Scala and the JVM for Big Data: Lessons from Spark

YOW!
December, 2015

Typesafe

©Dean Wampler 2014-2016, All Rights Reserved

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

All photos are copyright (C) 2014-2016, Dean Wampler, except where otherwise noted. All Rights Reserved.

You can find this and my other talks here.

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

A Distributed Computing Engine for the JVM

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 Dataset

The core abstraction in the core of Spark.

- Your data set is held in an RDD and partitioned over a cluster.
- If a partition is lost, it's reconstructed.

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..

Productivity?

Very concise, elegant, functional APIs.

- Scala, Java
- Python, R
- ... and SQL!

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

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

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.

Example: Inverted Index

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

Web Crawl

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

Compute Inverted Index

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

Compute 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))
}.flatMap {
 case (id, contents) =>
  toWords(contents).map(w => ((w,id),1))
```


Here is an actual Spark program, written as a script.
Start with the imports. The SparkContext is the entry point for all programs.

```
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))
}.flatMap {
 case (id, contents) =>
  toWords(contents).map(w => ((w,id),1))
```


Create a SparkContext, specifying the “master” (local for single core on the local machine, “local[*]” for all cores, “mesos://...” for Mesos, “yarn-*” for YARN, etc.)

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

val sparkCo
 SparkContex
sparkContext.textFile("/path/to/input").
map { line =>
 val array = line.split(",", 2)
 (array(0), array(1))
}.flatMap {
 case (id,
  toWords(contents).map(w => ((w,id),1))
```

RDD[String]: .../hadoop, Hadoop provides...

RDD[(String,String)]: (.../hadoop,Hadoop provides...)

Using the sparkContext, read test data (the web crawl data). Assume it's comma-delimited and split it into the identifier (e.g., URL) and the contents. If the input is very large, multiple, parallel tasks will be spawned across the cluster to read and process each file block as a partition, all in parallel.

```
  val array = line.split(", ", 2)
  (array(0), array(1))
}.flatMap {
  case (id, contents) =>
    toWords(contents).map(w => ((w,id),1))
}.reduceByKey(_ + _).
map {
  case ((word,id),n) => (word,(id,n))
}.groupByKey.
mapValues {
  seq => sortByCount(seq)
}.saveAsTextFile("/path/to/output")
```

RDD[((String,String),Int)]: ((Hadoop,.../hadoop),1)

An idiom for counting...
Flat map to tokenize the contents into words (using a “toWords” function that isn’t shown), then map over those words to nested tuples, where the (word, id) is the key, and see count of 1 is the value.
Then use reduceByKey, which is an optimized groupBy where we don’t need the groups, we just want to apply a function to reduce the values for each unique key. Now the records are unique (word,id) pairs and counts >= 1.

```
 val array = line.split(", ", 2)
 (array(0), array(1))
}.flatMap {
 case (id, contents) =>
  toWords(contents).map(w => ((w,id),1))
}.reduceByKey(_ + _).
map {
 case ((word,id),n) => (word,(id,n))
}.groupByKey.
mapValues {
 seq => sortBy(seq)
}.saveAsTextFile("/path/to/output")
```

RDD[(String,Iterable((String,Int))]: (Hadoop,seq(.../hadoop,1),...))

I love how simple this line is anon. function is; by moving the nested parentheses, we setup the final data set, where the words alone are keys and the values are (path, count) pairs. Then we group over the words (the “keys”).

```
 val array = line.split(", ", 2)
 (array(0), array(1))
}.flatMap {
 case (id, contents) =>
  toWords(contents).map(w => ((w,id),1))
}.reduceByKey(_ + _).
map {
 case ((word,id),n) => (word,(id,n))
}.group
mapValues {
 seq => sortByCount(seq)
}.saveAsTextFile("/path/to/output")
```

RDD[(String,Iterable((String,Int))]: (Hadoop,seq(.../hadoop,1),...))

Finally, for each unique word, sort the nested collection of (path,count) pairs by count descending (this is easy with Scala's Seq.sortBy method). Then save the results as text files.

Altogether

```
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))
}.flatMap {
 case (id, contents) =>
  toWords(contents).map(w => ((w,id),1))
}.reduceByKey(_ + _).
map {
 case ((word,id),n) => (word,(id,n))
}.groupByKey.
mapValues {
 seq => sortByCount(seq)
}.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.

Performance?

Lazy API, inspired by Scala collections.

- Dataflow DAG of steps.
- Distributed over a cluster.

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.

Performance?

Lazy API, inspired by Scala collections.

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

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

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.

DStreams: Spark Streaming

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.

Streams can run a very long time. Lots of problems can arise, such as consumers getting backed up or producers producing more and more stuff...

What about backpressure in Spark? In v1.5, Typesafe contributed a dynamic backpressure mechanism for flow control.

This is what can happen without backpressure. If the rate of incoming data increases to the point where each minibatch job can't finish before the next batch is ready, then it backs up and will eventually exhaust the heap. In this case, the rate of incoming data dropped again, causing the red peak.

This is what can happen with backpressure. The producer of data is told to slow down (eg., the connection to Kafka pulls fewer events/interval). The algorithm uses previous intervals data to adjust the flow rate.

There's a standard for addressing this problem, called Reactive Streams, which will be part of Java 9. reactive-streams.org. It's mechanism is out-of-band messaging between the consume and the producer to control the flow rate. Hence, it implements backpressure, but also allows the consumers and producers of different stream implementations to interoperate.

Typesafe is contributing an RS-compliant consumer implementation to Spark, hopefully in the 1.6 or 1.7 release.

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).

DataFrames:

- Wraps RDDs.
- Records with known schemas.
- Uses a query planner/optimizer.
- Provides relational-like operations.
- Inspired by Python/R DataFrames.

The underlying collection type is the DataFrame, which wraps an RDD. The query optimizer is an engine called “Catalyst”.
Because Scala is flexible for embedded DSLs, the Scala API looks very similar to the original Python API.

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
```


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 a DataFrame, which wraps an RDD.

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


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)
```


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.

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.

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.

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.

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

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?

Tools and Libraries

Akka
Breeze
Algebird
Spire & Cats
Axle
...

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.

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.

Big Data Ecosystem

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/

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

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 heaps 10s-100s of GB.
  - Less common in "generic" (non-data) services.

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.
  - Potentially 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

- Garbage First GC (G1 - JVM 1.6).
  - Balance latency and throughput.
  - More flexible mem. region mgmt.

Image: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html
Much better for Spark it turns out...

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

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

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.
  - 20 bytes for array overhead.
  - 8 bytes for UTF16 chars.
  - 8 bytes for hash code.

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

Arrays

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

Class Instances

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

Hash Maps

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?

Spark jobs are CPU bound:

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

So, we need to look elsewhere...

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.

Why?

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

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

Why?

- Smarter use of I/O
  - Prune unneeded data sooner.
  - Use caching more effectively.
  - Better data formats, like Parquet.

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

Why?

- But more CPU use:
  - Serialization.
  - Compression.
  - Hashing.

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

Can We Fix these Issues?

Recall the DataFrame API...

We discussed DataFrames already. There is a lot of work happening in their implementation to address the problems we just outlined.

Performance

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

This is a major step forward. Previously for Hadoop, Data Scientists often developed models in Python or R, then an engineering team ported them to Java MapReduce. Previously with Spark, you got good performance from Python code, but about 1/2 the efficiency of corresponding Scala code. Now, the performance is the same.
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 more limited set of abstractions than the more general RDD API. This narrower interface enables broader (more aggressive) optimizations and other implementation choices underneath.

```
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 col(colName: Column): DataFrame = {
def map[R: ClassTag](f: Row => R): RDD[R] = {
def flatMap[R: ClassTag](f: Row => Traversable[R]): RDD[R] ={
def foreach(f:
def take(n: Int
def count(): Lo
def distinct():
def agg(exprs:                                  {
```

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

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

Recall our previous DataFrame example, where the “anonymous function” passed to the join method specifying the join criteria was actually an instance of a Column type that we constructed using a mini-DSL.

Project Tungsten

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

Initiative to greatly improve performance of DataFrames.

Project Tungsten

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

Four Problems

- Java object model overhead suboptimal for data.
- GC expensive.
- Minimize CPU cache misses.
- Minimize function overhead.

The default behaviors for these things work fine for general purposes, but not for the special needs of large-scale data systems.

Fewer References?

Bad for cache utilization.

Fewer (Bigger) Objects to GC?

Too much small garbage.

Fewer Cache Misses?

Too much small garbage.

Less Expression Overhead?

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

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

Project Tungsten

Project Tungsten tries to improve in all these areas using three lines of attack...

Project Tungsten

- Custom memory management and object layouts.
- Cache-aware algorithms and data structures.
- Code gen. for optimal performance.

Three areas of focus.

Object Layout

- Off (using sun.misc.Unsafe) and on Heap.
- New CompactRow type:

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

Object Layout

- BytesToBytesMap:

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

Tungsten

- These data structures:
  - Improve cache hits.
  - Reduce GC drastically, because objects are now more coarse-grained, with fewer of them.

Tungsten

- Byte code generation for expressions to eliminate:
  - Boxing/unboxing.
  - Virtual method call overhead.
  - Branching.

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


Their test results indicate ~3x improvement in performance, very close to hand-written, optimized code!

No Unsigned Types

What's factorial(-1)?

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!

2GB is quite small in modern big data apps and servers now approaching TBs of memory!

No Value Types

Planned for Java

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.

Python and R have a longer history as Data Science languages, so they have much richer and more mature libraries for Data Science, e.g., statistics, machine learning, natural language processing, etc.

Okay, all things considered, the JVM is a great platform for Big Data.
Scala has emerged as the de facto “data engineering” language, as opposed to data science, where Python, R, SAS, Matlab, and Mathematica still dominate (although Scala has its passionate advocates here, too.)

Pragmatic OOP + FP

While some people hate the fact that Scala embraces OOP (and I know that you know that I know who you are...), this is pragmatic to make Scala accessible to Java developers and it provides a convenient, familiar modularity tool. However, data science is the killer app for FP, IMHO, so the core problem needs to use FP, which Scala enables nicely.

- Abstractions vs. implementations:
  - Pure functional abstractions?
  - Mutable implementations, when necessary.
  - Performance is critical.

Raw performance is extremely important for competitive data computation systems. Providing ~pure, high-level abstractions with well-defined semantics, enabling flexibility in the implementations to exploit mutability for performance (but hopefully not prematurely applied).

Scala Big Data Sandwich

Image: https://en.wikipedia.org/wiki/Hamburger#/media/File:NCI_Visuals_Food_Hamburger.jpg

Scala Big Data Sandwich

scopes

Objects as Modules

Functional APIs

Optimized (Mutable?) Code

Image: https://en.wikipedia.org/wiki/Hamburger#/media/File:NCI_Visuals_Food_Hamburger.jpg

Interactive exploration is a powerful tool for exploring data, algorithms, and analysis approaches. Data scientists are familiar with notebook metaphors, such as iPython. Now Spark Notebook brings this way of using the REPL to Scala and Spark, providing a richer experience the REPL or IDE alone.

Collections API

Collections API a natural foundation for data flows.

* Powerful, yet concise.
* Abstracts over implementation - can be parallelized, distributed.
* Generalizes to streaming or a fixed collection.

Inspired Spark's API

Spark's API looks very similar to the "conceptual" abstractions of Scala's collections API, i.e., what you see in the simplified method signatures shown in the Scaladoc, as opposed to the actual signatures with the extra type parameters, CanBuildFrom implicit, etc.
Arguably, Spark's API offers a cleaner separation between interface and implementation.

```
.map { line =>
  val array = line.split(",", 2)
  (array(0), array(1))
}.flatMap {
  case (id, contents) => toWords(contents)
}.reduceByKey {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path,
.groupByKey
.map {
  case (word, list) => (word, sortByCount(
}.saveAsTextFile("/path/to/output")
```

Dataflow and query abstractions (that mostly don't leak)

Spark does a good job exposing a reasonably intuitive model of dataflow and query abstractions so developers can focus on the problems at hand without a lot of ceremony. It's not completely leak free. There are implementation concerns developers have to think about, such as when to explicitly cache results in memory for subsequent reuse. There are far too may configuration properties, too.

Claim

Spark does fine without the I/O Monad.

While the purists amongst us might prefer that all state mutation be explicitly encapsulated in monads, in fact the “informal” approach of Spark makes it approachable by a wide class of developers and the boundaries between lazy “transformations” and side-effecting “actions”, while not explicitly obvious, become so once you gain some experience with the API.

Are Java 8 Lambdas Enough?

Spark is untenable in Java 7. The verbosity of anonymous inner classes started a flight to Scala. Then Java 8 added lambdas. Do they make Java good enough? No...

Tuples

First, Tuples are very useful for writing concise code, as we saw in the previous example.

Pattern matching makes it easy to extract fields in records, match strings with regular expressions, test types, etc.

Type Inference

Type inference removes the tedium of explicit typing, as in Java, while providing type safety and useful feedback in interactive sessions.

Expressive DSLs are relatively easy to create in Scala. The collections API itself is a DSL of sorts for data flows. Another simple example is the ability to define your own mathematical types, like vectors, matrices, complex numbers, etc. and operators for them, like +, -, *, and / that you use as if the types were built in. We'll another Spark DSL shortly.

Finally, the REPL and Notebook UIs for it are essential for exploring data and experimenting with algorithms.

Conclusions

Spark Is Driving Scala Adoption

Spark could be more pure.

Spark could learn from FP enthusiasts in the Scala community. The Spark code base quality could be made better with a more rigorous application of pure FP, including specifying the semantics of immutable behaviors of public APIs.

Spark has too much technical debt.

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.

Spark Streaming is complex and needs to mature.

Mini-batch processing is actually a complex beast to implement robustly. A lot of progress has been made already, but it's far from "perfection".

Scala should more cleanly separate interface from implementation in the collections API.

Going the other way, implementation details leak in Scala's collections API. The separation of implementation from abstraction could be cleaner. Lazy evaluation is very appealing, because of the possibility of materializing the code optimally on demand, more or less like Spark is forced to do.

Scala could adopt some RDD operations.

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

Going the other way, Martin has even said publicly that Scala's collections API could adopt extensions in Spark.

```
✔ 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
```

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

Scala should adopt some Tungsten optimizations.

Going the other way, Scala could also adopt some Tungsten optimizations, such as the optimized Record and HashMap types.

The JVM needs long indexing, value & unsigned types

We discussed these three limitations of the JVM today.

Could the JVM adopt Tungsten's object layouts?

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

We at Typesafe think the future of Spark is bright. We also think the next generation cluster management framework, Mesos, is the future of Big Data platforms, while also supporting cluster management for all your applications, databases, web services, etc., etc. That's why Typesafe is investing in this infrastructure and now offering commercial support for it.

typesafe.com/reactive-big-data
polyglotprogramming.com/talks
dean.wampler@typesafe.com