Scala and the JVM for Big Data: Lessons from Spark

Strata San Jose
March 30, 2016

Photos from Olympic National Park, Washington State, USA, Aug. 2015.

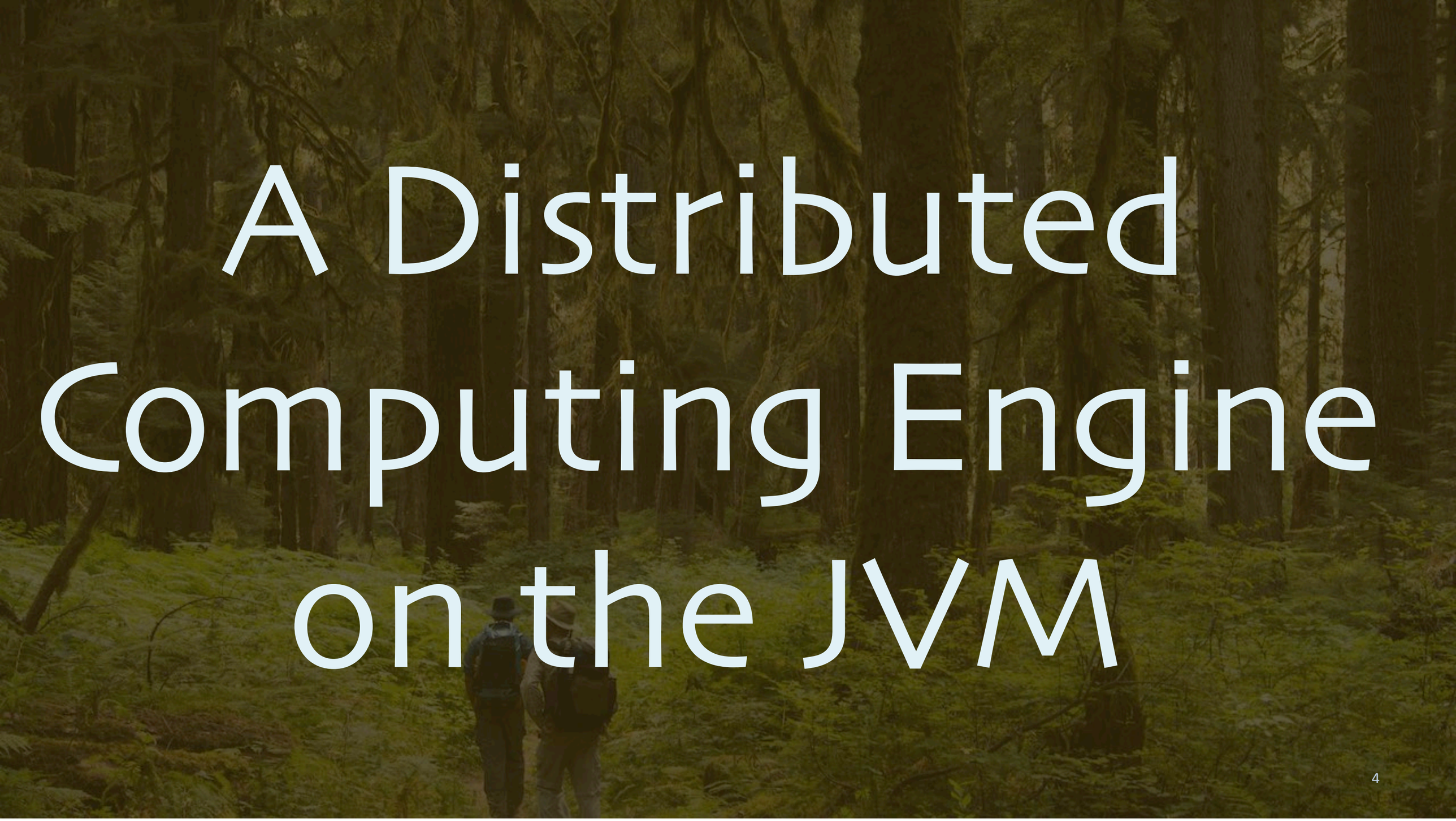
Wednesday, March 30, 16

You can find this and my other talks here.

Spark

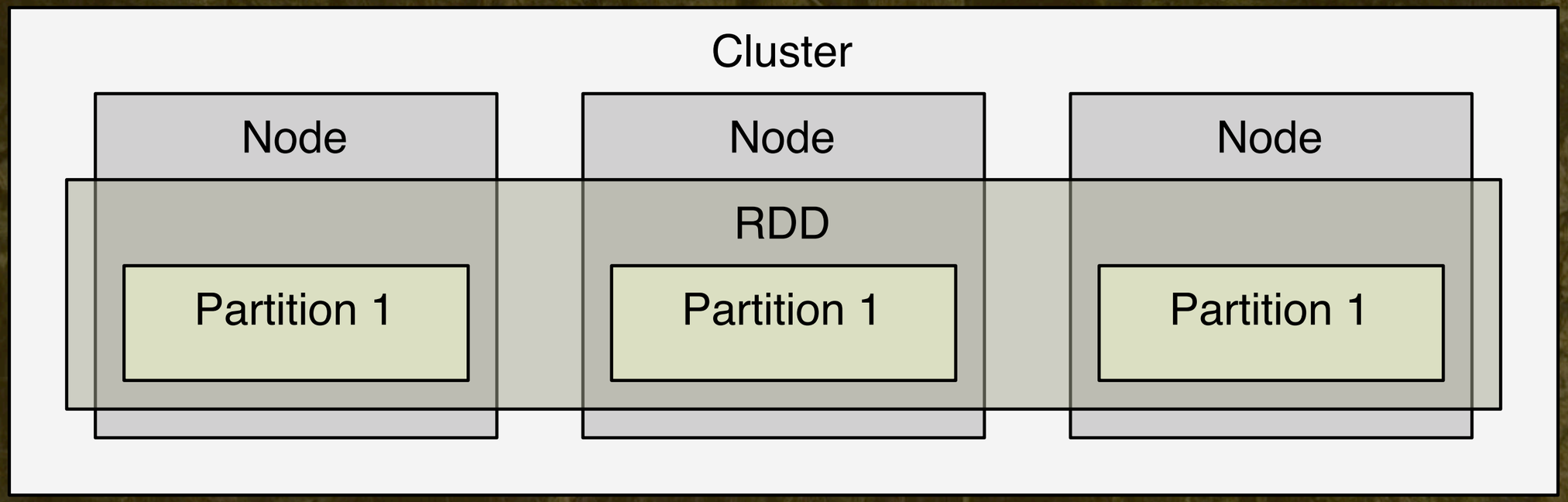
Wednesday, March 30, 16

Scala has become popular for Big Data tools and apps, such as Spark. Why is Spark itself so interesting?



Wednesday, March 30, 16

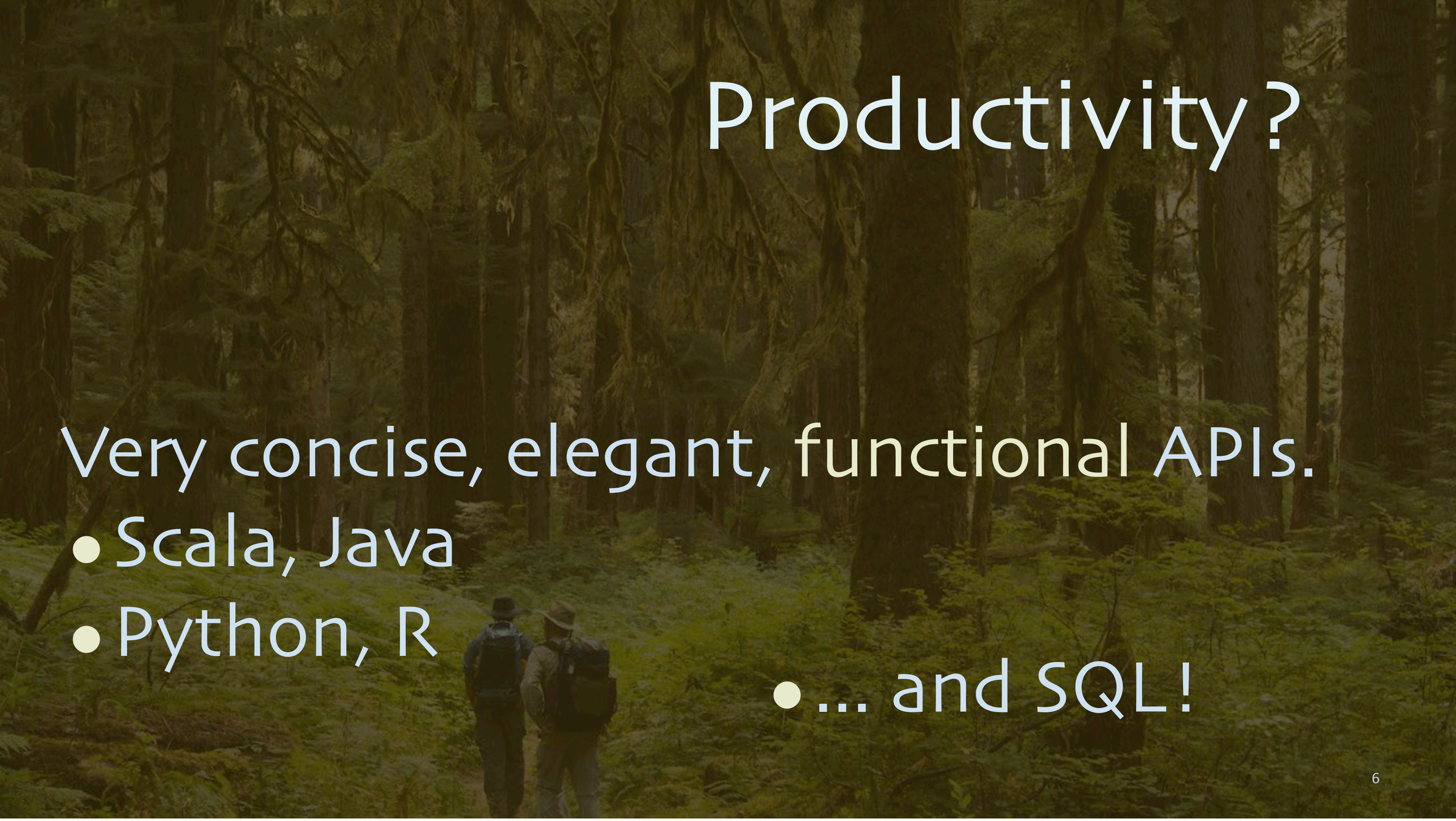
Spark is a general-purpose engine for writing JVM apps to analyze and manipulate massive data sets (although it works well for small ones, too), with the ability to decompose “jobs” into “tasks” that are distributed around a cluster.



Resilient Distributed Datasets

Wednesday, March 30, 16

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. However, you can also cache RDDs to eliminate having to walk back too far. RDDs are immutable, but they are also an abstraction, so you don't actually instantiate a new RDD with wasteful data copies for each step of the pipeline, but rather the end of each stage..



Wednesday, March 30, 16

We saw an example why this true.
While Spark was written in Scala, it has Java, Python, R, and even SQL APIs, too, which means it can be a single tool used across a Big Data organization, engineers and data scientists.

Productivity?

Interactive shell (REPL)

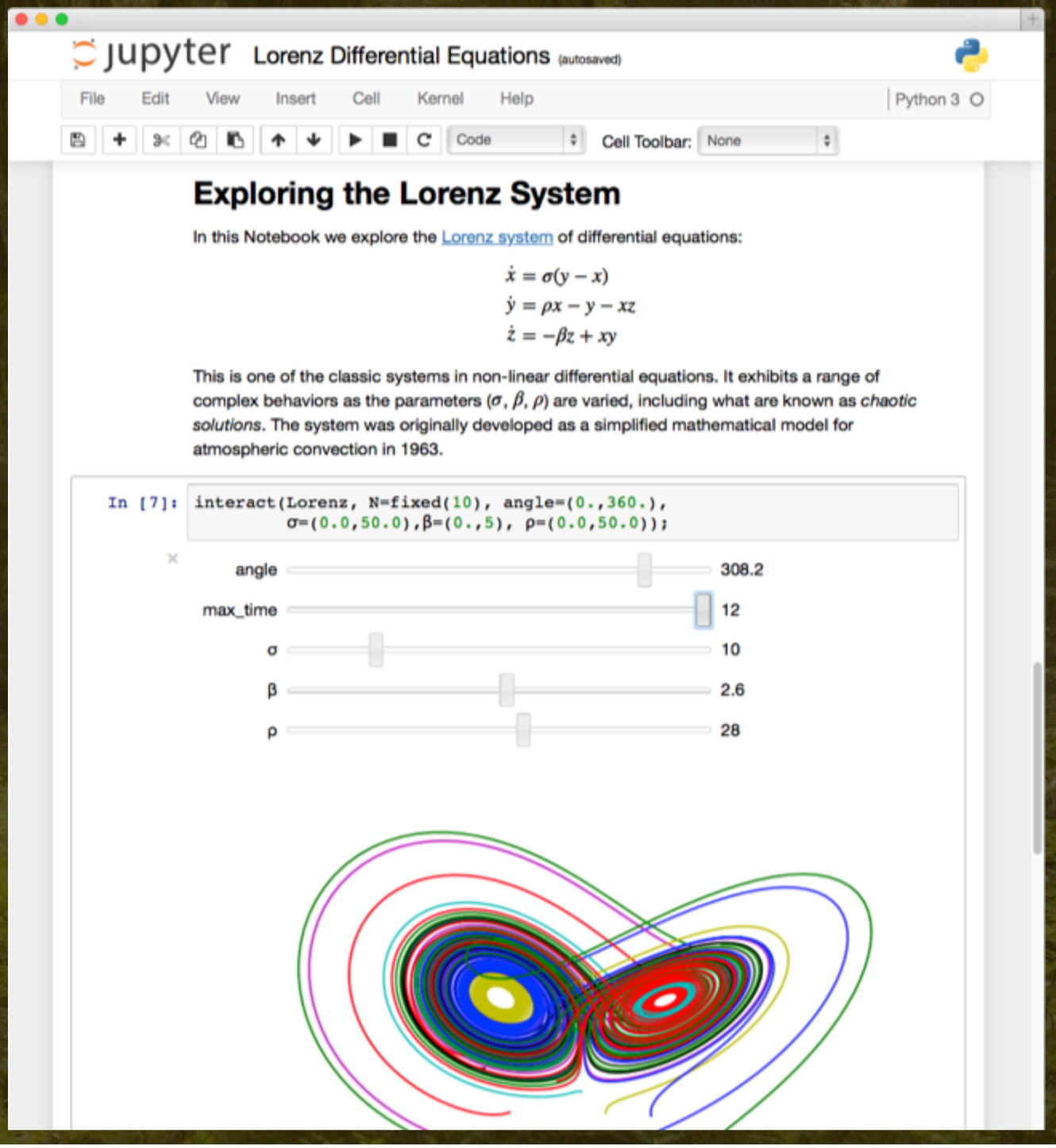
- Scala, Python, R, and SQL

Wednesday, March 30, 16

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. In fact, I tend to experiment in the REPL or Spark Notebook (next slide), then copy the code to a more "permanent" form, when it's supposed to be compiled and run as a batch program.

Notebooks

- Jupyter
- Spark Notebook
- Zeppelin
- Databricks



Wednesday, March 30, 16

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. In fact, I tend to experiment in the REPL or Spark Notebook, then copy the code to a more "permanent" form, when it's supposed to be compiled and run as a batch program. Databricks is a commercial, hosted notebook offering.

Wednesday, March 30, 16

Example: Inverted Index

Let’s look at a small, real actual Spark program, the Inverted Index.

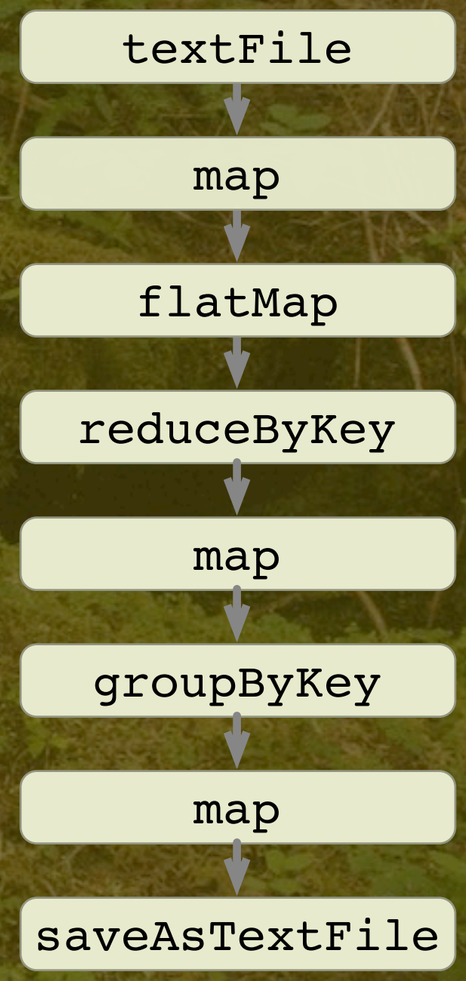
```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sparkContext = new SparkContext(master, “Inv. Index”)
sparkContext.textFile("/path/to/input").
map { line =>
 val array = line.split(",", 2)
 (array(0), array(1))  // (id, content)
}.flatMap {
 case (id, content) =>
  toWords(content).map(word => ((word,id),1)) // toWords not shown
}.reduceByKey(_ + _).
map {
 case ((word,id),n) => (word,(id,n))
}.groupByKey.
mapValues {
 seq => sortByCount(seq)  // Sort the value seq by count, desc.
}.saveAsTextFile("/path/to/output")
```


All on one slide. A very short program, which means your productivity is high and the “software engineering process” is drastically simpler. See an older version of this talk on polyglotprogramming.com/talks for a walkthrough of this code.

Productivity?

Intuitive API:

- Dataflow of steps.
- Inspired by Scala collections and functional programming.

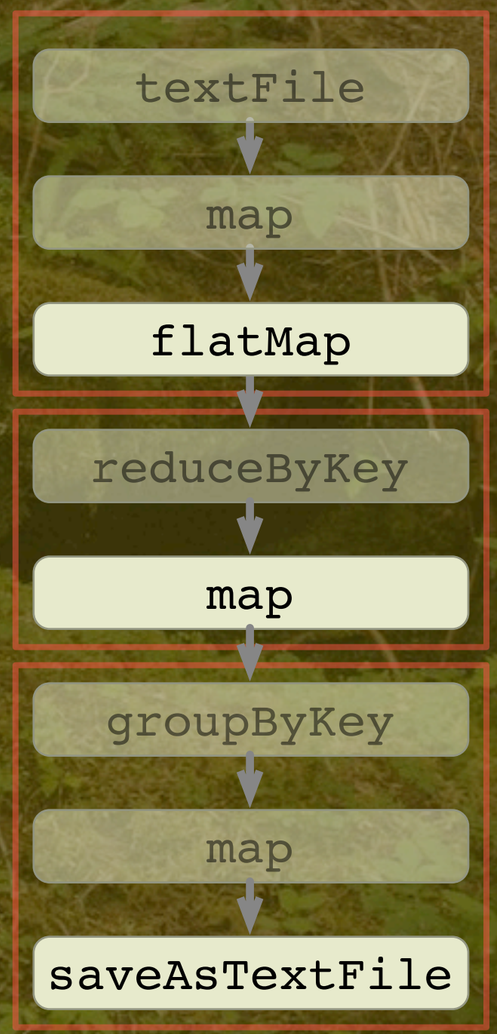


One you learn all these “operators” from functional programming, as expressed first in the Scala collections API and then adapted by Spark, you can quickly construct data flows that are intuitive, yet powerful.

Performance?

Lazy API:

- Combines steps into "stages".
- Cache intermediate data in memory.



Wednesday, March 30, 16

How is Spark more efficient? 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.

The transformation steps that don't require data from other partitions can be pipelined together into a single JVM process (per partition), called a Stage. When you do need to bring together data from different partitions, such as group-bys, joins, reduces, then data must be "shuffled" between partitions (i.e., all keys of a particular value must arrive at the same JVM instance for the next transformation step). That triggers a new stage, as shown. So, this algorithm requires three stages and the RDDs are materialized only for the last steps in each stage.

Higher-Level APIs

Wednesday, March 30, 16

Composable operators, performance optimizations, and general flexibility are a foundation for higher-level APIs...
A major step forward compared to MapReduce. Due to the lightweight nature of Spark processing, it can efficiently support a wider class of algorithms.

Wednesday, March 30, 16

For data with a known, fixed schema.
Like Hive for MapReduce, a subset of SQL (omitting transactions and the U in CRUD) is relatively easy to implement as a DSL on top of a general compute engine like Spark. Hence, the SQL API was born, but it's grown into a full-fledged programmatic API supporting both actual SQL queries and an API similar to Python's DataFrame API for working with structured data in a more type-safe way (errr, at least for Scala).

Example

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

val sparkContext = new
 SparkContext(master, “...”)
val sqlContext = new
 SQLContext(sparkContext)
val flights =
 sqlContext.read.parquet(".../flights")
val planes =
 sqlContext.read.parquet(".../planes")
flights.registerTempTable("flights")
planes.registerTempTable("planes")
flights.cache(); planes.cache()

val planes_for_flights1 = sqlContext.sql("""
  SELECT * FROM flights f
  JOIN planes p ON f.tailNum = p.tailNum LIMIT 100""")
```

```
val planes_for_flights2 =
 flights.join(planes,
  flights("tailNum") ===
  planes ("tailNum")).limit(100)
```


Example using SparkSQL to join two data sets (adapted from Typesafe’s Spark Workshop training), data for flights and information about the planes.

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

val sparkContext = new
 SparkContext(master, "...")
val sqlContext = new
 SQLContext(sparkContext)
val flights =
 sqlContext.read.parquet("../flights")
val planes =
 sqlContext.read.parquet("../planes")
```


Now we also need a SQLContext.

```
 SQLContext(sparkContext)
val flights =
 sqlContext.read.parquet(".../flights")
val planes =
 sqlContext.read.parquet(".../planes")
flights.registerTempTable("flights")
planes.registerTempTable("planes")
flights.cache(); planes.cache()
```

```
val planes_for_flights1 =
sqlContext.sql("""
  SELECT * FROM flights f
```


Wednesday, March 30, 16

Read the data as Parquet files, which include the schemas. Create temporary tables (purely virtual) for SQL queries, and cache the tables for faster, repeated access.

```
val planes_for_flights1 =
sqlContext.sql("""
  SELECT * FROM flights f
  JOIN planes p
  ON f.tailNum = p.tailNum
  LIMIT 100""")
```

Returns another DataFrame.

```
val planes_for_flights2 =
 flights.join(planes,
 flights("tailNum") ===
 planes ("tailNum")).limit(100)
```


Wednesday, March 30, 16

Use SQL to write the query. There are DataFrame operations we can then use on this result (the next part of the program uses this API to redo the query). A DataFrame wraps an RDD, so we can also all RDD methods to work with the results. Hence, we can mix and match SQL and programmatic APIs to do what we need to do.

```
val planes_for_flights1 =
sqlContext.sql("""
  SELECT * FROM flights f
  JOIN planes p
  ON f.tailNum = p.tailNum
  LIMIT 100""")
```

```
val planes_for_flights2 =
 flights.join(planes,
 flights("tailNum") ===
 planes ("tailNum")).limit(100)
```


Wednesday, March 30, 16

Use the DataFrame DSL to write the query (more type safe). Also returns a new DataFrame.

```
val planes_for_flights2 =
  flights.join(planes,
    flights("tailNum") ===
    planes ("tailNum")).limit(100)
```

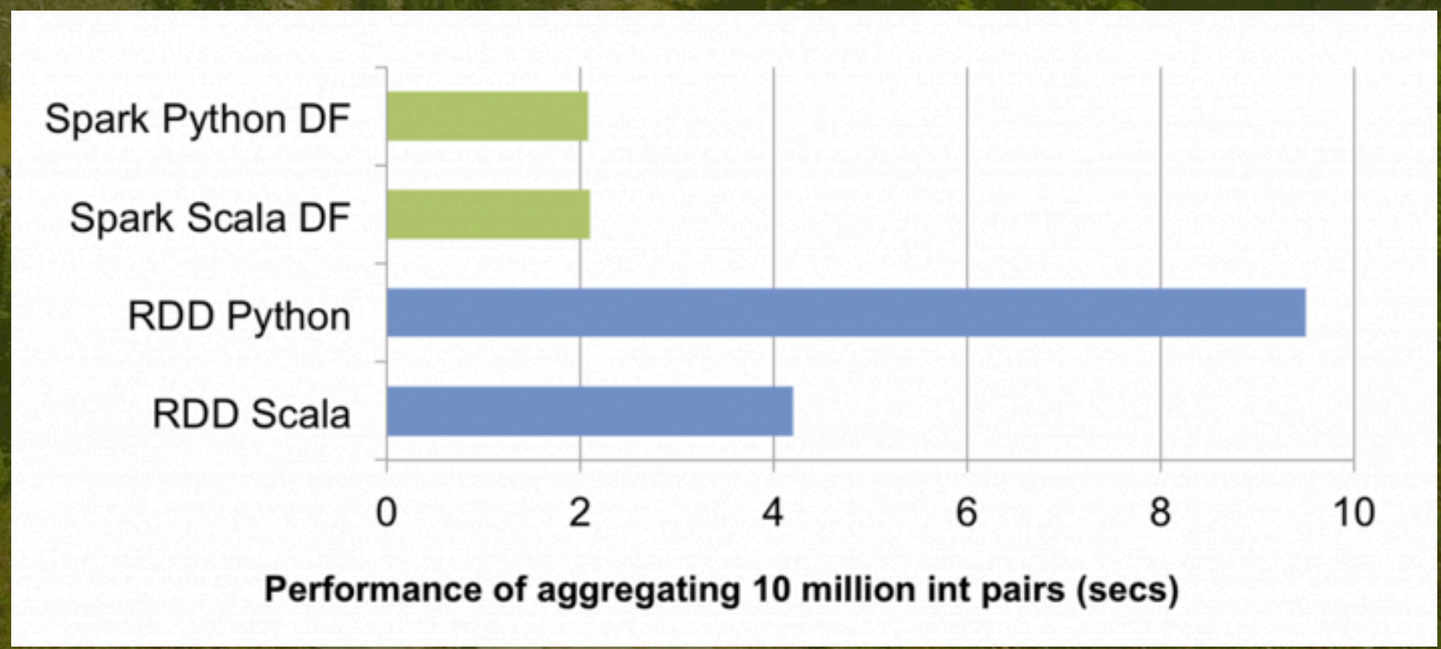
Not an “arbitrary” predicate anon. function, but a “Column” instance.

Wednesday, March 30, 16

Looks like an anonymous function, but actually it’s actually a join expression that constructs an instance of a “Column” type. By constraining what’s allowed here, Spark knows the exact expression used here and it can apply aggressive optimizations at run time.

Performance

The DataFrame API has the same performance for all languages: Scala, Java, Python, R, and SQL!



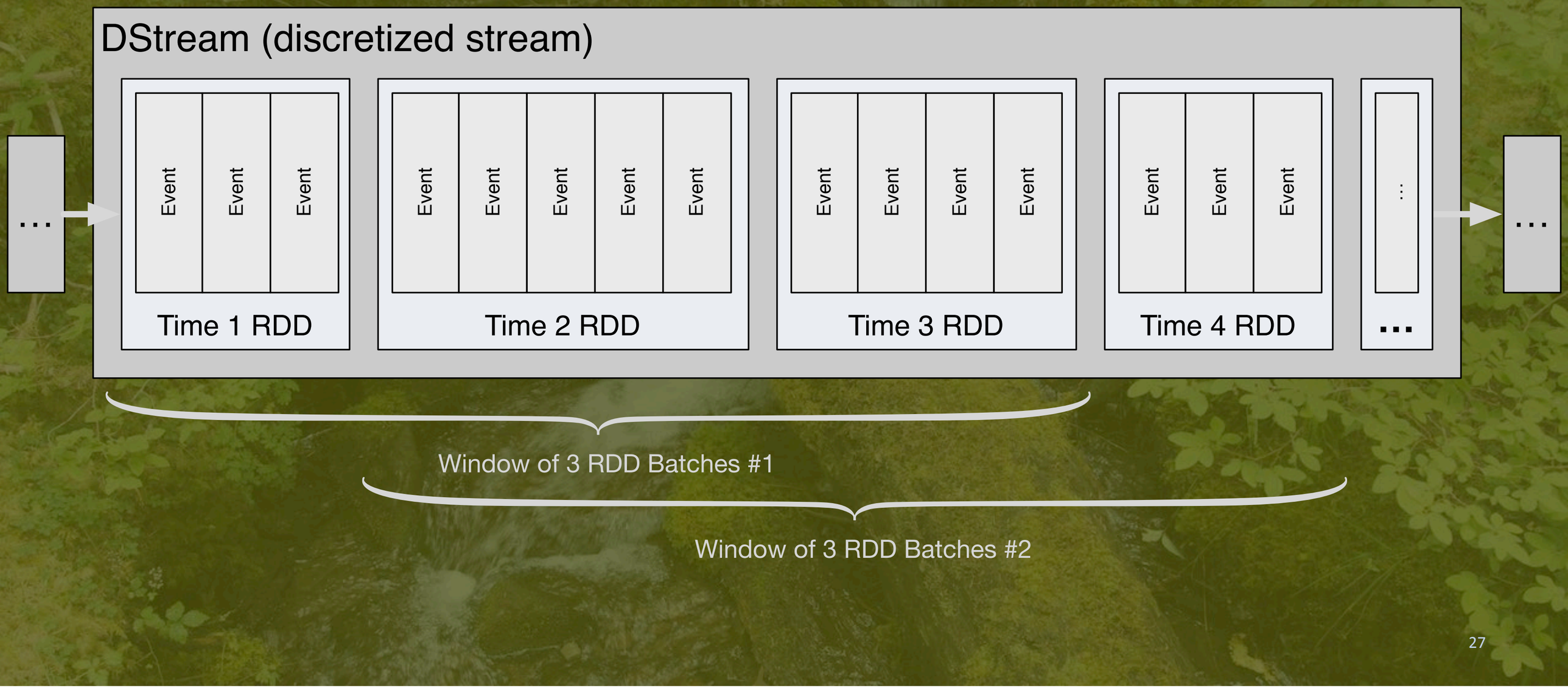
All the different language APIs are thin veneers on top of the Catalyst query optimizer and other Scala-based code. This performance independent of language choice is a major step forward. Previously for Hadoop, Data Scientists often developed models in Python or R, then an engineering team ported them 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.
Graph from: https://databricks.com/blog/2015/04/24/recent-performance-improvements-in-apache-spark-sql-python-dataframes-and-more.html

```
def join(right: DataFrame, joinExprs: Column): DataFrame = {
def groupBy(cols: Column*): GroupedData = {
def orderBy(sortExprs: Column*): DataFrame = {
def select(cols: Column*): DataFrame = {
def where(condition: Column): DataFrame = {
def limit(n: Int): DataFrame = {
def unionAll(other: DataFrame): DataFrame = {
def intersect(other: DataFrame): DataFrame = {
def sample(withReplacement: Boolean, fraction, seed) = {
def drop(col: Column): DataFrame = {
def map[R: ClassTag](f: Row => R): RDD[R] = {
def flatMap[R: ClassTag](f: Row => Traversable[R]): RDD[R] ={
def foreach(f: Row => Unit): Unit = {
def take(n: Int): Array[Row] = {
def count(): Long = {
def distinct(): DataFrame = {
def agg(exprs: Map[String, String]): DataFrame = {
```


So, the DataFrame exposes a set of relational (-ish) operations, more limited than the general RDD API. This narrower interface enables broader (more aggressive) optimizations and other implementation choices underneath.

Wednesday, March 30, 16


DStreams:
Spark Streaming
26



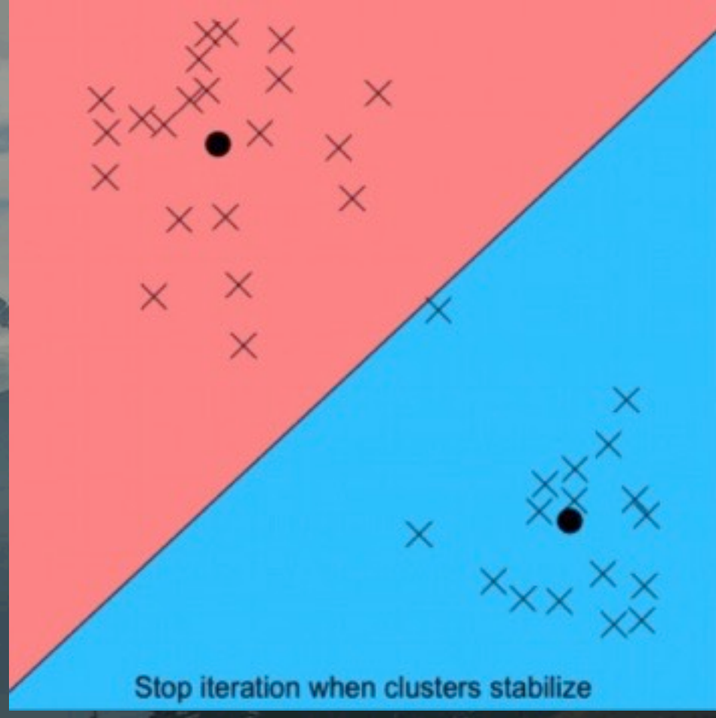
Wednesday, March 30, 16

For streaming, one RDD is created per batch iteration, with a DStream (discretized stream) holding all of them, which supports window functions. Spark started life as a batch-mode system, just like MapReduce, 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.

Wednesday, March 30, 16

Many machine learning libraries are being implemented on top of Spark. An important requirement is the ability to do linear algebra and iterative algorithms quickly and efficiently, which is used in the training algorithms for many ML models.

K-Means



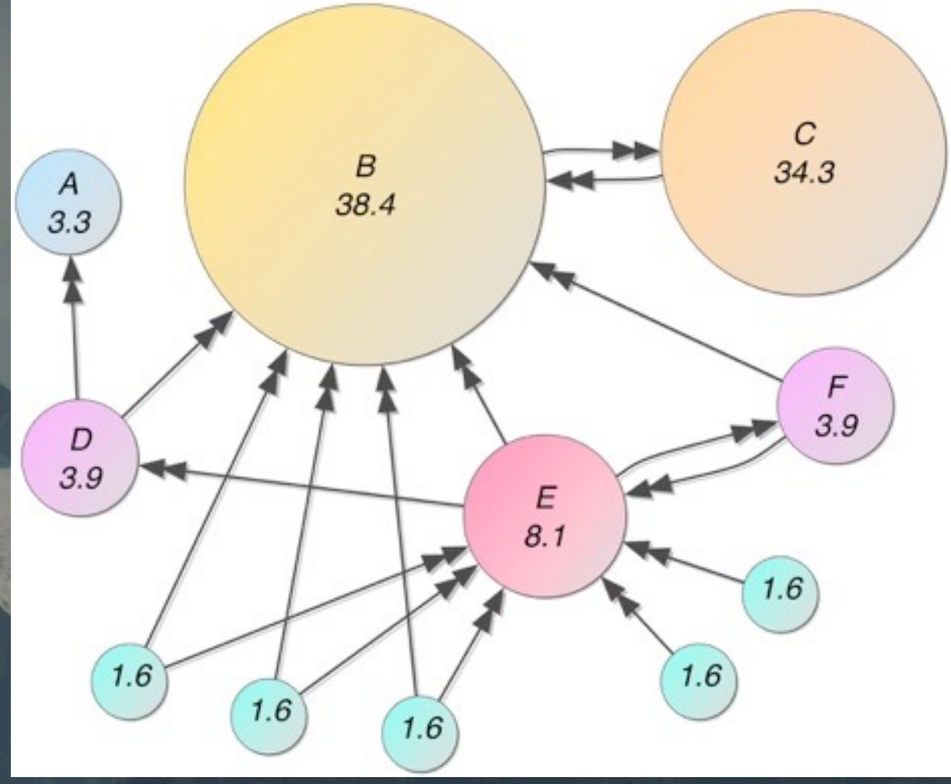
- Machine Learning requires:
- Iterative training of models.
- Good linear algebra perf.

Wednesday, March 30, 16

Many machine learning libraries are being implemented on top of Spark. An important requirement is the ability to do linear algebra and iterative algorithms quickly and efficiently, which is used in the training algorithms for many ML models.
K-Means Clustering simulation: https://en.wikipedia.org/wiki/K-means_clustering

Similarly, efficient iteration makes graph traversal algorithms tractable, enabling “first-class” graph representations of data, like social networks.

PageRank



- Graph algorithms require:
- Incremental traversal.
- Efficient edge and node reps.

Wednesday, March 30, 16

Similarly, efficient iteration makes graph traversal algorithms tractable, enabling “first-class” graph representations of data, like social networks.
Page Rank example: https://en.wikipedia.org/wiki/PageRank

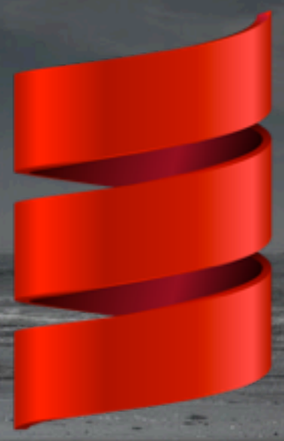
Wednesday, March 30, 16

With that introduction, let's step back and look at the larger context, starting at the bottom; why is the JVM the platform of choice for Big Data?

Wednesday, March 30, 16

We have 20 years of operations experience running JVM-based apps in production. We have many Java developers, some with 20 years of experience, writing JVM-based apps.

Tools and Libraries



Akka
Breeze
Algebird
Spire & Cats
Axle

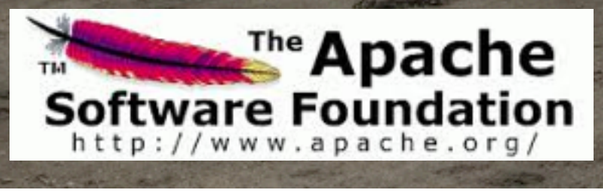
...

Wednesday, March 30, 16

We have a rich suite of mature(-ish) languages, development tools, and math-related libraries for building data applications...

http://akka.io - For resilient, distributed middleware.
https://github.com/scalanlp/breeze - A numerical processing library around LAPACK, etc.
https://github.com/twitter/algebird - A big data math library, developed at Twitter
https://github.com/non/spire - A numerical library
https://github.com/non/cats - A Category Theory library
http://axle-lang.org/ - DSLs for scientific computing.

Big Data Ecosystem



Wednesday, March 30, 16

All this is why most Big Data tools in use have been built on the JVM, including Spark, especially those for OSS projects created and used by startups.

http://spark.apache.org
https://github.com/twitter/scalding
https://github.com/twitter/summingbird
http://kafka.apache.org
http://hadoop.apache.org
http://cassandra.apache.org
http://storm.apache.org
http://samza.apache.org
http://lucene.apache.org/solr/
http://h2o.ai

Wednesday, March 30, 16

But no system is perfect. The JVM wasn't really designed with Big Data systems, as we call them now, in mind.

Wednesday, March 30, 16

First, a small, but significant issue; since Python and R have longer histories as Data Science languages, they have much richer and more mature libraries, e.g., statistics, machine learning, natural language processing, etc., compared to the JVM.

Garbage Collection

Garbage collection is a wonder thing for removing a lot of tedium and potential errors from code, but the default settings in the JVM are not ideal for Big Data apps.

GC Challenges

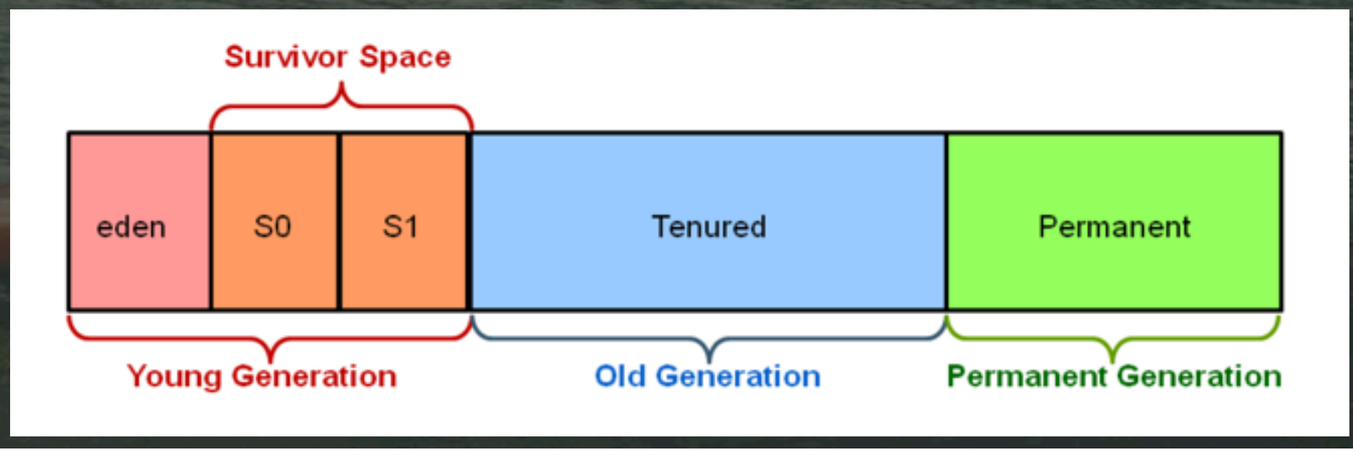
- Typical Spark heaps: 10s-100s GB.
- Uncommon for "generic", non-data services.

Wednesday, March 30, 16

We'll explain the details shortly, but the programmer controls which data sets are cached to optimize usage.
See https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html for a detailed overview of GC challenges and tuning for Spark.

GC Challenges

- Too many cached RDDs leads to huge old generation garbage.
- Billions of objects => long GC pauses.



We'll explain the details shortly, but the programmer controls which data sets are cached to optimize usage.
See https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html for a detailed overview of GC challenges and tuning for Spark.
Image from http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.
Another example is the Hadoop Distributed File System Name Node service, which holds the file system's metadata in memory, often 10s-100sGB. At these sizes, there have been occurrences of multi-hour GC pauses. Careful tuning is required.

Tuning GC

- Best for Spark:
  - -XX:UseG1GC -XX:-ResizePLAB -Xms... -Xmx... -XX:InitiatingHeapOccupancyPercent=... -XX:ConcGCThread=...

databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

Wednesday, March 30, 16

Summarizing a long, detailed blog post (https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html) in one slide!
Their optimal settings for Spark for large data sets, before the “Tungsten” optimizations. The numbers elided (“...”) are scaled together.

JVM Object Model

Wednesday, March 30, 16

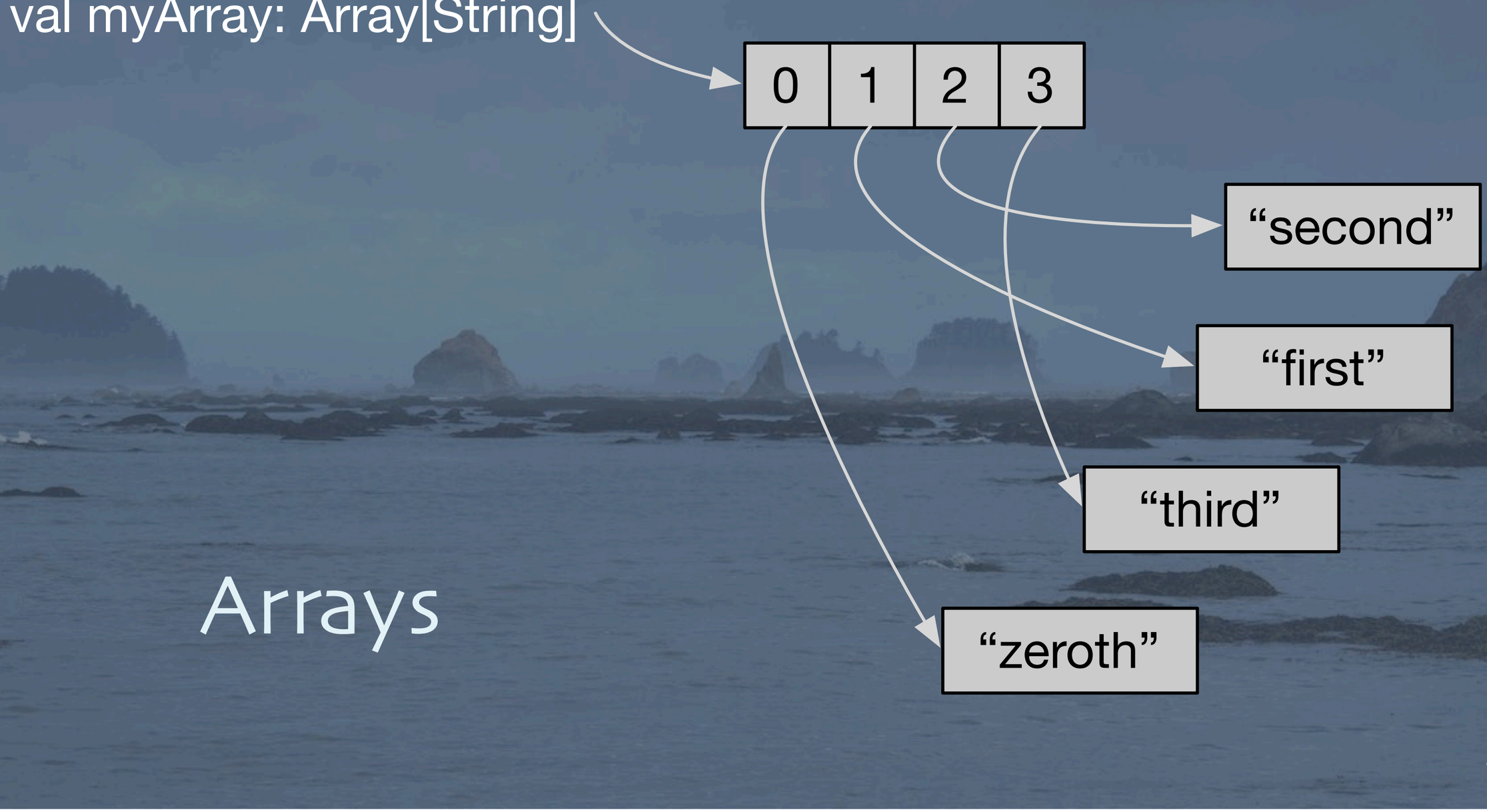
For general-purpose management of trees of objects, Java works well, but not for data orders of magnitude larger, where you tend to have many instances of the same “schema”.

Java Objects?

- "abcd": 4 bytes for raw UTF8, right?
- 48 bytes for the Java object:
  - 12 byte header.
  - 8 bytes for hash code.
  - 20 bytes for array overhead.
  - 8 bytes for UTF16 chars.

From http://www.slideshare.net/SparkSummit/deep-dive-into-project-tungsten-josh-rosen

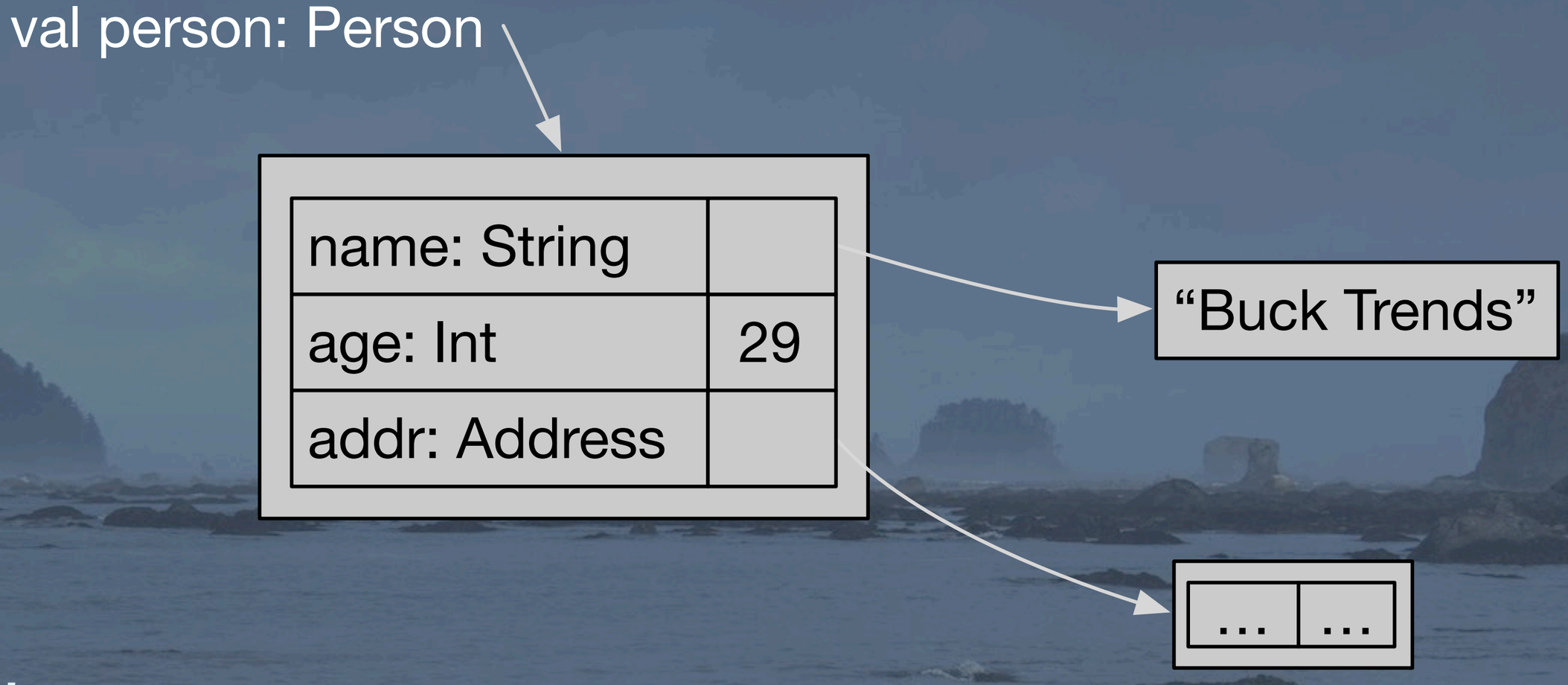
Arrays



Wednesday, March 30, 16

There’s the memory for the array, then the references to other objects for each element around the heap.

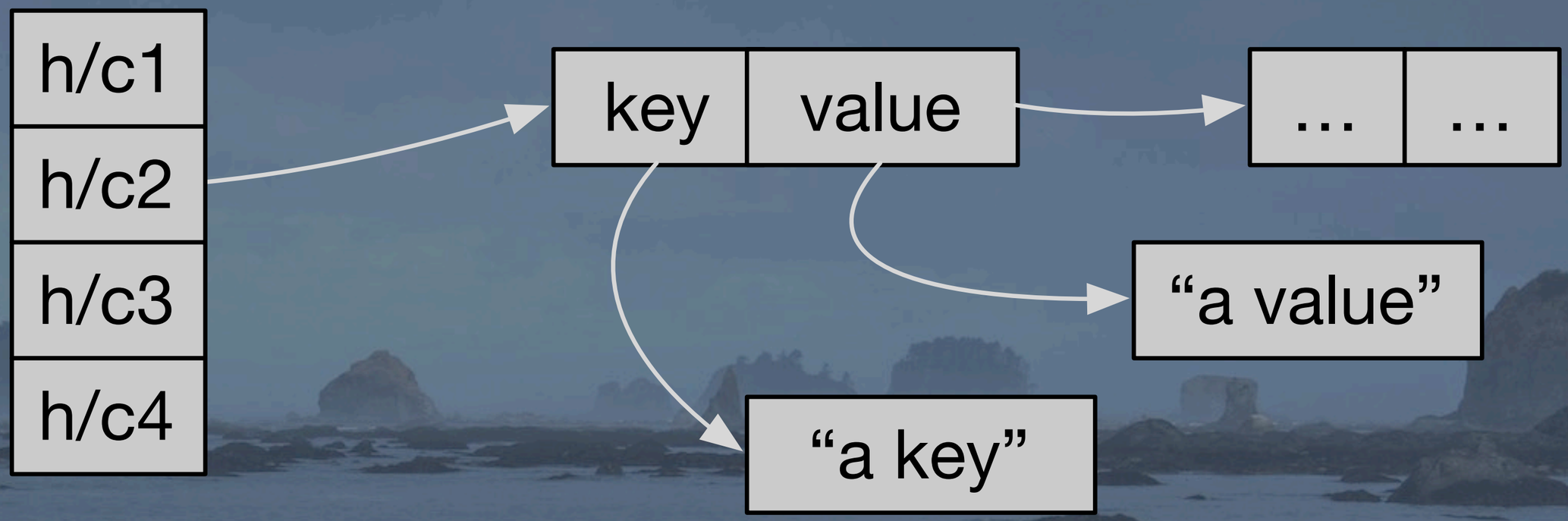
Class Instances



Wednesday, March 30, 16

An example of the type that might represent a record for a “persons” table.
There’s the memory for the array, then the references to other objects for each element around the heap.

Hash Maps



Wednesday, March 30, 16

For a given hash bucket, there’s the memory for the array, then the references to other objects for each element around the heap.

Performance?

Why obsess about this?
Spark jobs are CPU bound:

- Improve network I/O? ~2% better.
- Improve disk I/O? ~20% better.

Wednesday, March 30, 16

Okay, so memory handling is an issue, but is it the major issue? Aren't Big Data jobs I/O bound anyway? MapReduce jobs tend to be CPU bound, but research by Kay Ousterhout (http://www.eecs.berkeley.edu/~keo/talks/2015_06_15_SparkSummit_MakingSense.pdf) and other observers indicate that for Spark, optimizing I/O only improve performance ~5% for network improvements and ~20% for disk I/O. So, Spark jobs tend to be CPU bound.

What changed?

- Faster HW (compared to ~2000)
  - 10Gbs networks
  - SSDs.

Wednesday, March 30, 16

These are the contributing factors that make today's Spark jobs CPU bound while yesterday's MapReduce jobs were more I/O bound.

What changed?

- Smarter use of I/O
  - Pruning unneeded data sooner.
  - Caching more effectively.
  - Efficient formats, like Parquet.

Wednesday, March 30, 16

These are the contributing factors that make today's Spark jobs CPU bound while yesterday's MapReduce jobs were more I/O bound.
We also use compression more than we used to.

What changed?

- But more CPU use today:
  - More Serialization.
  - More Compression.
  - More Hashing (joins, group-bys).

Wednesday, March 30, 16

These are the contributing factors that make today's Spark jobs CPU bound while yesterday's MapReduce jobs were more I/O bound.

Performance?

To improve performance, we need to focus on the CPU, the:

- Better algorithms, sure.
- And optimize use of memory.

Wednesday, March 30, 16

Okay, so memory handling is an issue, but is it the major issue? Aren't Big Data jobs I/O bound anyway? MapReduce jobs tend to be CPU bound, but research by Kay Ousterhout (http://www.eecs.berkeley.edu/~keo/talks/2015_06_15_SparkSummit_MakingSense.pdf) and other observers indicate that for Spark, optimizing I/O only improve performance ~5% for network improvements and ~20% for disk I/O. So, Spark jobs tend to be CPU bound.

Project Tungsten

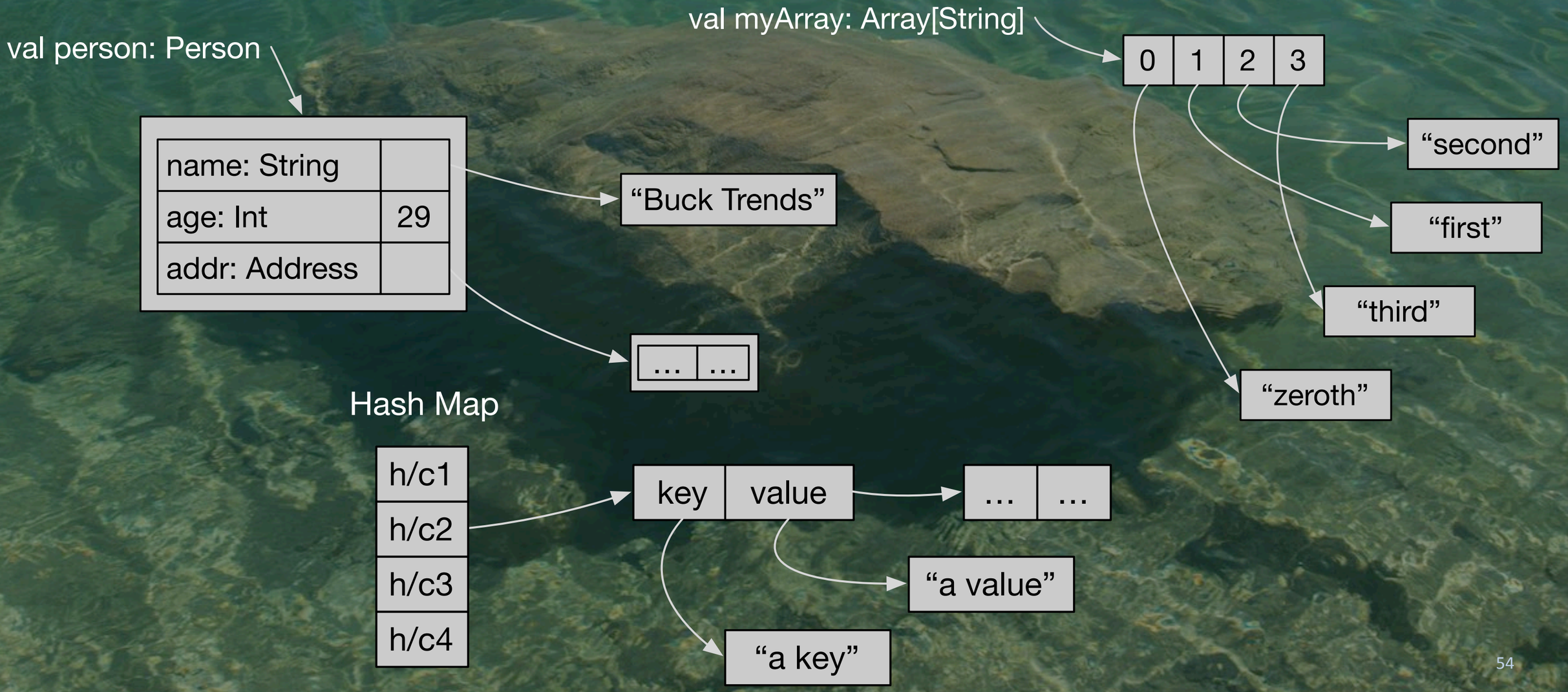
Initiative to greatly improve DataFrame performance.

Wednesday, March 30, 16

Project Tungsten is a multi-release initiative to improve Spark performance, focused mostly on the DataFrame implementation. References:
http://www.slideshare.net/databricks/2015-0616-spark-summit
http://www.slideshare.net/SparkSummit/deep-dive-into-project-tungsten-josh-rosen
https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

Goals

Reduce the # of References

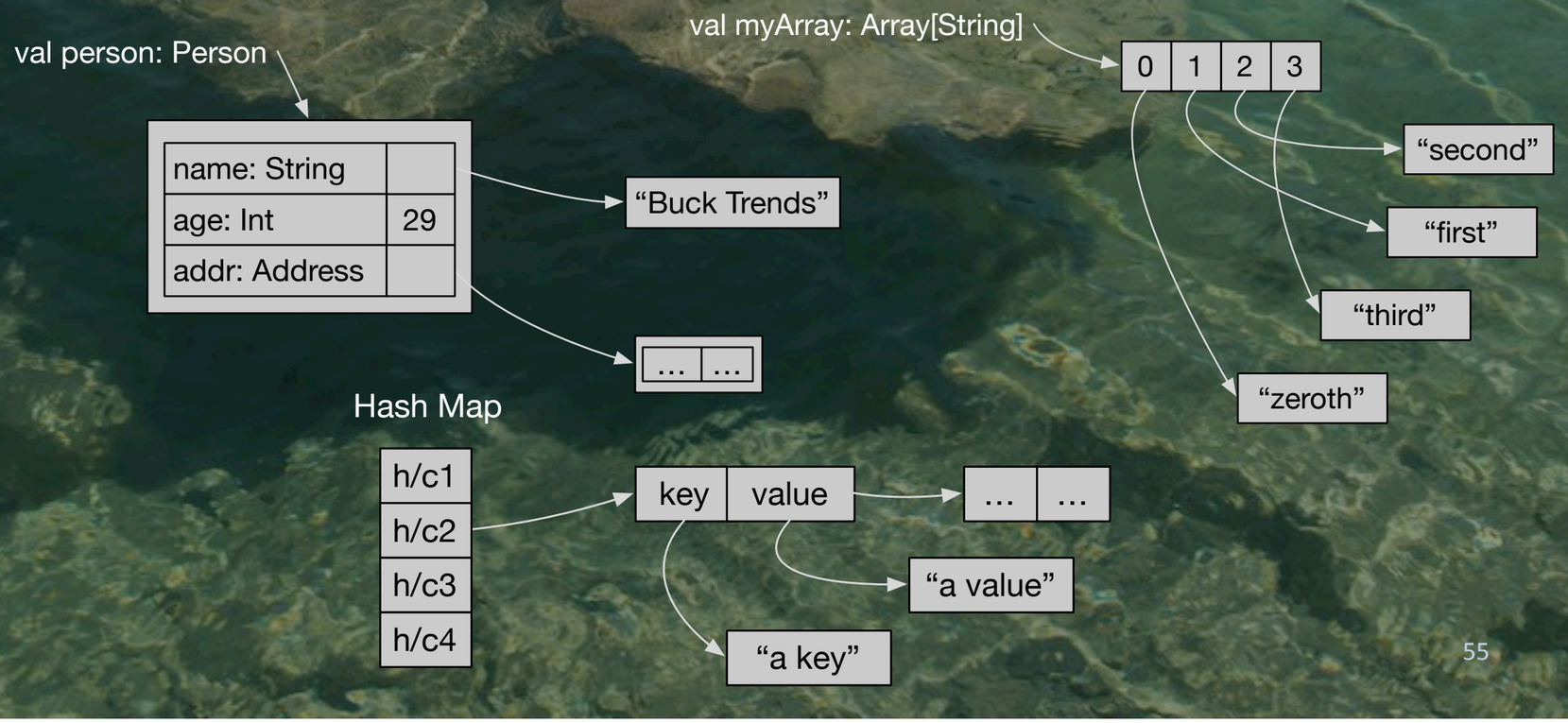


Wednesday, March 30, 16

While this general-purpose model has given us the flexibility we need, it’s bad for big data, where we have a lot of data with the same format, same types, etc.

Reduce the # of References

- Fewer, bigger objects to GC.
- Fewer cache misses



Wednesday, March 30, 16

Eliminating references (the arrows) means we have better GC performance, because we'll have far fewer (but larger) to manage and collect. The size doesn't matter; it's just as efficient to collect a 1MB block as a 1KB block.
With fewer arrows, it's also more likely that the data we need is already in the cache!

Less Expression Overhead

```
sql("SELECT a + b FROM table")
```

- Evaluating expressions billions of times:
  - Virtual function calls.
  - Boxing/unboxing.
  - Branching (if statements, etc.)

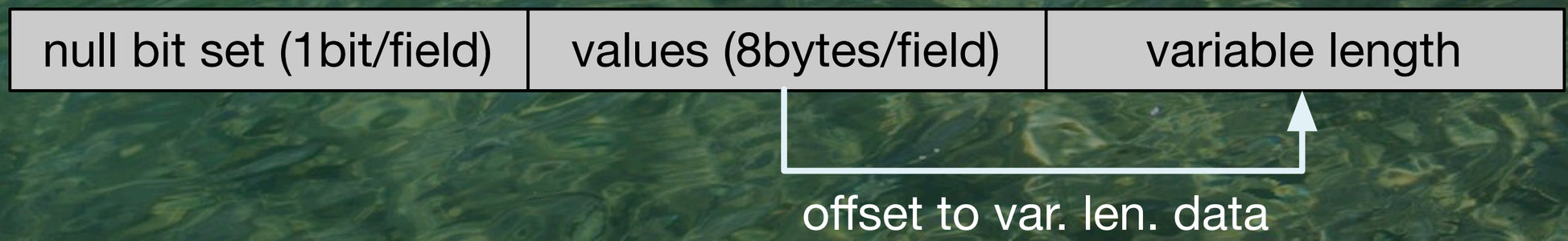
Wednesday, March 30, 16

This is perhaps less obvious, but when you evaluate an expression billions of times, then the overhead of virtual function calls (even with the JVMs optimizations for polymorphic dispatch), the boxing and unboxing of primitives, and the evaluate of conditions adds up to noticeable overhead.

Implementation

Object Encoding

New CompactRow type:

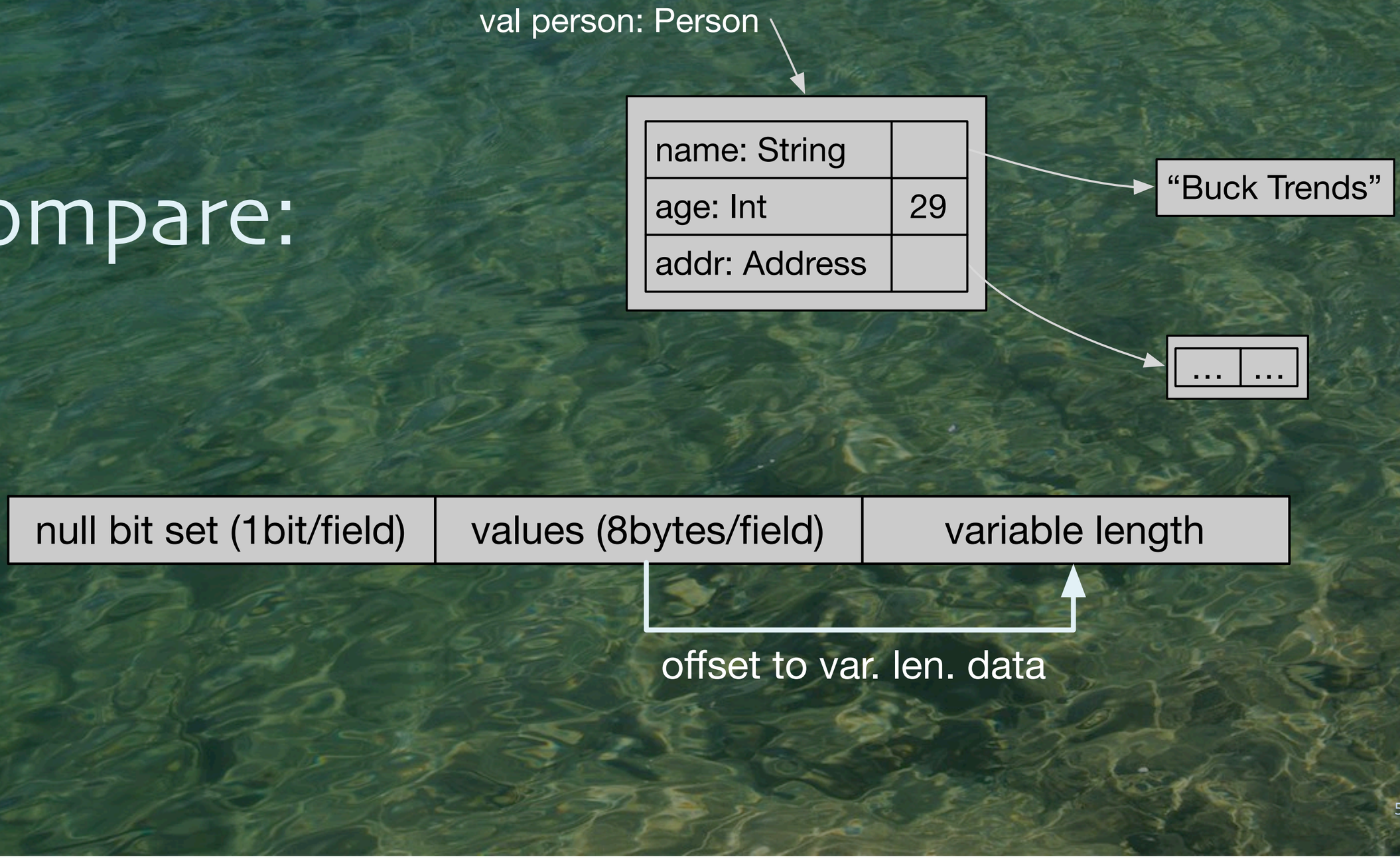


- Compute hashCode and equals on raw bytes.

Wednesday, March 30, 16

The new Compact Row format (for each record) uses a bit vector to mark values that are null, then packs the values together, each in 8 bytes. If a value is a fixed-size item and fits in 8 bytes, it's inlined. Otherwise, the 8 bytes holds a reference to a location in variable-length segment (e.g., strings).
Rows are 8-byte aligned. Hashing and equality can be done on the raw bytes.

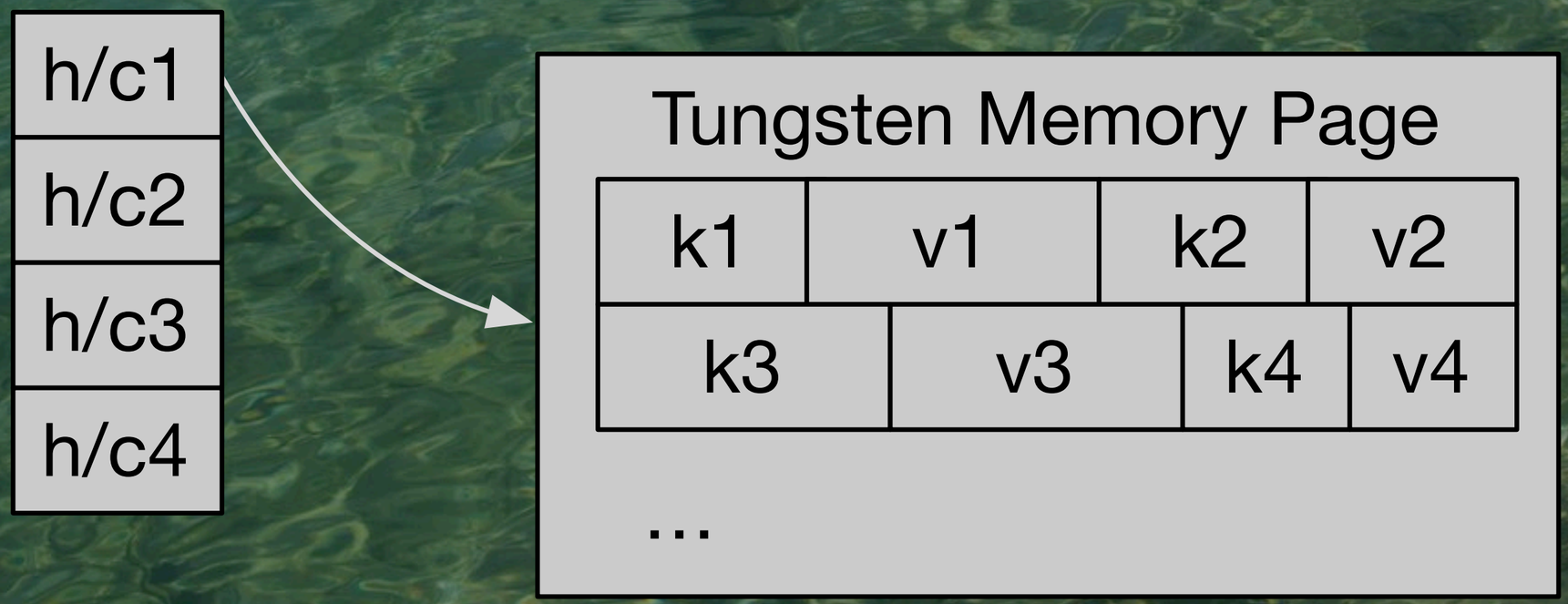
- Compare:



Wednesday, March 30, 16

An object spread over the heap vs. a single byte array encoding.

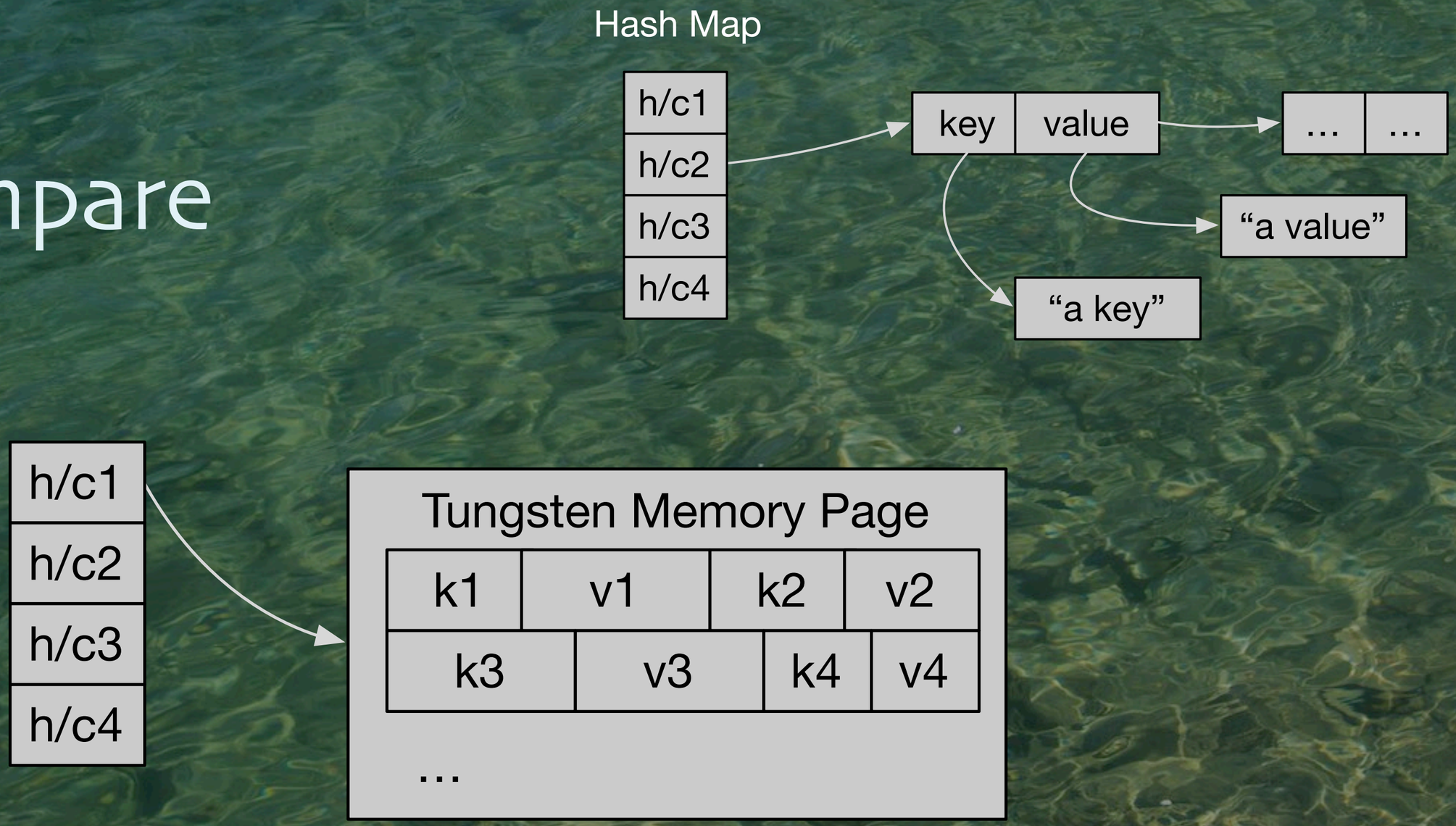
- BytesToBytesMap:



Wednesday, March 30, 16

BytesToBytesMap eliminates almost all the reference indirections. The memory pages are on or off heap, but managed by Tungsten using sun.misc.Unsafe.

- Compare



Wednesday, March 30, 16

Many fewer arrows!

Memory Management

- Some allocations off heap.
  - sun.misc.Unsafe.

Wednesday, March 30, 16

Unsafe allows you to manipulate memory directory, like in C/C++.

Less Expression Overhead

```
sql("SELECT a + b FROM table")
```

- Solution:
  - Generate custom byte code.

To eliminate virtual function calls, boxing overhead and branching, generate custom byte code for expressions.

Less Expression Overhead

```
sql("SELECT a + b FROM table")
```

- Solution:
  - Generate custom byte code.
  - (Spark 2.0 - custom code for whole queries.)

Wednesday, March 30, 16

To eliminate virtual function calls, boxing overhead and branching, generate custom byte code for expressions.

No Value Types (Planned for Java)

Wednesday, March 30, 16

Value types would let you write simple classes with a single primitive field, then the compiler doesn't heap allocate instances, but puts them on the stack, like primitives today. Under consideration for Java 10. Limited support already in the Scala compiler.

```
case class Timestamp(epochMillis: Long) {

  def toString: String = { ... }

  def add(delta: TimeDelta): Timestamp = {
    /* return new shifted time */
  }

  ...
}
```

Don’t allocate on the heap; just push the primitive long on the stack. (scalac does this now.)

Value types would let you write simple classes with a single primitive field, then the compiler doesn’t heap allocate instances, but puts them on the stack, like primitives today. Under consideration for Java 10. Limited support already in the Scala compiler.

Long operations aren't atomic

According to the JVM spec

Wednesday, March 30, 16

That is, even though longs are quite standard now and we routinely run 64bit CPUs (small devices the exception), long operations are not guaranteed to be atomic, unlike int operations.

No Unsigned Types

What's factorial(-1)?

Wednesday, March 30, 16

Back to issues with the JVM as a Big Data platform...
Unsigned types are very useful for many applications and scenarios. Not all integers need to be signed!
Arrays are limited to 2^(32-1) elements, rather than 2^(32)!

Arrays Indexed with Ints

Byte Arrays limited to 2GB!

Wednesday, March 30, 16

2GB is quite small in modern big data apps and servers now approaching TBs of memory! Why isn't it 4GB, because we *signed* ints, not *unsigned* ints!

```
scala> val N = 1100*1000*1000
N2: Int = 1100000000 // 1.1 billion

scala> val array = Array.fill[Short](N)(0)
array: Array[Short] = Array(0, 0, ...)

scala> import
org.apache.spark.util.SizeEstimator

scala> SizeEstimator.estimate(array)
res3: Long = 2200000016  // 2.2GB
```


One way this bites you is when you need to serialize a data structure larger than 2GB. Keep in mind that modern Spark heaps could be 10s of GB and 1TB of memory per server is coming on line!
Here's a real session that illustrates what can happen.

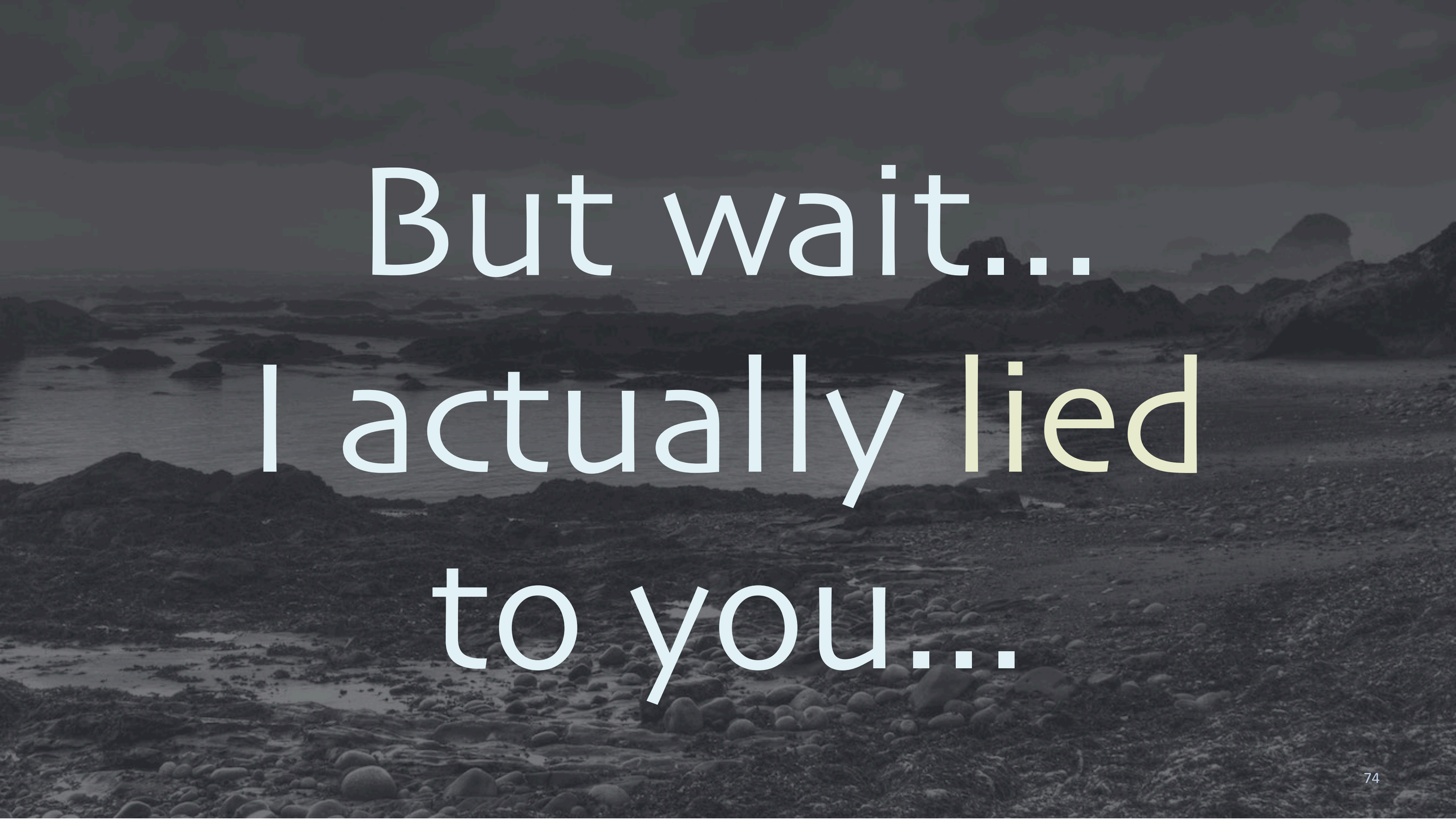
```
scala> val b = sc.broadcast(array)
...broadcast.Broadcast[Array[Short]] = ...

scala> SizeEstimator.estimate(b)
res0: Long = 2368

scala> sc.parallelize(0 until 100000).
     | map(i => b.value(i))
```


One way this bites you is when you need to serialize a data structure larger than 2GB. Keep in mind that modern Spark heaps could be 10s of GB and 1TB of memory per server is coming on line!
Here's a real session that illustrates what can happen.

```
scala> SizeEstimator.estimate(b)
res0: Long = 2368

scala> sc.parallelize(0 until 100000).
     | map(i => b.value(i))

java.lang.OutOfMemoryError:
        Requested array size exceeds VM limit
    at java.util.Arrays.copyOf(...)
    ...
```

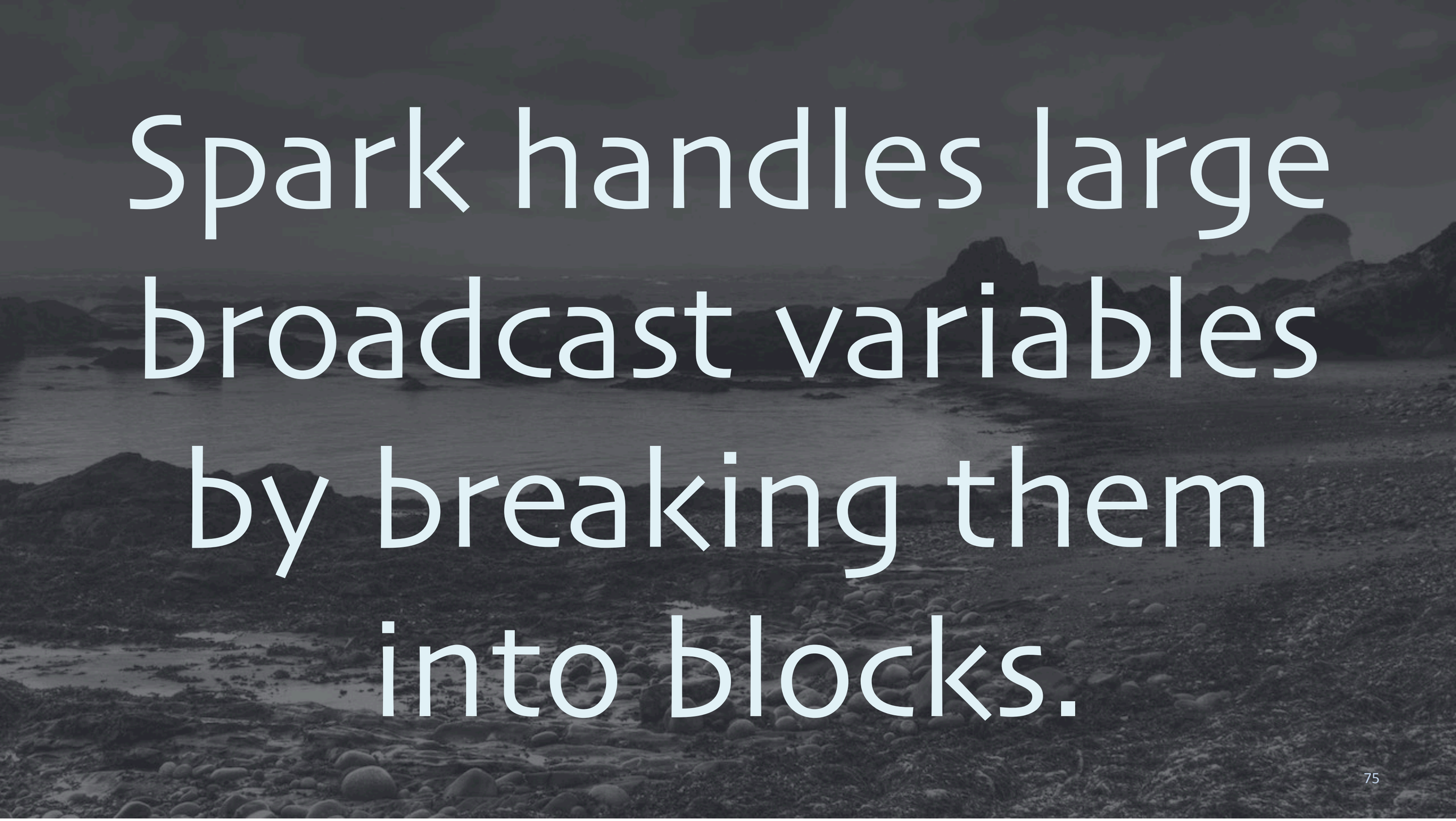
Boom!

Even though a 1B-element Short array is fine, as soon as you attempt to serialize it, it won't fit in a 2B-element Byte array. Our 2.2GB short array can't be serialized into a byte array, because it would take more than 2^31-1 bytes and we don't have that many available elements, due to (signed) integer indexing.



Wednesday, March 30, 16

I implied this is a problem with broadcast variables, but it’s not...



Wednesday, March 30, 16

Spark serializes and distributes smaller blocks for broadcast variables (and also when shuffling data between tasks), so it doesn't attempt to create an Array[Byte] that's larger than 2B elements. So what really happened?

Wednesday, March 30, 16

This is actually a limitation of the Scala REPL (“read, evaluate, print, loop” - i.e., the interpreter), which was never intended to support very large heaps. Let’s see what actually happened...

```
java.lang.OutOfMemoryError:
      Requested array size exceeds VM limit

  at java.util.Arrays.copyOf(...)
  ...
  at java.io.ByteArrayOutputStream.write(...)
  ...
  at java.io.ObjectOutputStream.writeObject(...)
  at ...spark.serializer.JavaSerializationStream
          .writeObject(...)
  ...
  at ...spark.util.ClosureCleaner$.ensureSerializable(..)
  ...
  at org.apache.spark.rdd.RDD.map(...)
```


This is actually a limitation of the Scala REPL (shell), which was never intended to support very large heaps. Let's see what actually happened...

```
java.lang.OutOfMemoryError:
      Requested array size exceeds VM limit

  at java.util.Arrays.copyOf(...)
  ...
  at java.io.ByteArrayOutputStream.write(...)
  ...
  at java.io.ObjectOutputStrea
  at ...spark.serializer.JavaS
          .writeObject(...)
  ...
  at ...spark.util.ClosureClea
  ...
  at org.apache.spark.rdd.RDD.map(...)
```

Pass this closure to RDD.map:

```
i => b.value(i)
```


When RDD.map is called, Spark verifies that the “closure” (anonymous function) that’s passed to it is clean, meaning it can be serialized. Note that it references “b” outside its body, i.e., it “closes over” b.

```
java.lang.OutOfMemoryError:
      Requested array size exceeds VM limit

  at java.util.Arrays.copyOf(...)
  ...
  at java.io.ByteArrayOutputSt
  ...
  at java.io.ObjectOutputStrea
  at ...spark.serializer.JavaS
          .writeObject(...)
  ...
  at ...spark.util.ClosureCleaner$.ensureSerializable(..)
  ...
  at org.apache.spark.rdd.RDD.map(...)
```

Verify that it's "clean" (serializable).

```
i => b.value(i)
```


Wednesday, March 30, 16

ClosureCleaner does this serialization check (among other things).

```
java.lang.OutOfMemoryError:
      Requested array size exceeds VM limit

  at java.util.Arrays.copyOf(...)
  ...
  at java.io.ByteArrayOutputStream.write(...)
  ...
  at java.io.ObjectOutputStream.writeObject(...)
  at ...spark.serializer.JavaSerializationStream
          .writeObject(...)
  ...
  at ...spark.util.ClosureClea
  ...
  at org.apache.spark.rdd.RDD.r
```

...which it does by serializing to a byte array...

ClosureCleaner does this serialization check (among other things).

```
java.lang.OutOfMemoryError:
      Requested array size exceeds VM limit

  at java.util.Arrays.copyOf(...)
  ...
  at java.io.ByteArrayOutputSt
  ...
  at java.io.ObjectOutputStrea
  at ...spark.serializer.JavaSe
          .writeObject(...)
  ...
  at ...spark.util.ClosureClea
```

...which requires copying an array...

What array???

```
i => b.value(i)
```

```
...
scala> val array = Array.fill[Short](N)(0)
...
```


ClosureCleaner does this serialization check (among other things).

Wednesday, March 30, 16

It’s because of the way Scala has to wrap your REPL expressions into objects...

- You write:

```
scala> val array = Array.fill[Short](N)(0)
scala> val b = sc.broadcast(array)
scala> sc.parallelize(0 until 100000).
     | map(i => b.value(i))
```


Wednesday, March 30, 16

I'm greatly simplifying the actual code that's generated. In fact, there's a nested object for every line of REPL code! The synthesized name $iwC is real.

```
scala> val array = Array.fill[Short](N)(0)
scala> val b = sc.broadcast(array)
scala> sc.parallelize(0 until 100000).
     | map(i => b.value(i))
```

- Scala compiles:

```
class $iwC extends Serializable {
 val array = Array.fill[Short](N)(0)
 val b = sc.broadcast(array)

 class $iwC extends Serializable {
   sc.parallelize(...).map(i => b.value(i))
  }
}
```


Wednesday, March 30, 16

I'm greatly simplifying the actual code that's generated. In fact, there's a nested object for every line of REPL code! The synthesized name $iwC is real.

```
scala> val array = Array.fill[Short](N)(0)
scala> val b = sc.broadcast(array)
scala> sc.parallelize(0 until 100000).
     | ma
```

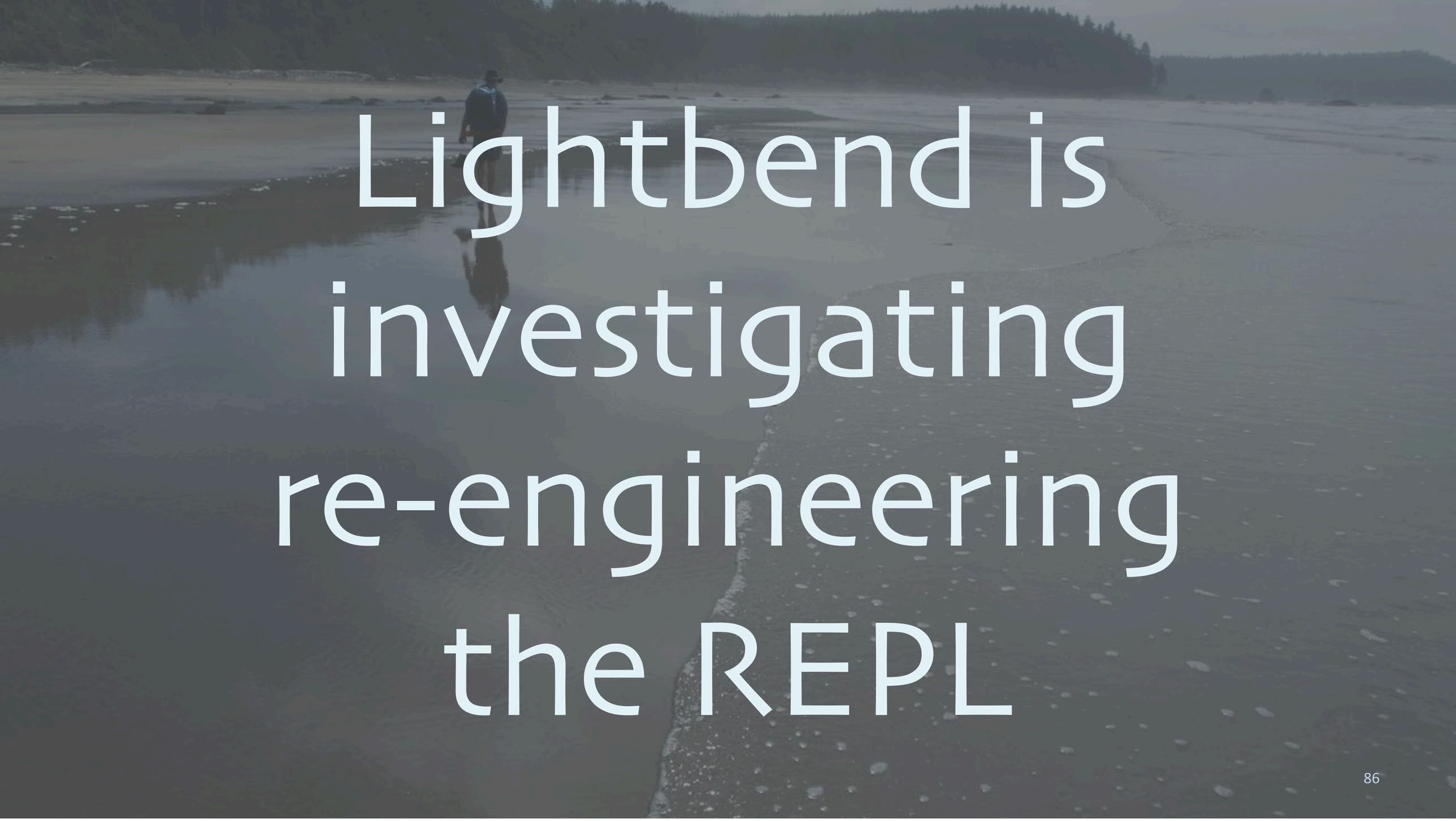
- Scala compiles:

... sucks in all this!

```
class $iwC extends Serializable {
  val array = Array.fill[Short](N)(0)
  val b = sc.broadcast(array)

  class $iwC extends Se
    sc.parallelize(...).map(i => b.value(i))
  }
}
```

So, this closure over “b”...

I’m greatly simplifying the actual code that’s generated. In fact, there’s a nested object for every line of REPL code!
The synthesized name $iwC is real.
Note that “array” and “b” become fields in these classes.



Wednesday, March 30, 16

We’re looking into design changes that would better support Big Data scenarios and avoid these issue.


Workarounds:
Use @transient,
local vals,
and objects

- Transient is often all you need:

```
scala> @transient val array =
     |   Array.fill[Short](N)(0)
scala> ...
```


Wednesday, March 30, 16

This is enough in this example because when serializing over those generated objects, the array will be skipped.

```
object Data {   // Encapsulate in objects!
  val N = 1100*1000*1000
  val array = Array.fill[Short](N)(0)
  val getB = sc.broadcast(array)
}
object Work {
  def run(): Unit = {
    val b = Data.getB // local ref!
    val rdd = sc.parallelize(...).
      map(i => b.value(i)) // only needs b
    rdd.take(10).foreach(println)
}}
```


It's not shown, but you could paste this into the REPL. A more robust approach. You don't need to wrap everything in objects. "Work" is probably not needed, but the local variable "b" is very valuable to grabbing just what you need inside an object and not serialiazing the whole "Data.

Conclusions


Spark Is Driving
Scala Adoption

Spark has some technical debt.

Wednesday, March 30, 16

The Spark project is adding features rapidly and accumulating technical debt. This is a threat to its long term success. Test coverage also needs to be better.

Scala collections need a refresh.

spark-summit.org/eu-2015/events/spark-the-ultimate-scala-collections/

Implementation details leak in Scala's collections API; it could more cleanly separate implementation from abstraction. Bringing some of Spark's operations, like joins, and distributed processing would benefit all Scala projects. Martin Odersky has said publicly that Scala's collections API could adopt extensions in Spark.
In fact, a rewrite is planned for Scala 2.13, the next major version (after the current one in RC status).

```
✔ grep 'def.*ByKey' ~/projects/spark/spark-git/core/src/main/scala/org/apache/spark/rdd/PairRDD
def combineByKey[C](createCombiner: V => C,
def combineByKey[C](createCombiner: V => C,
def aggregateByKey[U: ClassTag](zeroValue: U, partitioner: Partitioner)(seqOp: (U, V) => U,
def aggregateByKey[U: ClassTag](zeroValue: U, numPartitions: Int)(seqOp: (U, V) => U,
def aggregateByKey[U: ClassTag](zeroValue: U)(seqOp: (U, V) => U,
def foldByKey(
def foldByKey(zeroValue: V, numPartitions: Int)(func: (V, V) => V): RDD[(K, V)] = self.withScop
def foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)] = self.withScope {
def sampleByKey(withReplacement: Boolean,
def sampleByKeyExact(
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = self.withScope {
def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)] = self.withScope {
def reduceByKey(func: (V, V) => V): RDD[(K, V)] = self.withScope {
def reduceByKeyLocally(func: (V, V) => V): Map[K, V] = self.withScope {
def reduceByKeyToDriver(func: (V, V) => V): Map[K, V] = self.withScope {
def countByKey(): Map[K, Long] = self.withScope {
def countByKeyApprox(timeout: Long, confidence: Double = 0.95)
def countApproxDistinctByKey(
def countApproxDistinctByKey(
def countApproxDistinctByKey(
def countApproxDistinctByKey(relativeSD: Double = 0.05): RDD[(K, Long)] = self.withScope {
def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])] = self.withScope {
def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])] = self.withScope {
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) =>
def groupByKey(): RDD[(K, Iterable[V])] = self.withScope {
def subtractByKey[W: ClassTag](other: RDD[(K, W)]): RDD[(K, V)] = self.withScope {
def subtractByKey[W: ClassTag](
def subtractByKey[W: ClassTag](other: RDD[(K, W)], p: Partitioner): RDD[(K, V)] = self.withScop
```

Wednesday, March 30, 16

Here's a subset of possibilities, "by key" functions that assume a schema of (key,value).

Scala should adopt some Tungsten optimizations...

Wednesday, March 30, 16

More specifically, Scala could adopt Tungsten optimizations, such as the optimized Record and HashMap types and off-heap memory management.

… & could the JVM adopt Tungsten's object encoding?

Wednesday, March 30, 16

I'm not sure this is possible, but it would be great. Perhaps "Record Types"?

The JVM needs long indexing, value types & unsigned types

Wednesday, March 30, 16

We discussed these three limitations of the JVM today.

Office hour: 1100-1140 tomorrow.

Wednesday, March 30, 16

You can find this talk and an older version of it with more details at polyglotprogramming.com/talks

Wednesday, March 30, 16

You can find this talk and an older version of it with more details at polyglotprogramming.com/talks

lightbend.com/fast-data

dean.wampler@lightbend.com
@deanwampler

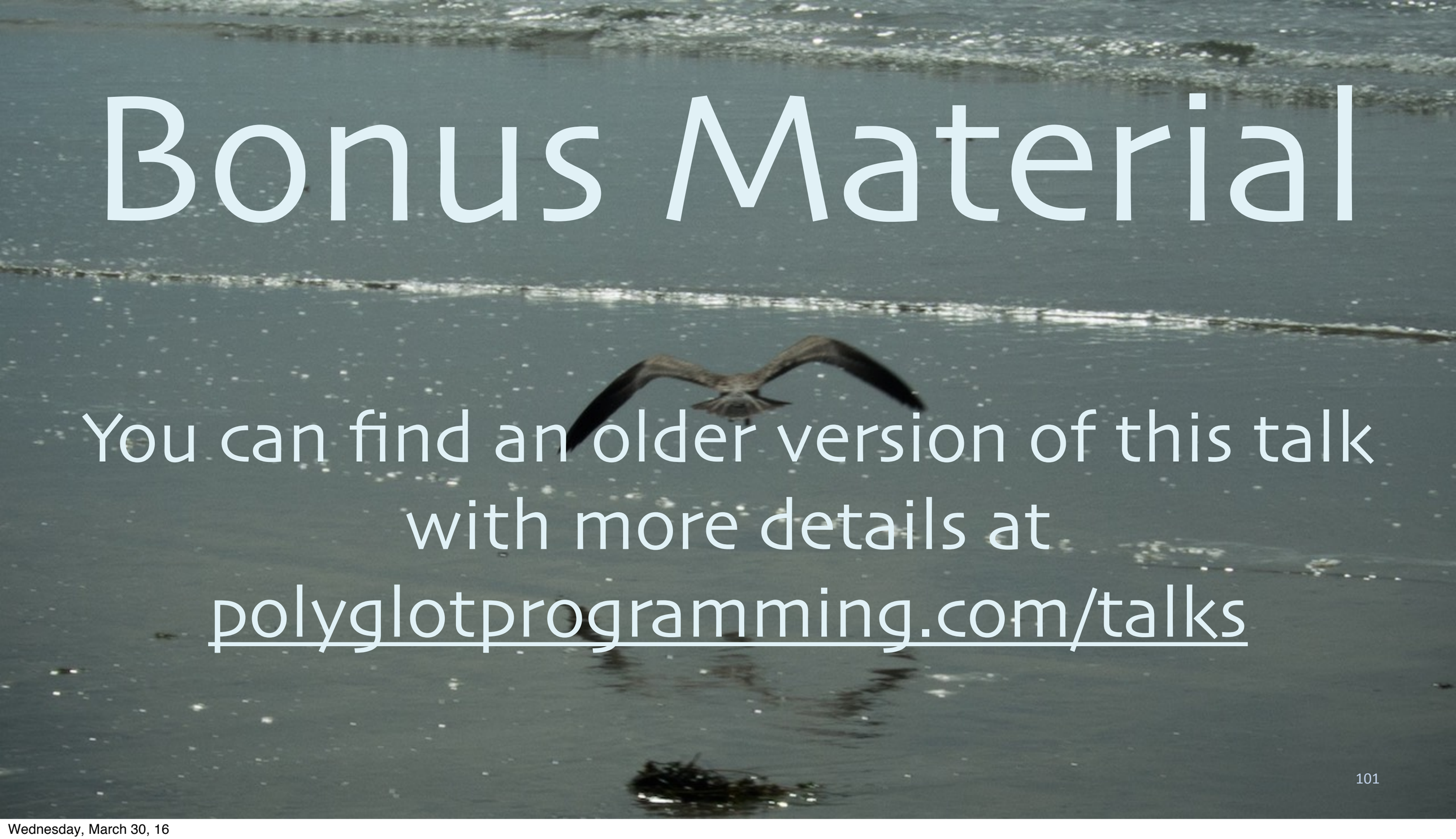
Thank You!

Lightbend

Wednesday, March 30, 16

I've thought a lot about the evolution of big data to more stream-oriented "fast data". Please follow this link for a whitepaper to learn more.

Bonus Material



You can find an older version of this talk with more details at

polyglotprogramming.com/talks