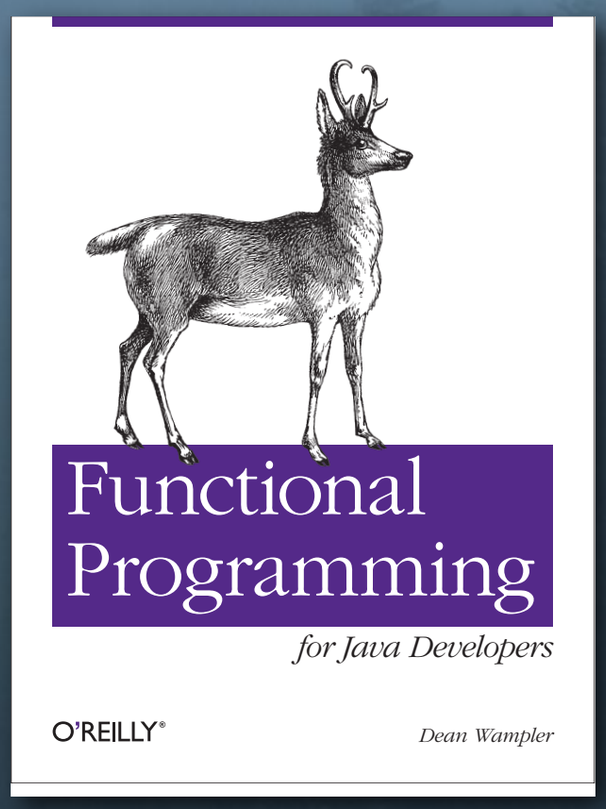
BIG DATA STATE OF THE ART: SPARK AND THE SQL RESURGENCE

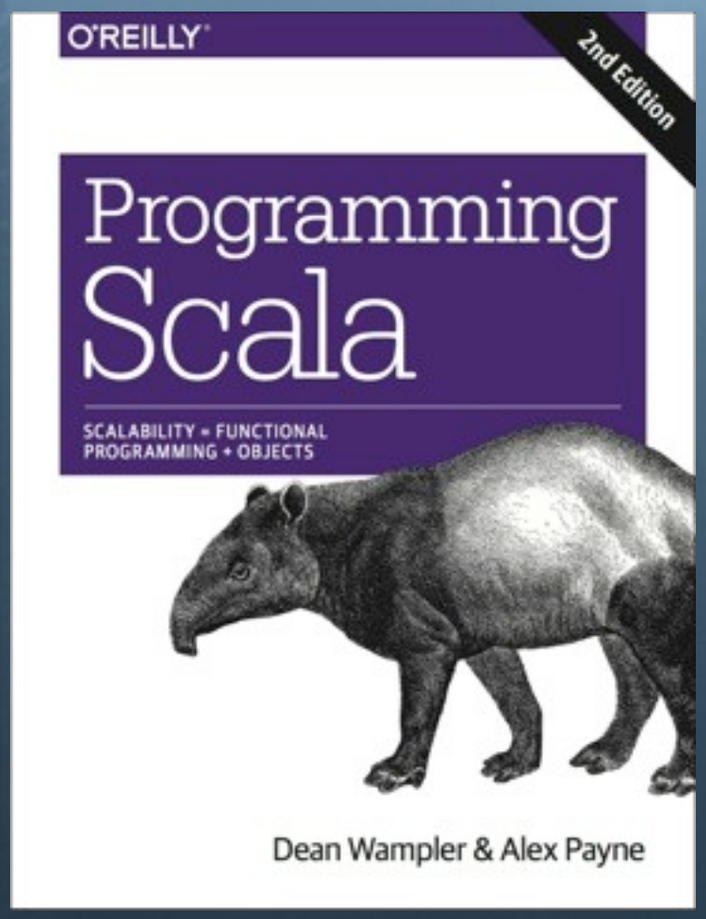
Dean Wampler, Ph.D.
Typesafe

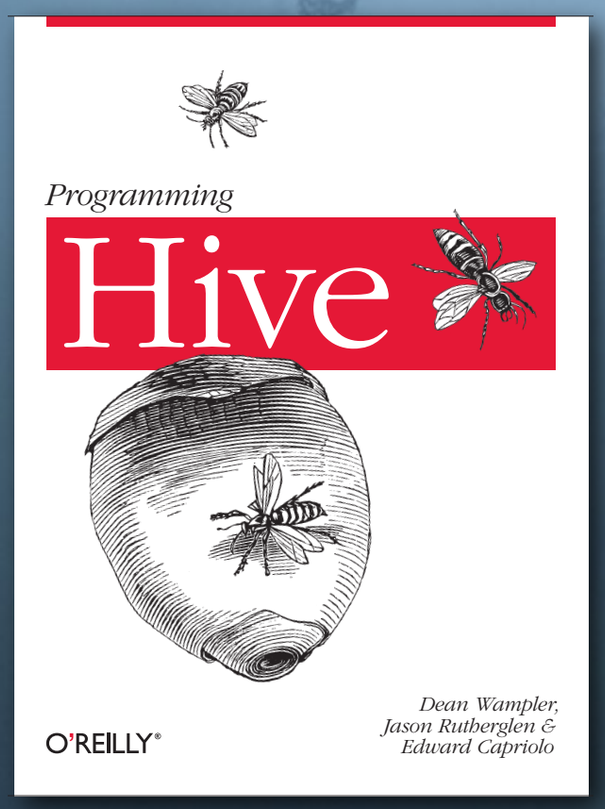
INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

Dean Wampler

Typesafe







dean.wampler@typesafe.com

polyglotprogramming.com/talks

@deanwampler

Tuesday, September 30, 14

About me. You can find this presentation and others on Big Data and Scala at polyglotprogramming.com.

It's 2014

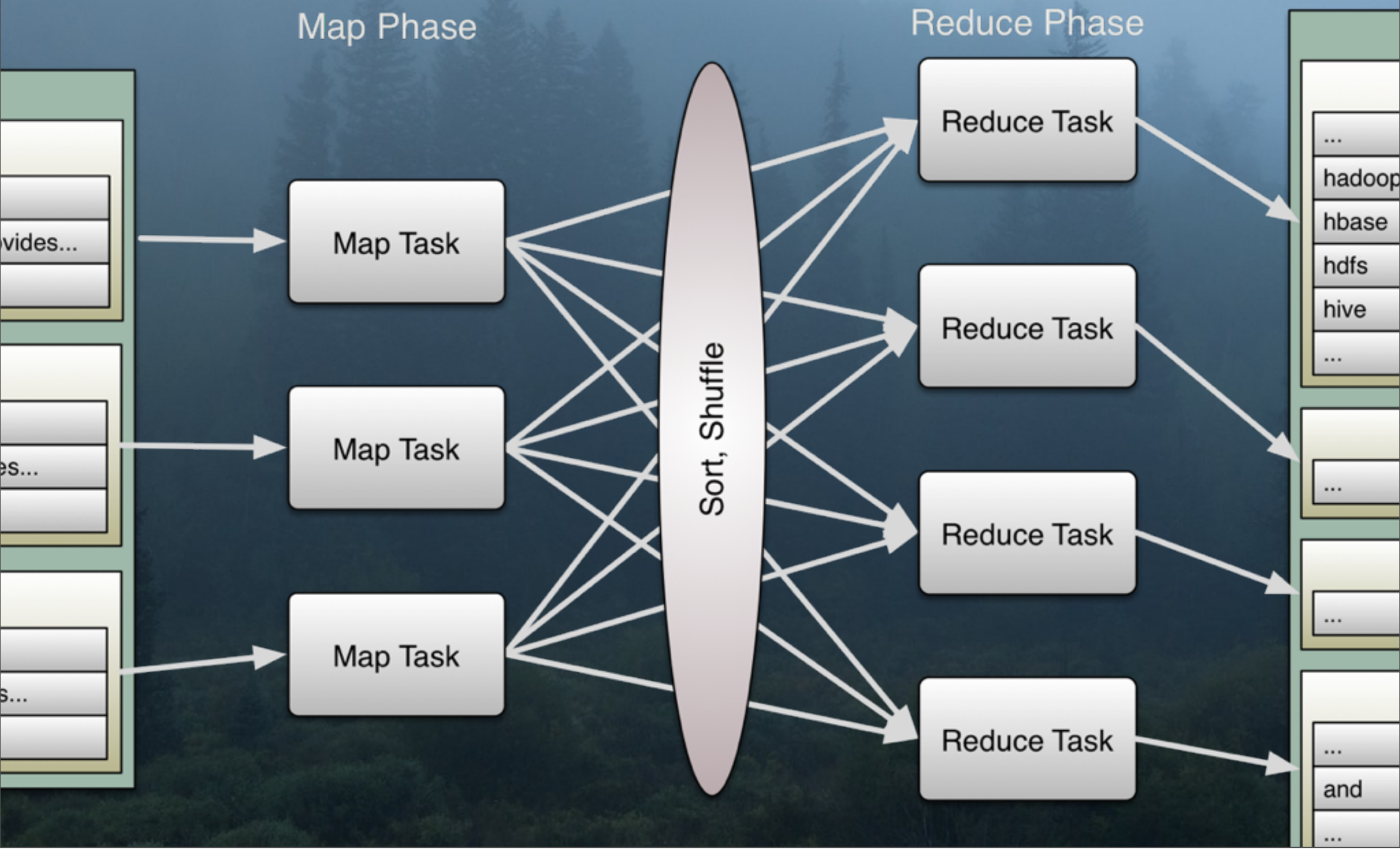

Hadoop has been
very successful.


But it's not perfect

Tuesday, September 30, 14

The limitations of MapReduce have become increasingly significant...

1 Map step + 1 Reduce step



You get one map step and one reduce step. You can sequence jobs together when necessary.

Problems

Limited programming model

Tuesday, September 30, 14

MapReduce is a restrictive model. Writing jobs requires arcane, specialized skills that few master.
It's not easy mapping arbitrary algorithms to this model. For example, algorithms that are naturally iterative are especially hard, because MR doesn't support iteration efficiently. For a good overview, see http://lintool.github.io/MapReduceAlgorithms/.

The limited model doesn't just impede developer productivity, it makes it much harder to build tools on top of the model, as we'll discuss.

Problems

The Hadoop API is horrible

Tuesday, September 30, 14

And Hadoop's Java API only makes the problem harder, because it's very low level and offers limited or no support for many common idioms.

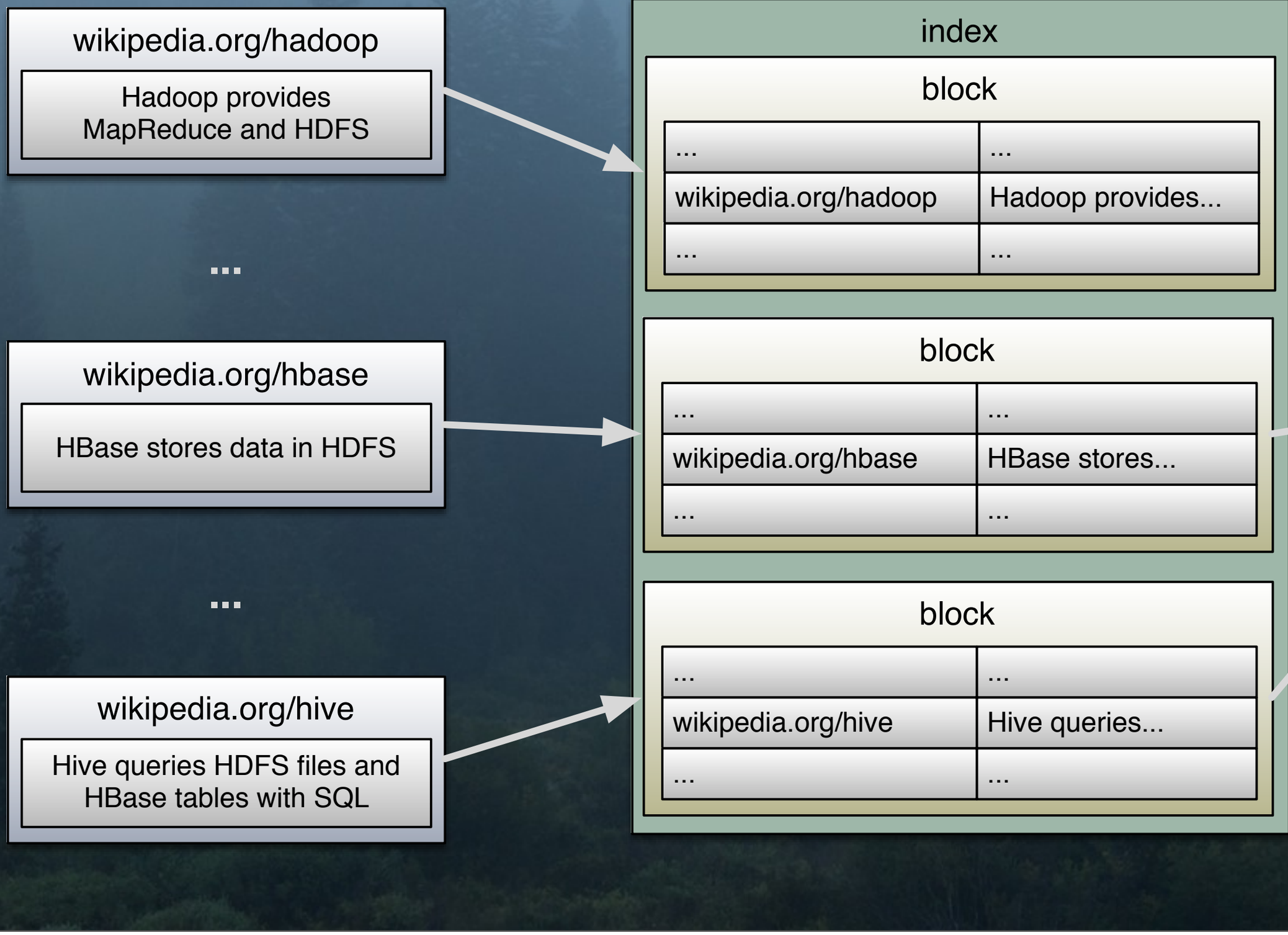
Example

Tuesday, September 30, 14

See compare and contrast MR with Spark, let's use this classic algorithm.

Inverted Index

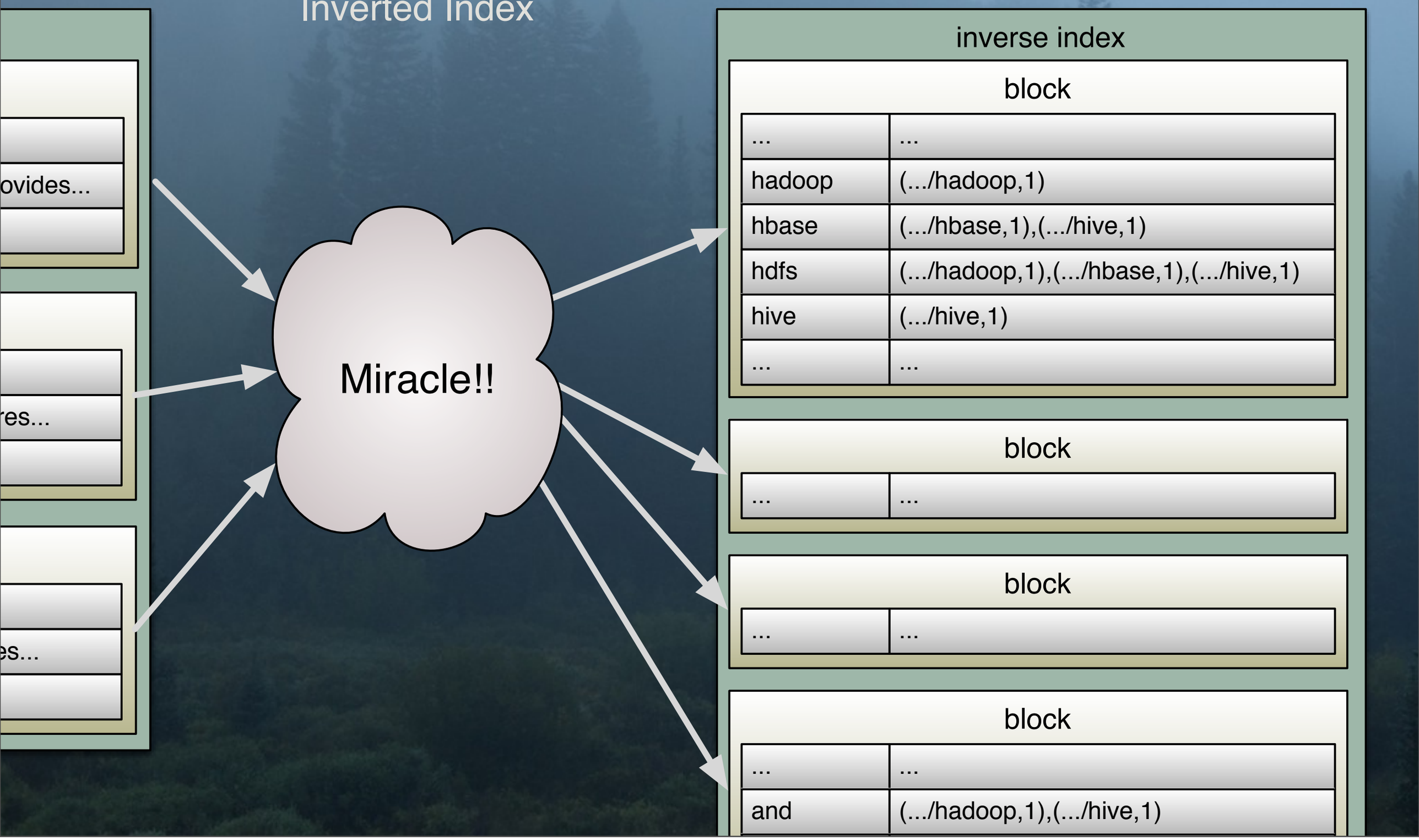
Web Crawl



First we crawl the web to build a data set of document names/ids and their contents. Then we “invert” it; we tokenize the contents into words and build a new index from each word to the list of documents that contain the word and the count in each document. This is a basic data set for search engines.

Inverted Index

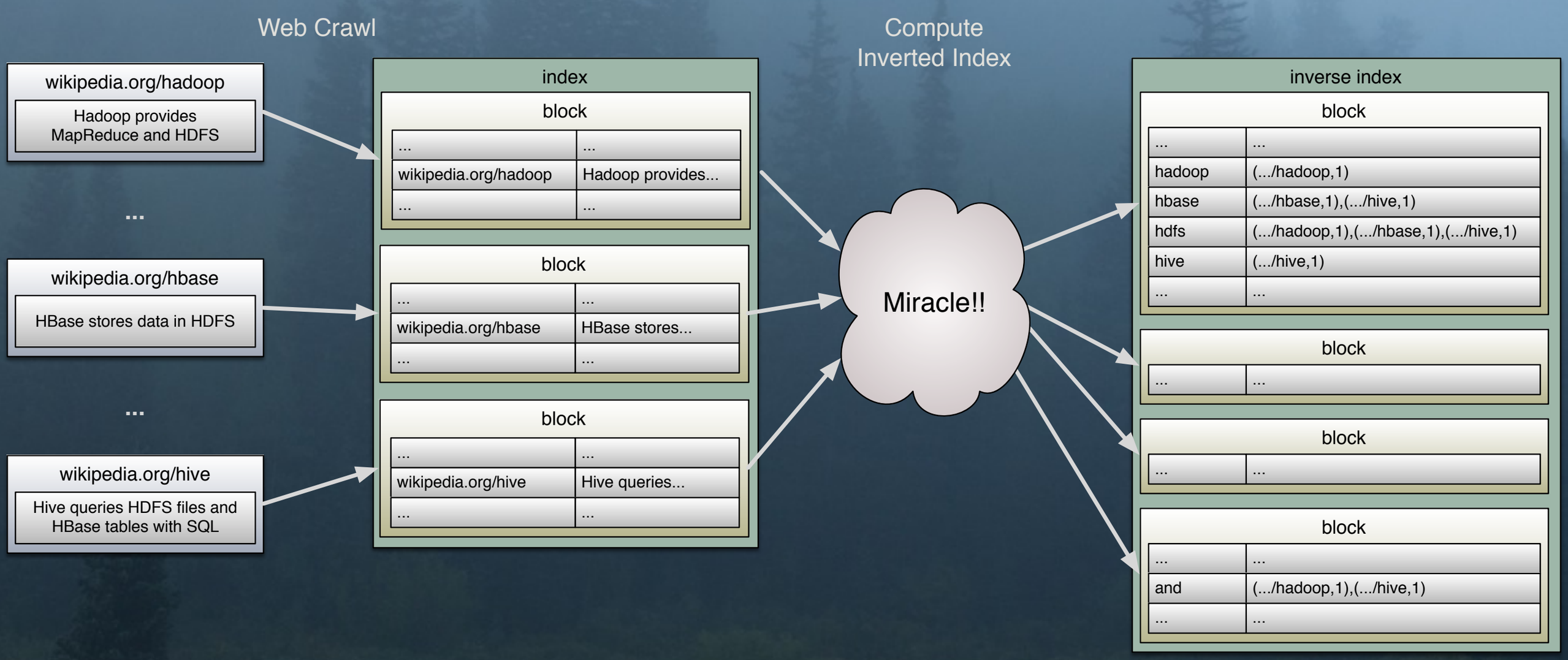
Compute
Inverted Index



Tuesday, September 30, 14

First we crawl the web to build a data set of document names/ids and their contents. Then we “invert” it; we tokenize the contents into words and build a new index from each word to the list of documents that contain the word and the count in each document. This is a basic data set for search engines.

Inverted Index



Altogether

Tuesday, September 30, 14

We'll implement the "miracle".

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {

 public static void main(String[] args) {
    JobClient client = new JobClient();
    JobConf conf =
     new JobConf(LineIndexer.class);

    conf.setJobName("LineIndexer");
    conf.setOutputKeyClass(Text.class);
```


I'm not going to explain many of 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...

```
    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> {
```


Tuesday, September 30, 14

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);
  }
 }
}
```


Tuesday, September 30, 14

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.

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


EOF

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {

 public static void main(String[] args) {
  JobClient client = new JobClient();
  JobConf conf =
   new JobConf(LineIndexer.class);

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

  client.setConf(conf);

  try {
   JobClient.runJob(conf);
  } catch (Exception e) {
   e.printStackTrace();
  }
 }

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

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

   FileSplit fileSplit =
    (FileSplit)reporter.getInputSplit();
   String fileName =
    fileSplit.getPath().getName();
   location.set(fileName);

   String line = val.toString();
   StringTokenizer itr = new
    StringTokenizer(line.toLowerCase());
   while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    output.collect(word, location);
   }
  }
 }

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

Altogether

Tuesday, September 30, 14

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.

Problems

MR only supports "batch-mode"; no streaming

Tuesday, September 30, 14

You can't do event-stream processing (a.k.a. "real-time") with MapReduce, only batch mode processing.

Problems

Wasteful disk IO between jobs

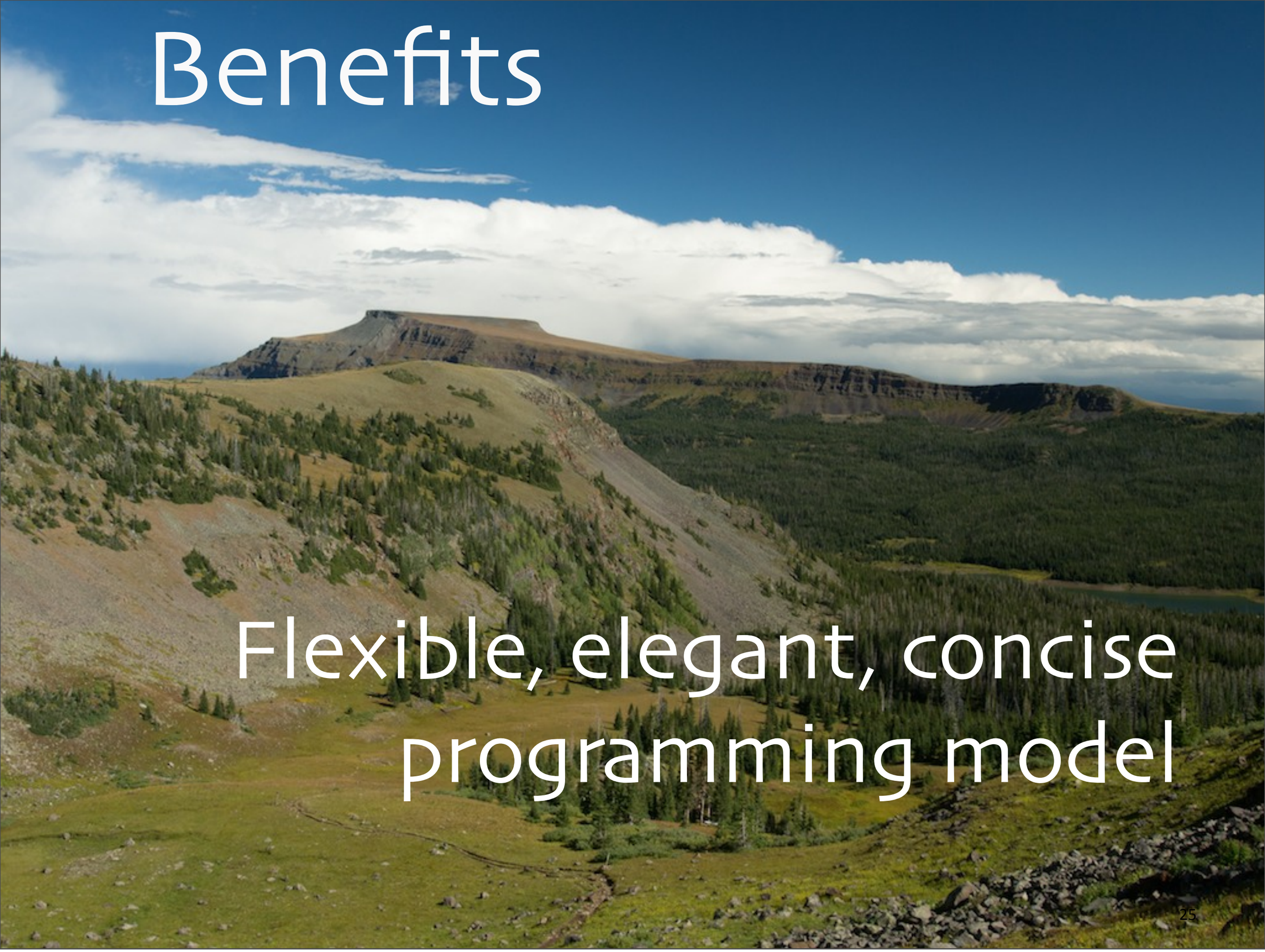
Tuesday, September 30, 14

A complex sequence of jobs results in fully flushing data to disk at the end of each job in the sequence, even though it will be immediately reread by the next job!

Tuesday, September 30, 14

http://spark.apache.org

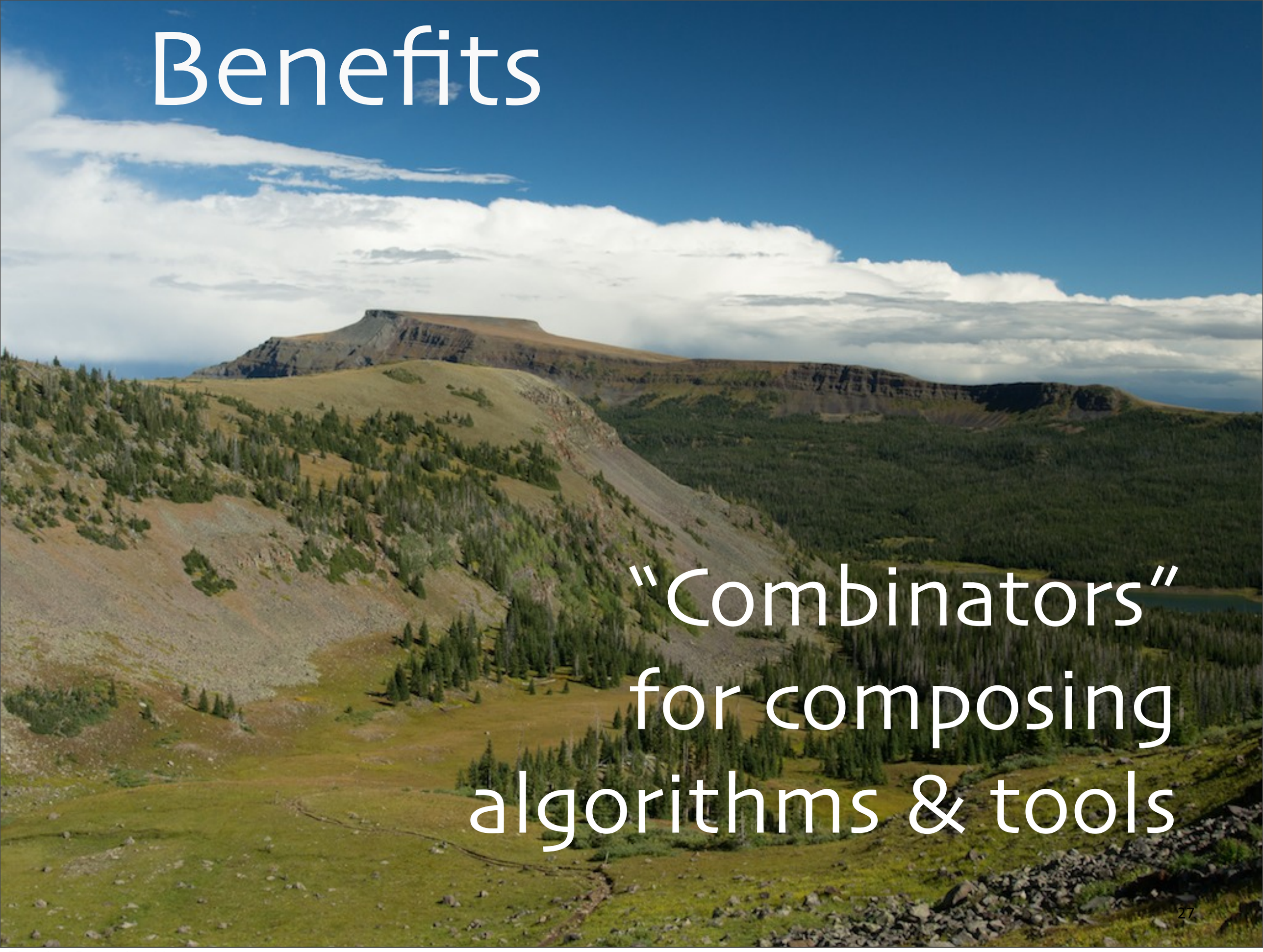
Benefits



Flexible, elegant, concise programming model

Benefits

Benefits



Tuesday, September 30, 14

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

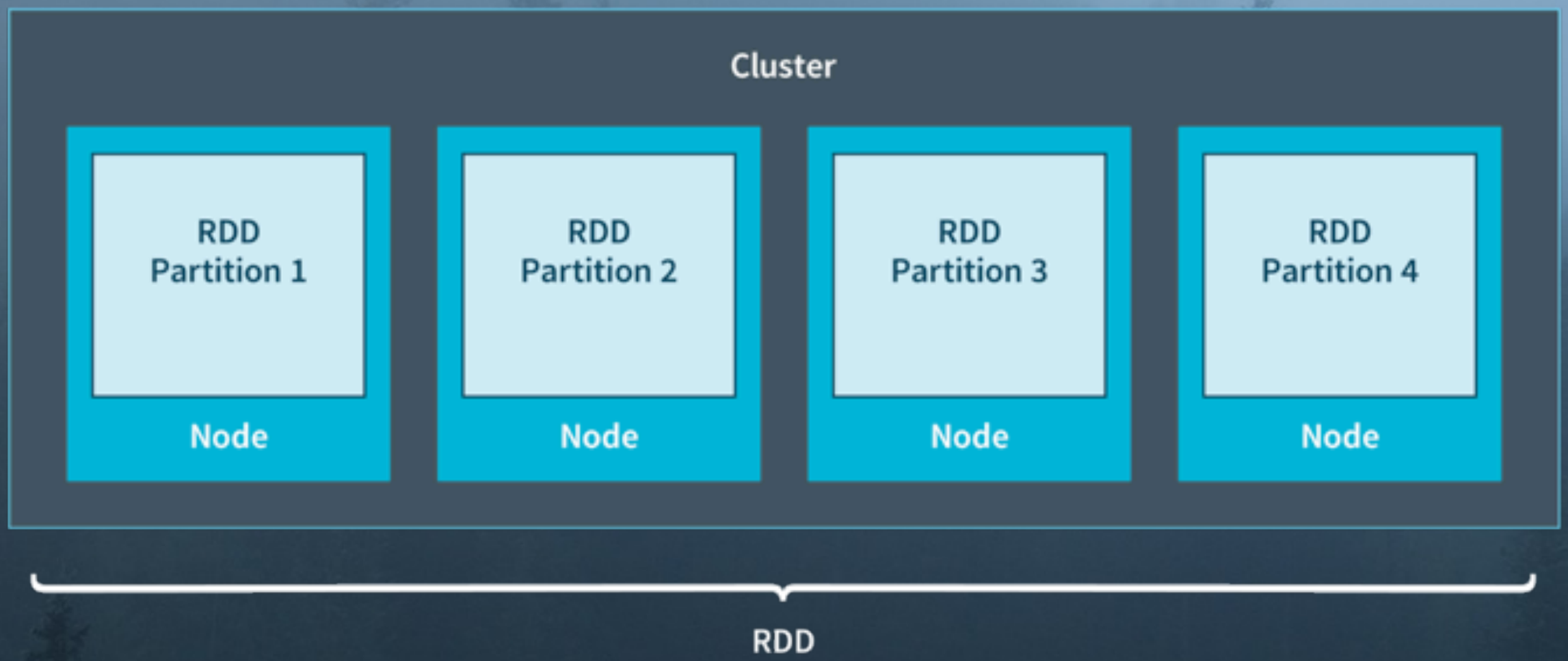
Tuesday, September 30, 14

Not restricted to Hadoop, when you don’t need it, e.g., because you want to “enhance” existing applications with data analytics, running in the same infrastructure.

Resilient Distributed Datasets

Tuesday, September 30, 14

Data is shared over the cluster in RDDs. This is the core abstraction everyone else builds on.


Cluster
RDD Partition 1
Node
RDD Partition 2
Node
RDD Partition 3
Node
RDD Partition 4
Node
RDD

Example

Tuesday, September 30, 14

Let’s see our example rewritten in Spark.

```
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 {
```


Tuesday, September 30, 14

It starts with imports, then declares a singleton object (a first-class concept in Scala), with a main routine (as in Java).
The methods are colored yellow again. Note this time how dense with meaning they are this time.

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

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

sc.textFile returns an RDD with a string for each line of input text. So, the first thing we do is map over these strings to extract the original document id (i.e., file name), followed by the text in the document, all on one line. We assume tab is the separator. “(array(0), array(1))” returns a two-element “tuple”. Think of the output RDD has having a schema “String fileName, String text”.

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


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

Then we map over these pairs and add a single count of 1.

reduceByKey does an implicit “group by” to bring together all occurrences of the same (word, path) and then sums up their counts.
Note the input to the next map is now ((word, path), n), where n is now >= 1. We transform these tuples into the form we actually want, (word, (path, n)).

```
    }
    .groupBy {
      case (w, (p, n)) => w
    }
    .map {
      case (w, seq) =>
        val seq2 = seq map {
          case (_, (p, n)) => (p, n)
        }
        (w, seq2.mkString(", "))
    }
    .saveAsTextFile(argz.outpath)

    sc.stop()
  }
}
```


Now we do an explicit group by to bring all the same words together. The output will be (word, (word, (path1, n1)), (word, (path2, n2)), ...).
The last map removes the redundant “word” values in the sequences of the previous output. It outputs the sequence as a final string of comma-separated (path,n) pairs.
We finish by saving the output as text file(s) and stopping the workflow.

```
  }
  .map {
    case (w, p) => ((w, p), 1)
  }
  .reduceByKey {
    case (n1, n2) => n1 + n2
  }
  .map {
    case ((w, p), n) => (w, (p, n))
  }
  .groupBy {
    case (w, (p, n)) => w
  }
  .map {
    case (w, seq) =>
      val seq2 = seq map {
        case (_, (p, n)) => (p, n)
```

Chain combinators together

The yellow functions are our composable - and elegant - combinators.

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

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

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

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

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

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

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

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

  sc.textFile("data/crawl")
  .map { line =>
    val array = line.split("\t", 2)
    (array(0), array(1))
  }
  .flatMap {
    case (path, text) =>
     text.split("""\W+""") map {
      word => (word, path)
     }
  }
  .map {
    case (w, p) => ((w, p), 1)
  }
  .reduceByKey {
    case (n1, n2) => n1 + n2
  }
  .map {
    case ((w, p), n) => (w, (p, n))
  }
  .groupBy {
    case (w, (p, n)) => w
  }
  .map {
    case (w, seq) =>
      val seq2 = seq map {
        case (_, (p, n)) => (p, n)
      }
      (w, seq2.mkString(", "))
   }
   .saveAsTextFile(argz.outpath)

  sc.stop()
 }
}
```

Altogether

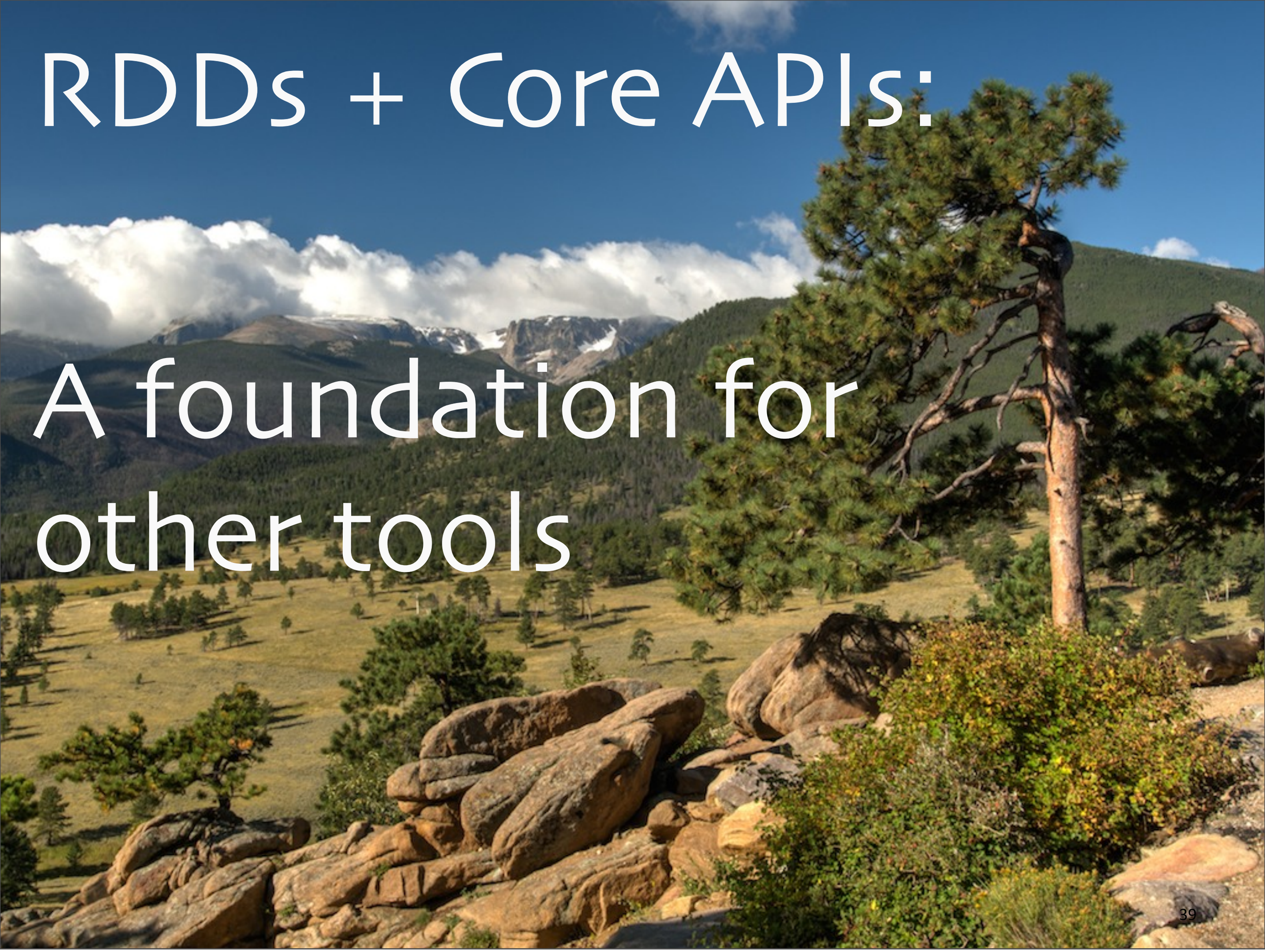
That version took me ~30 mins. to write

Tuesday, September 30, 14

When you have a concise, flexible API, you can turn a “software development project” into a script! It transforms your productivity.

RDDs + Core APIs:

A foundation for other tools



Tuesday, September 30, 14

The good API also provides an excellent foundation for other tools to build on.

Extensions

MLlib

GraphX

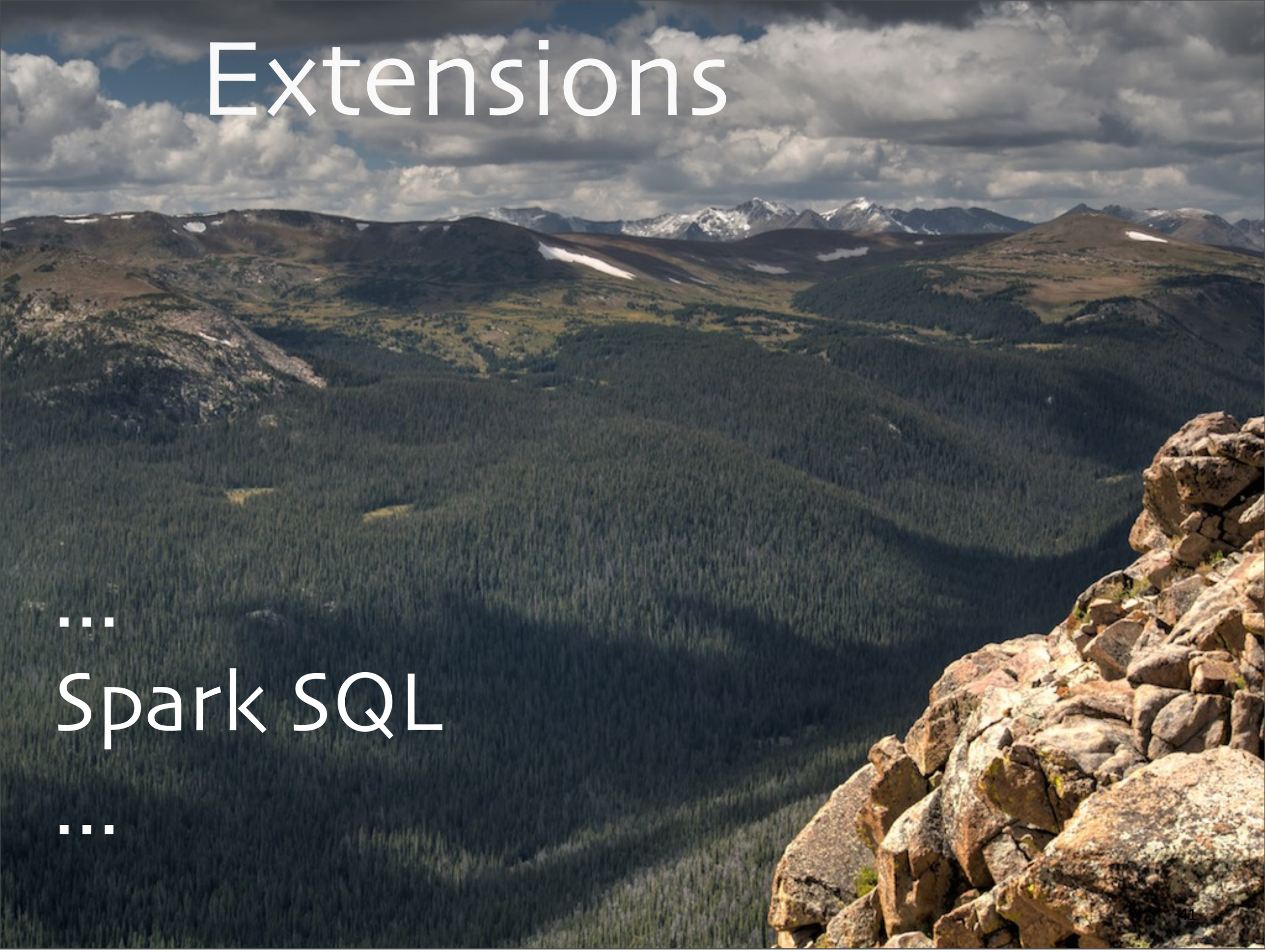
Tachyon

...

Tuesday, September 30, 14

MLlib - a growing library of machine learning algorithms.
GraphX - for representing data as a graph and applying graph algorithms to it.
Tachyon - an experiment to generalize Spark's caching mechanism into a standalone service, so data is shareable between apps and more durable. I believe it will be transformative!

Extensions



Tuesday, September 30, 14

Let’s look at the SQL abstractions layered on top.

Tuesday, September 30, 14

Best of both worlds: SQL for concision, the RDD API for Turing-complete, general-purpose computing. Also adds elegant handling of the “schema” for data.

Hive Interop

Tuesday, September 30, 14

Let’s us query Hadoop Hive “tables”. We can create or delete them, too.

Tuesday, September 30, 14

New feature. Can read JSON records and infer their schema. Can write RDD records as JSON.

Example

Use the Crawl data for Inverted Index

It's a bit tricky to use the inverted index data, because of the variable list of (docid, N) values, so we'll use the crawl data, which is easier for our purposes.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.sql.{
  SQLContext, SchemaRDD}
import org.apache.spark.rdd.RDD

case class CrawlRecord(
  docid: String, contents: String)

def makeCrawlRecord(line: String) = {...}

def dosql(qry: String, n: Int = 100) =
  sql(qry).collect.take(n) foreach println

val crawlData = "/path/to/directory"

val sc = new SparkContext("...", "Crawl")
```


Starts out like a typical Spark program...
Defines a “case class” (think normal Java class where the args are automatically turned into fields) to represent each record.
Defines a method to parse each input line and output a CrawlRecord (details omitted)
Defines a helper method to take a SQL query as a string, run it using the “sql(...)” method provided by SparkSQL, grab the first n elements and print them, one per line.
Finally, defines the path to the input for the crawl data.

```
val sc = new SparkContext("...", "Crawl")

val crawl = for {
  line <- sc.textFile(crawlData)
  cr <- makeCrawlRecord(line)
} yield cr

crawl.registerAsTable("crawl")
crawl.printSchema

dosql("""
  SELECT docid, contents FROM crawl
  LIMIT 10""")

val crawlPerWord = crawl flatMap {
  case CrawlRecord(docid, contents) =>
    contents.trim.split("""[^\w']""") map
```


The for loop takes each line of input and parses it into a CrawlRecord. So, “crawl” has the type RDD[CrawlRecord]. Then we register it as a table and print its schema.
Now run a query!

```
val crawlPerWord = crawl flatMap {
  case CrawlRecord(docid, contents) =>
    contents.trim.split("""[^\w']""") map
(word => CrawlRecord(docid, word))
}

crawlPerWord.registerAsTable("crawl2")
crawlPerWord.cache

dosql("SELECT * FROM crawl2 LIMIT 10")
dosql("""
  SELECT DISTINCT * FROM crawl2
  WHERE contents = 'management'""")
dosql("""
  SELECT contents, COUNT(*) AS c
  FROM crawl2
  GROUP BY contents
```


Create a new, similar RDD where the records are the docids with each word in the document, not one record for each docid with the entire contents.

Register this RDD as a table and this time cache it in memory, since we'll write several queries against it.

```
crawlPerWord.registerAsTable("crawl2")
crawlPerWord.cache

dosql("SELECT * FROM crawl2 LIMIT 10")
dosql("""
  SELECT DISTINCT * FROM crawl2
  WHERE contents = 'management'""")
dosql("""
  SELECT contents, COUNT(*) AS c
  FROM crawl2
  GROUP BY contents
  ORDER BY c DESC LIMIT 100""")
```


Run several queries.

Extensions

... and Streaming.

A clever extension to the existing batch-oriented RDD model; use smaller batches! So, it's not a replacement for true event-processing systems, like Storm, message queues.

Each slice is an RDD. Plus window functions

We get all the familiar RDD functions, “for free”, plus functions for working with windows of batches.


DStream (discretized stream)
...
Event
Event
Event
Time 1 RDD
Event
Event
Event
Event
Event
Time 2 RDD
Event
Event
Event
Event
Time 3 RDD
Event
Event
Event
Time 4 RDD
...
...
Window of 3 RDD Batches #1
Window of 3 RDD Batches #2

Example

Use “live” Crawl data for Inverted Index

Continue using the crawl data, but “pretend” we’re reading it live from a socket.

```
// ... imports, etc.
val sc = new SparkContext(...)
val ssc = new StreamingContext(
  sc, Seconds(60))
ssc2.addStreamingListener(
  /*... listener for end of data ...*/)
val sqlc = new SQLContext(sc)
import sqlc._

val inputDStream =
  sc.socketTextStream(server,
port).flatMap(_.split("\n"))

val crawlWords = for {
  line <- inputDStream
  cr1  <- makeCrawlRecord(line)
```

"Batch" size

Tuesday, September 30, 14

We won't show everything now, just the interesting bits...
We create a SparkContext, then wrap it with a new StreamingContext object, where we'll grab the records in 60-second increments, AND a SQLContext as before (optional).
We also add listener for stream events, such as end of input (details omitted).
Finally, create a DStream that will ingest events from a socket of plain text data. It uses flatMap to split the input stream into "records" on newlines.

```
port).flatMap(_.split("\n"))

val crawlWords = for {
  line <- inputDStream
  cr1  <- makeCrawlRecord(line)
  word <- cr1.contents.trim.split(
    """[^\w']""")
} yield (CrawlRecord(cr1.docid, word))

crawlWords.window(
  Seconds(300), Seconds(60))
.foreachRDD { rdd =>
  rdd.registerAsTable("crawlWords")
  dosql("""
    SELECT contents, COUNT(*) AS c
    FROM crawlWords
    GROUP BY contents
```


As before, parse each line into a CrawlRecord of docids and contents, then immediately convert to docids and individual words. (We used separate RDDs for these two steps previously.)

```
} yield (CrawlRecord(cr1.docid, word))

crawlWords.window(
  Seconds(300), Seconds(60))
  .foreachRDD { rdd =>
    rdd.registerAsTable("crawlWords")
    dosql("""
      SELECT contents, COUNT(*) AS c
      FROM crawlWords
      GROUP BY contents
      ORDER BY c DESC LIMIT 100""")
  }

ssc2.start()
ssc2.awaitTermination()
```

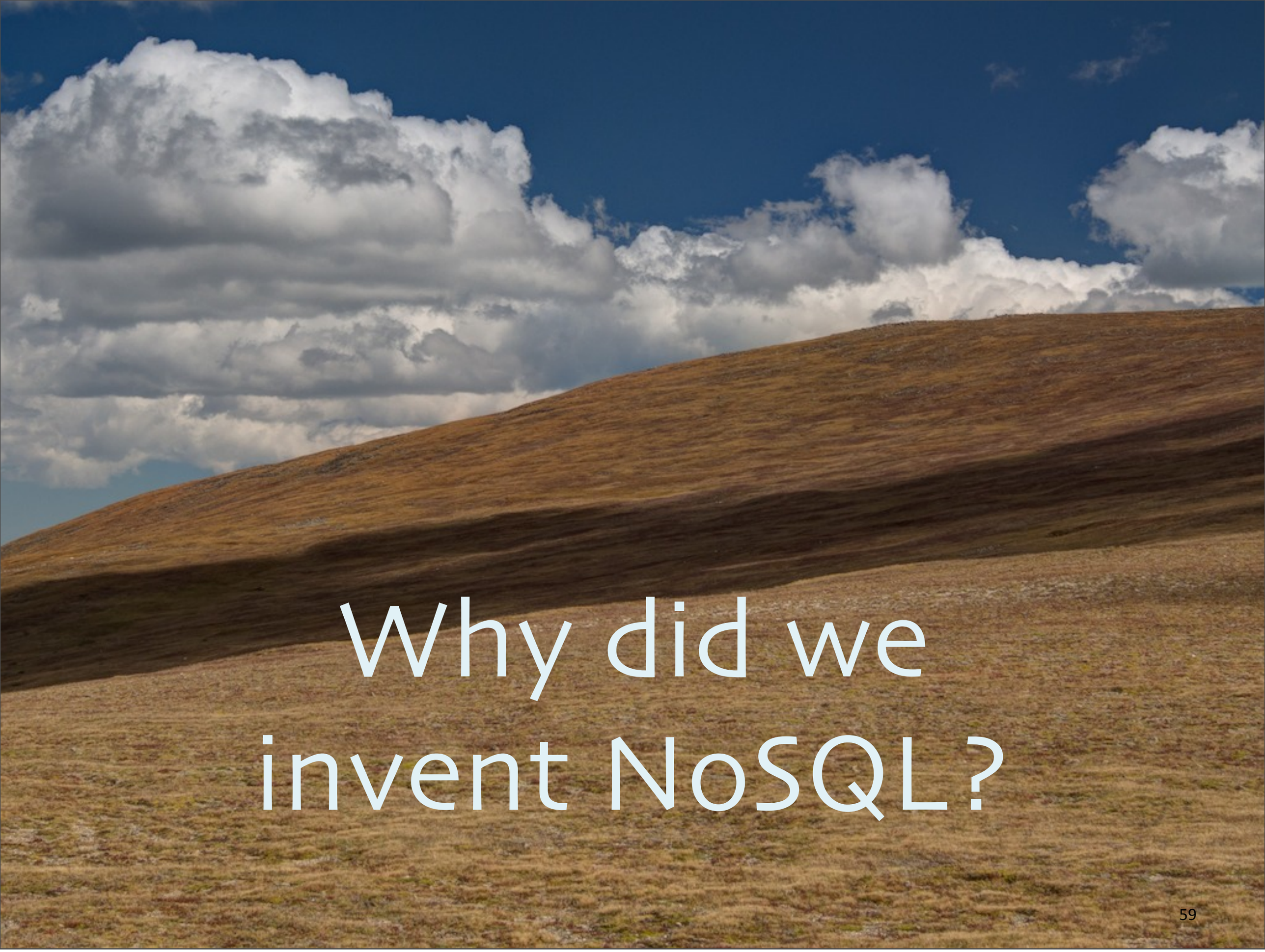
Window size, "skip size.

Tuesday, September 30, 14

Now process event batches in windows of 300 seconds (5 batches of 60 seconds) and each time we move the window, move by 60 seconds (one batch). Then, for each RDD, register as a table and run a query.
Finally, start processing and wait for it to finish (if ever!).

Tuesday, September 30, 14

So, SQL is very useful for “structured” data in Hadoop. In fact, SQL has experienced a renaissance in Big Data



Tuesday, September 30, 14

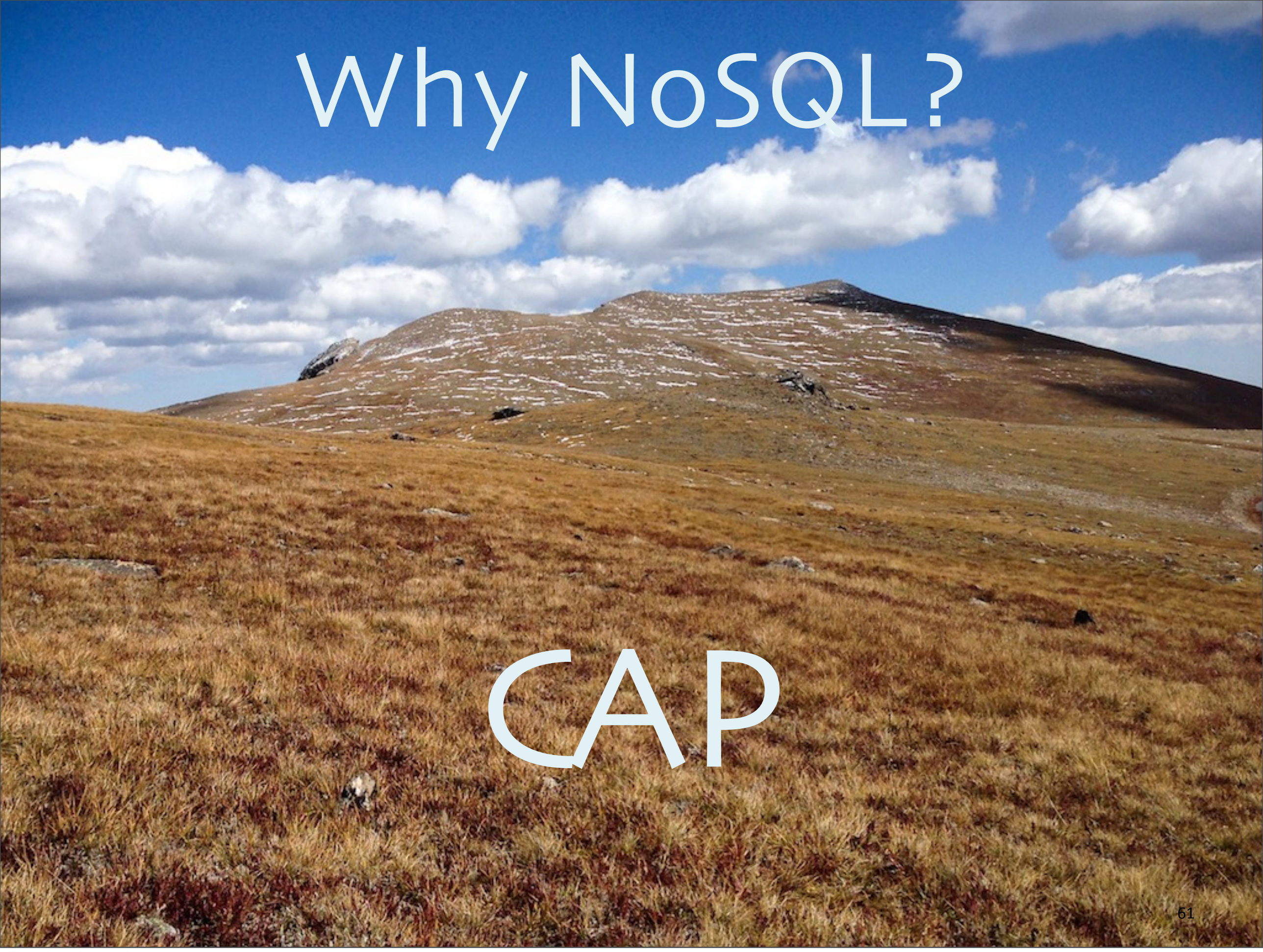
First, why did NoSQL emerge in the first place?

Why NoSQL?

Tuesday, September 30, 14

1. We needed to manage unprecedented data set sizes, economically. Existing Relational tools couldn't handle the size, especially at low cost.

Why NoSQL?



Tuesday, September 30, 14

Sometimes remaining available and accepting eventual consistency is the tradeoff we want when partitions occur. Relational is CP, if a partition happens we prefer consistency, so the DB won't be available until the partition is resolved. But many apps can accept lack of consistency if they can still remain available.

Why NoSQL?

Not all data is relational

Tuesday, September 30, 14

Key-value stores, hierarchical data, e.g.,JSON/XML docs, etc. are alternative forms that work better for many scenarios.

Tuesday, September 30, 14

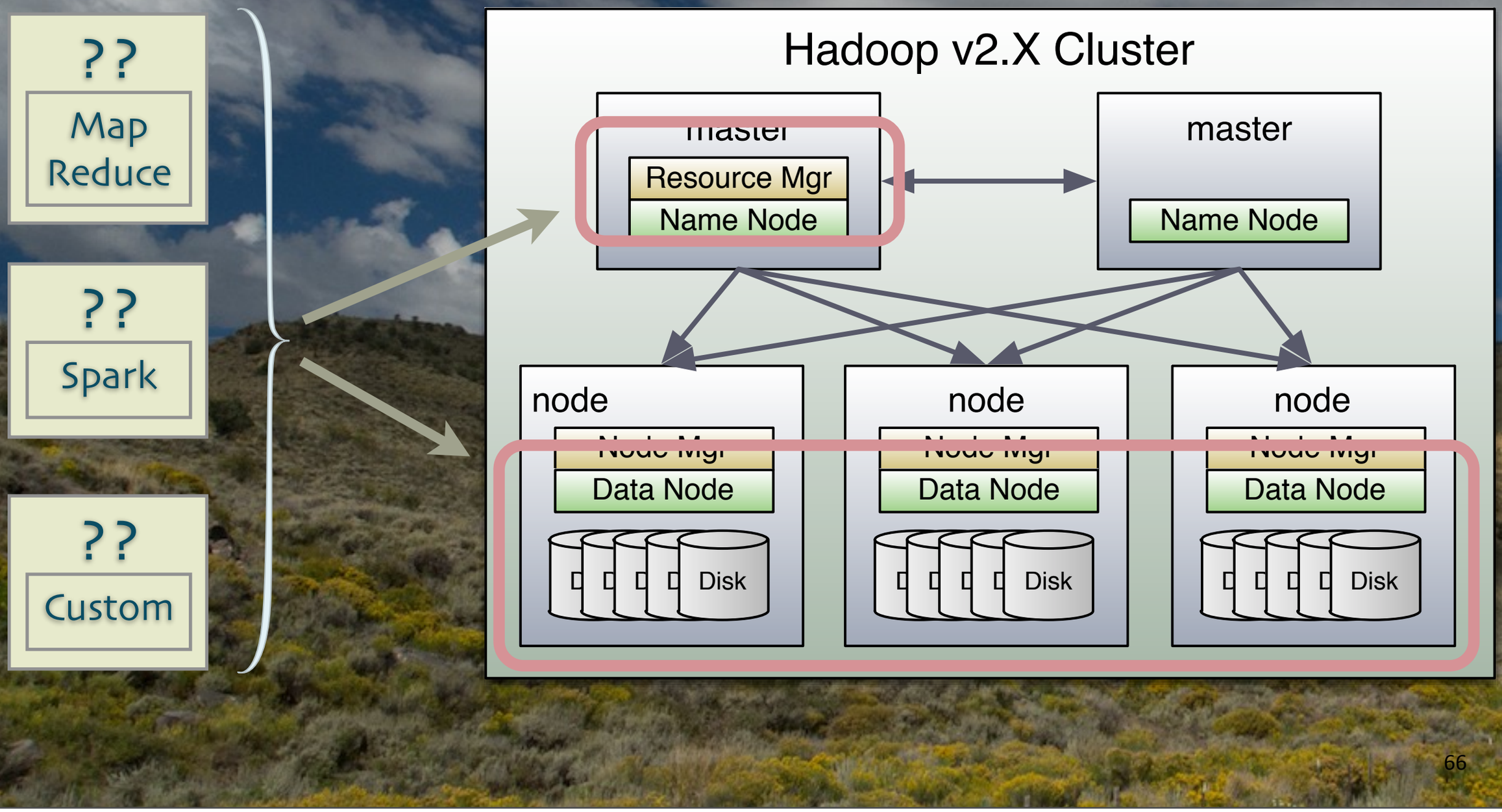
Even so-called “unstructured data”, like tweets and other text, are quickly processed to extract structure of some kind.


Two New Approaches
for SQL

1. Query Engine + HDFS

First idea, put SQL-based query abstractions on top of simple storage, like flat files in HDFS, MapRFS, S3, etc. The query abstractions can be a “DSL” implemented in a generic framework like MapReduce or Spark, or with a custom query engine optimized for the job.

1. Query Engine + HDFS



Tuesday, September 30, 14

You could write engines in MR, Spark, or something custom, which may offer less flexibility, but better performance.

1. Query Engine + HDFS

Tuesday, September 30, 14

Hive, developed at Facebook, pioneered SQL on Hadoop. It has recently been ported to a new, higher-performance engine called Tez. Tez is a competitor to Spark, but isn't gaining the same traction.

The Spark team ported Hive to Spark and achieved 30x+ performance improvements. Shark is now deprecated; it's being replaced with a better engineered query engine called Catalyst, inside SparkSQL.

Impala was the first custom query engine, inspired by Google's Dremel. It holds the current performance records for SQL on Hadoop. Presto is a Facebook project. Drill is another Apache project. It's more portable across Hadoop platforms than Impala, but not as fast.

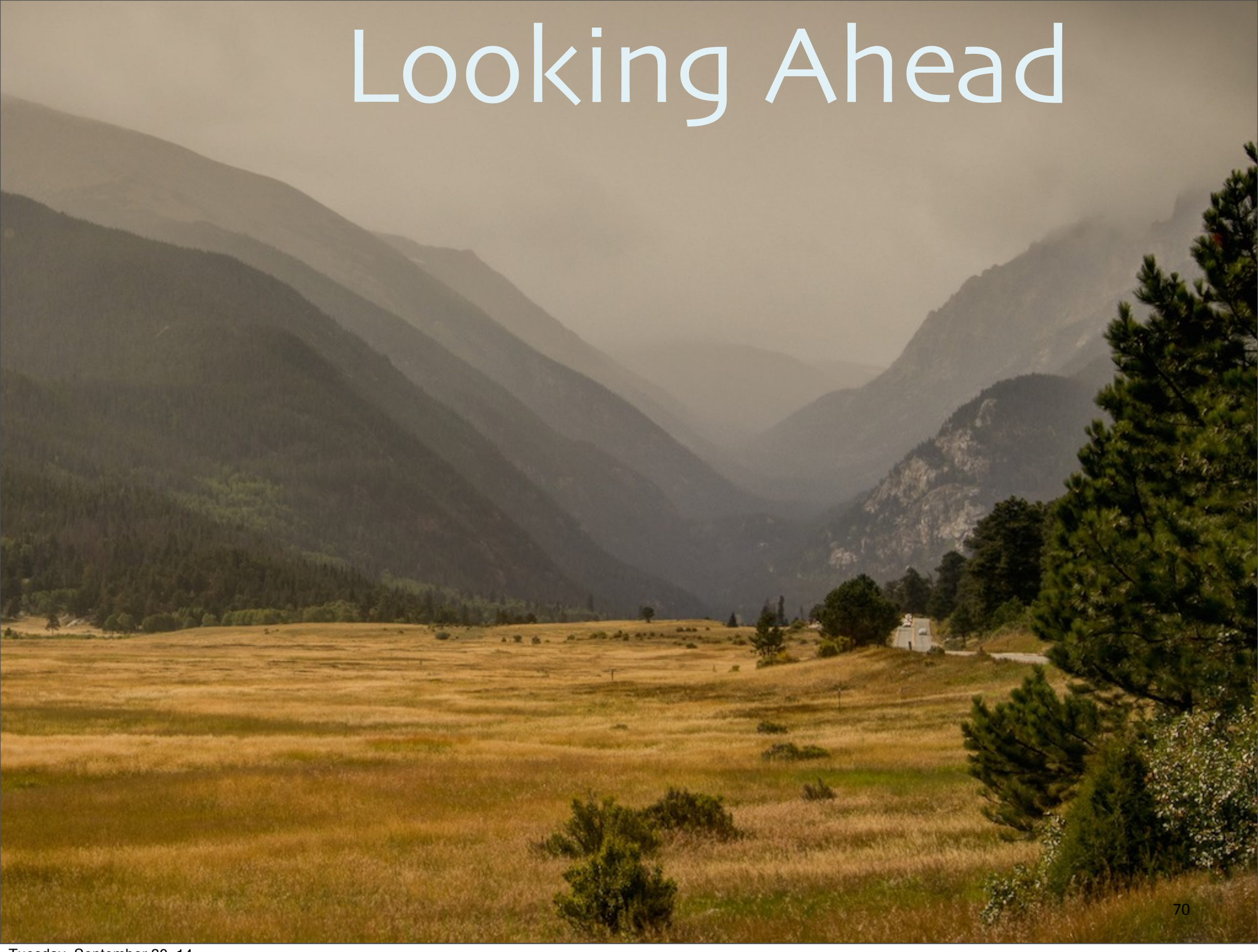
Tuesday, September 30, 14

These are new, relational databases, separate from Hadoop altogether. The leverage the scalability and resiliency lessons of NoSQL, but restore the relational model.

Tuesday, September 30, 14

Google Spanner is globally distributed with *global transactions*. The others are commercial projects.

Looking Ahead



Tuesday, September 30, 14

Gazing into the foggy future...

Spark + Mesos

Tuesday, September 30, 14

Mesos may be all that many teams need, if they don't already have Hadoop (YARN), and especially if they have other infrastructure running on Mesos. (Technically, you can run Hadoop on Mesos, too.)

Flexible cloud deployments

In general, people are pursuing flexible ways of deploying big data apps, especially when they integrate with other systems running in different cloud or virtualization environments.

Tuesday, September 30, 14

I think Tachyon will be disruptive when it’s mature.

H2O

https://github.com/0xdata/h2o

Tuesday, September 30, 14

Check out this high-performance computing engine. https://github.com/0xdata/h2o They are integrating it with Spark.
See Cliff Click's talk!

Recap

Spark

Replaces MapReduce

Spark

Supports Streaming and Batch

Spark

Integrates
SQL, ML, & Graph
libraries

SQL

Works great in Hadoop!

SQL

NewSQL DBs improve Relational scalability

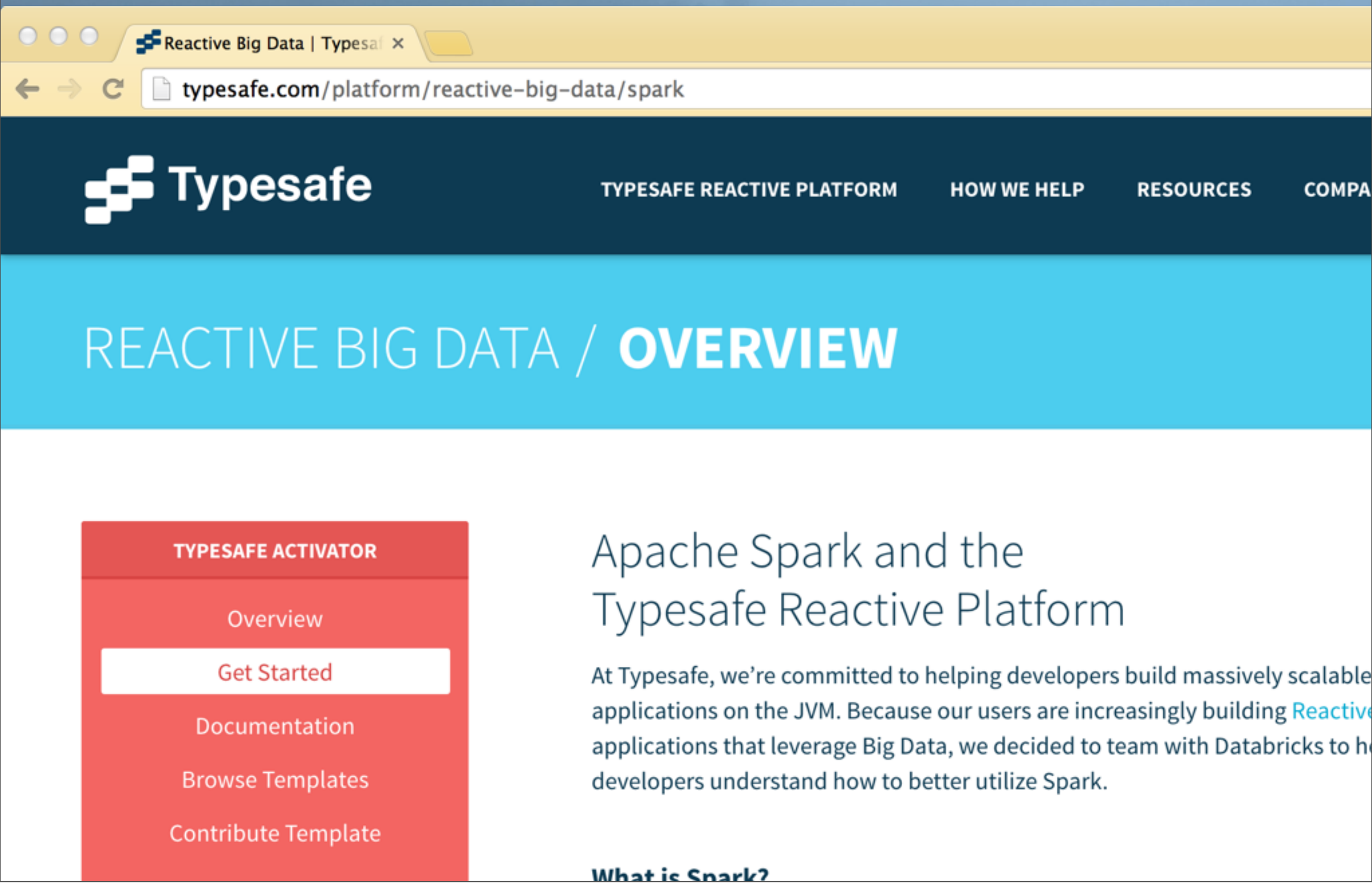
SQL

The world needs NoSQL and NewSQL

Prediction

Mesos-based environments will grow.

More Stuff by Me...



I have a 1-day Spark Workshop I'm teaching for Typesafe. See this page for details, as well as a whitepaper and blog post on Spark that I wrote.

Thank You

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

About me. You can find this presentation and others on Big Data and Scala at polyglotprogramming.com.
Photo: Time lapse at night in Colorado.