Scala and the JVM for Big Data: Lessons from Spark

Typesafe

©Dean Wampler 2014-2015, All Rights Reserved

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

All photos are copyright (C) 2015, Dean Wampler, except where otherwise noted. All Rights Reserved.

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

Tuesday, September 15, 15

Let's start at the bottom. Why the JVM?

Tools and Libraries

Akka
Breeze
Algebird
Spire
...

We have a rich suite of core tools for building data applications...

Tuesday, September 15, 15

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 for OSS and startups..

But it's
not perfect...

Garbage Collection

Pauses

Object
Overhead

No Unsigned
Types

Richer data libs. in Python & R

Java OOP Thinking!

Why Scala?

Pragmatic OOP + FP

REPL & Notebooks

Spark Notebook | Magic Plots

localhost:9000/notebooks/viz/Magic%20Plots.snb#

SPARK NOTEBOOK Magic Plots (unsaved changes)

File Edit View Insert Cell Kernel Help | Scala [2.11.6] Spark [1.5.0] Hadoop [2.2.0] {Hive ✓} {Parquet ✓}

Code | Cell Toolbar: None

Plot them all!

In [14]:
```
widgets.pairs(list)
```

```
res9: scala.xml.Elem =
<table class="table" style="width: 600px">
    <thead><tr><th>a</th><th>b</th><th>c</th><th>d</th></tr></thead>
    <tbody><tr><td><div>
      <script data-this="{&quot;dataId&quot;:&quot;anonc5e491684fb8b82707472efb5a8c5673&quo
t;,&quot;dataInit&quot;:[{&quot;a&quot;:&quot;b&quot;},{&quot;a&quot;:&quot;a&quot;},{&quot;a&q
uot;:&quot;b&quot;},{&quot;a&quot;:&quot;a&quot;},{&quot;a&quot;:&quot;c&quot;}],&quot;genId&qu
ot;:&quot;270004360&quot;}" type="text/x-scoped-javascript">/*<![CDATA[*/req(['../javascripts/n
otebook/playground','../javascripts/notebook/magic/tableChart'],
      function(playground, _magictableChart) {
        // data ==> data-this (in observable.js's scopedEval) ==> this in JS => { dataId, dataI
nit, ... }
        // this ==> scope (in observable.js...
```

Out[14]:

a	b	c	d
100 items (Out of 100 items, only the 5 first items are shown)	100 items (Out of 100 items, only the 25 first items are shown)	100 items (Out of 100 items, only the 25 first items are shown)	100 items (Out of 100 items, only the 5 first items are shown)

a	a
b	b
a	a

a	d
b	true
a	true
b	false

Collections API

Basis of Spark

Collections API a natural foundation for data flows.

* Powerful, yet concise.
* Abstracts over implementation - can be parallelized, distributed.
* Generalizes to streams.

Example:

Inverted Index

Tuesday, September 15, 15

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

wikipedia.org/hadoop

Hadoop provides
MapReduce and HDFS

...

wikipedia.org/hbase

HBase stores data in HDFS

...

inverse index

block

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

block

...	...

block

...	...

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

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

```scala
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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


Here is an actual Spark program, written as a script.

```
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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


Start with 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))
}.reduceByKey {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


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 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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


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.

```
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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


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.

```
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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


reduceByKey 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.

```
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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


I love how simple this line 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.

```scala
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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


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 {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```

```
.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, n
.groupByKey
.map {
  case (word, list) => (word, sortByCount(l
}.saveAsTextFile("/path/to/output")
```

Beautiful, concise operators

Tuesday, September 15, 15

Beautiful “combinators”.

Tuples

Java 8 has lambdas, which improve enormously over Java 7. Is this all we need? No. First, Tuples are very useful, ...

... pattern matching makes it easy to work with fields in records, ...

Type Inference

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

Expressive DSLs are relatively easy to create in Java.

Spark

Productivity?

Very concise, elegant, functional APIs.

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

Tuesday, September 15, 15

We saw an example why this true.
While Spark was written in Scala, it has a Java and Python API, too, and an R API is almost released.

Productivity?

Interactive shell (REPL)

- Scala, Python, R, and SQL

Spark Notebook (iPython/Jupyter-like)

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.

Resilient Distributed Datasets

Tuesday, September 15, 15

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.

DStreams: Spark Streaming

Tuesday, September 15, 15

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.

Lambda Architecture

This reuse of code between batch and mini-batch is very handy for the Lambda Architecture.

Flexible for Algorithms

Composable operators, performance optimizations, and general flexibility are foundation for higher-level APIs...

Tuesday, September 15, 15

A major step forward. Due to the lightweight nature of Spark processing, it can efficiently support a wider class of algorithms.

Tuesday, September 15, 15

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).

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.

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

Performance

Builds a dataflow DAG:

- Combines steps into "stages"
- Can cache intermediate data

How is Spark more efficient? As we'll see, Spark programs are actually "lazy" dataflows definitions that are only evaluated on demand. Because Spark has this directed acyclic graph of steps, it knows what data to attempt to cache in memory between steps (with programmable tweaks) and it can combine many logical steps into one "stage" of computation, for efficient execution while still providing an intuitive API experience.

Inverted 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 =>
}.reduceByKey {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```

Inverted 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 =>
}.reduceByKey {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```


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.

Performance

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

Tuesday, September 15, 15

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.

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

Initiative to greatly improve performance of DataFrames.

Project Tungsten

Tuesday, September 15, 15

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

Observations

Spark jobs are CPU bound:

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

So, we need to look elsewhere...

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?

- HW: 10Gbs networks, SSDs.
- Smart pruning of unneeded data.
- Effective use of caching in Spark.
- Better data formats, like Parquet.
- Serialization, compression, and hashing are CPU intensive.

First, the general-purpose object structure and memory management in the JVM is suboptimal for data-centric applications with huge datasets. So, Spark

Why?

- Java object model overhead suboptimal for data.
- GC expensive.
- Cache awareness important.

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

Java Objects?

- "abcd": 4 bytes for raw UTF8.
- 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

Java Objects?

- Case classes, tuples, & Spark's Row type used for "records", but their reference indirections add lots of overhead.
- ... and lead to cache misses.

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

Case Classes

Tuesday, September 15, 15

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

Arrays

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

Tuesday, September 15, 15

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

GC Challenges

- Typical heaps 10s-100s of GB.
  - Uncommon in generic services.
  - Too many cached RDDs leads to huge old generation sections.
  - Puts extreme pressure on GC.

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.

Expensive Logic

- Overhead evaluating expressions, like in SQL queries:
  - Virtual function calls.
  - Object creation from boxing.
  - Branching (if statements, etc.)

Tungsten

Tuesday, September 15, 15

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

Tungsten

- Custom memory management.
  - Rather than Java objects.
- Cache-aware algorithms and data structures.
- Code gen for optimal performance.

Three areas of focus.

sun.misc.Unsafe

- Off and On Heap.
- Compact Row:

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.

sun.misc.Unsafe

- 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.

Three areas of focus.

Tungsten

- Code generation for expression evaluation is used to eliminate:
  - Boxing/Unboxing.
  - Virtual method call overhead.
  - Branching

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

Lastly, let's talk about streams again.

Recall:

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

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...

Necessary for flow control. In Spark 1.5, Typesafe contributed a dynamic backpressure mechanism. (Blog post coming...) Going forward, we hope to contribute a Reactive Streams-compliant consumer.

Necessary for flow control. In Spark 1.5, Typesafe contributed a dynamic backpressure mechanism. (Blog post coming...) Going forward, we hope to contribute a Reactive Streams-compliant consumer.

Spark Is Driving Scala Adoption

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