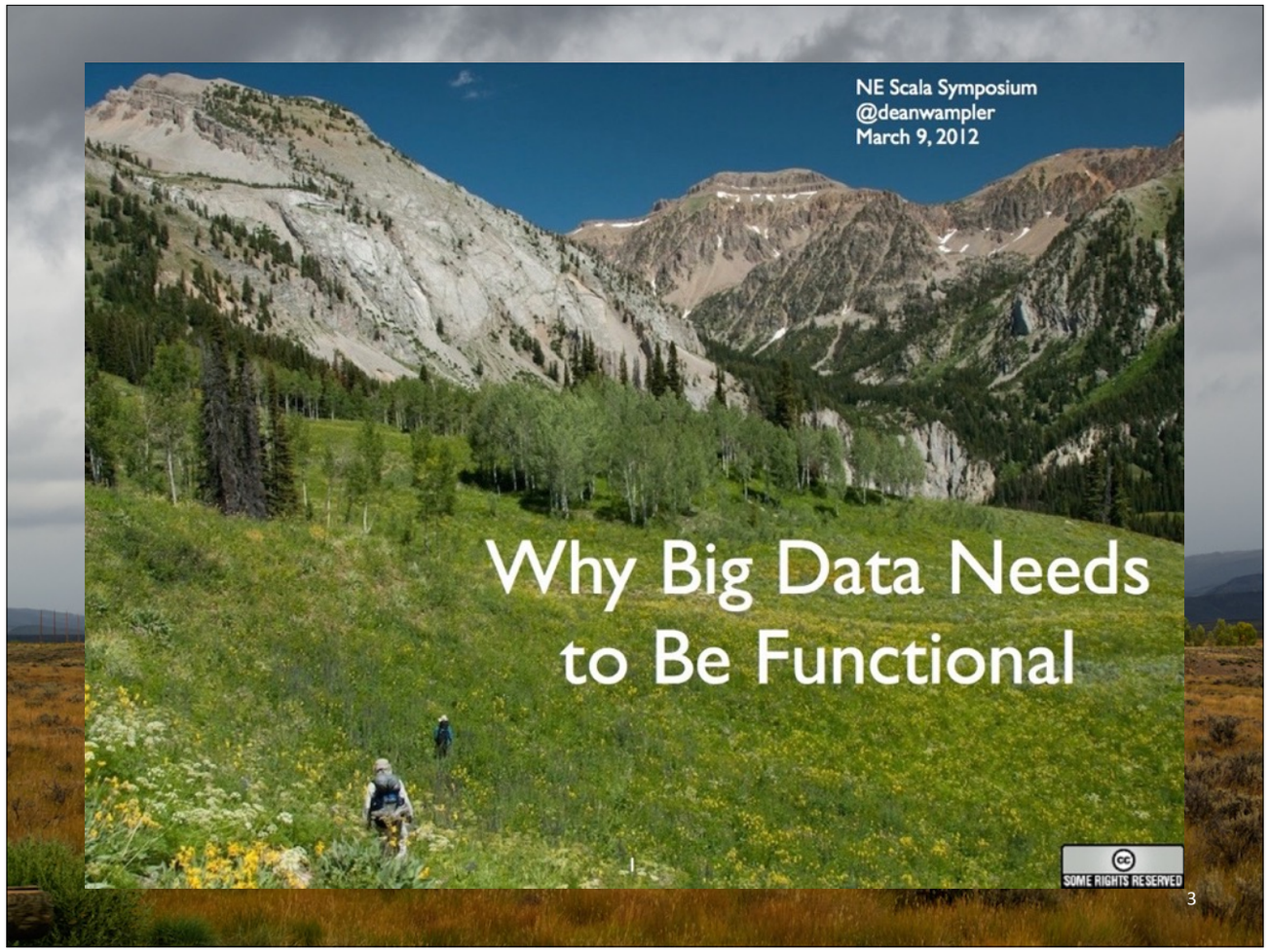
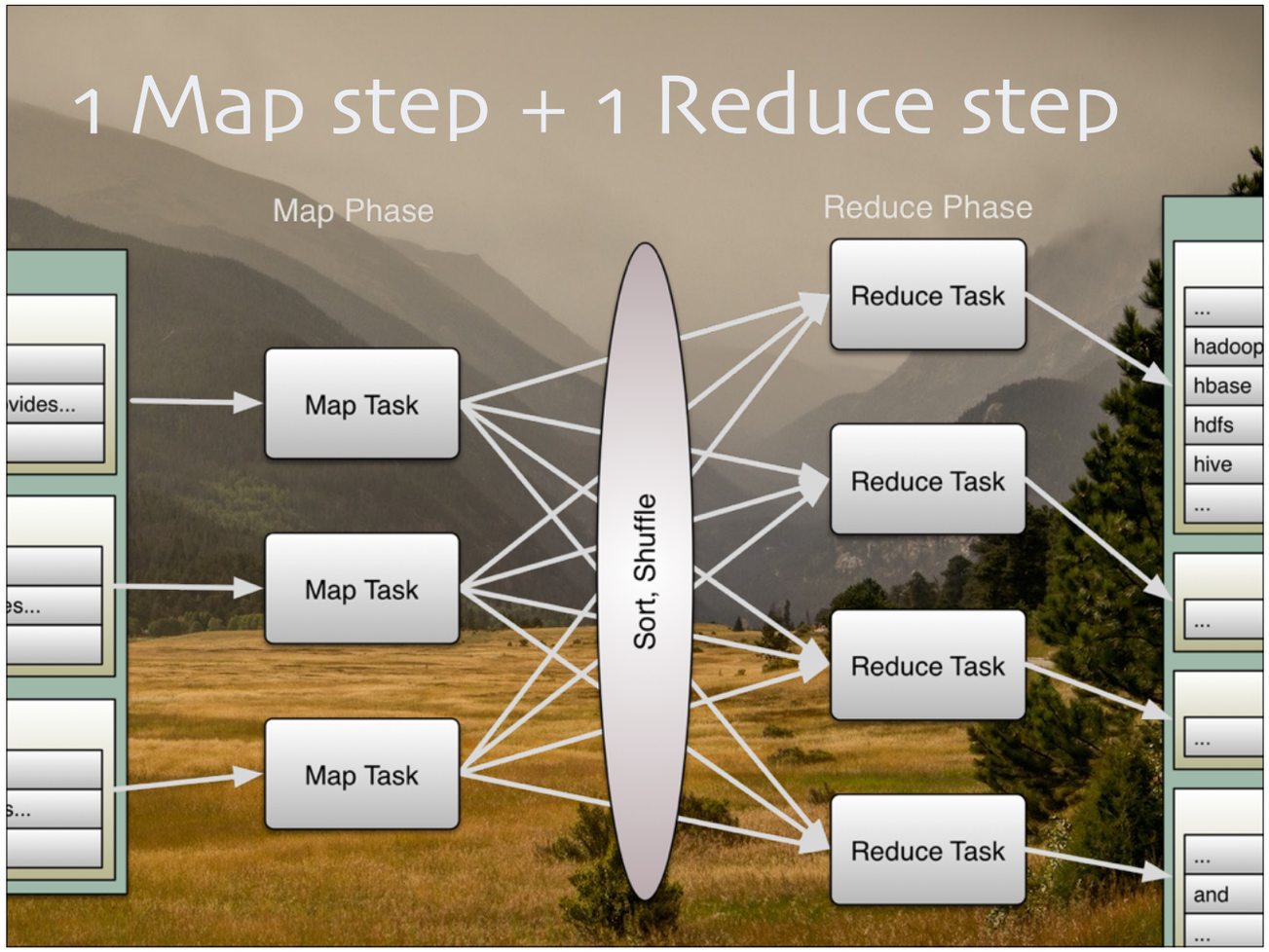
Dean Wampler

"Trolling the Hadoop community since 2012..."

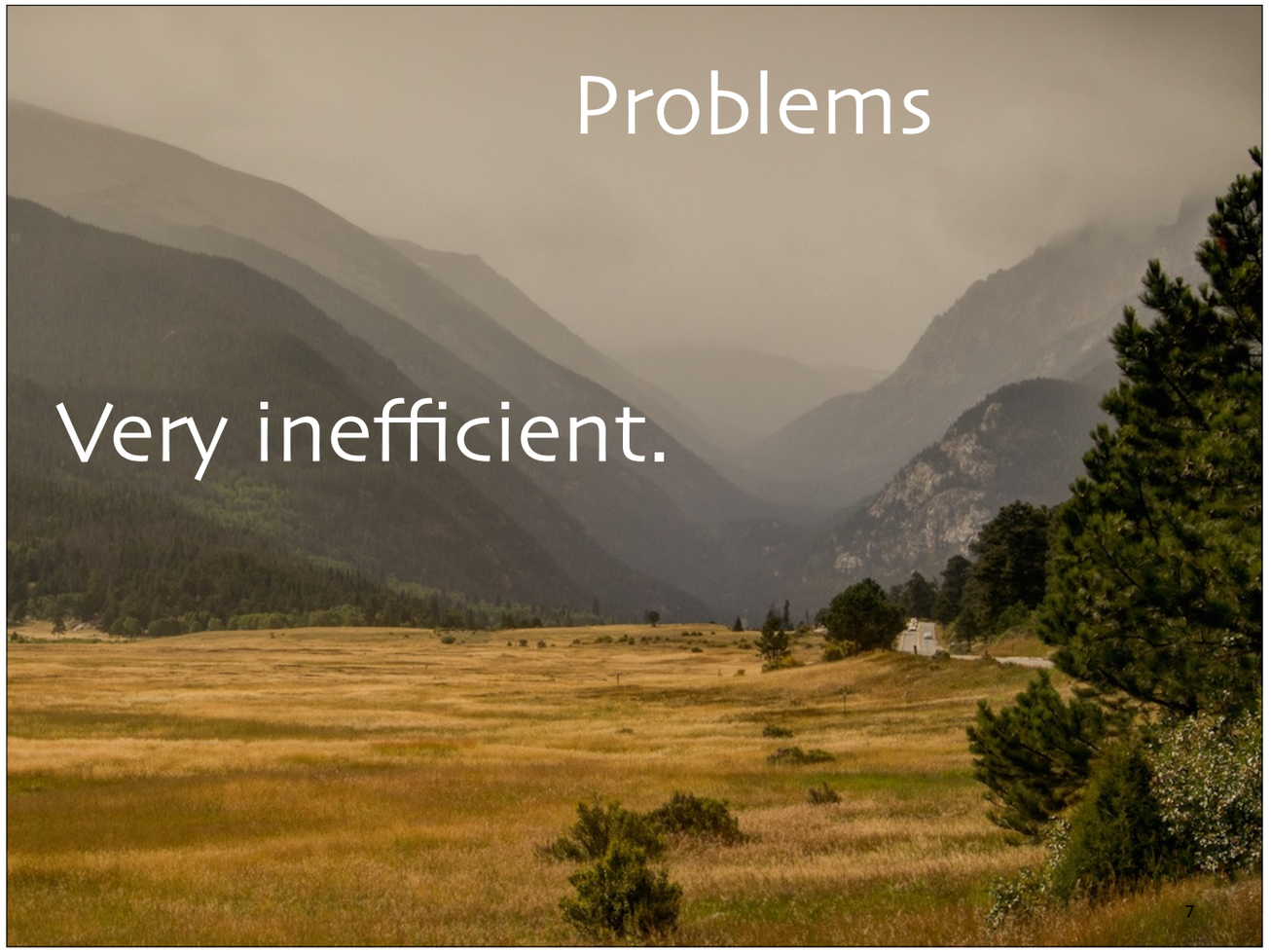

NE Scala Symposium
@deanwampler
March 9, 2012
Why Big Data Needs
to Be Functional
SOME RIGHTS RESERVED

In which I claimed that:

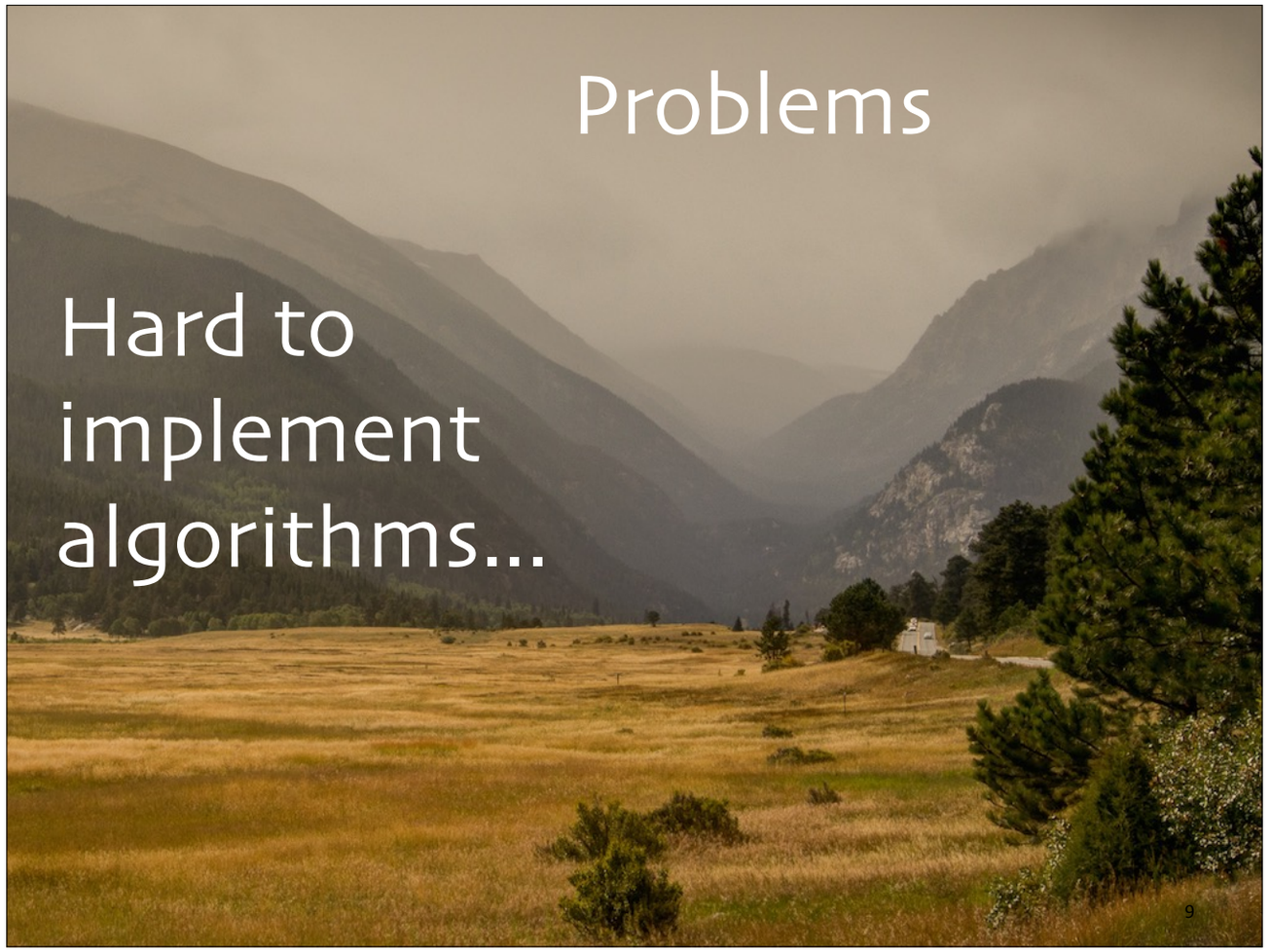
Historically, up to 2013, MapReduce was the officially-supported compute engine for writing all compute jobs.

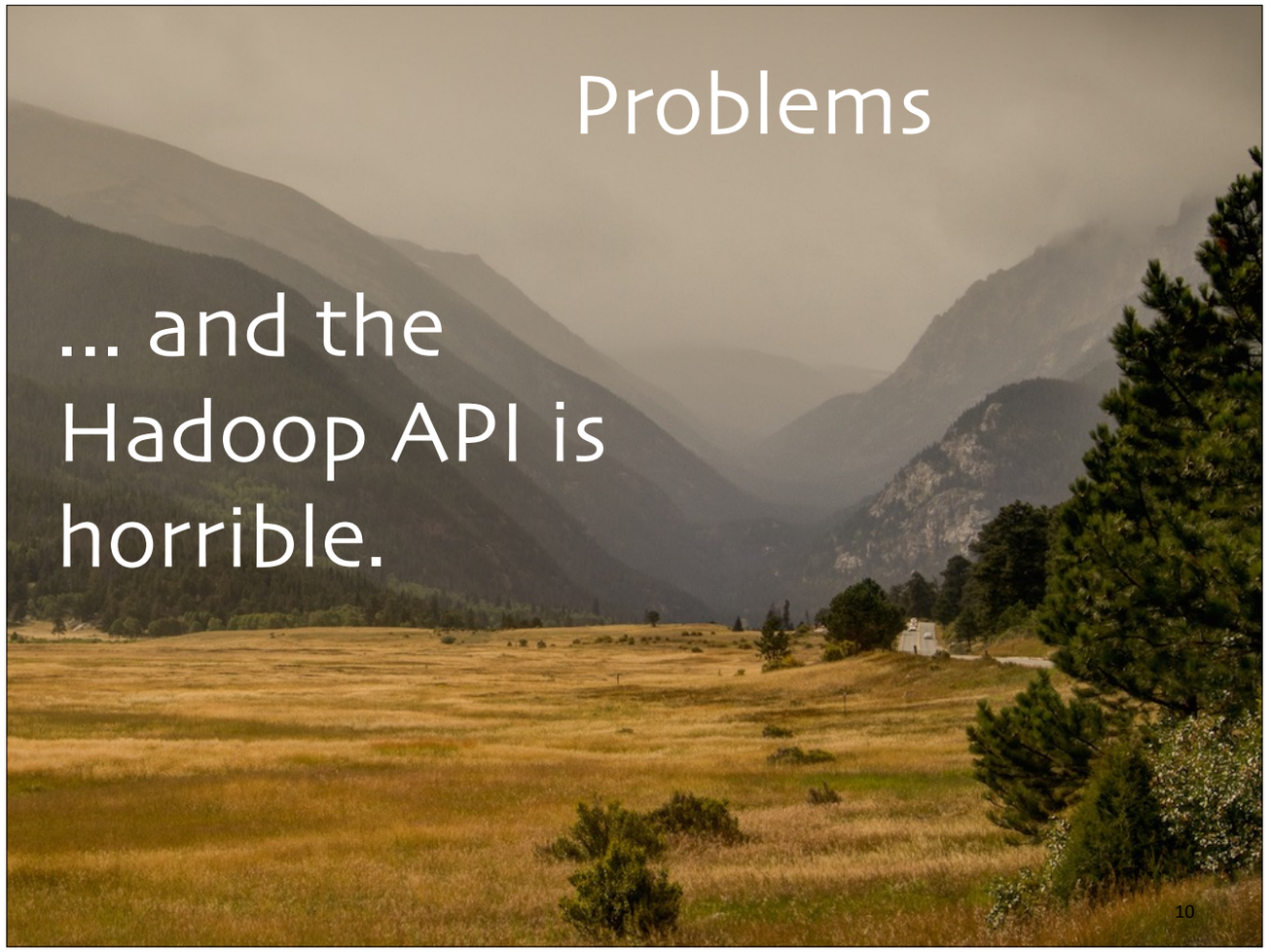


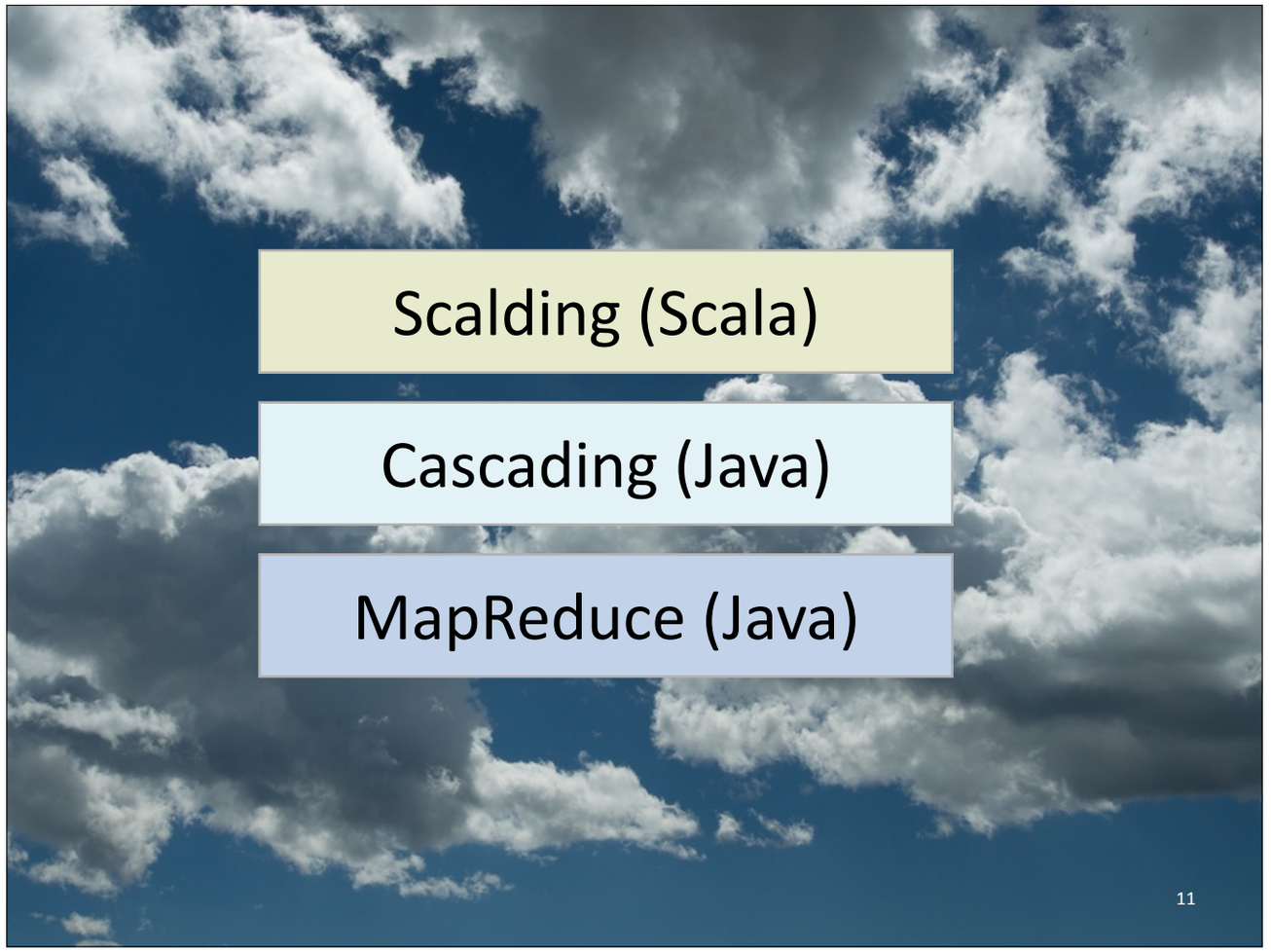
Historically, up to 2013, MapReduce was the officially-supported compute engine for writing all compute jobs.


Problems
Very inefficient.

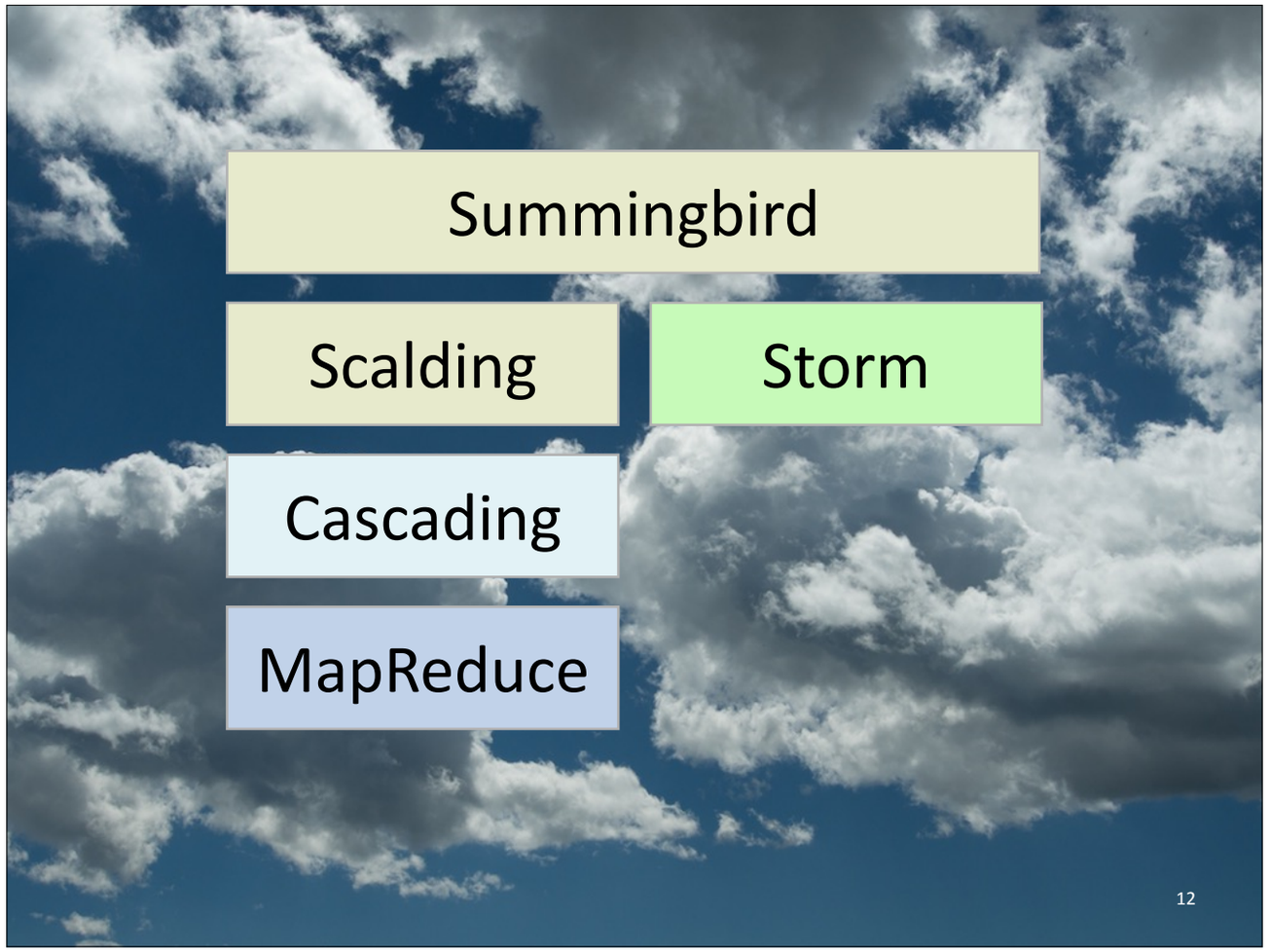

Problems
MapReduce is
“batch-mode”
only


Problems
Hard to implement algorithms...


Problems
... and the
Hadoop API is
horrible.



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”.



Twitter added another layer, Summingbird, to abstract over batch-mode Scalding/MR and streaming Storm.

Started in 2009, interest in Spark as a replacement for MR grew, because it addressed the major MR issues. Finally, in late 2013, Cloudera, the largest Hadoop vendor embraced Spark officially as the replacement for MR.

https://spark.apache.org/sql/
https://spark.apache.org/streaming/

```
val sparkContext =
  new SparkContext("local[*]", "Much Wow!")
val streamingContext =
  new StreamingContext(
    sparkContext, Seconds(60))
val sqlContext =
  new SQLContext(sparkContext)
import sqlContext._

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

```
val sparkContext =
  new SparkContext("local", "connections")
val streamingContext =
  new StreamingContext(
    sparkContext, Seconds(60))
val sqlContext =
  new SQLContext(sparkContext)
import sqlContext._

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


Create the SparkContext that manages for the driver program, followed by context object for streaming and another for the SQL extensions.
Note that the latter two take the SparkContext as an argument. The StreamingContext is constructed with an argument for the size of each batch of events to capture, every 60 seconds here.

```
val sparkContext =
  new SparkContext("local", "connections")
val streamingContext =
  new StreamingContext(
    sparkContext, Seconds(60))
val sqlContext =
  new SQLContext(sparkContext)
import sqlContext._

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


For some APIs, Spark uses the idiom of importing the members of an instance.

```
import sqlContext._

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 dStream =
```


Define a case class to represent a schema, and a companion object to define a parse method for parsing a string into an instance of the class. Return an option in case a string can’t be parsed.
In this case, we’ll simulate a data stream of data about airline flights, where the records contain only the flight number, carrier, and the origin and destination airports, and other data we’ll ignore for this example, like times.

```
}

val server = ...   // IP address or name
val port = ...     // integer
val dStream =
  streamingContext.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,
```


Read the data stream from a socket originating at a given server:port address.

```
}

val server = ...   // IP address or name
val port = ...     // integer
val dStream =
  streamingContext.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,
```


Parse the text stream into Flight records.

```
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)
}

streamingContext.start()
streamingContext.awaitTermination()
streamingContext.stop()
```


A DStream is a collection of RDDs, so for each RDD (effectively, during each batch interval), invoke the anonymous function, which takes as arguments the RDD and current timestamp (epoch milliseconds), then we register the RDD as a “SQL” table named “flights” and run a query over it that groups by the carrier, origin, and destination, selects for those fields, plus the hard-coded timestamp (i.e., “hardcoded” for each batch interval), and the count of records in the group. Also order by the count descending, and return only the first 20 records.

```
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)
}

streamingContext.start()
streamingContext.awaitTermination()
streamingContext.stop()
```


Start processing the stream, await termination, then close it all down.

The title of this talk is in the present tense (present participle to be precise?), but has Scala already won? Is the game over?

Elegant DSLs

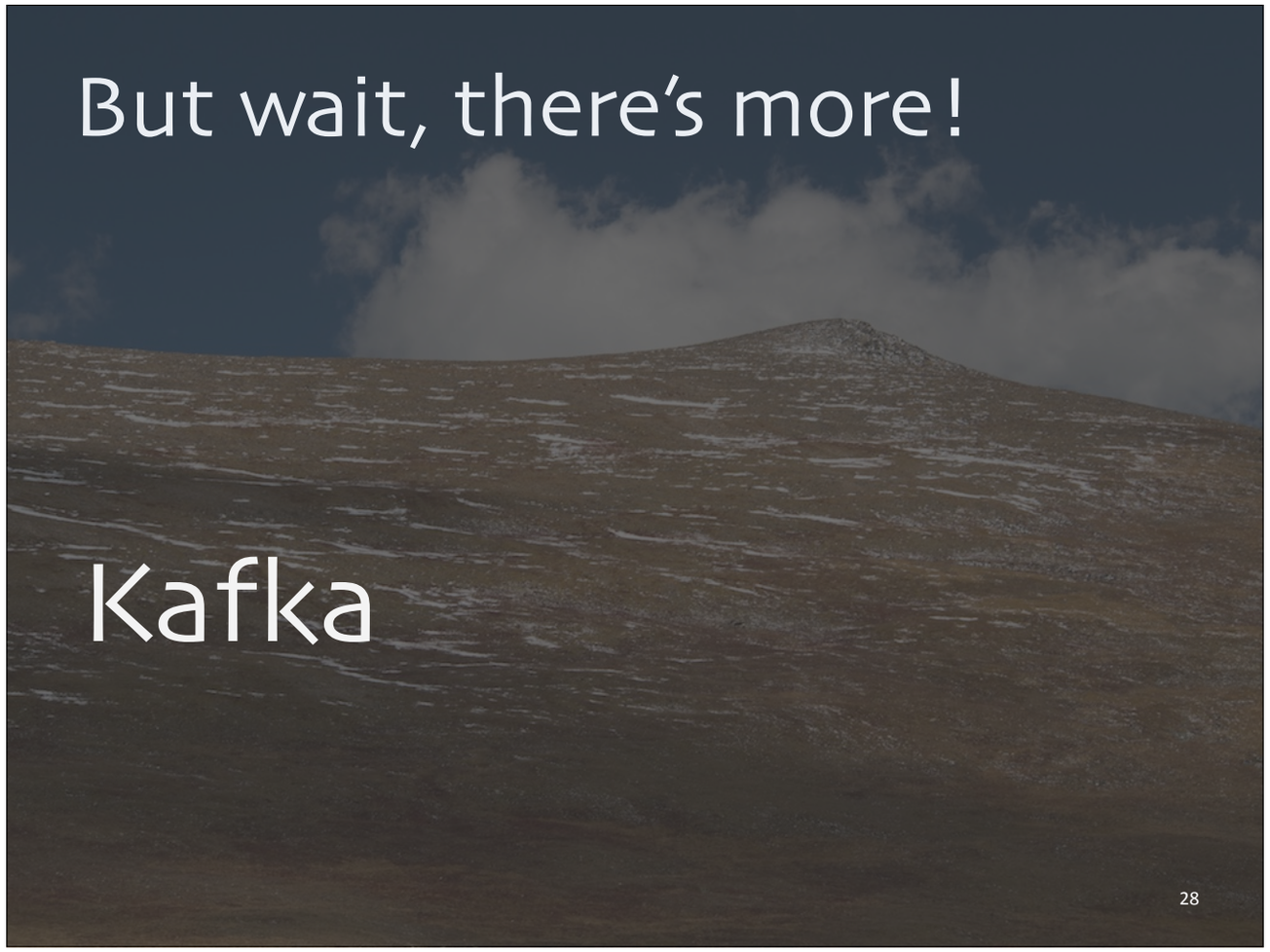
```
...
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
...
```


You would be hard pressed to find a more concise DSL for big data!!

You have the rich Java ecosystem at your fingertips.

You have the rich Java ecosystem at your fingertips.

Shout out to Helena Edelstein



Linkedin's hot message queue, spun off as a company, Clnfluent.


dean.wampler@typesafe.com
polyglotprogramming.com/papers
@deanwampler
Typesafe
SOME RIGHTS RESERVED