the groups when all we want is their size, in this case.) The output of reduceByKey is indicated with the bubble; we'll have one record per (word,path) pair, with a count >= 1.

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

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

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

I love this step! It simply moves parentheses to reorganize the tuples, where now the "key" is each word, setting us up for the final group by to bring together all (path, n) "subtuples" for each word. I'm showing both the new schema and the previous schema.

```
      case ((word,path),n) => (word,(path,n))
    }
    .groupByKey
    .mapValues { iter =>
      it
        c
      }.mkString(", ")
    }
    .saveAsTextFile(argz.outpath)

    sc.stop()
  }
}
```

(word, Seq((path1, n1), (path2, n2), (path3, n3), ...))

...

Now we do an explicit group by using the word as the key (there's also a more general groupBy that lets you specify how to treat each record). The output will be (word, iter((path1, n1), (path2, n2), ...)), where "iter" is used to indicate that we'll have a Scala abstraction for iterable sequences, e.g., Lists, Vectors, etc.

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

    sc.stop()
  }
}
```

(word, “(path4, 80), (path19, 51), (path8, 12), ...”)

...

The last step could use map, but mapValues is a convenience when we just need to manipulate the values, not the keys. Here we convert the iterator to a sequence (it may already be one...) so we can sort the sequence by the count descending, because we want the first elements in the list to be the documents that mention the word most frequently. It secondary sorts by the path, which isn’t as useful, except for creating repeatable results for testing!. Finally, the sequence is converted into a string. A “sample” record is shown.

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

    sc.stop()
  }
}
```


Tuesday, May 12, 15

We finish the sequence of steps by saving the output as one or more text files (it could be other formats, too, including writes to a database through JDBC). Note that in Spark, everything shown UNTIL the saveAsTextFile is lazy; it builds up a pipeline of steps but doesn't actually process any data. Such steps are called "transformations" in Spark. saveAsTextFile is an example of an "action", which triggers actual processing to happen. Finally, after processing, we stop the workflow to clean up.

```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 {
      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

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

```
  }
  .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)
```

Powerful, beautiful combinators

Tuesday, May 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!!

Tuesday, May 12, 15

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

SQL Revisited

Spark SQL

- Integrates with Hive
- Has its own query engine "Catalyst":
  - Query optimizations
- Write SQL
- Use the new DataFrame API

See http://spark.apache.org/docs/latest/sql-programming-guide.htm

Spark SQL

- Integrates with Hive
- Has its own query engine "Catalyst":
  - Query optimizations
  - SQL API
  - DataFrame API

See http://spark.apache.org/docs/latest/sql-programming-guide.htm

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

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

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

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

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


Example adapted from http://spark.apache.org/docs/latest/sql-programming-guide.html#hive-tables
Assume we're using word count data, abbreviated "wc" to fit.
Spark has it's own dialect of SQL for working outside Hive. The intention is to eventual replace the need for Hive. Simultaneously, Cloudera is sponsoring an effort to replace MapReduce inside Hive with Spark.

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

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

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

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

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


Tuesday, May 12, 15

Create a SparkContext as before, then a HiveContext that knows about Hive metastores, can talk to a Hive server to run HiveQL queries, even DDL statements, etc.

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

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

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

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

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


Create a Hive table for Word Count data.

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

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

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

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

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


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...)

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

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

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

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

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


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.

I think it’s funny how similar Scala and Python Spark code are. The Java code is also very similar for this simple example at least.

Spark SQL

- Integrates with Hive
- Has its own query engine "Catalyst":
  - Query optimizations
  - SQL API
  - DataFrame API

See http://spark.apache.org/docs/latest/sql-programming-guide.htm

```
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")
```


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.

```scala
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")
```


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.

```scala
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")
```


Tuesday, May 12, 15

We'll assume the data is in Parquet, the default for load(), then write an equivalent query in the DataFrame API, plus other queries. The path shown is treated as a directory to read, by default (normal Hadoop behavior which Spark follows.)

```scala
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")
```


Tuesday, May 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.

```scala
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")
```


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.

```scala
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")
```


Find all the words of length > 20

```scala
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")
```


Save the long words data to another location as one or more Parquet files.

Machine Learning

MLlib

Tuesday, May 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

Tuesday, May 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.

```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)
```


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)
```


Tuesday, May 12, 15

Imports from MLlib and Spark Streaming.

```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)
```


Tuesday, May 12, 15

Create a SparkContext and StreamingContext, with a batch interval of 10 seconds.

```
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)
```


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.

```scala
  .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()
```


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()
```

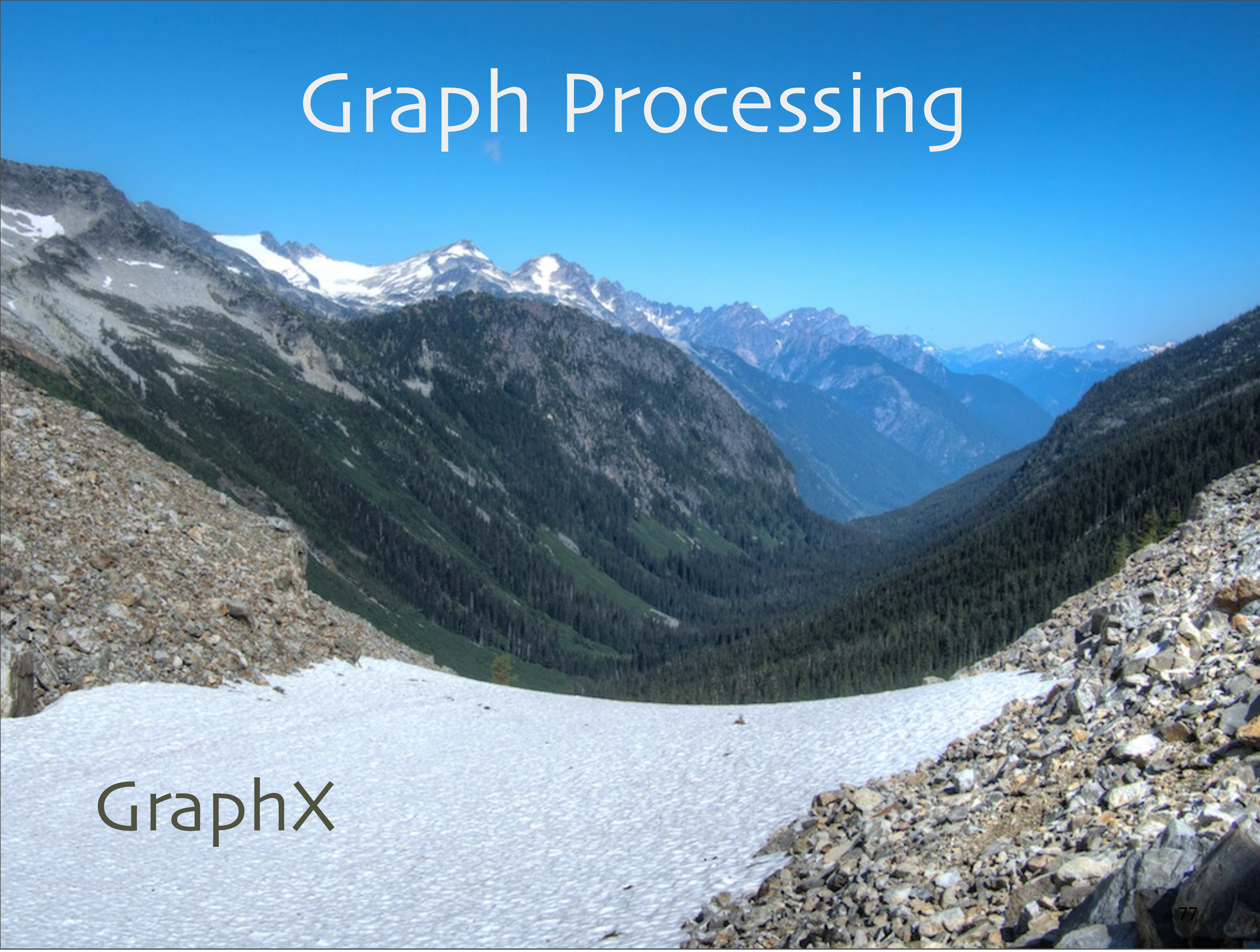

This is a Scala named function (as opposed to a method or an anonymous function). It takes in a labeled point and returns a tuple of the label and the vector of features.

```
  .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()
```


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.

```scala
  .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()
```


Actually start stream processing and wait forever for termination.

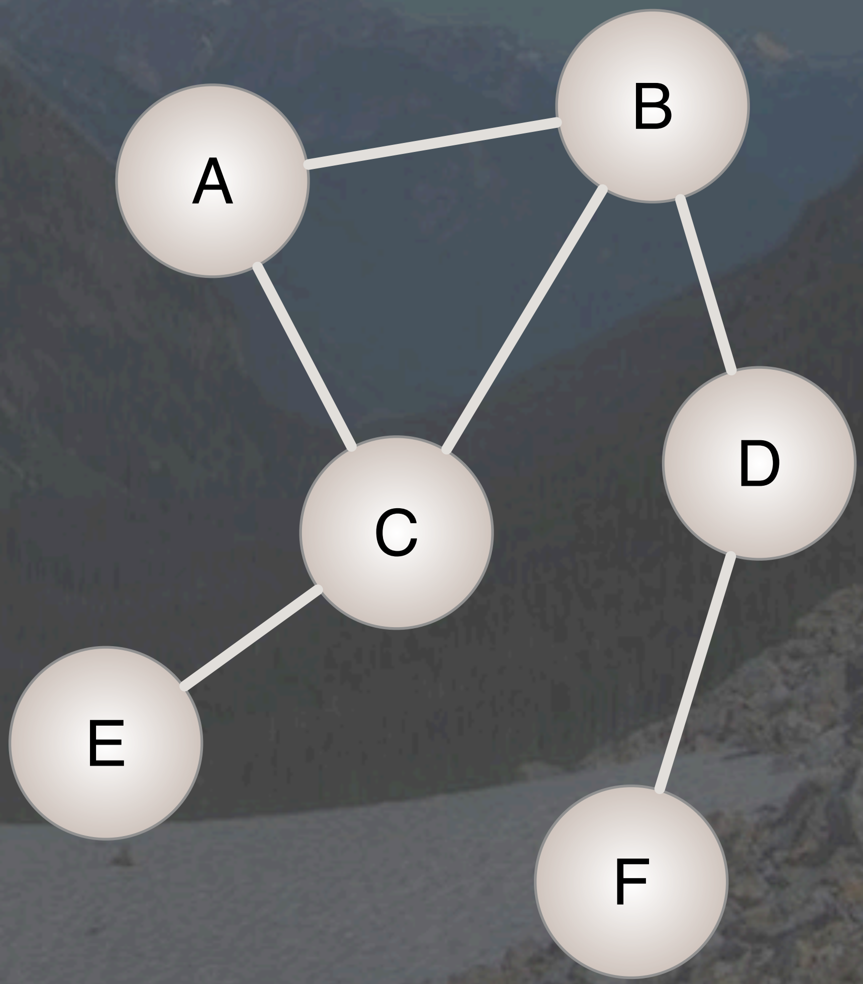


Tuesday, May 12, 15

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
- ...



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.

```scala
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 = "..."
```


In this example (for Spark 1.2 - the API changed in 1.3), we'll pretend to ingest real-time data about flights between airports, we'll process this stream in 60 second intervals using a SQL query to find the top 20 origin and destination airport pairs, in each interval. (Real-world are traffic data probably isn't that large in a 60-second window...)

```scala
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 = "..."
```


Lot 'o
imports

```scala
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 = "..."
```


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.

```scala
val tolerance = 0.001f
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(
```


Construct the SparkContext then the graph, pre-partitioning, and cache the underlying RDD.

```
val tolerance = 0.001f
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(
```


Partition the graph across the cluster. Print out the number of vertices and edges.

```scala
  "# vertices " + graph.vertices.count)
println("# edges " + graph.edges.count)

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

println("Total rank: " +
  pr.map(_._2).reduce(_ + _))

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

sc.stop()
```

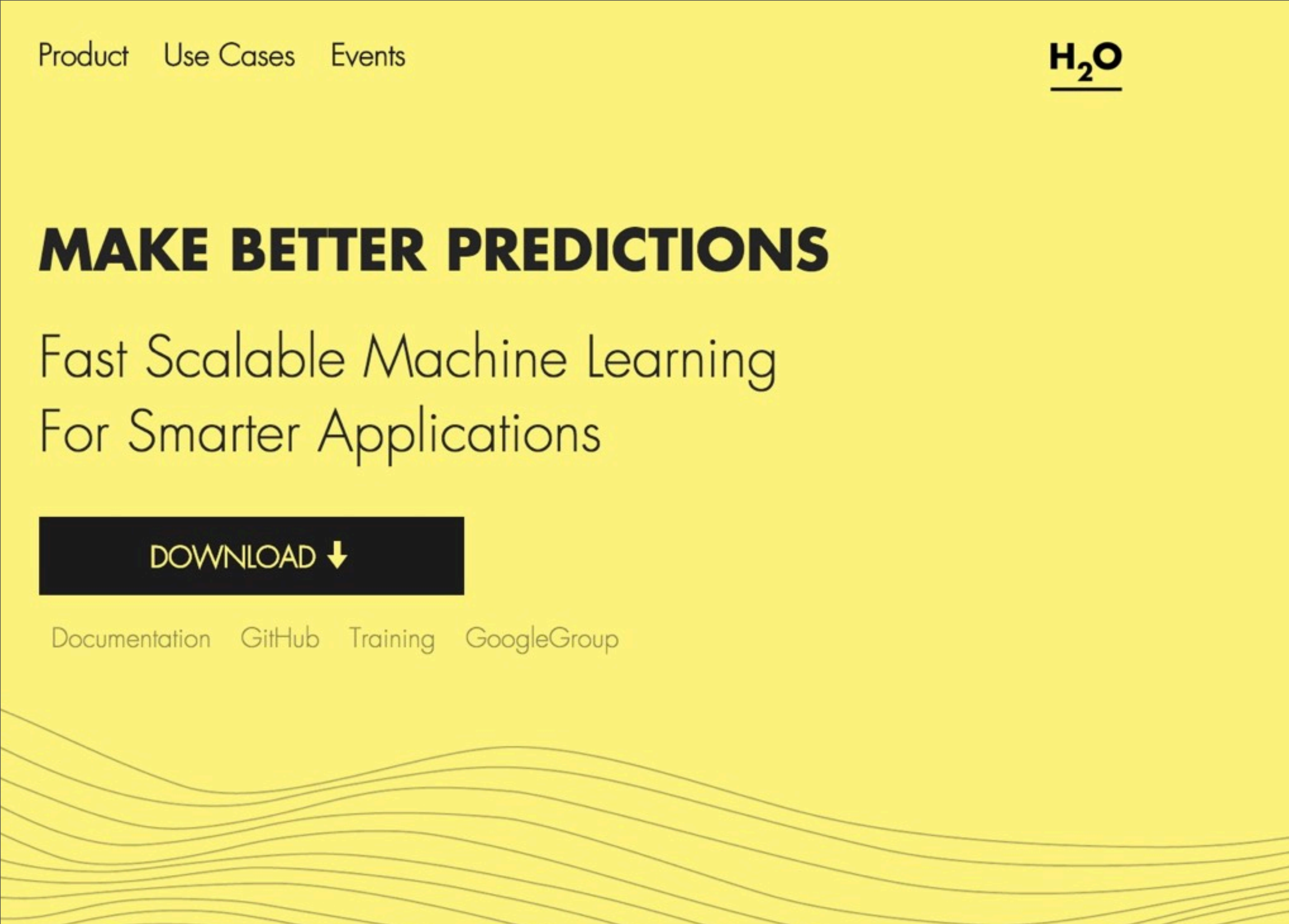

Run PageRank. Cache the vertices (RDD). The print the total rank, which is the sum of the edge weights.

```
  "# vertices " + graph.vertices.count)
println("# edges " + graph.edges.count)

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

println("Total rank: " +
  pr.map(_._2).reduce(_ + _))

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

sc.stop()
```


Tuesday, May 12, 15

Make tab-delimited output and save it. Then stop the job.

Other (Non-Spark) Things to Watch



Tuesday, May 12, 15

From: http://h2o.ai A high-performance, in-memory compute engine. Cliff Click (of JVM HotSpot fame) is the lead engineer. Also integrates with Spark. Provides a rich library of ML algorithms and a mature R binding.

Tuesday, May 12, 15

From: http:/julialang.org

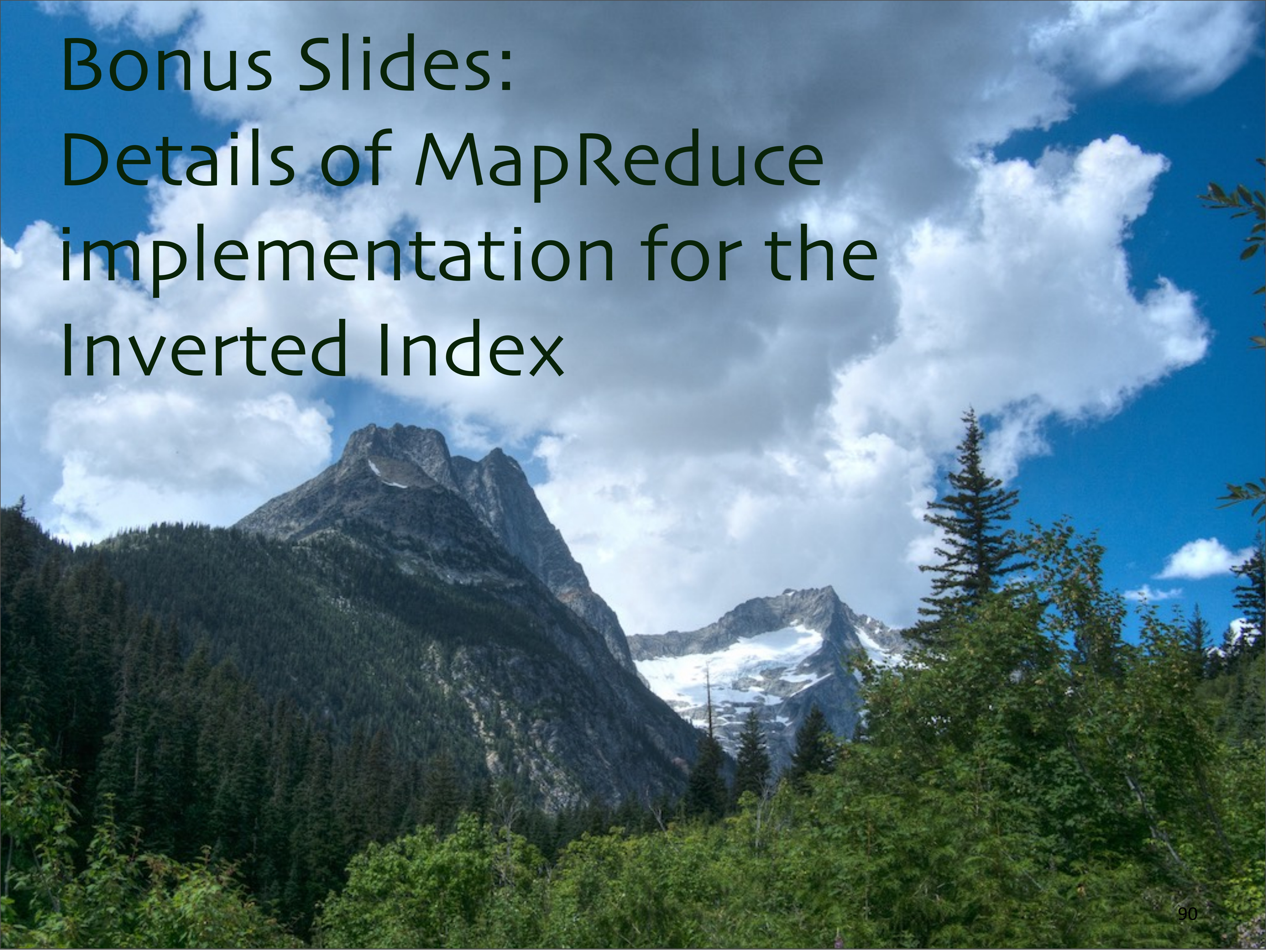
Questions?

Please remember to evaluate via the GOTO Guide App

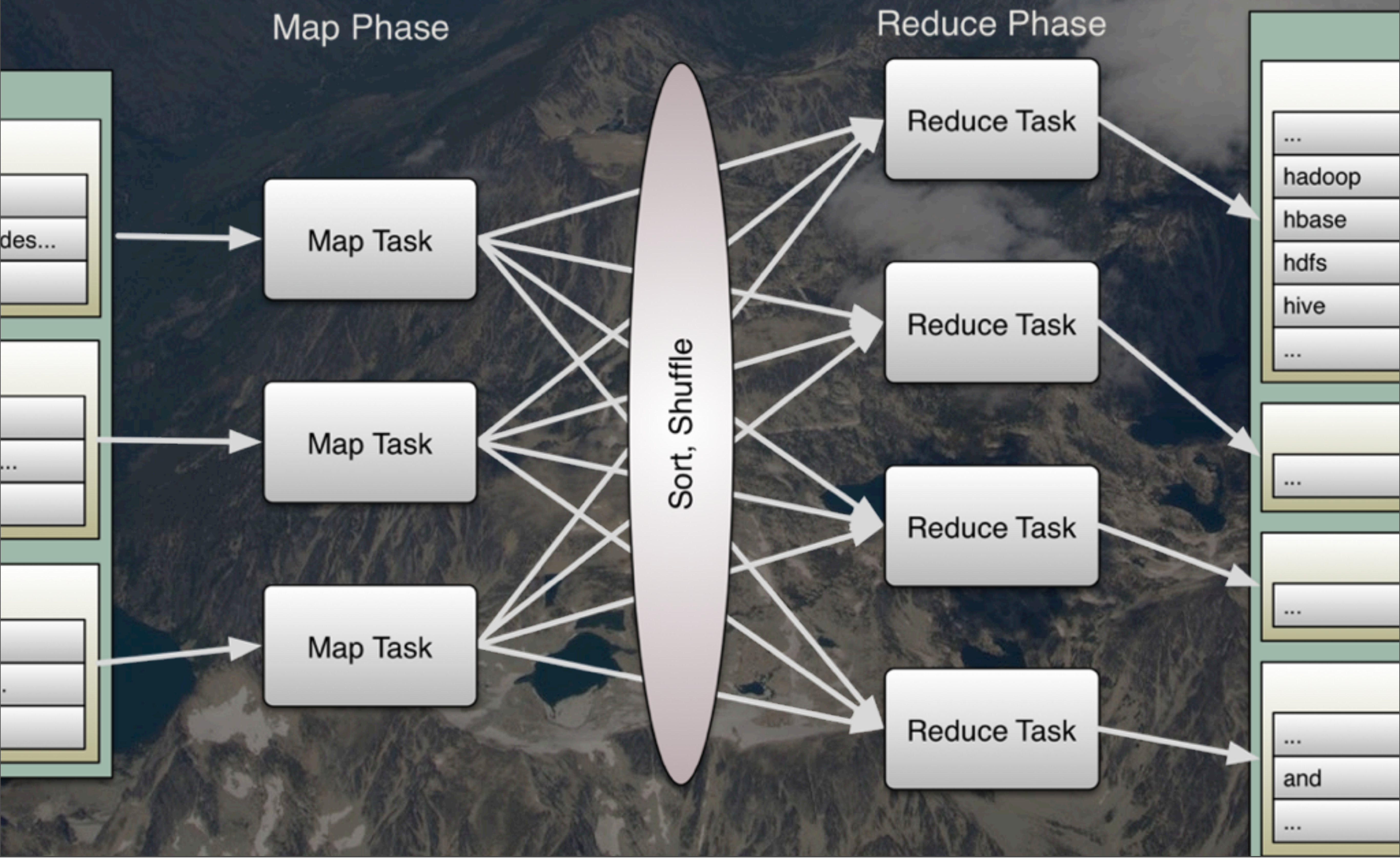
dean.wampler@typesafe.com
@deanwampler

Typesafe

Bonus Slides: Details of MapReduce implementation for the Inverted Index

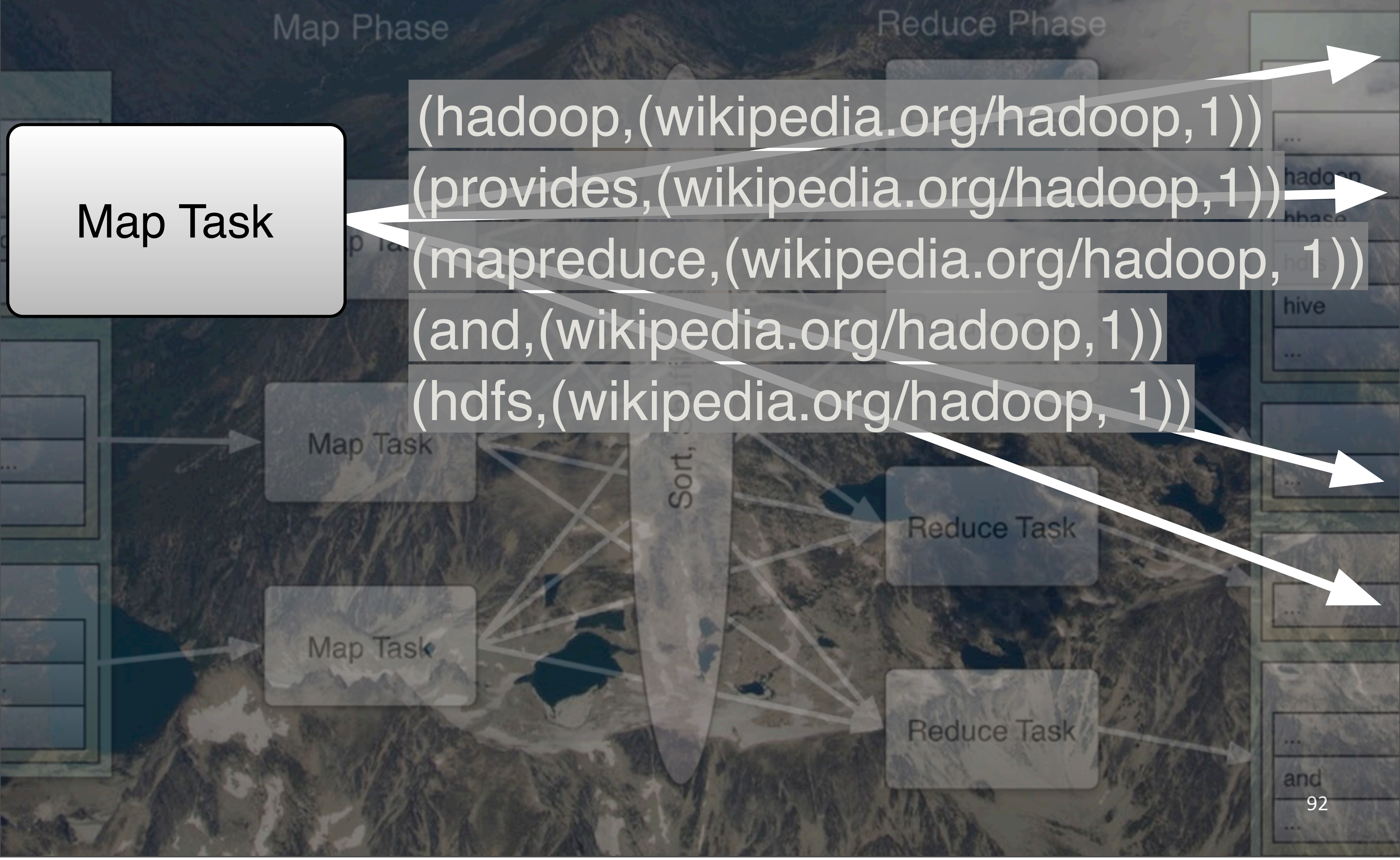


1 Map step + 1 Reduce step



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

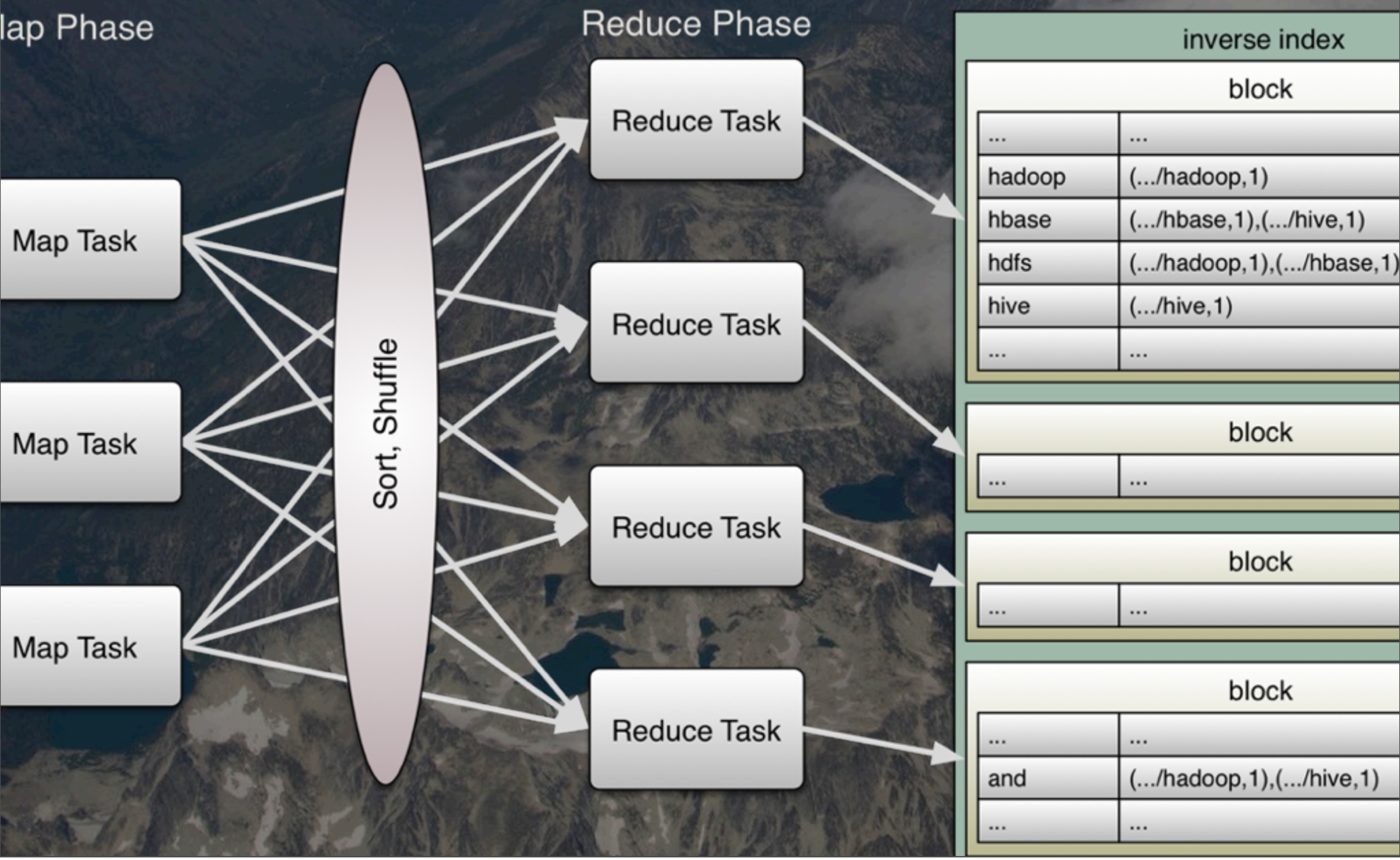
1 Map step + 1 Reduce step



Tuesday, May 12, 15

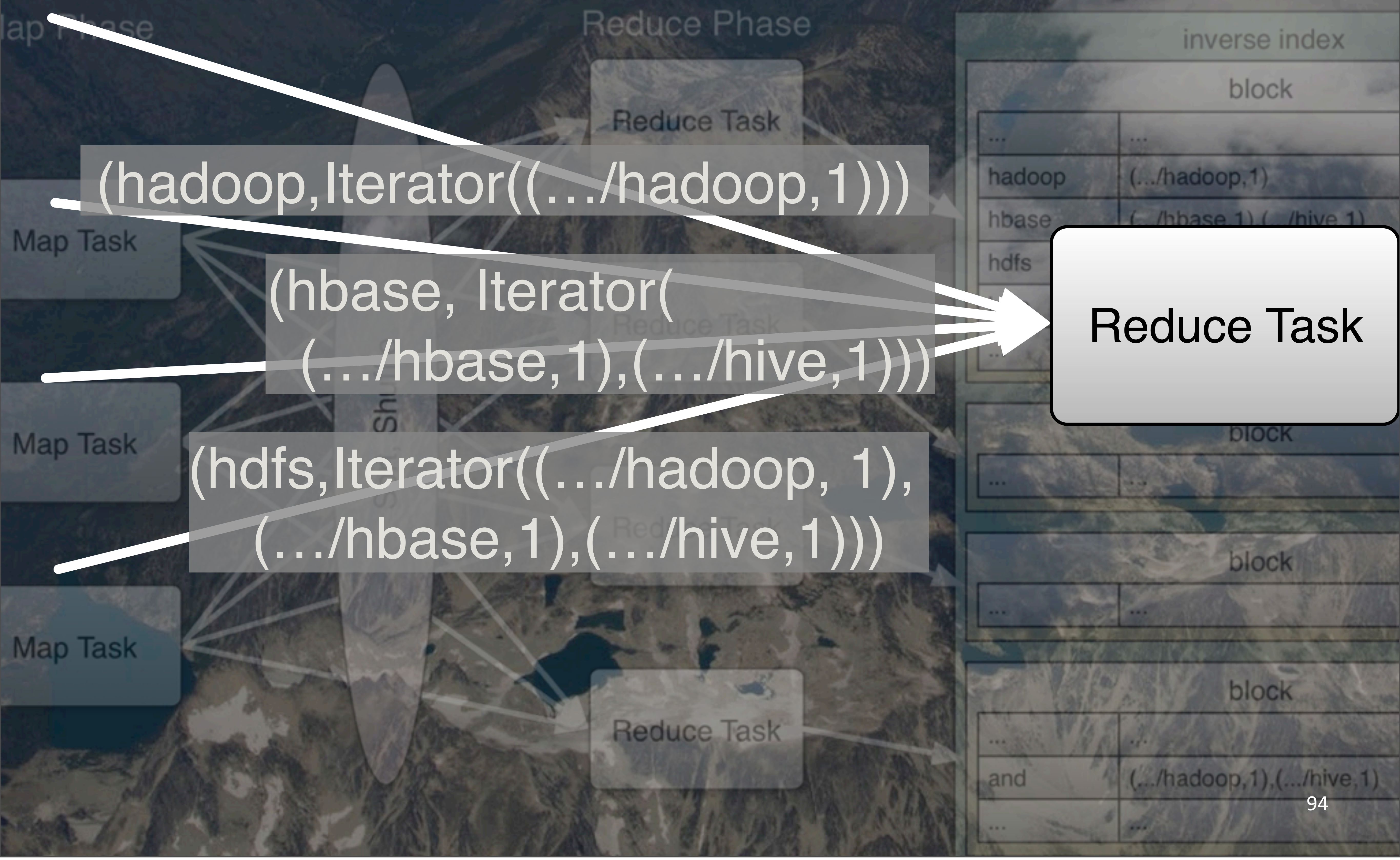
Each map task parses the records. It tokenizes the contents and write new key-value pairs (shown as tuples here), with the word as the key, and the rest, shown here as a second element that is itself a tuple, which holds the document id and the count.

1 Map step + 1 Reduce step



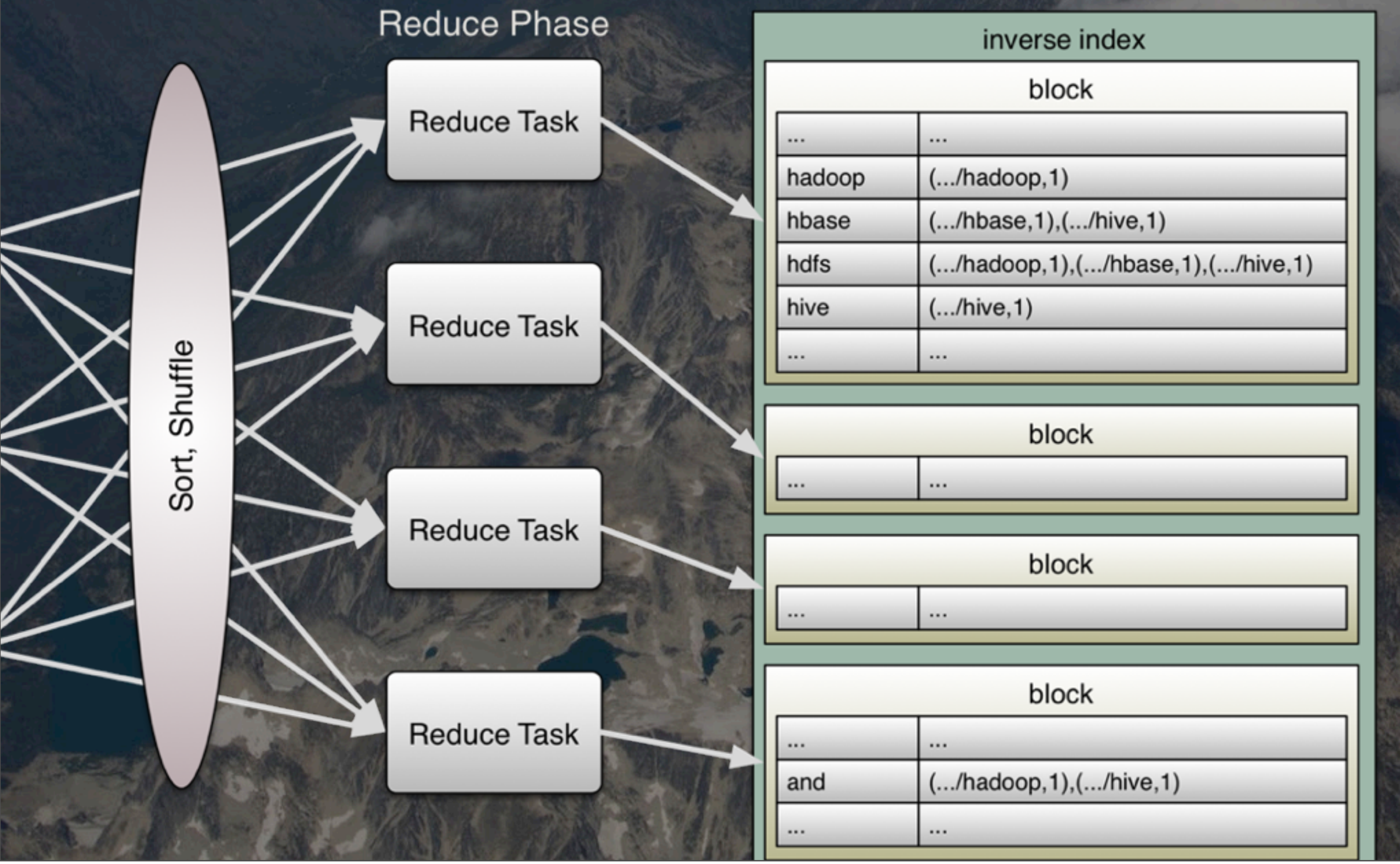
The output key,value pairs are sorted by key within each task and then “shuffled” across the network so that all occurrences of the same key arrives at the same reducer, which will gather together all the results for a given set of keys.

1 Map step + 1 Reduce step



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

1 Map step + 1 Reduce step



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.