The Unreasonable Effectiveness of Scala for Big Data

Scala Days 2015

dean.wampler@typesafe.com

Typesafe

Also check out:

- Vitaly Gordon's talk on Scala for Data Science.
- Yesterday's two Spark talks.

Typesafe

Wednesday, March 18, 15

All three were yesterday, but the videos and slides will be available.

San Francisco is very health conscious...

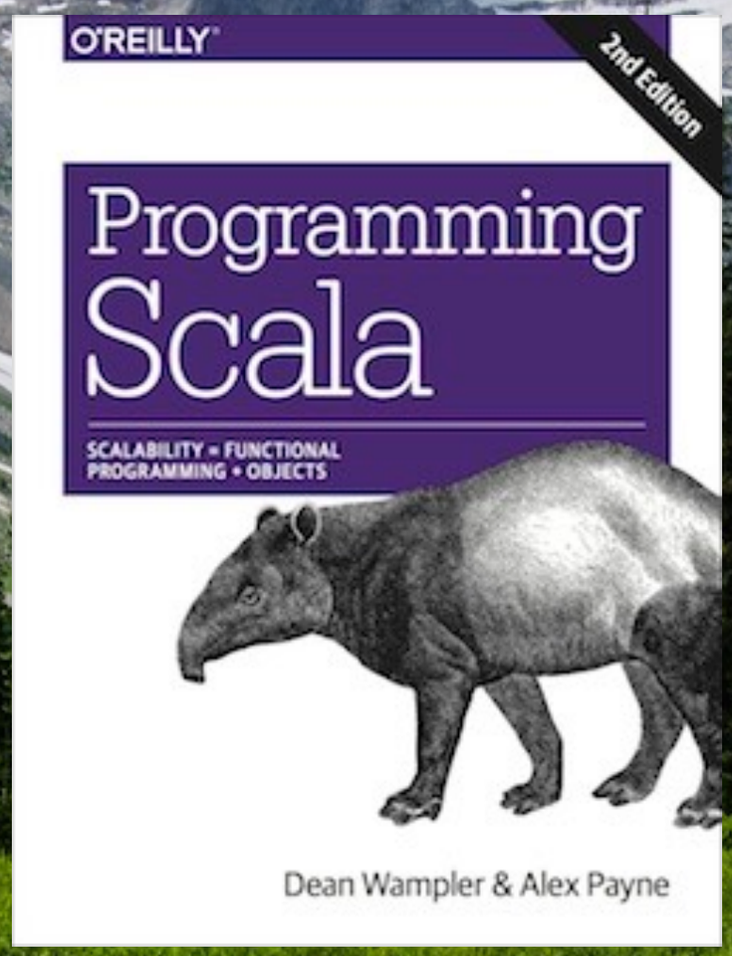
Wednesday, March 18, 15

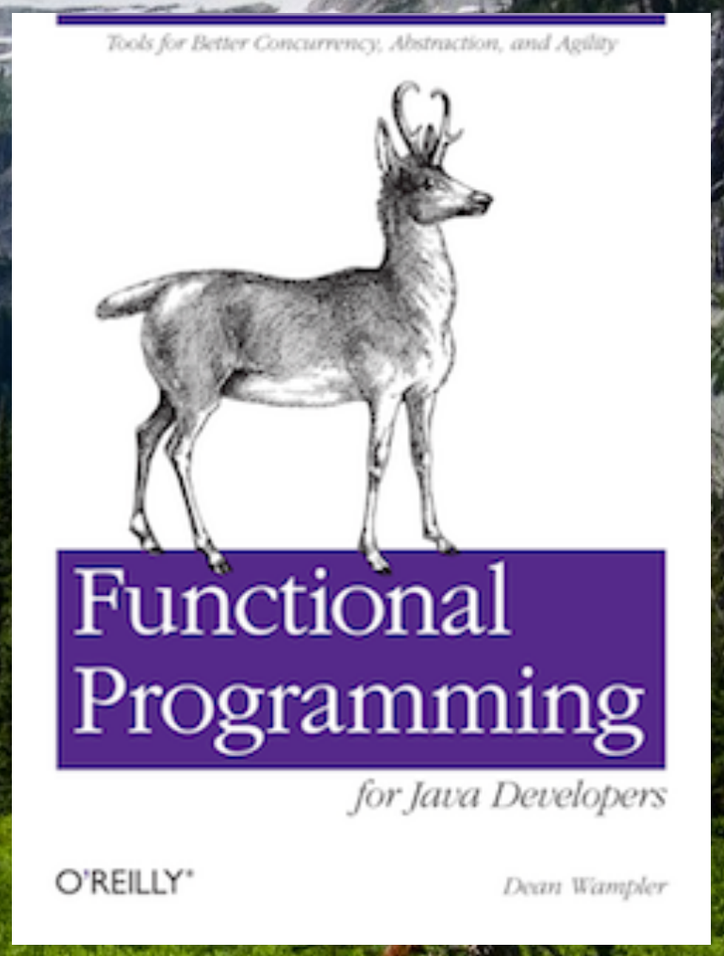
Maybe you've noticed that San Francisco is very health conscious. I didn't realize how much, until I noticed...

Wednesday, March 18, 15

... they have a ship named “No Smoking”!!

<shameless>
<plug>





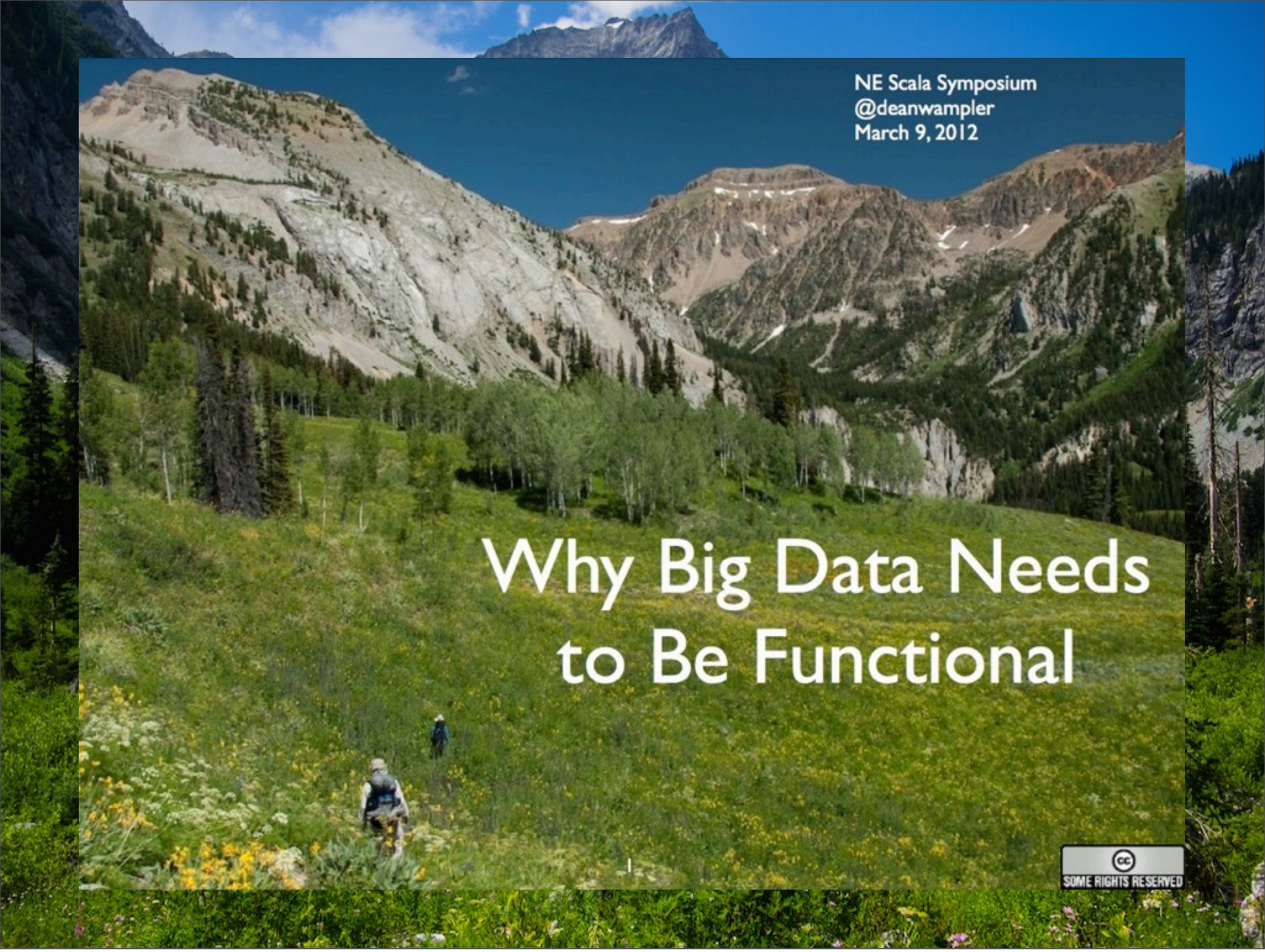
</plug>
</shameless>

Every developer talk should have some XML!!

“Trolling the Hadoop community since 2012...”

Wednesday, March 18, 15

My linkedin profile since 2012??



Wednesday, March 18, 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.

In which I claimed that:

Hadoop is the
Enterprise Java Beans
of our time.

Shock!!

Hadoop

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 Blog RSS Vide

4000

Scaling Hadoop to 4000 nodes at Yahoo!

By aanand – Tue, Sep 30, 2008 10:04 AM EDT

Recommend 1 Tweet 0

Tue, Sep 30, 2008

We recently ran Hadoop on what we believe is the single largest Hadoop installation, ever:

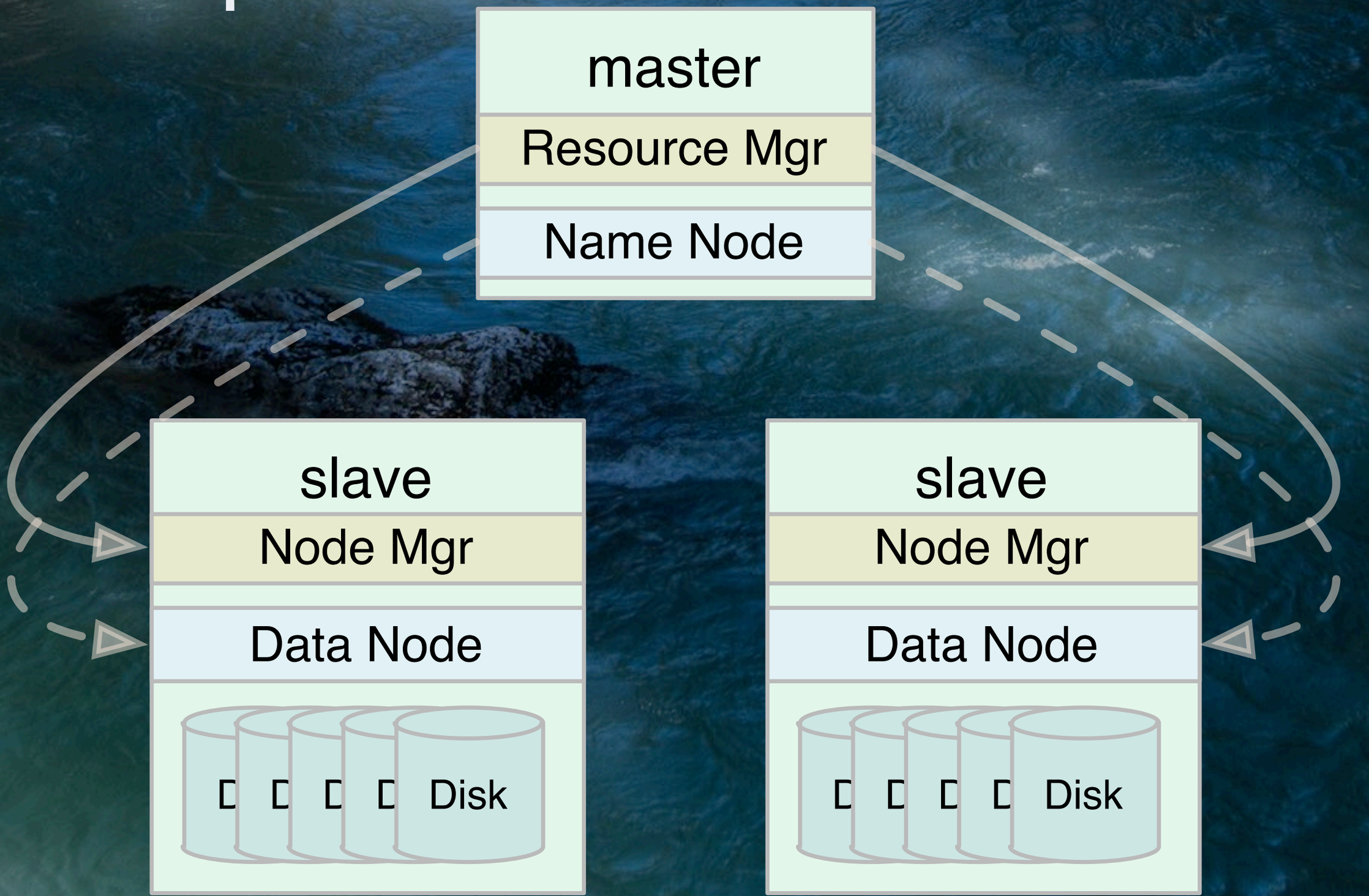
- 4000 nodes
- 2 quad core Xeons @ 2.5ghz per node
- 4x1TB SATA disks per node
- 8G RAM per node
- 1 gigabit ethernet on each node
- 40 nodes per rack
- 4 gigabit ethernet uplinks from each rack to the core (unfortunately a misconfiguration, we usually do 8 uplinks)
- Red Hat Enterprise Linux AS release 4 (Nahant Update 5)
- Sun Java JDK 1.6.0_05-b13
- So that's well over 30,000 cores with nearly 16PB of raw disk!

Quant, by today's standards

16PB

The largest clusters today are roughly 10x in size.

Hadoop



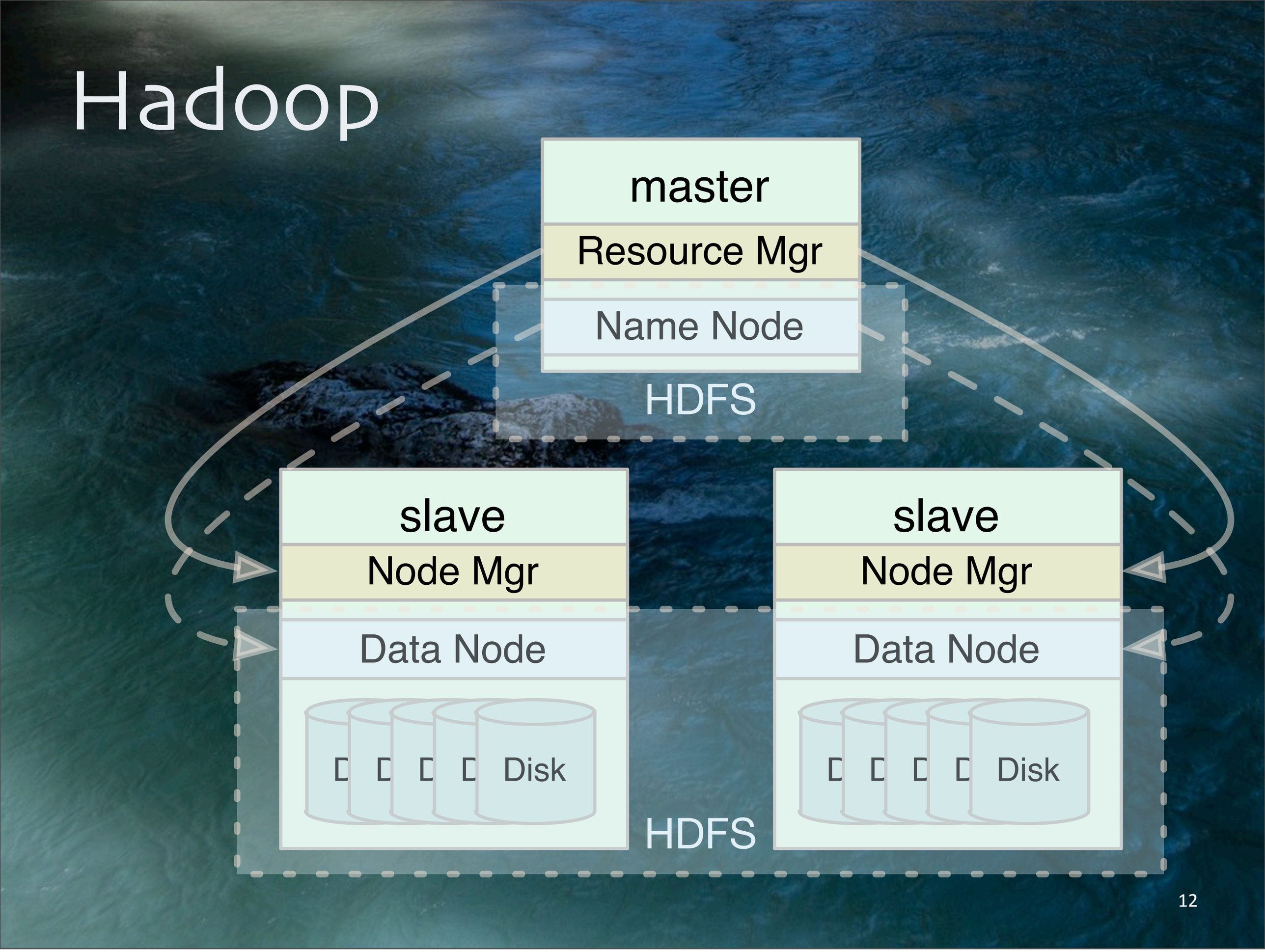
Wednesday, March 18, 15

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

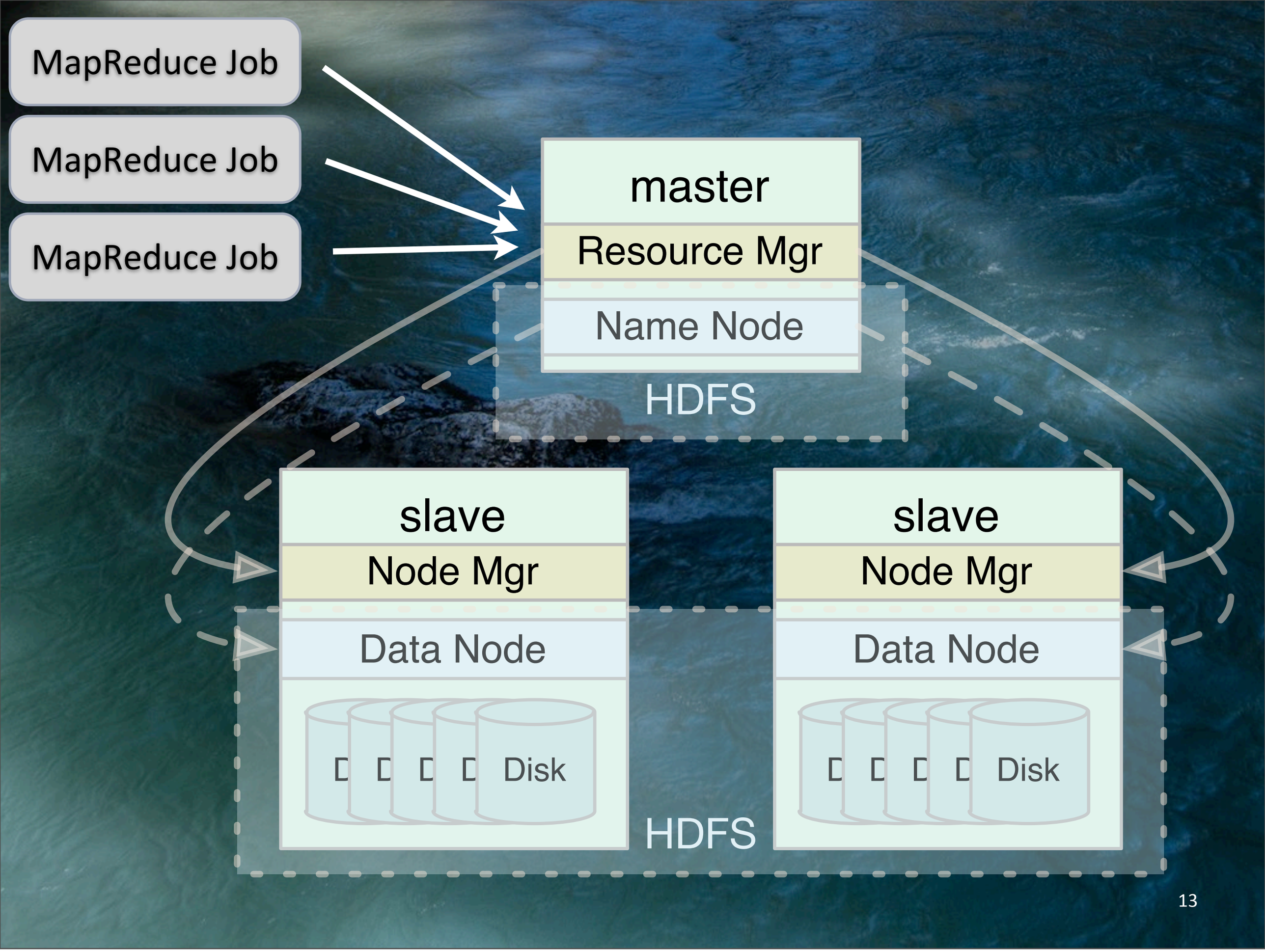


Wednesday, March 18, 15

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.



Wednesday, March 18, 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

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

Example: Inverted Index

wikipedia.org/hadoop

Hadoop provides MapReduce and HDFS

...

wikipedia.org/hbase

HBase stores data in HDFS

...

wikipedia.org/hive

Hive queries HDFS files and

inverse index

block

...	...
hadoop	(.../hadoop,1)
hbase	(.../hbase,1),(.../hive,1)
hdfs	(.../hadoop,1),(.../hbase,1),(.../hive,1)
hive	(.../hive,1)
...	...

block

...	...

block

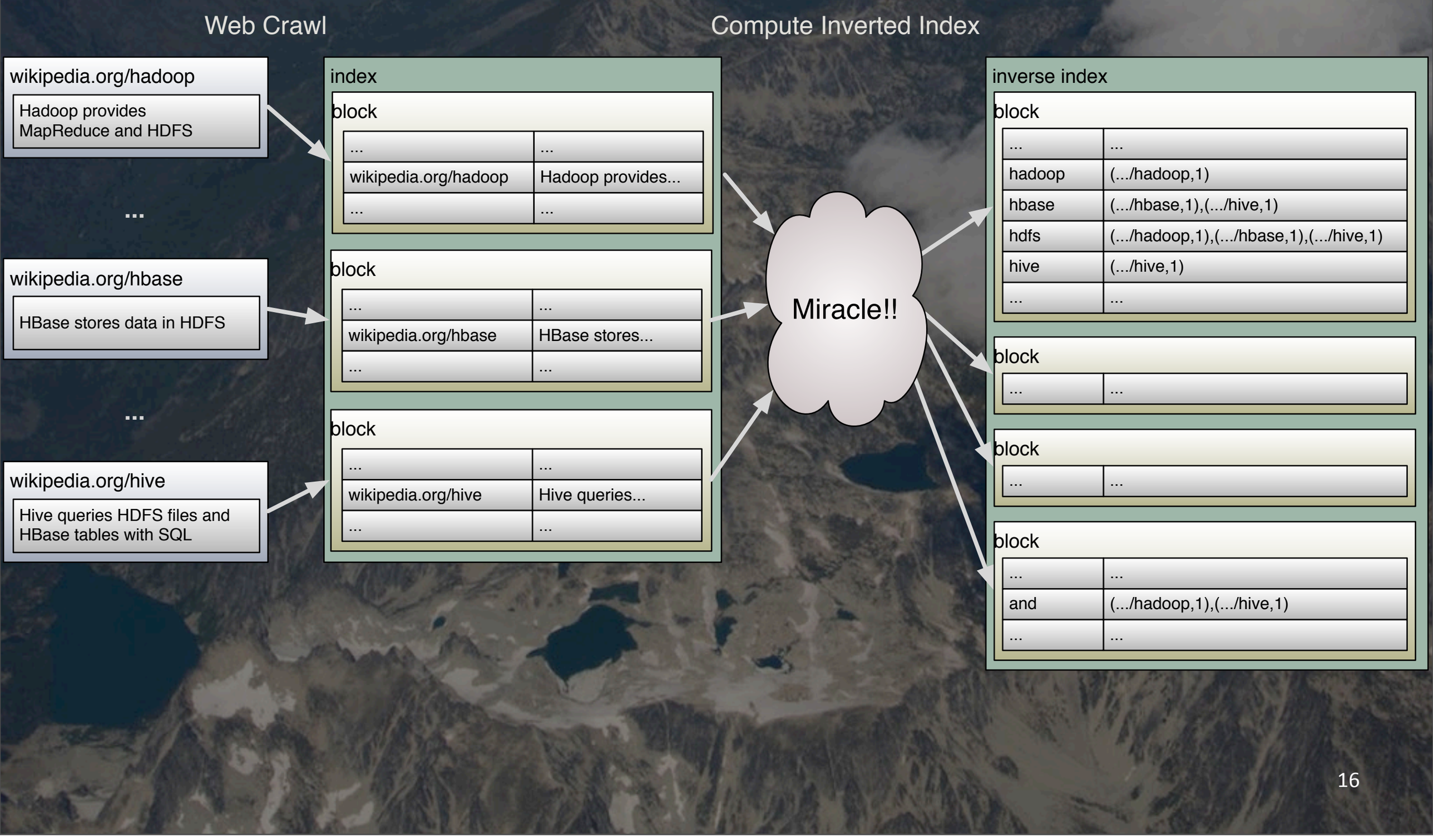
...	...

block

Wednesday, March 18, 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



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.

Web Crawl

wikipedia.org/hadoop

Hadoop provides
MapReduce and HDFS

...

wikipedia.org/hbase

HBase stores data in HDFS

...

index

block

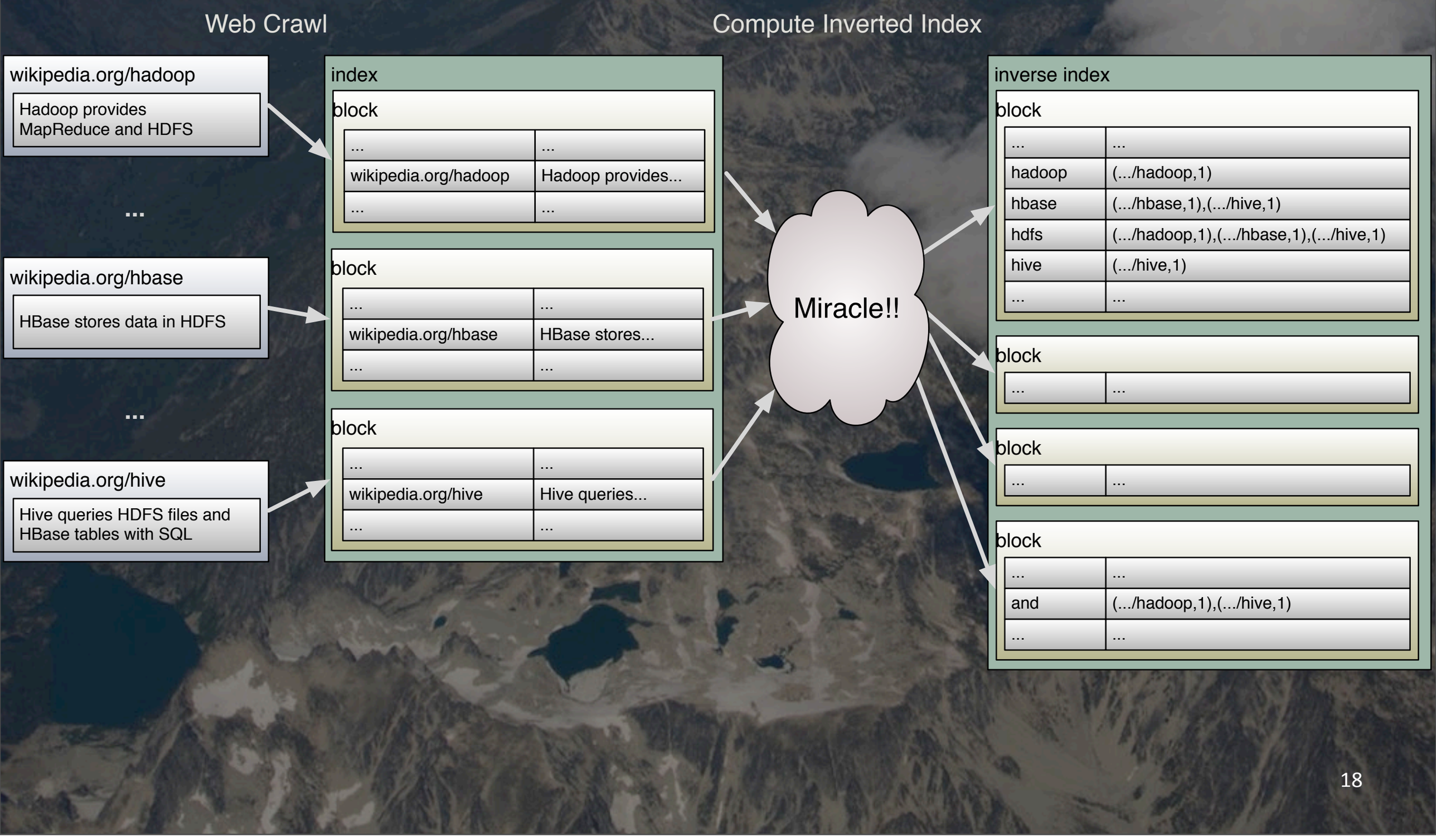
...	...
wikipedia.org/hadoop	Hadoop provides...
...	...

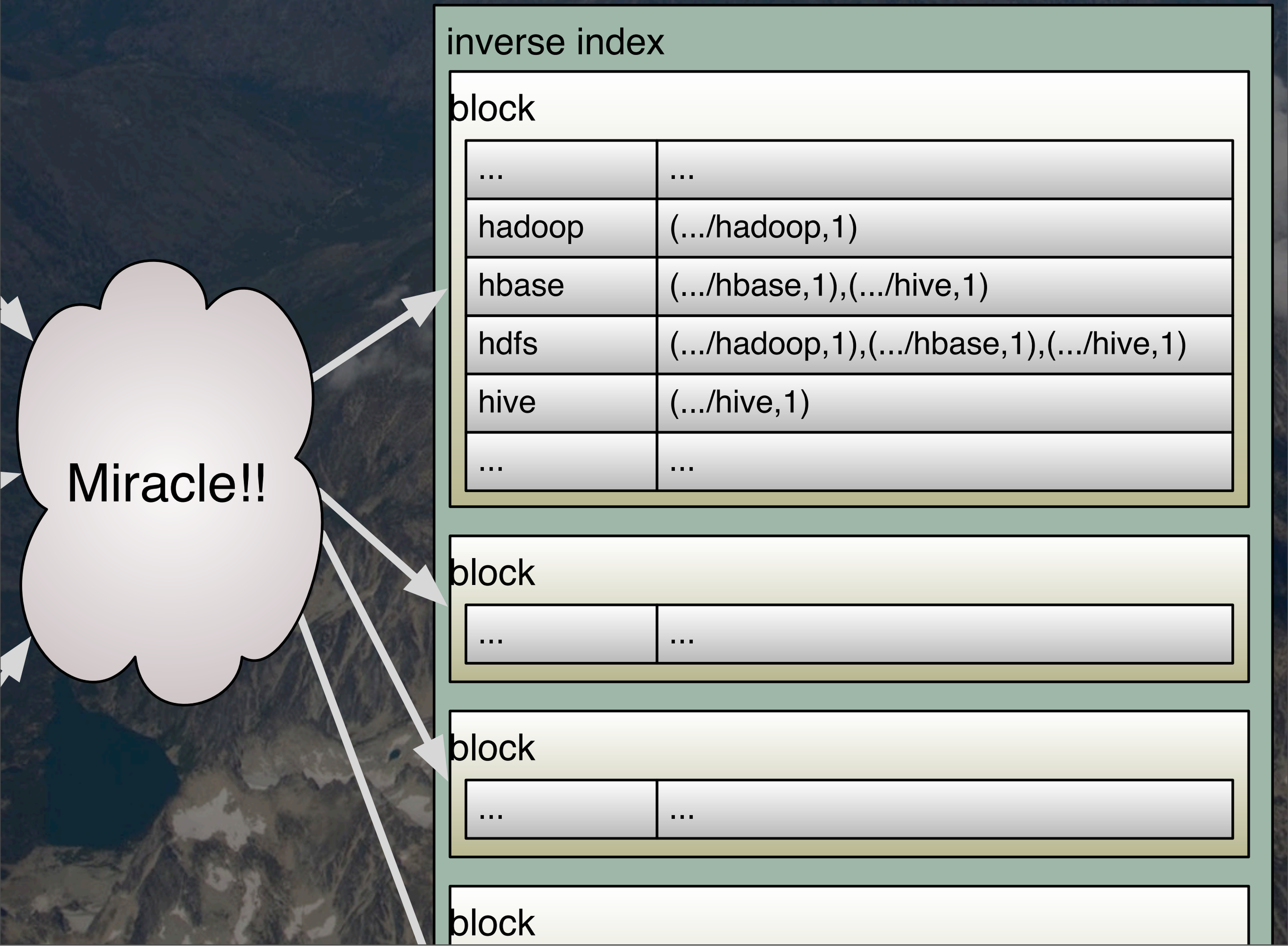
block

...	...
wikipedia.org/hbase	HBase stores...
...	...

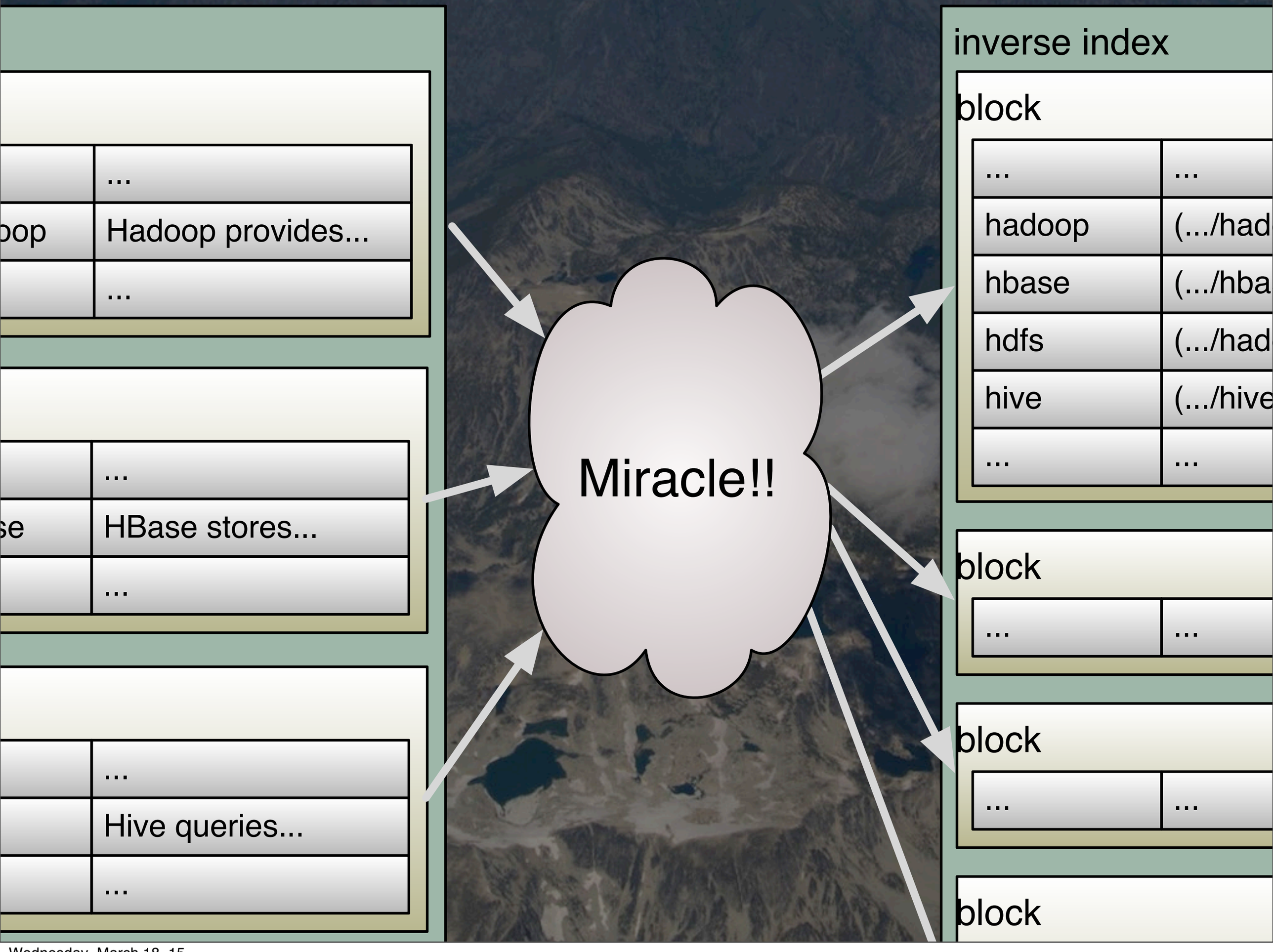
block

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).


Web Crawl
Compute Inverted Index
wikipedia.org/hadoop
Hadoop provides MapReduce and HDFS
...
wikipedia.org/hbase
HBase stores data in HDFS
...
wikipedia.org/hive
Hive queries HDFS files and HBase tables with SQL
index
block
... ...
wikipedia.org/hadoop Hadoop provides...
... ...
block
... ...
wikipedia.org/hbase HBase stores...
... ...
block
... ...
wikipedia.org/hive Hive queries...
... ...
Miracle!!
inverse index
block
... ...
hadoop (.../hadoop,1)
hbase (.../hbase,1),(.../hive,1)
hdfs (.../hadoop,1),(.../hbase,1),(.../hive,1)
hive (.../hive,1)
... ...
block
... ...
block
... ...
block
... ...
and (.../hadoop,1),(.../hive,1)
... ...

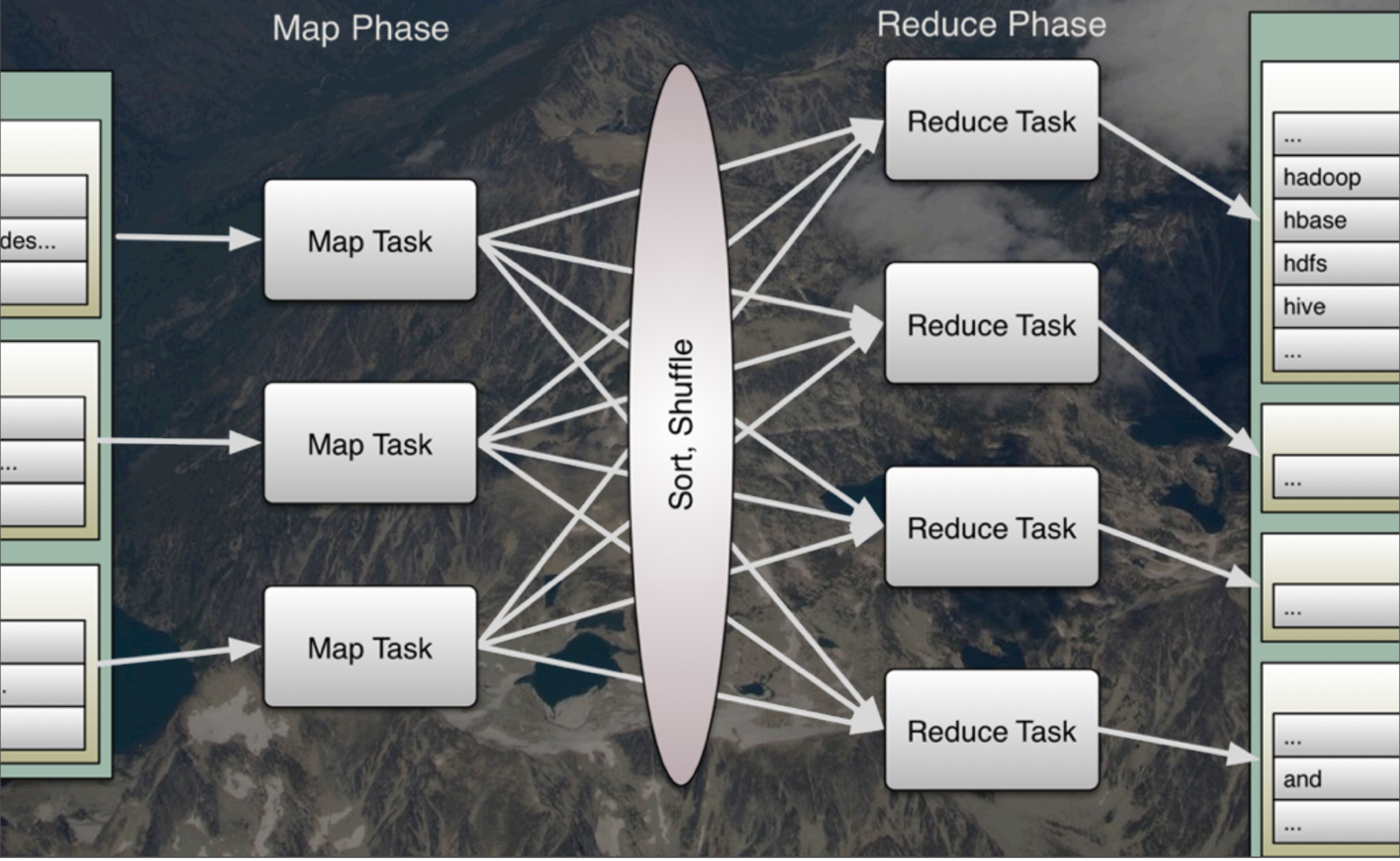


Zoom into details. This is the output we expect, a two-column dataset with word keys and a list of tuples with the doc id and count for that document.



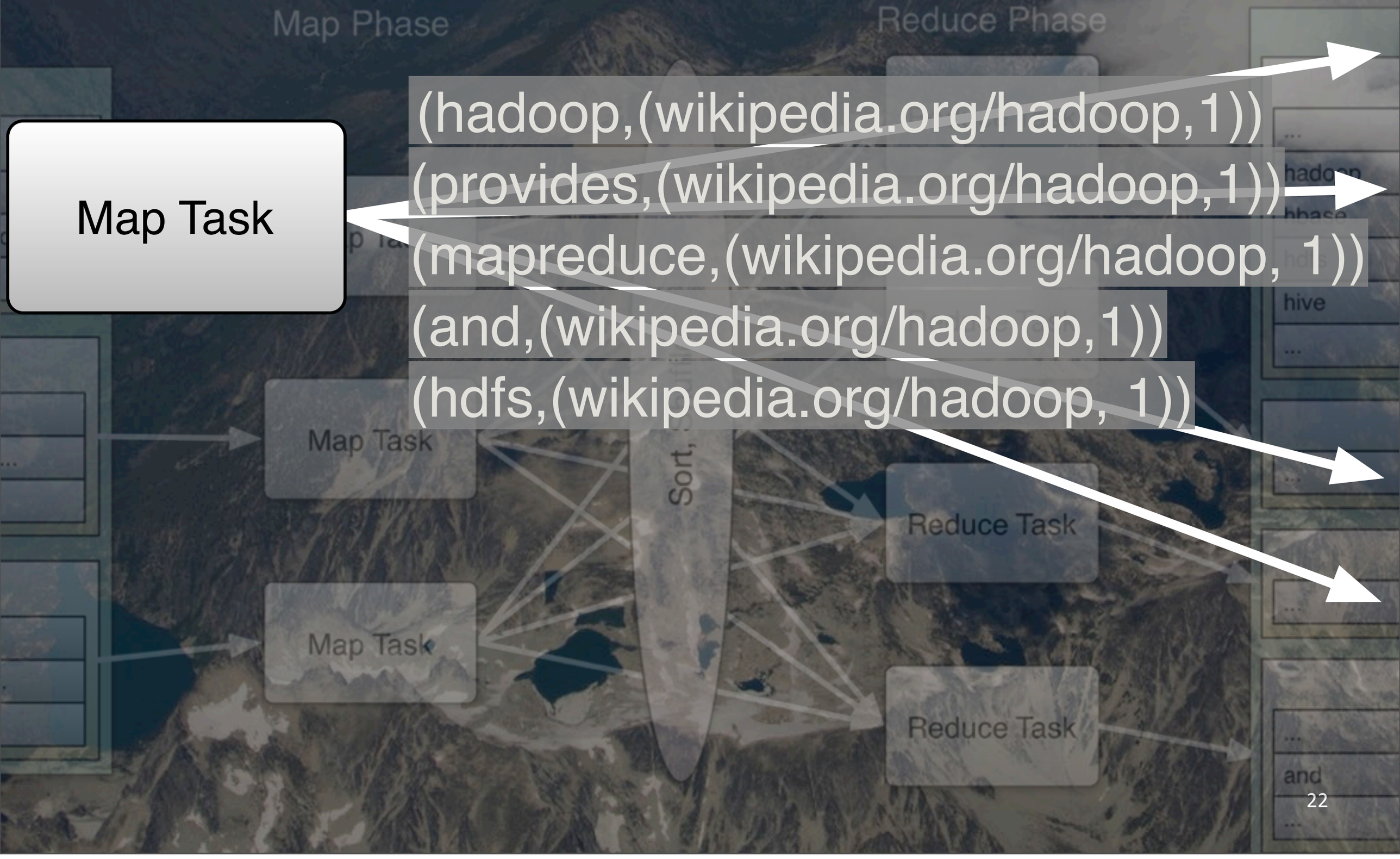
Let’s look at the “miracle”

1 Map step + 1 Reduce step



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

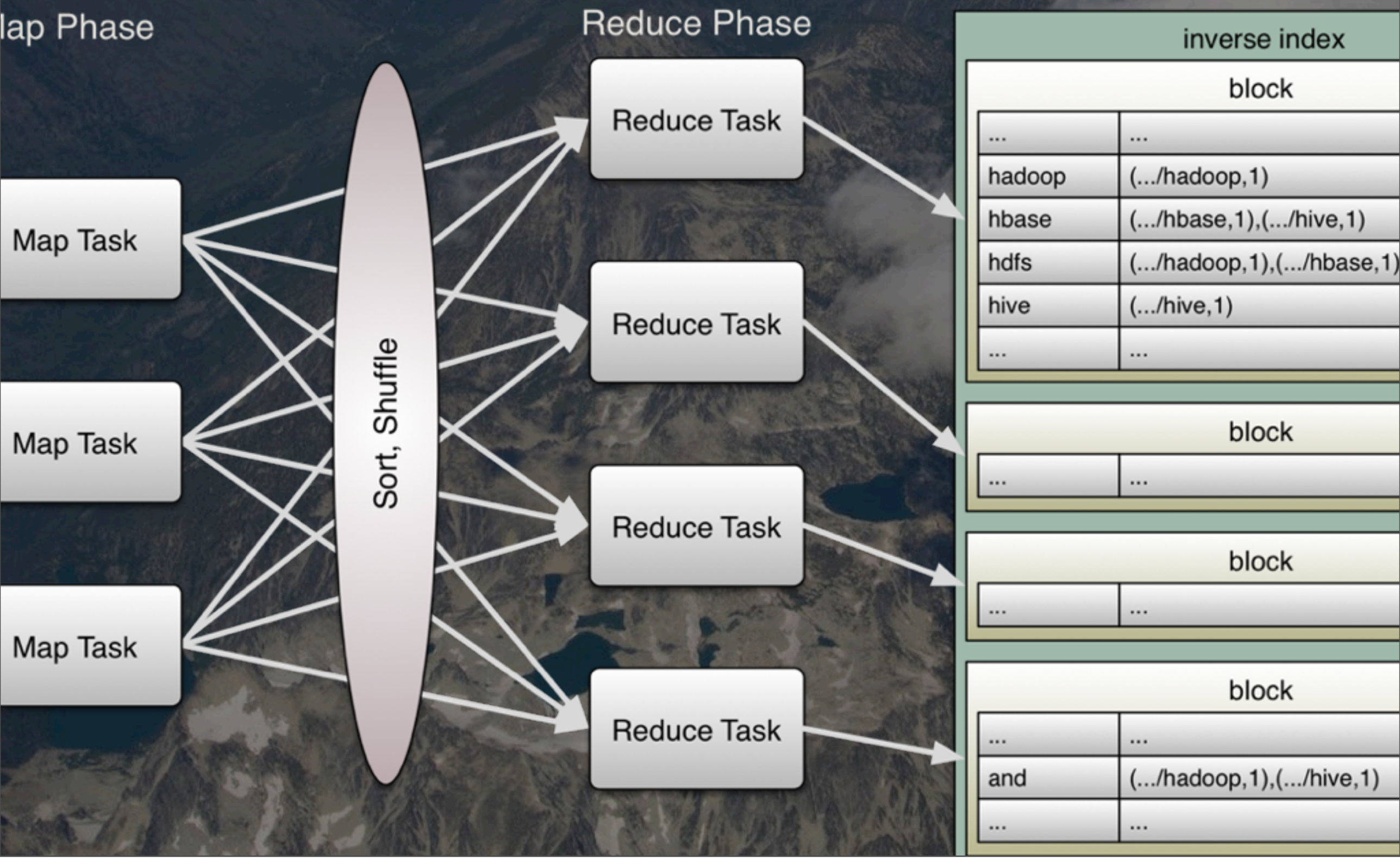
1 Map step + 1 Reduce step



Wednesday, March 18, 15

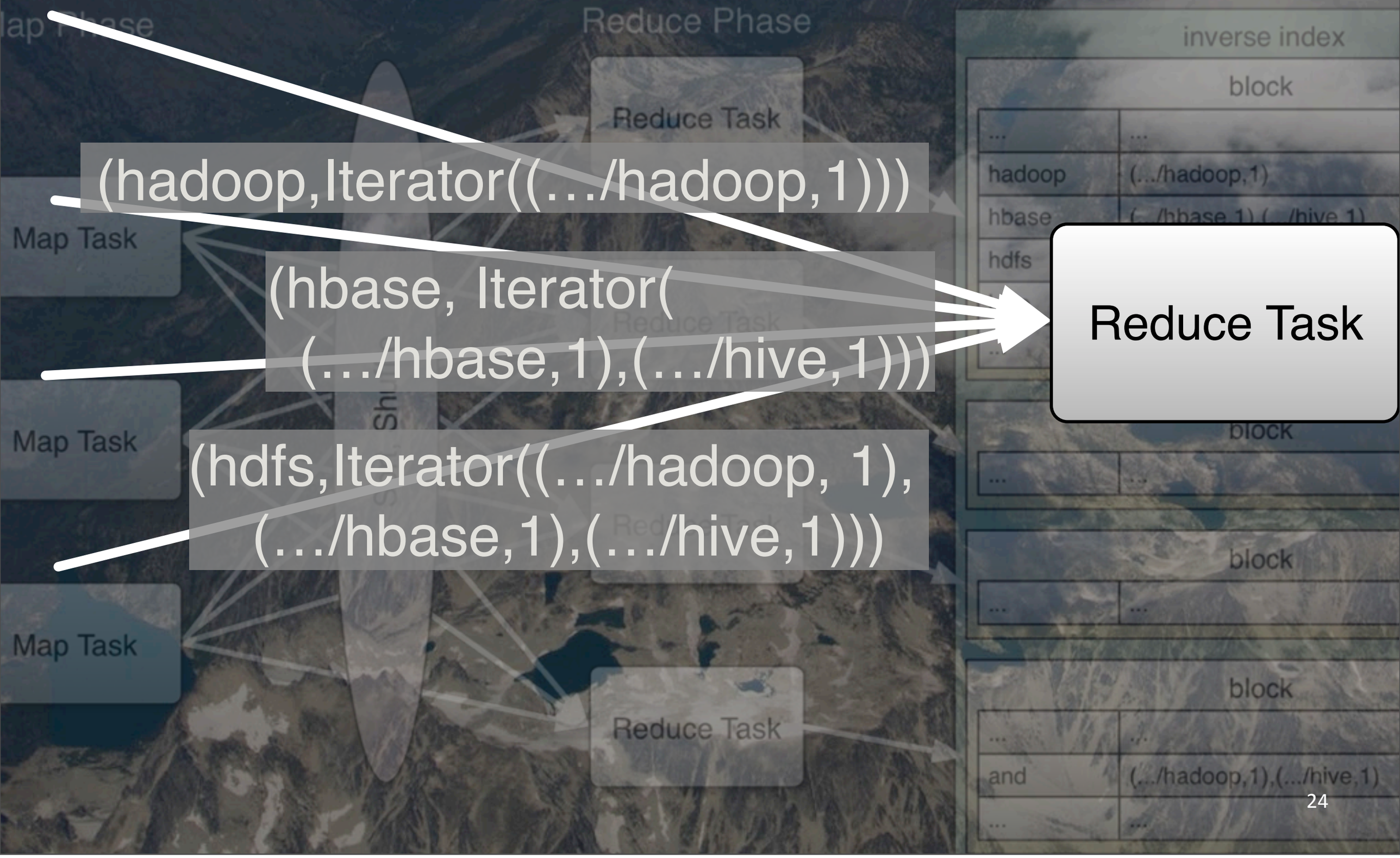
Each map task parses the records. It tokenizes the contents and write new key-value pairs (shown as tuples here), with the word as the key, and the rest, shown here as a second element that is itself a tuple, which holds the document id and the count.

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.

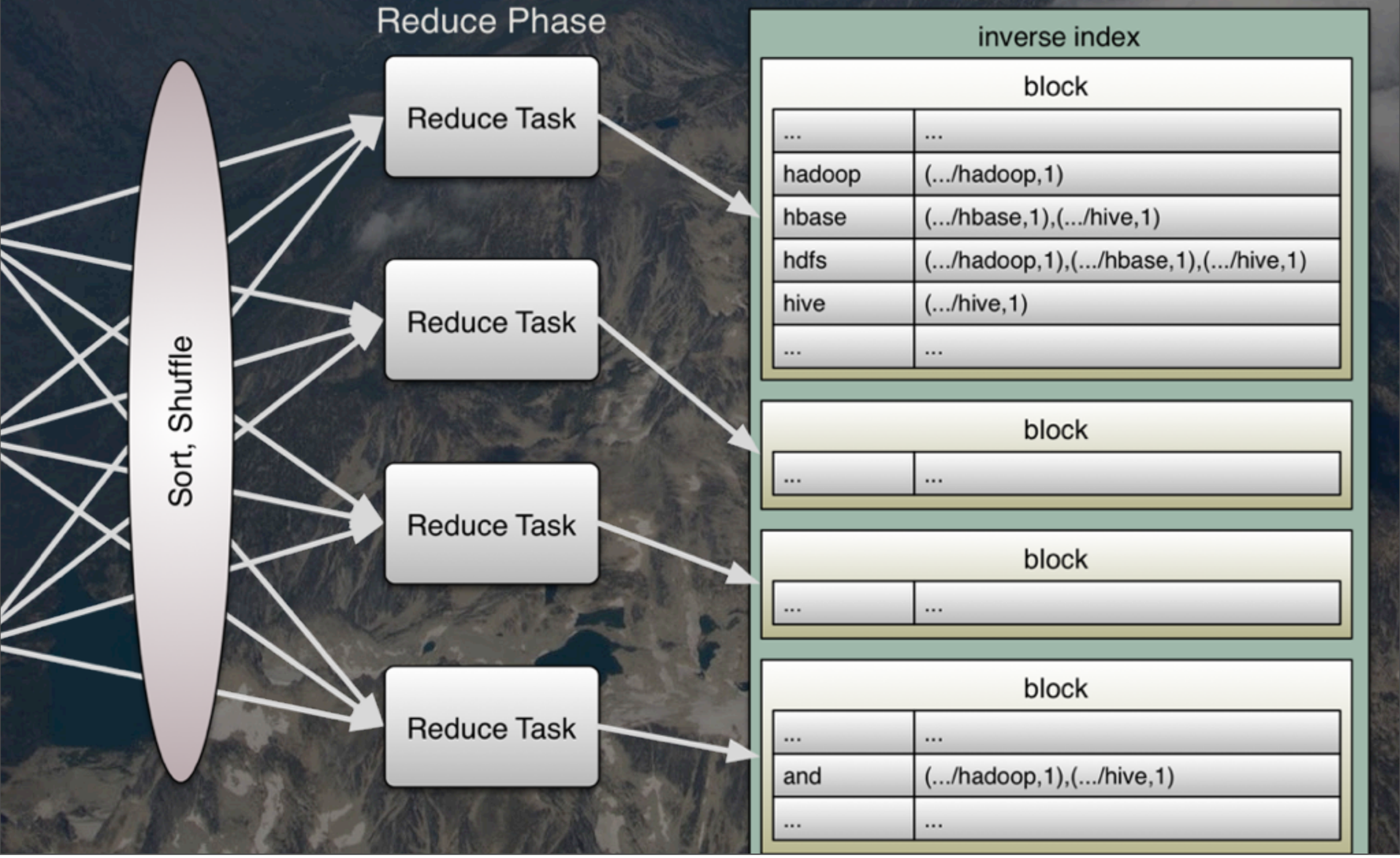
1 Map step + 1 Reduce step



Wednesday, March 18, 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



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.

Problems

Hard to implement algorithms...

Wednesday, March 18, 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.

Problems

... and the Hadoop API is horrible...

Wednesday, March 18, 15

The Hadoop API is very low level and tedious...

```
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);
```


For example, the classic inverted index, used to convert an index of document locations (e.g., URLs) to words into the reverse; an index from words to doc locations. It's the basis of search engines.

I'm not going to explain the details. The point is to notice all the boilerplate that obscures the problem logic.

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

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

```
JobClient client = new JobClient();
JobConf conf =
 new JobConf(LineIndexer.class);

conf.setJobName("LineIndexer");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
FileInputFormat.addInputPath(conf,
 new Path("input"));
FileOutputFormat.setOutputPath(conf,
 new Path("output"));
conf.setMapperClass(
   LineIndexMapper.class);
conf.setReducerClass(
   LineIndexReducer.class);

client.setConf(conf);
```


Boilerplate: The red are methods and it should be true that we care most about red, because methods/functions to the real work. However, these are trivial property setters. They take up a lot of space and don't deliver much value.

```
    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
```


Wednesday, March 18, 15

main ends with a try-catch clause to run the job.

```
public static class LineIndexMapper
 extends MapReduceBase
 implements Mapper<LongWritable, Text,
                    Text, Text> {
 private final static Text word =
  new Text();
 private final static Text location =
  new Text();

 public void map(
 LongWritable key, Text val,
 OutputCollector<Text, Text> output,
 Reporter reporter) throws IOException {

 FileSplit fileSplit =
  (FileSplit)reporter.getInputSplit();
 String fileName =
```


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
```


Wednesday, March 18, 15

The rest of the LineIndexMapper class and map method.

```
public static class LineIndexReducer
 extends MapReduceBase
 implements Reducer<Text, Text,
                    Text, Text> {
 public void reduce(Text key,
  Iterator<Text> values,
  OutputCollector<Text, Text> output,
  Reporter reporter) throws IOException {
  boolean first = true;
  StringBuilder toReturn =
   new StringBuilder();
 while (values.hasNext()) {
  if (!first)
   toReturn.append(", ");
  first=false;
  toReturn.append(
   values.next().toString());
```


The reducer class, LineIndexReducer, with the reduce method that is called for each key and a list of values for that key. The reducer is stupid; it just reformats the values collection into a long string and writes the final (word,list-string) output.

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

I lied; I didn't count occurrences

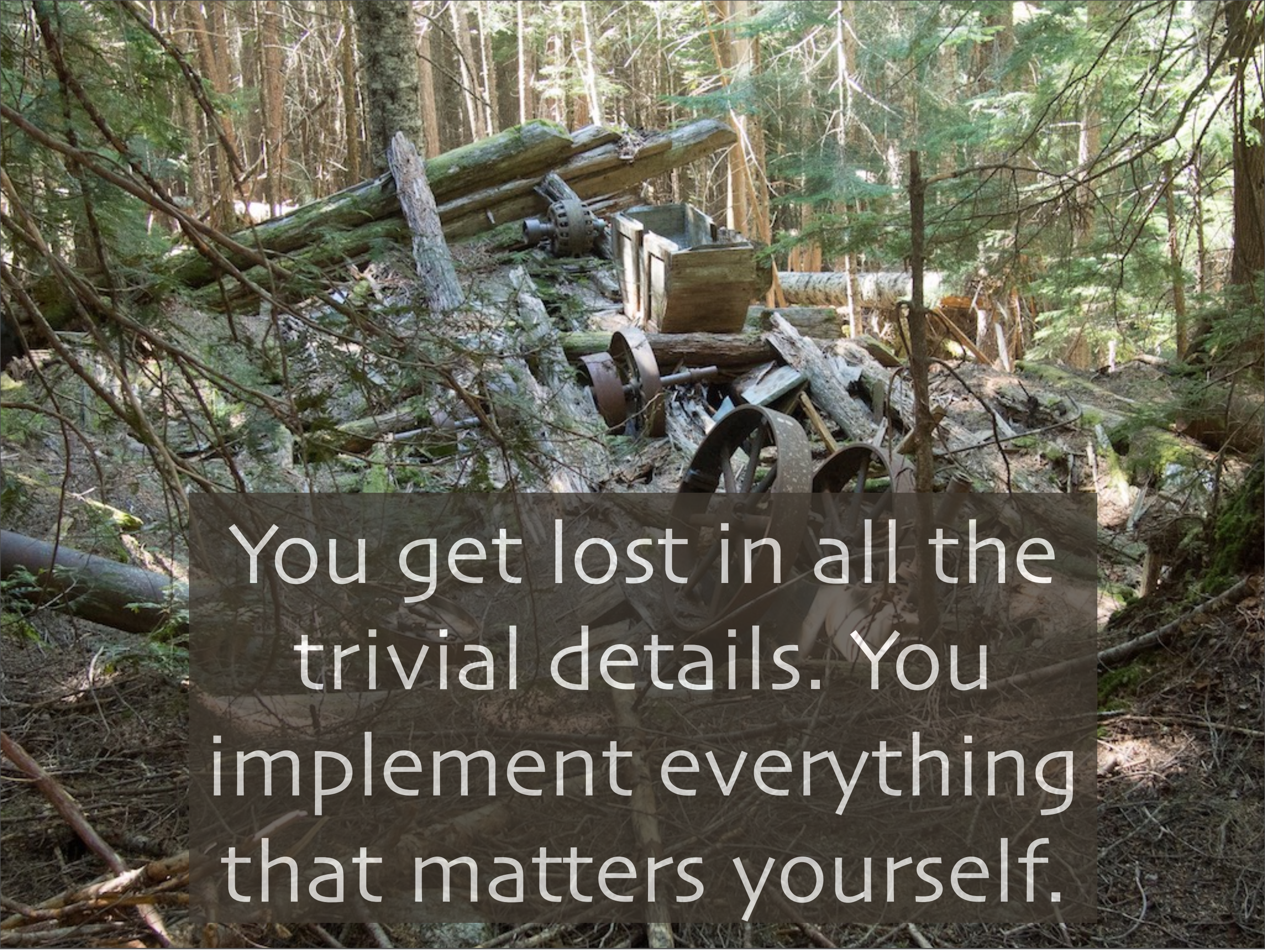
Wednesday, March 18, 15

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. Note that I'm not computing the counts of words per document, so I lied...
EOF

```
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

The whole shebang (6pt. font) This would take a few hours to write, test, etc. assuming you already know the API and the idioms for using it.


You get lost in all the
trivial details. You
implement everything
that matters yourself.

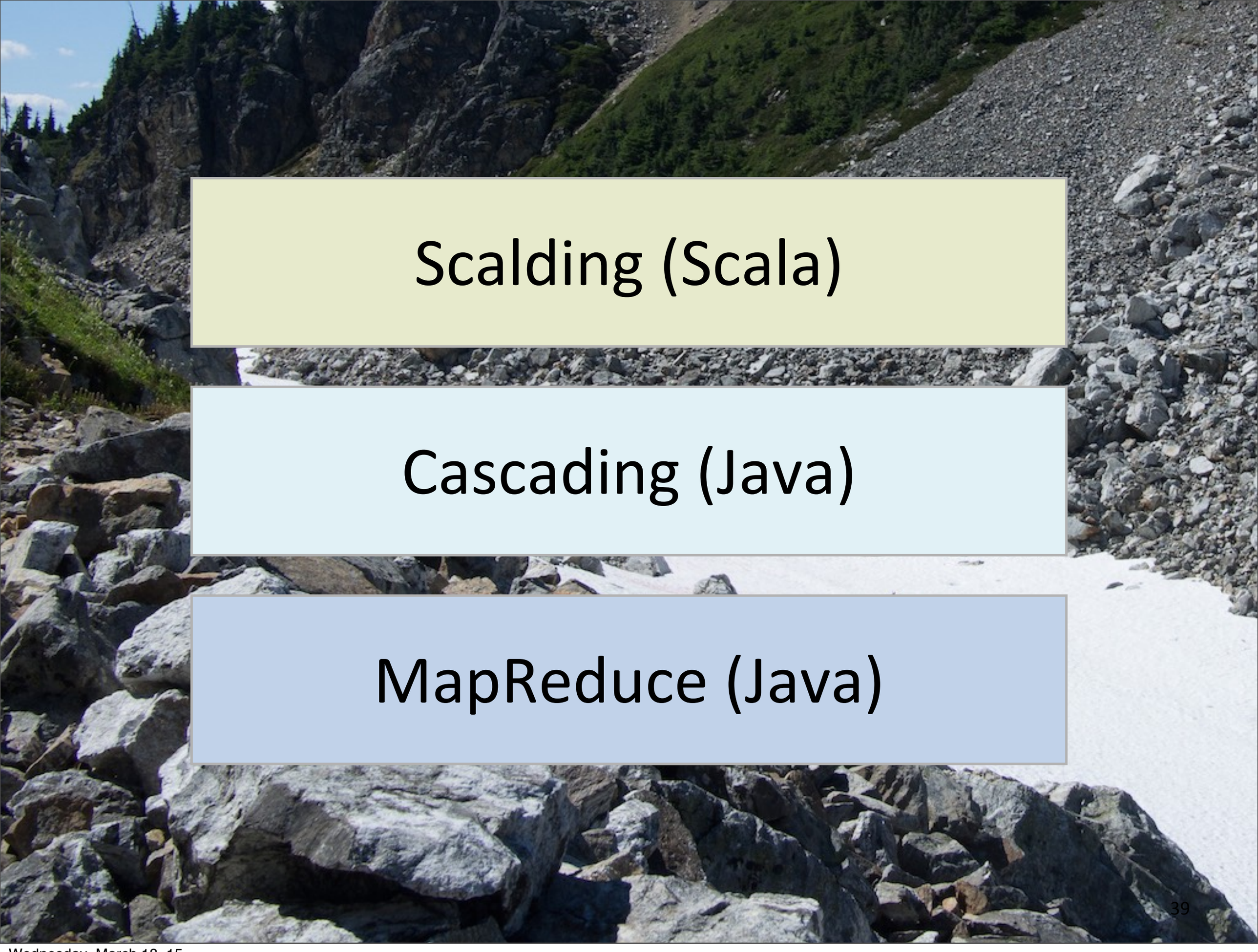
Seems a little daunting

. . .

How do we get around these problems??

Wednesday, March 18, 15

Twitter wrote a Scala API, https://github.com/twitter/scalding, to hide the mess. Actually, Scalding sits on top of Cascading (http://cascading.org) a higher-level Java API that exposes more sensible “combinators” of operations, but is still somewhat verbose due to the pre-Java 8 conventions it must use. Scalding gives us the full benefits of Scala syntax and functional operations, “combinators”.



Wednesday, March 18, 15

Twitter wrote a Scala API, https://github.com/twitter/scalding, to hide the mess. Actually, Scalding sits on top of Cascading (http://cascading.org) a higher-level Java API that exposes more sensible “combinators” of operations, but is still somewhat verbose due to the pre-Java 8 conventions it must use. Scalding gives us the full benefits of Scala syntax and functional operations, “combinators”.

```
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)
     }
  }
```


Dramatically smaller, succinct code! (https://github.com/echen/rosetta-scone/blob/master/inverted-index/InvertedIndex.scala) Note that this example assumes a slightly different input data format; more than one document per file, with each document id followed by a tab and then the text of the document, all on a single line (embedded tabs removed!). Also, I'm using one of two Scalding APIs, the so-called "Fields" API. There is a newer "Typed" API that's similar, but provides better type checking.
Now the red methods are doing a lot of work with little code.

```
    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"))
}
```

That's it!

Problems

Only "Batch mode"

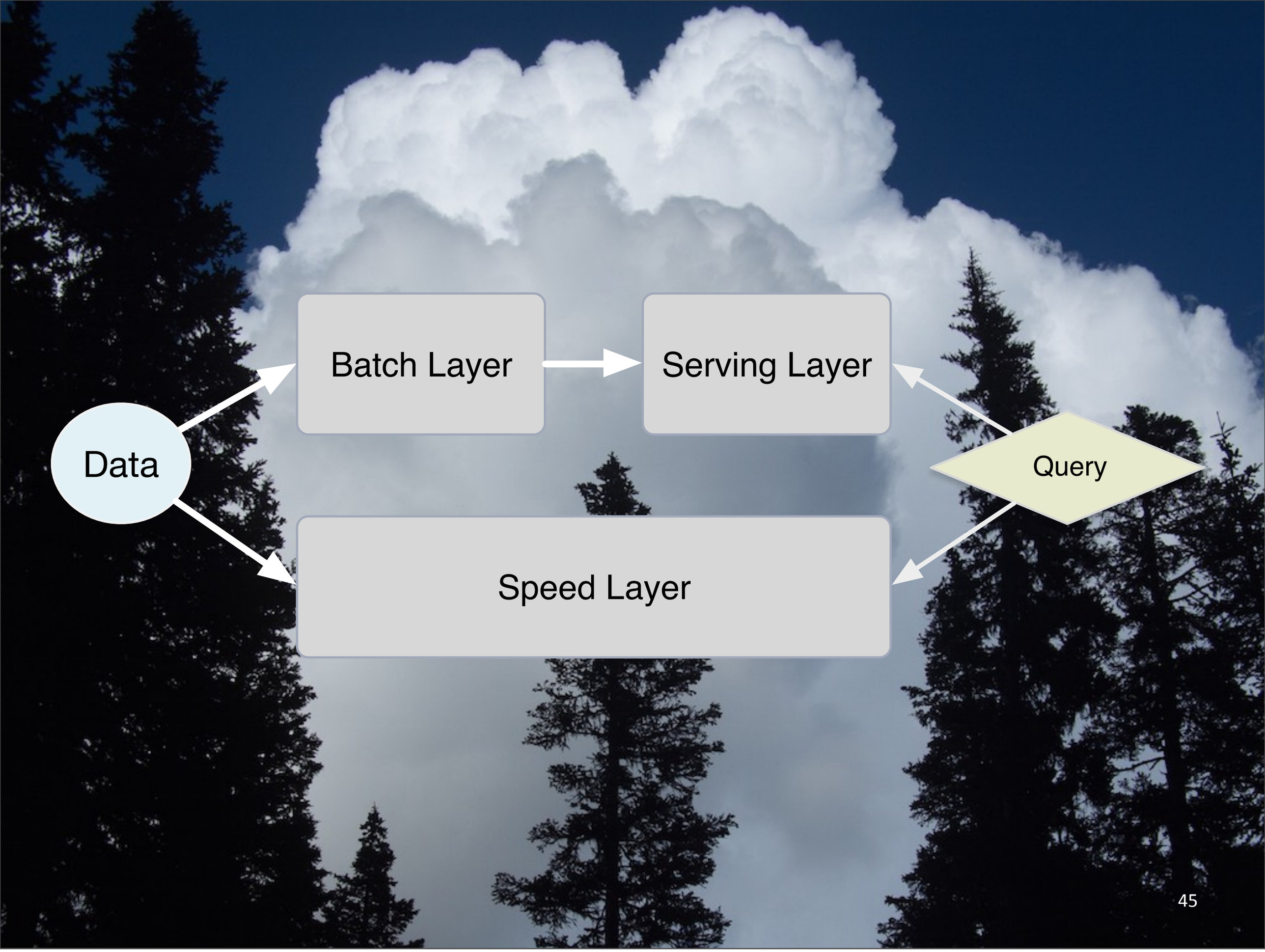
Back to MapReduce problems: Event stream processing is increasingly important, both because some systems have tight SLAs and because there is a competitive advantage to minimizing the time between data arriving and information being extracted from it, even when otherwise a batch-mode analysis would suffice. 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...).

Wednesday, March 18, 15

Can we support event streaming, of some kind?

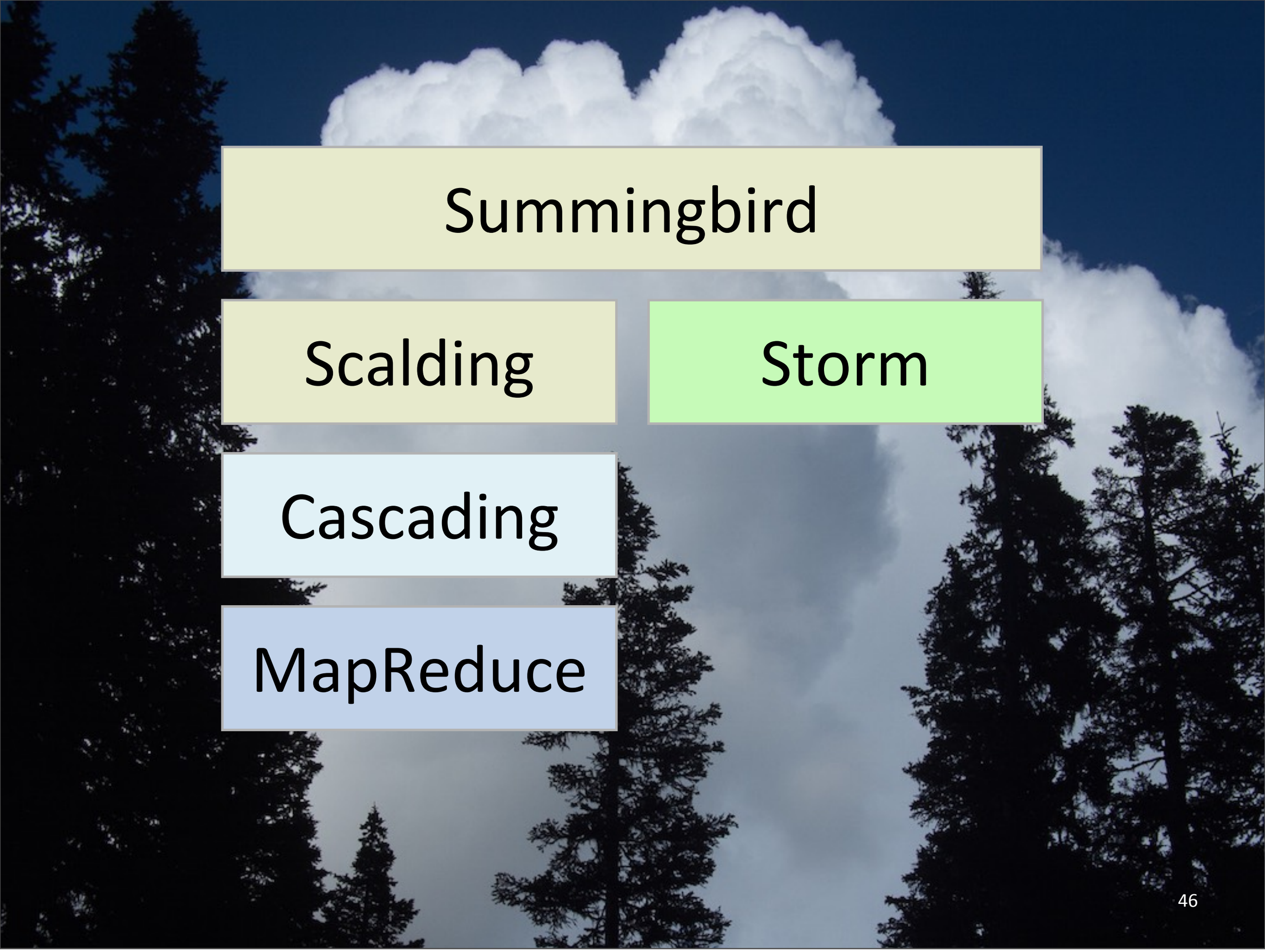
Wednesday, March 18, 15

Nathan Marz invented Storm as a durable, distributed event stream processing system to complement MapReduce. He coined the term Lambda Architecture for systems that combine batch mode analysis of historical data with real-time analysis of incoming event streams.



Wednesday, March 18, 15

The Lambda Architecture invented by Nathan that combines event stream handling of newly arrived data and batch mode of historical data, then queries go to both. Unfortunately, this effectively means duplicating logic in both the speed and batch layers!



Wednesday, March 18, 15

And Twitter realized that many of the same operations apply equally well to batch mode and streaming analysis, so they abstracted much of the Scalding API into a new API called Summingbird (https://github.com/twitter/summingbird) to be a layer over both Scalding and Storm to promote reuse. It did have mixed success, however, in part due to many feature differences between Storm and MapReduce. I won't show an example for time's sake. We'll discuss an alternative approach to this problem in a moment.

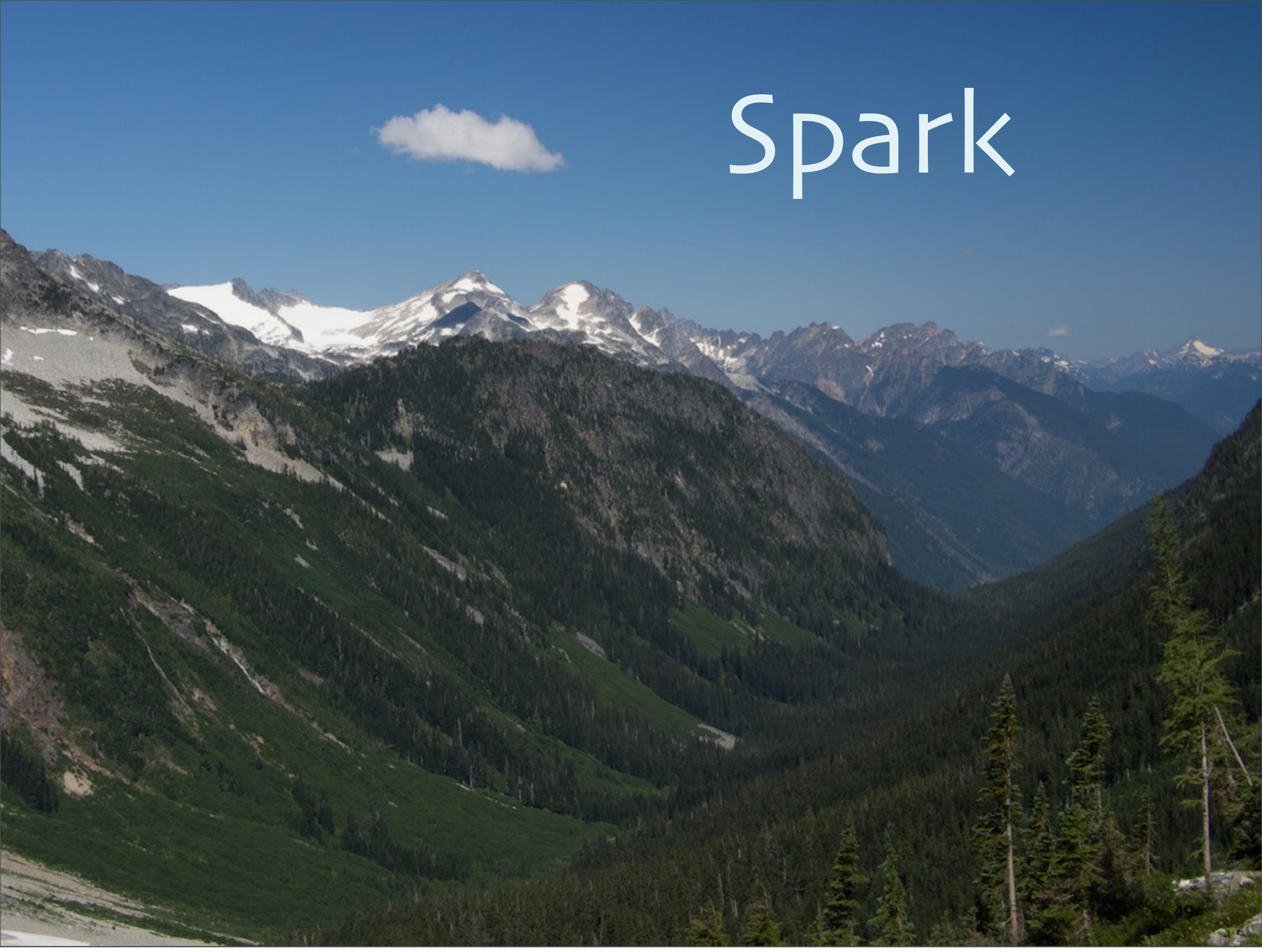
Problems

Very inefficient

More MR problems: The runtime doesn't understand the full dataflow, so if you sequence several MR jobs together, MR can't know to cache intermediate data in memory between jobs. Nor can it combine or otherwise optimize the flow.

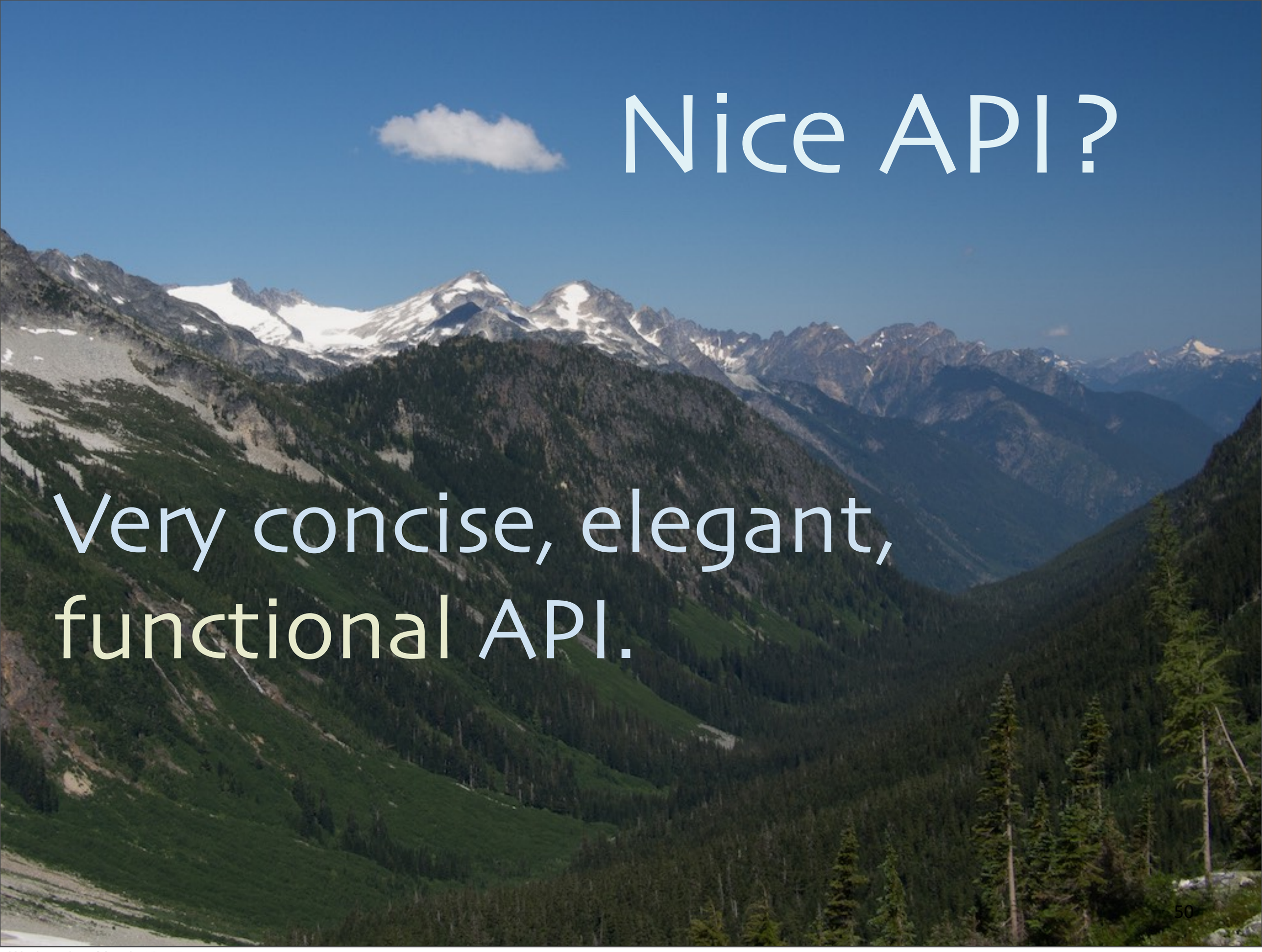
Wednesday, March 18, 15

Can we also address this need at the same time?



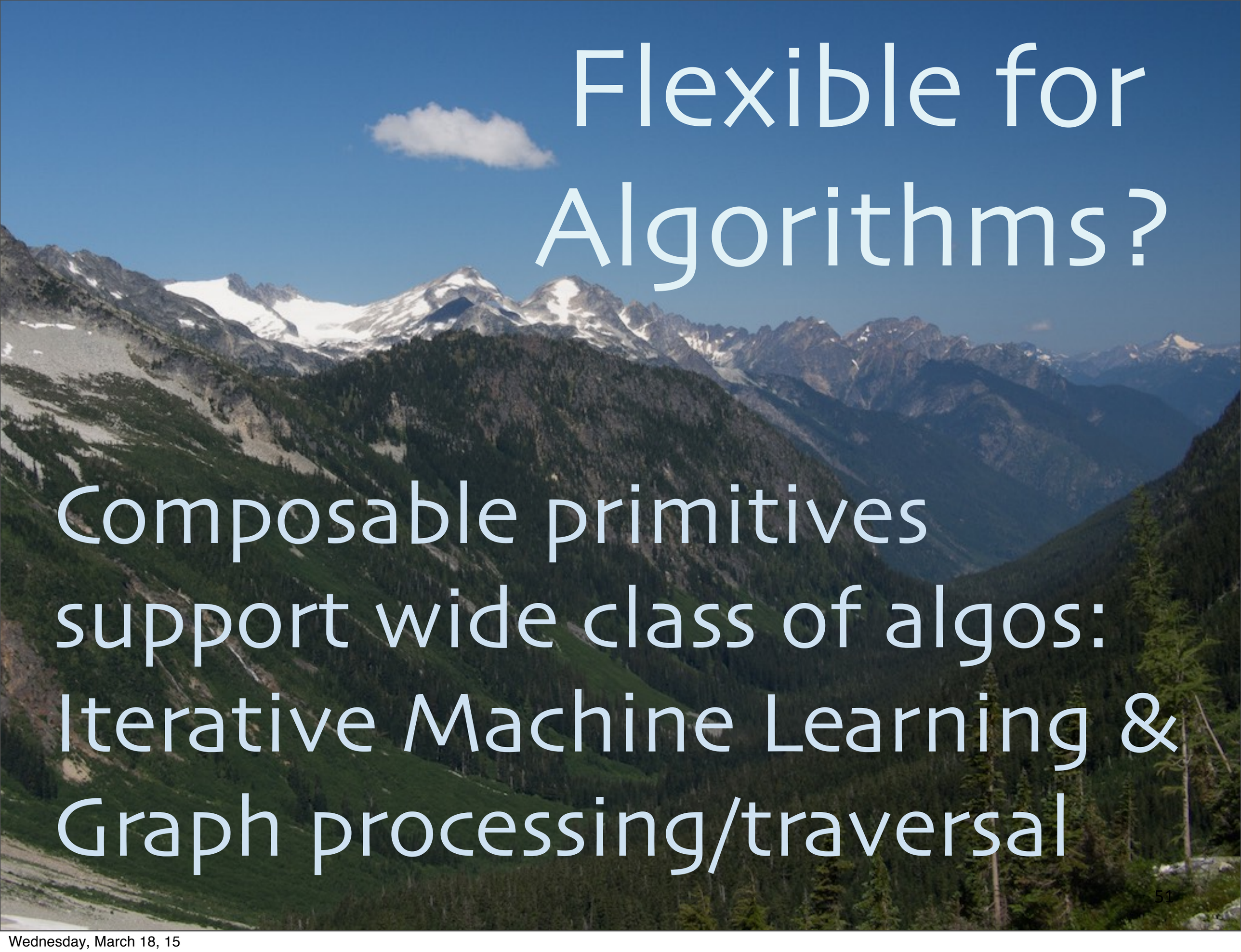
Wednesday, March 18, 15

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.



Wednesday, March 18, 15

We'll see by example shortly why this true.



Wednesday, March 18, 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.

Wednesday, March 18, 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.

Wednesday, March 18, 15

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.

```
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) =>
```


Wednesday, March 18, 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) =>
```


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.

```
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).

```
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.

```
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.

```
    }
  }
  .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)
…

Wednesday, March 18, 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)
…

Wednesday, March 18, 15

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), ...))

Wednesday, March 18, 15

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()
 }
}
```


Wednesday, March 18, 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.

```
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

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!!

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

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

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

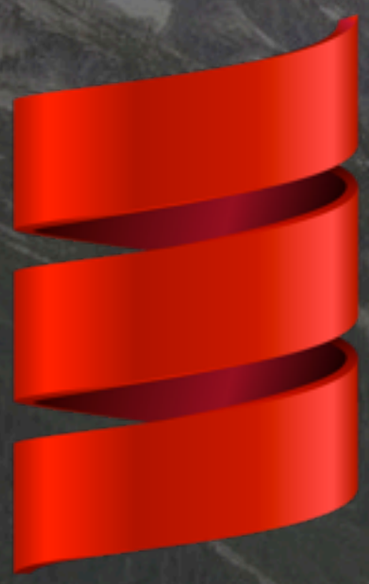
$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

Another example of a beautiful and profound DSL, in this case from the world of my first profession, Physics: Maxwell's equations that unified Electricity and Magnetism: http://upload.wikimedia.org/wikipedia/commons/c/c4/Maxwell'sEquations.svg

Wednesday, March 18, 15

Other reasons why Scala taking over the big data world.

The JVM



Algebird

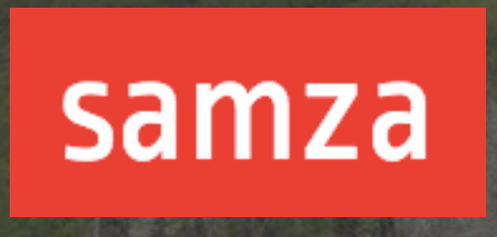
Spire

ScalaNLP

Wednesday, March 18, 15

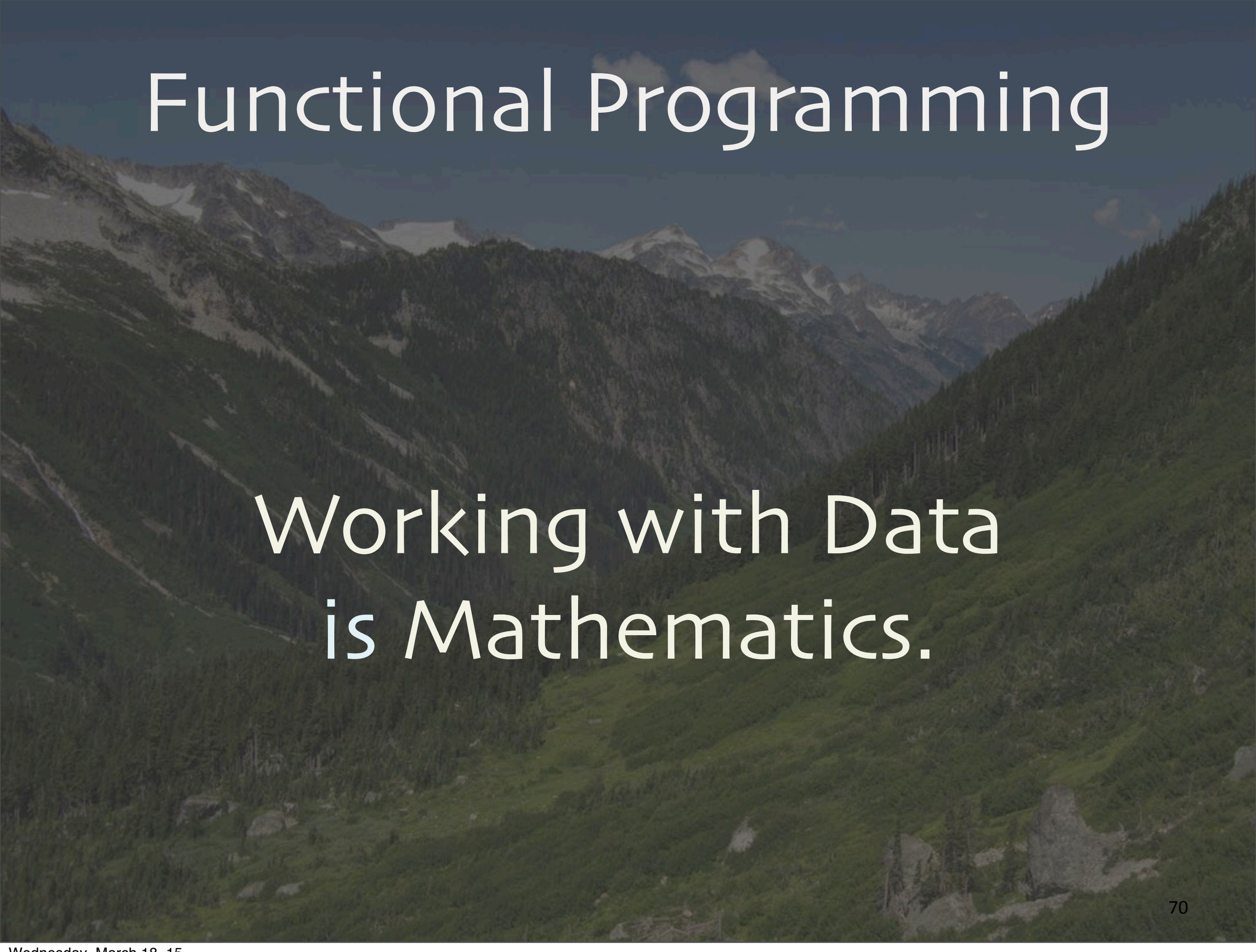
You have the rich Java ecosystem at your fingertips. Some core languages and tools.

Big Data Tools



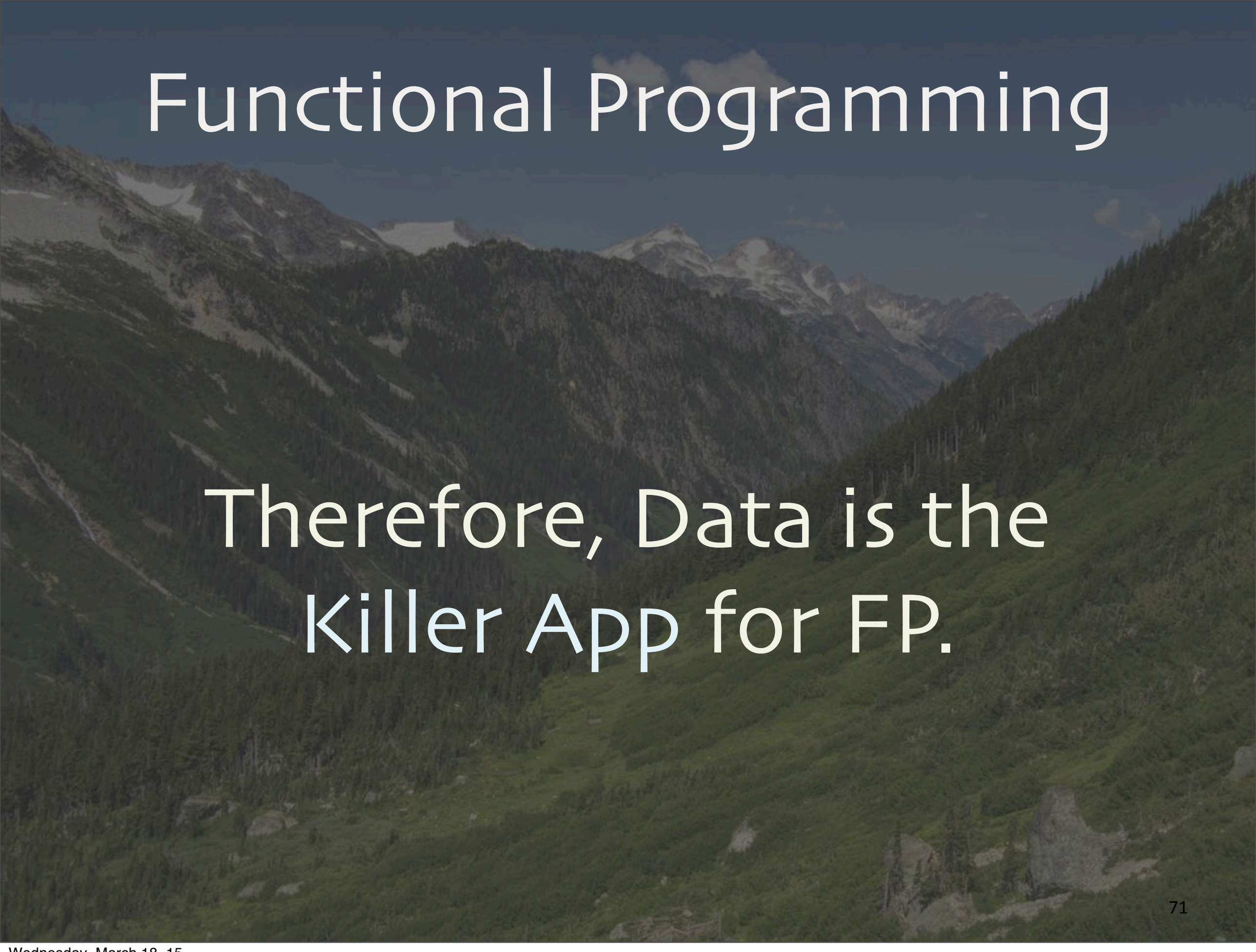
Wednesday, March 18, 15

You have the rich Java ecosystem at your fingertips. This is a small sample of tools...



Wednesday, March 18, 15

You could reasonably argue that Scala was in the right place at the right time, that the real winner in the Big Data world is FP, because it's such an obvious fit for working with data. As an FP language on the JVM, Scala was well placed for this opportunity.



Wednesday, March 18, 15

You could reasonably argue that Scala was in the right place at the right time, that the real winner in the Big Data world is FP, because it's such an obvious fit for working with data. As an FP language on the JVM, Scala was well placed for this opportunity.

Mathematics

Algebird

For example: Addition

https://github.com/twitter/algebird

Properties of Addition

–A set of elements:

$$\{1, 2, 3, 4, 5, 6, 7, \ldots\}$$

–An associative binary operation:

$$(a + b) + c = a + (b + c)$$

–An identity element:

$$a + o = o + a = a$$

Wednesday, March 18, 15

Addition? That seems kind of trivial!

Generalize addition: Monoid

–A set of elements:

{a, b, c, d, e, f, g, ...}

–An associative binary operation:

(a + b) + c = a + (b + c)

–An identity element: "zero"

a + zero = zero + a = a

Wednesday, March 18, 15

Generalize addition? That seems kind of trivial!

Other Monoids!

Examples:

–Top K

–Average

–Max/Min

—...

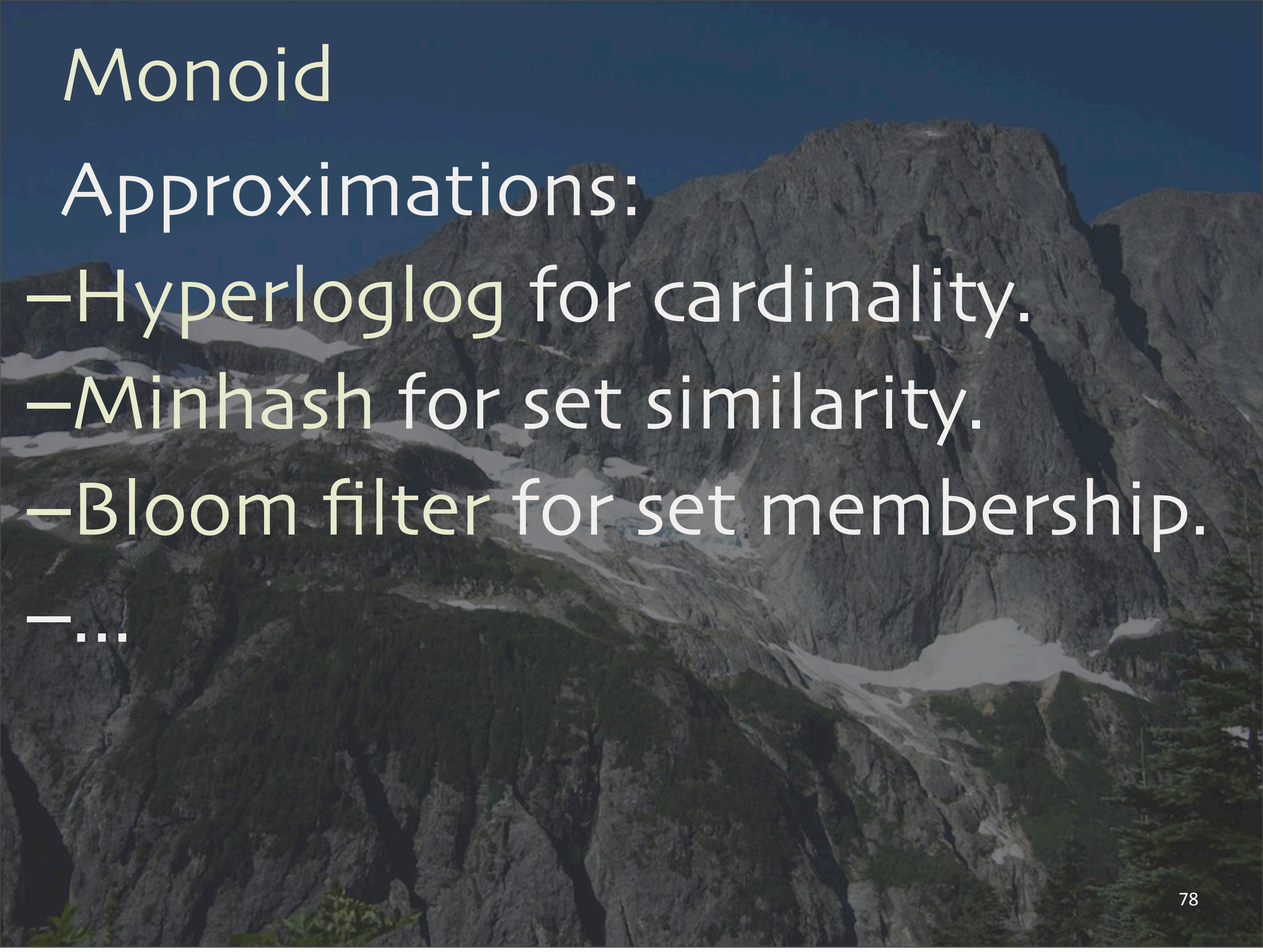
But it turns out that a generic infrastructure for addition can be applied to many calculations. These are easy to implement., but a naive implementation that does exact calculations of these items can require a lot of resources, prohibitive for big data!

Algebird

- Tunable accuracy vs. performance
  - Trade % error for low memory.
  - Approximate answers often good enough.

Wednesday, March 18, 15

For very big data sets, the CPU/memory requirements can be large for some algorithms, but often you don't need exact answers. In fact, you're willing to trade off space utilization vs. accuracy!



Wednesday, March 18, 15

For very big data sets, the CPU/memory requirements can be large for some algorithms, but often you don’t need exact answers. In fact, you’re willing to trade off space utilization vs. accuracy!

Wednesday, March 18, 15

Sampling was a common way to deal with excessive amounts of data. The new mantra, exemplified by this catch phrase from Twitter's data teams is to use approximation algorithms where the data is usually hashed into space-efficient data structures. You make a space vs. accuracy trade off. Often, approximate answers are good enough. With sampling, such tradeoffs are murkier and it can be expensive to extract the sample!

Algebird

Efficient approximation algorithms.

– “Add All the Things”, Avi Bryant:

–infoq.com/presentations/abstract-algebra-analytics

A great presentation for how the Monoid category generalizes to all sorts of useful approximation algorithms and datastructures.

Wednesday, March 18, 15

What if you need fast and robust numerical calculations, like floating point?

Spire

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

It exploits macros with generics to implement efficient computation without requiring custom types for particular primitives.

Functional Programming

You know what
else is functional?

SQL

Wednesday, March 18, 15

You could reasonably argue that Scala was in the right place at the right time, that the real winner in the Big Data world is FP, because it's such an obvious fit for working with data. As an FP language on the JVM, Scala was well placed for this opportunity.

Functional Combinators

SQL Analogs

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

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


Wednesday, March 18, 15

You have functional “combinators”, side-effect free functions that combine/compose together to create complex algorithms with minimal effort.
For simplicity, assume we only keep the two documents where the word appears most frequently, along with the counts in each doc and we’ll assume integer ids for the documents..
We’ll model the same data set in Scala with a Stream, because we’re going to process it in “batch”.

Functional Combinators

```
SELECT * FROM inverted_index
WHERE word LIKE 'sc%';
```

Restrict

```
inverted_index.filter {
  case (word, _) =>
    word startsWith "sc"
}
```


Wednesday, March 18, 15

The equivalent of a SELECT ... WHERE query in both SQL and Scala.

Functional Combinators

```
SELECT word FROM inverted_index;
```

Projection

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


Wednesday, March 18, 15

Projecting just the fields we want.

Functional Combinators

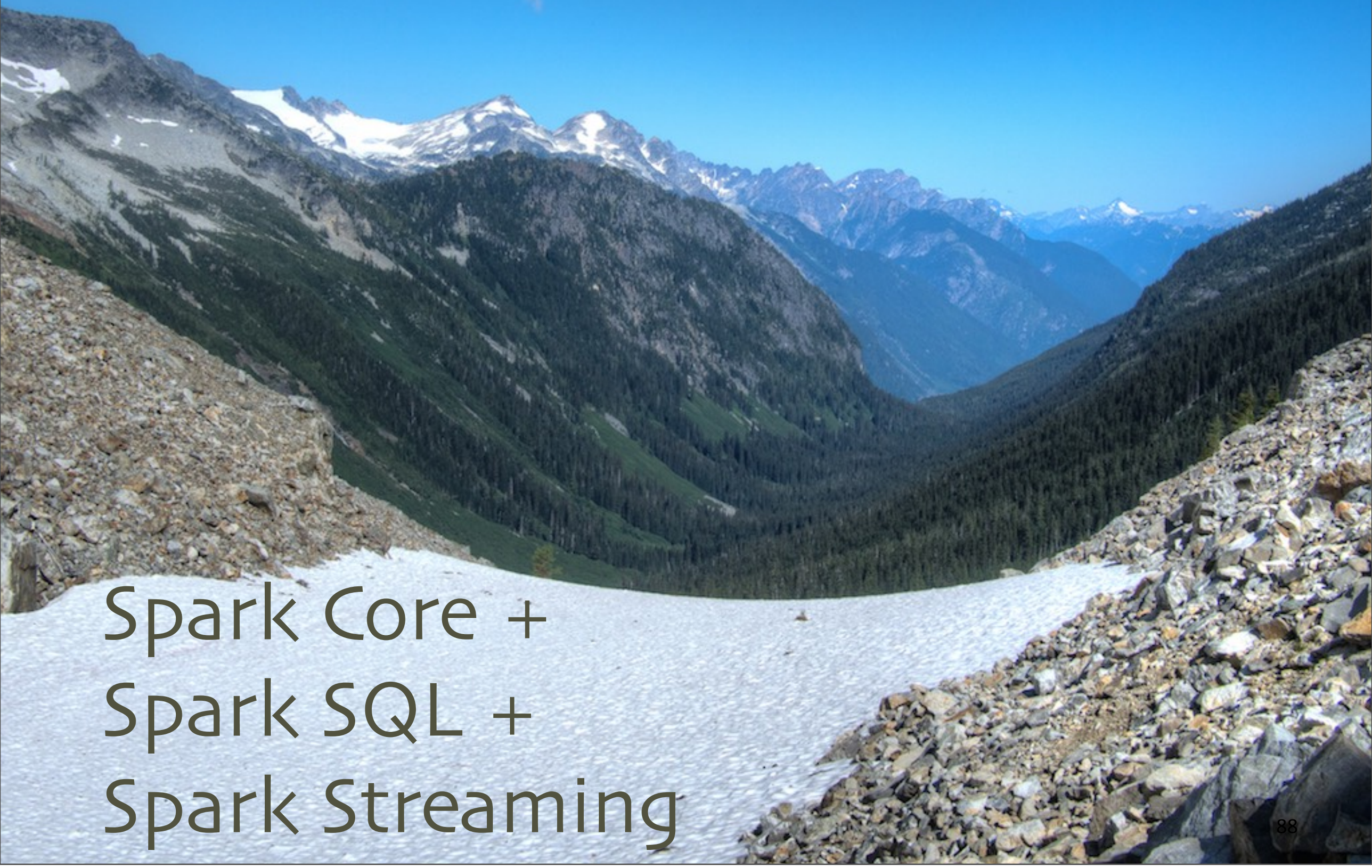
```
SELECT count1, COUNT(*) AS size
FROM inverted_index
GROUP BY count1
ORDER BY size DESC;
```

Group By and Order By

```
inverted_index.groupBy {
  case (_, _, count1, _, _) => count1
} map {
  case (count1, words) =>(count1,words.size)
} sortBy {
 case (count, size) => -size
}
```


Group By: group by the frequency of occurrence for the first document, then order by the group size, descending.

Unification



Wednesday, March 18, 15

Can we unify SQL and Spark? What about streaming?

```
case class Flight(
  number:      Int,
  carrier:     String,
  origin:      String,
  destination: String,
  ...)

object Flight {
  def parse(str: String): Option[Flight]=
    {...}
}

val server = ...   // IP address or name
val port   = ...   // integer
val master = ...   // Spark cluster master

val sc =
```


Wednesday, March 18, 15

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...)

```
case class Flight(
  number:      Int,
  carrier:     String,
  origin:      String,
  destination: String,
  ...)

object Flight {
  def parse(str: String): Option[Flight]=
    {...}
}
```

```
val server = ...   // IP address or name
val port   = ...   // integer
val master = ...   // Spark cluster master
```


Wednesday, March 18, 15

Let’s start by defining a case class to define the schema for our records. Also, we’ll add a parse method to the companion object that will know how to parse a string into a record. It returns Option[Flight] in case it can’t parse a string, in which case we’ll assume it somehow reports the error and returns None.

```
    {...}
}

val server = ...   // IP address or name
val port   = ...   // integer
val master = ...   // Spark cluster master

val sc =
  new SparkContext(master, "Much Wow!")
val strc =
  new StreamingContext(sc, Seconds(60))
val sqlc = new SQLContext(sc)
import sqlc._

val dStream =
  strc.socketTextStream(server, port)
```


Wednesday, March 18, 15

Normally, you would probably use a command-line argument to “main” to specify the server and port for a socket that will be the source of data. You would also use a command-line argument for the Spark master (e.g., “local”, “yarn-cluster”, “mesos”, etc.). For simplicity, we’ll assume they are hardcoded here.

```
    {...}
}

val server = ...   // IP address or name
val port   = ...   // integer
val master = ...   // Spark cluster master

val sc =
  new SparkContext(master, "Much Wow!")
val strc =
  new StreamingContext(sc, Seconds(60))
val sqlc = new SQLContext(sc)
import sqlc._

val dStream =
  strc.socketTextStream(server, port)
```


Wednesday, March 18, 15

Next create a SparkContext, specifying the master and some identifying string ;)
New for this example, create “wrapper” contexts for streaming and SQL, which bring in their additional capabilities. Then import the members of the created SQLContext (a Spark idiom...). Note the second argument to StreamingContext. It specifies the fixed-width time interval in which “batches” of events will be captured. So, we’ll capture and process the data in 60-second batches.

```
import sqlc._

val dStream =
  strc.socketTextStream(server, port)

val flights = for {
  line <- dStream
  flight <- Flight.parse(line)
} yield flight

flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
      destination, COUNT(*)
    FROM flights
    GROUP BY carrier, origin, destination
```


Using the server and port, construct a DStream (“discretized stream”) that will listen set up a socket connection at the specified server and port. We expect lines of text, separated by ‘\n’. As for the previous Spark Core example, we are constructing a lazy pipeline that won’t start processing data until we tell it to start.

Next, use a for comprehension to ingest each line from the stream (this will be invoked for each batch - 60 seconds of data - after the batch has been captured. We call Flight.parse to convert the string into a Flight instance. Hence, “flights” will be a DStream[Flight].

```
flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
      destination, COUNT(*)
    FROM flights
    GROUP BY carrier, origin, destination
    ORDER BY c4 DESC
    LIMIT 20""").foreach(println)
}
```

```
strc.start()
strc.awaitTermination()
strc.stop()
```


Wednesday, March 18, 15

Each DStream batch is stored in an RDD, so we can reuse everything we already have in Spark Core. Here we setup a loop that will be called after each batch is captured. The corresponding RDD and timestamp will be passed to the block, which uses Spark SQL to register the RDD as a “temporary table” named “flights”, then runs a SQL query to group all flight records in the batch by the carrier, origin, and destination. They it counts the size of each group and prints the top 20. Alternatively, we could send this data to a dashboard, database, filesystem, etc.

```
flights.foreachRDD { (rdd, time) =>
  rdd.registerTempTable("flights")
  sql(s"""
    SELECT $time, carrier, origin,
      destination, COUNT(*)
    FROM flights
    GROUP BY carrier, origin, destination
    ORDER BY c4 DESC
    LIMIT 20""").foreach(println)
}

strc.start()
strc.awaitTermination()
strc.stop()
```


Wednesday, March 18, 15

Finally, start the data pipeline using StreamingContext.start() and wait (forever?) for it to terminate. Then stop everything.

There are plenty of details I left out, but they are mostly incidental. The core program is still very concise.

Conclusions

We won!

Wednesday, March 18, 15

Monday night’s sunset in San Francisco, from the conference pier.