Learning Apache Spark by processing email

Markus Dale

2015

Talk Overview

(Spark Mail Tutorial Dale et al. 2015)

Speaker Background

Hadoop Ecosystem

- Based on Google GFS (2003)/MapReduce (2004) papers
- Extremely rich and robust
    - $\sim$ 2005 Nutch/2006 Yahoo - Doug Cutting/Mike Cafarella
- HDFS/Hadoop MapReduce
- DSLs: Pig, Cascading/Scalding, Crunch, Hive (SQL)
- Graph processing: Giraph
- Real-time streaming: Storm
- Machine Learning: Apache Mahout ...

Hadoop Challenges

- With rich ecosystem: installation, maintenance, cognitive load for each add-on framework
- MapReduce is batch only - no interactive shell
- Must write out to disk between each iteration
- No memory caching yet (Apache Tez working on complex DAGs of tasks)
- Hadoop MapReduce programming is very low-level
  - map phase - (internal shuffle/sort) - reduce phase
  - Progammer expresses logic in map/reduce

Why Apache Spark?

- Different trade-offs
  - Improved hardware (faster processors, more memory)
- High-level, scalable processing framework (programmer productivity)
- Iterative algorithms
- Interactive data exploration (Spark shell)

Apache Spark Unified Large Scale Processing System

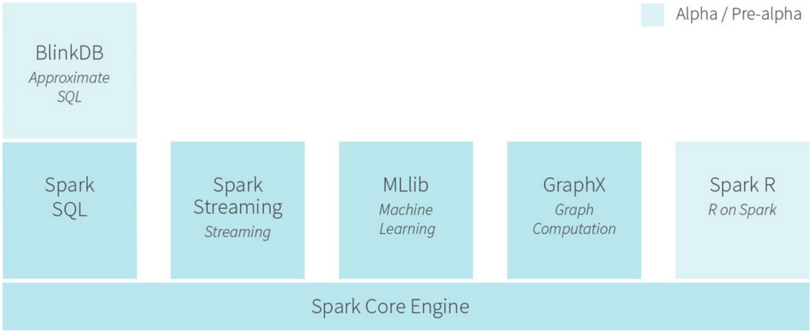

Figure : Databricks Spark Ecosystem (2015)

Spark Resilient Distributed Dataset (RDD)

- Treat distributed, immutable data set as a collection
- Resilient: Use RDD lineage to recompute failed partitions
- Two forms of RDD operations:
  - Transformations (applied lazily - optimized evaluation)
  - Actions (cause transformations to be executed)
- Scala, Java, Python APIs (Spark R coming)
  - Rich combinator functions on RDD abstraction

Exploration: Combinator functions on Scala collections

- Examples: map, flatMap, filter, reduce, fold, aggregate
- We will disregard type variance (covariance, contravariance) because RDD is invariant.
- Background - Combinatory logic, higher-order functions...

Combinatory Logic

Moses Schönfinkel and Haskell Curry in the 1920s

> *[C]ombinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments (Combinatory Logic Wikipedia 2014)*

- Higher-order function: Function that takes function as argument or returns function

map

- applies a given function to every element of a collection
- returns collection of output of that function (one per original element)
- input argument - same type as collection type
- return type - can be any type

map - Scala

```scala
def computeLength(w: String): Int = w.length

val words = List("when", "shall", "we", "three",
  "meet", "again")
val lengths = words.map(computeLength)

> lengths  : List[Int] = List(4, 5, 2, 5, 4, 5)
```

map - Scala syntactic sugar

```scala
//anonymous function (specifying input arg type)
val list2 = words.map((w: String) => w.length)

//let compiler infer arguments type
val list3 = words.map(w => w.length)

//use positionally matched argument
val list4 = words.map(_.length)
```

map - ScalaDoc

See immutable List ScalaDoc

```
List[A]
...
final def map[B](f: (A) => B): List[B]
```

- Builds a new collection by applying a function to all elements of this list.
- B - the element type of the returned collection (can be same as A or different)
- f - the function to apply to each element.
- returns - a new list resulting from applying the given function f to each element of this list and collecting the results.

flatMap

- ScalaDoc:

```
List[A]
...
def flatMap[B](f: (A) =>
          GenTraversableOnce[B]): List[B]
```

- GenTraversableOnce - List, Array, Option...
- can be empty collection or None
- flatMap takes each element in the GenTraversableOnce and puts it in order to output List[B]
- removes inner nesting - flattens
- output list can be smaller or empty (if intermediates were empty)

flatMap Example

```
val macbeth = """When shall we three meet again?
|In thunder, lightning, or in rain?""".stripMargin
val macLines = macbeth.split("\n")
// macLines: Array[String] = Array(
  When shall we three meet again?,
  In thunder, lightning, or in rain?)

//Non-word character split
val macWordsNested: Array[Array[String]] =
      macLines.map{line => line.split("""\W+""")}
//Array(Array(When, shall, we, three, meet, again),
//      Array(In, thunder, lightning, or, in, rain))

val macWords: Array[String] =
     macLines.flatMap{line => line.split("""\W+""")}
//Array(When, shall, we, three, meet, again, In,
//      thunder, lightning, or, in, rain)
```

filter

```
List[A]
...
def filter(p: (A) => Boolean): List[A]
```

- selects all elements of this list which satisfy a predicate.
- returns - a new list consisting of all elements of this list that satisfy the given predicate p. The order of the elements is preserved.

filter Example

```
val macWordsLower = macWords.map{_.toLowerCase}
//Array(when, shall, we, three, meet, again, in, thunder,
//       lightning, or, in, rain)

val stopWords = List("in","it","let","no","or","the")
val withoutStopWords =
  macWordsLower.filter(word => !stopWords.contains(word))
// Array(when, shall, we, three, meet, again, thunder,
//        lightning, rain)
```

reduce

```
List[A]
...
def reduce[A](op: (A, A) => A): A
```

- Creates one cumulative value using the specified associative binary operator.
- op - A binary operator that must be associative.
- returns - The result of applying op between all the elements if the list is nonempty. Result is same type as list type.
- UnsupportedOperationException if this list is empty.

reduce Example

```scala
//beware of overflow if using default Int!
val numberOfAttachments: List[Long] =
  List(0, 3, 4, 1, 5)
val totalAttachments =
  numberOfAttachments.reduce((x, y) => x + y)
//Order unspecified/non-deterministic, but one
//execution could be:
//0 + 3 = 3, 3 + 4 = 7,
//7 + 1 = 8, 8 + 5 = 13

val emptyList: List[Long] = Nil
//UnsupportedOperationException
emptyList.reduce((x, y) => x + y)
```

fold

```
List[A]
...
def fold[A](z: A)(op: (A, A) => A): A
```

- Very similar to reduce but takes start value z (a neutral value, e.g. 0 for addition, 1 for multiplication, Nil for list concatenation)
- returns start value z for empty list
- Note: See also foldLeft/Right (return completely different type)

  ```
  foldLeft[B](z: B)(f: (B, A)   B): B
  ```

fold Example

```
val numbers = List(1, 4, 5, 7, 8, 11)
val evenCount = numbers.fold(0) { (count, currVal) =>
  println(s"Count: $count, value: $currVal")
  if (currVal % 2 == 0) {
    count + 1
  } else {
    count
  }
}
Count: 0, value: 1
Count: 0, value: 4
Count: 1, value: 5
Count: 1, value: 7
Count: 1, value: 8
Count: 2, value: 11
evenCount: Int = 2
```

aggregate

```
List[A]
...
def aggregate[B](z: B)(seqop: (B, A) => B,
                       combop: (B, B) => B): B
```

- More general than fold or reduce. Can return different result type.
- Apply seqop function to each partition of data.
- Then apply combop function to combine all the results of seqop.
- On a normal immutable list this is just a foldLeft with seqop (but on a parallelized list both operations are called).

aggregate Example

```
val wordsAll = List("when", "shall", "we", "three",
  "meet", "again", "in", "thunder", "lightning",
  "or", "in", "rain")
//Map(5 letter words ->3, 9->1, 2->4, 7->1, 4->3)
val lengthDistro = wordsAll.aggregate(Map[Int, Int]())(
  seqop = (distMap, currWord) =>
  {
    val length = currWord.length()
    val newCount = distMap.getOrElse(length, 0) + 1
    val newKv = (length, newCount)
    distMap + newKv
  },
  combop = (distMap1, distMap2) => {
    distMap1 ++ distMap2.map {
      case (k, v) =>
      (k, v + distMap1.getOrElse(k, 0))
    }
  })
```

So what does this have to do with Apache Spark?

- Resilient Distributed Dataset (RDD)
- From API docs: "immutable, partitioned collection of elements that can be operated on in parallel"
- map, flatMap, filter, reduce, fold, aggregate...

com.uebercomputing.analytics.basic.BasicRddFunctions

```
//compiler can infer bodiesRdd type - reader clarity
val bodiesRdd: RDD[String] =
  analyticInput.mailRecordRdd.map { record =>
  record.getBody
}
val bodyLinesRdd: RDD[String] =
  bodiesRdd.flatMap { body => body.split("\n") }
val bodyWordsRdd: RDD[String] =
  bodyLinesRdd.flatMap { line => line.split("""\W+""") }
val stopWords = List("in", "it", "let", "no", "or", "the")
val wordsRdd = bodyWordsRdd.filter(!stopWords.contains(_))

//Lazy eval all transforms so far - now action!
println(s"There were ${wordsRdd.count()} words.")
```

Spark - RDD API

- RDD API
- Transforms - map, flatMap, filter, reduce, fold, aggregate...
  - Lazy evaluation (not evaluated until action! Optimizations)
- Actions - count, collect, first, take, saveAsTextFile...

Spark - From RDD to PairRDDFunctions

- If an RDD contains tuples (K,V) - can apply PairRDDFunctions
- Uses implicit conversion of RDD to PairRDDFunctions
- In 1.3 conversion is defined in RDD singleton object
- In 1.2 and previous versions available by importing org.apache.spark.SparkContext._

```
From 1.3.0 org.apache.spark.rdd.RDD (object):

implicit def rddToPairRDDFunctions[K, V](rdd: RDD[(K, V)])
(implicit kt: ClassTag[K], vt: ClassTag[V],
  ord: Ordering[K] = null): PairRDDFunctions[K, V] = {
  new PairRDDFunctions(rdd)
}
```

PairRDDFunctions

- keys, values - return RDD of keys/values
- mapValues - transform each value with a given function
- flatMapValues - flatMap each value (0, 1 or more output per value)
- groupByKey - RDD[(K, Iterable[V])]
  - Note: expensive for aggregation/sum - use reduce/aggregateByKey!
- reduceByKey - return same type as value type
- foldByKey - zero/neutral starting value
- aggregateByKey - can return different type
- lookup - retrieve all values for a given key
- join (left/rightOuterJoin), cogroup ...

From RDD to DoubleRDDFunctions

- From API docs: "Extra functions available on RDDs of Doubles through an implicit conversion."
- mean, stddev, stats (count, mean, stddev, min, max)
- sum
- histogram ...

MailRecord

- We want to analyze email data
- Started with Enron email dataset from Carnegie Mellon University
  - Nested directories for each user/folder/subfolder
  - Emails as text files with headers (To, From, Subject...)
  - over 500,000 files (= 500,000 splits for FileInputFormat)
- Don't want our analytic code to worry about parsing

Solution: Create Avro record format, parse once, store (MailRecord)

Apache Avro

- JSON - need to encode binary data
- Hadoop Writable - Java centric
- Apache Avro
  - Binary serialization framework created by Doug Cutting in 2009 (Hadoop, Lucene)
  - Language bindings for: Java, Scala, C, C++, C#, Python, Ruby
  - Schema in file - can use generic or specific processing

(Apache Avro Cutting 2009)

Avro Container File

- Contains many individual Avro records ($\sim$ SequenceFile)
- Schema for each record at the beginning of file
- Supports compression
- Files can be split

Avro Schema for MailRecord

```
record MailRecord {
  string uuid;
  string from;
  union{null, array<string>} to = null;
  union{null, array<string>} cc = null;
  union{null, array<string>} bcc = null;
  long dateUtcEpoch;
  string subject;
  union{null, map<string>} mailFields = null;
  string body;
  union{null, array<Attachment>} attachments = null;
}
```

Avro Schema for Attachment

```
record Attachment {
  string fileName;
  int size;
  string mimeType;
  bytes data;
}
```

com.uebercomputing.mailrecord.MailRecord

- Avro Maven plugin translates schema into Java source code
- spark-mail/mailrecord
  - src/main/avro/
    - com/uebercomputing/mailrecord/MailRecord.avdl ->
  - src/main/java
    - com/uebercomputing/mailrecord/MailRecord.java

MailRecord.java

```
//Autogenerated by Avro DO NOT EDIT DIRECTLY
package com.uebercomputing.mailrecord;

public class MailRecord extends
   org.apache.avro.specific.SpecificRecordBase...
   public java.lang.String getFrom() {
     return from;
   }
   public java.lang.String getBody() {
     return body;
   }
   public List<Attachment> getAttachments() {
     return attachments;
   }
}
```

Converting emails to Avro

- See spark-mail/README.md
- spark-mail/PstProcessing.md

for details on how to go from Enron/PST files to Avro.

Apache Spark execution environments

- Local, standalone process (can be started command line or Eclipse)
- Spark Standalone Cluster (master/workers - http://spark.apache.org/docs/1.3.0/spark-standalone.html)
- Mesos resource manager http://spark.apache.org/docs/1.3.0/running-on-mesos.html
- Hadoop YARN resource manager http://spark.apache.org/docs/1.3.0/running-on-yarn.html

Running Spark

- Command line interactive shell environment (spark-shell)
- Submit job (spark-submit)

Both methods can be used in all execution environments.

Some Spark command arguments

```
spark-shell --help
```

- --master MASTER - e.g. yarn or local.
- --driver-memory MEM - Memory for driver (e.g. 1000M, 2G) (Default: 512M)
- --executor-memory MEM - Memory per executor (e.g. 1000M, 2G) (Default: 1G).
- --jars JARS - Comma-separated list of local jars for driver and executor classpaths.
- --conf PROP=VALUE Arbitrary Spark configuration property.
- --properties-file FILE Path for extra properties. If not specified, conf/spark-defaults.conf.

Spark Serialization

- Default - Java Serialization (java.io.ObjectOutputStream). Classes must implement java.io.Serializable otherwise:

  ```
  java.io.NotSerializableException:
     ...
       at java.io.ObjectOutputStream.writeObject0
     (ObjectOutputStream.java:1183)
  ```

- Better: Kryo "significantly faster and more compact than Java serialization (often as much as 10x)"

com.uebercomputing.mailrecord.MailRecordRegistrator

```scala
import org.apache.spark.serializer.KryoRegistrator
import com.esotericsoftware.kryo.Kryo
import com.twitter.chill.avro.AvroSerializer

//Uses Twitter's chill-avro library.
class MailRecordRegistrator extends KryoRegistrator {

  def registerClasses(kryo: Kryo): Unit = {
    kryo.register(classOf[MailRecord],
      AvroSerializer.
      SpecificRecordBinarySerializer[MailRecord])
  }
}
```

Spark Kryo Configurations

- spark.serializer - org.apache.spark.serializer.KryoSerializer
- spark.kryo.registrator
- spark.kryoserializer.buffer.mb
- spark.kryoserializer.buffer.max.mb

Kryo configurations

From command line:

```
--conf spark.serializer=\
org.apache.spark.serializer.KryoSerializer \
--conf spark.kryo.registrator=\
com.uebercomputing.mailrecord.MailRecordRegistrator \
--conf spark.kryoserializer.buffer.mb=128 \
--conf spark.kryoserializer.buffer.max.mb=512 \
```

Kryo configuration properties file

spark-mail/mailrecord-utils/mailrecord.conf

```
spark.serializer=org...serializer.KryoSerializer
spark.kryo.registrator=com...MailRecordRegistrator
spark.kryoserializer.buffer.mb=128
spark.kryoserializer.buffer.max.mb=512
```

Starting Spark interactive exploration

From spark-mail directory:

```
spark-shell --master local[4] --driver-memory 4G \
--executor-memory 4G \
--jars mailrecord-utils/target/mailrecord-*-shaded.jar \
--properties-file mailrecord-utils/mailrecord.conf \
--driver-java-options \
 "-Dlog4j.configuration=log4j.properties"
```

Getting an RDD of MailRecords

With spark-mail utilities:

```
import com.uebercomputing.mailrecord._
import com.uebercomputing.mailrecord.Implicits._

val args =
  Array("--avroMailInput",
        "/opt/rpm1/enron/filemail.avro")
val config =
  CommandLineOptionsParser.getConfigOpt(args).get
val recordsRdd =
  MailRecordAnalytic.getMailRecordRdd(sc, config)
```

Under the Hood - newAPIHadoopRDD in SparkContext

com.uebercomputing.mailrecord.MailRecordAnalytic.scala

```
val sparkHadoopConf = sc.hadoopConfiguration
hadoopConf.addResource(sparkHadoopConf)
hadoopConf.setBoolean(
  FileInputFormat.INPUT_DIR_RECURSIVE, true)
val mailRecordsAvroRdd =
  sc.newAPIHadoopFile(config.avroMailInput,
  classOf[MailRecordInputFormat],
  classOf[AvroKey[MailRecord]],
  classOf[FileSplit], hadoopConf)
```

mailrecord-utils - MailRecordInputFormat.scala

```
class MailRecordInputFormat extends
   FileInputFormat[AvroKey[MailRecord], FileSplit]
...
class MailRecordRecordReader(val readerSchema: Schema,
   val fileSplit: FileSplit) extends
     AvroRecordReaderBase
```

Hadoop InputFormats - Minimize object creation!

- WARNING: Hadoop InputFormats generally reuse the key/value objects
- Same with AvroRecordReaderBase in MailRecordInputFormat
- Generally, not a problem if you just map out the fields you need (getFrom etc.)
- However, if you want to cache the whole MailRecord you need to copy the original:

```
val mailRecordsRdd = mailRecordsAvroRdd.map {
  case (mailRecordAvroKey, fileSplit) =>
    val mailRecord = mailRecordAvroKey.datum()
    //make a copy - MailRecord gets reused!!!
    MailRecord.newBuilder(mailRecord).build()
  }
```

Analytic 1 - Mail Folder Statistics

- What are the least/most/average number of folders per user?
- Each MailRecord has user name and folder name

```
lay-k/        <- mailFields(UserName)
   business   <- mailFields(FolderName)
   family
   enron
   inbox
   ...
```

Hadoop Mail Folder Stats - Mapper

- read each mail record
- emits key: userName, value: folderName for each email

Hadoop Mail Folder Stats - Reducer

- reduce method
  - create set from values for a given key (unique folder names per user)
  - set.size == folder count
  - keep adding up all set.size (totalNumberOfFolders)
  - one up counter for each key (totalUsers)
  - keep track of min/max count
- cleanup method
  - compute average for this partition: totalNumberOfFolders/totalUsers
  - write out min, max, totalNumberOfFolders, totalUsers, avgPerPartition

Hadoop Mail Folder Stats - Driver

- Set Input/OutputFormat
- Number of reducers

Hadoop Mail Folder Stats - Results

- if only one reducer - results are overall lowest/highest/avg
- if multiple reducers
  - post-processing overall lowest/highest
  - add totalNumberOfFolders and totalUsers to compute overall average

Hadoop Mapper

```java
public void map(AvroKey<MailRecord> key,
NullWritable value, Context context) throws ... {
  MailRecord mailRecord = key.datum();
  Map<CharSequence, CharSequence> mailFields =
      mailRecord.getMailFields();
  CharSequence userName =
      mailFields.get(AvroMailMessageProcessor.USER_NAME);
  CharSequence folderName =
      mailFields.get(AvroMailMessageProcessor.FOLDER_NAME);
  userKey.set(userName.toString());
  folderValue.set(folderName.toString());
  context.write(userKey, folderValue);
}
```

Hadoop Reducer

```
public void reduce(Text userKey,
  Iterable<Text> folderValues,
  Context context) throws ... {
  Set<String> uniqueFolders = new HashSet<String>();
  for (Text folder : folderValues) {
    uniqueFolders.add(folder.toString());
  }
  int count = uniqueFolder.size();
  if (count > maxCount) maxCount = count;
  if (count < minCount) minCount = count;
  totalNumberOfFolder += count
  totalUsers++
}
...
public void cleanup...
//write min, max, totalNumberOfFolders,
//totalUsers, avgPerPartition
```

Spark Mail Folder Stats

- Create (user,folder) tuple for each email
- Aggregate by key (PairRDDFunctions)- for each key, create set of folders (distinct)
- Map values for each key (set) to the set's size:
  - (String, Int) represents (userName, # of folders for that user)
- Create an RDD from just the values (folder sizes for all users)
- Gather statistics on values (DoubleRDDFunction) (count, min, max, mean, stddev)
- Create a histogram (DoubleRDDFunction)

Spark - Creating an RDD of 2-Tuples via flatMap

```
val userFolderTuplesRdd: RDD[(String, String)] =
  analyticInput.mailRecordsRdd.flatMap {
    mailRecord =>
  val userNameOpt =
    mailRecord.getMailFieldOpt(UserName)
  val folderNameOpt =
    mailRecord.getMailFieldOpt(FolderName)

  if (userNameOpt.isDefined &&
      folderNameOpt.isDefined) {
    Some((userNameOpt.get, folderNameOpt.get))
    } else {
      None
    }
  }

userFolderTuplesRdd.cache()
```

Spark - applying PairRDDFunctions

```scala
//pre Spark 1.3.0: import org.apache.spark.SparkContext._
import scala.collection.mutable.{ Set => MutableSet }
...
//mutable set - reduce object creation/garbage collection
val uniqueFoldersByUserRdd:
 RDD[(String, MutableSet[String])] =
   userFolderTuplesRdd.aggregateByKey(
     MutableSet[String]())(
    seqOp = (folderSet, folder) => folderSet + folder,
    combOp = (set1, set2) => set1 ++ set2)

val folderPerUserRddExact: RDD[(String, Int)] =
   uniqueFoldersByUserRdd.mapValues { set => set.size }
```

DoubleRDDFunctions - Stats

```
val folderCounts: RDD[Int] =
   folderPerUserRddExact.values

val stats = folderCounts.stats()
> stats: org.apache.spark.util.StatCounter =
(count: 150, mean: 22.033333, stdev: 26.773474,
 max: 193.000000, min: 2.000000)

//buckets 0-25, 25-50 etc.
val buckets = Array(0.0,25,50,75,100,125,150,175,200)
folderCounts.histogram(buckets, evenBuckets=true)
res13: Array[Long] = Array(116, 16, 11, 3, 2, 1, 0, 1)
```

Who has 193 folders?

- RDD - def max()(implicit ord: Ordering[T]): T

```
folderPerUserRddExact.max()(
  Ordering.by(tuple => tuple._2))
> res2: (String, Int) = (kean-s,193)
```

RDD Lineage - transformations

```
folderCounts.toDebugString
> res18: String =
(22) MappedRDD[27] at values at <console>:35 []
|   MappedValuesRDD[26] at mapValues at <console>:33 []
|   ShuffledRDD[25] at aggregateByKey at <console>:31 []
+-(22) FlatMappedRDD[2] at flatMap at <console>:26 []
|       CachedPartitions: 22; MemorySize: 76.3 MB;
         TachyonSize: 0.0 B; DiskSize: 0.0 B
|   MappedRDD[1] at map at MailRecordAnalytic.scala:48 []
|   NewHadoopRDD[0] at newAPIHadoopRDD at
         MailRecordAnalytic.scala:94 []
```

References I

Cutting, Doug. 2009. “Apache Avro.” http://avro.apache.org/.

Dale, Markus, Jeff Schmidt, JT Halbert, and Jason Morris. 2015. “Spark Mail Tutorial.” https://github.com/medale/spark-mail.

Ecosystem. 2015. “Databricks Spark Ecosystem.” https://databricks.com/spark/about.

Wikipedia. 2014. “Combinatory Logic.” http://en.wikipedia.org/wiki/Combinatory_logic.