Analytic Cloud with Spark

Shelly Garion
IBM Research -- Haifa
Why Spark?

- **Apache Spark™** is a fast and general open-source cluster computing engine for big data processing

- **Speed:** Spark is capable to run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk

- **Ease of use:** Write applications quickly in Java, Scala, Python and R, also with notebooks

- **Generality:** Combine SQL, streaming, and complex analytics – machine learning, graph processing

- **Runs everywhere:** runs on Apache Mesos, Hadoop YARN cluster manager, standalone, or in the cloud, and can read any existing Hadoop data, and data from HDFS, object store, databases etc.
History of Spark

- Started in 2009 as a research project of UC Berkley
- Now it is an open source Apache project
  - Built by a wide set of developers from over 200 companies
  - more than 1000 developers have contributed to Spark
- IBM has decided to “bet big on Spark” at June 2015
  - Created Spark Technology Center (STC) - [http://www.spark.tc/](http://www.spark.tc/)
  - “Spark as a Service” on Bluemix
How to Analyze BigData?
Basic Example: Word Count (Spark & Python)

```python
> lines = sc.textFile("hamlet.txt")
> counts = lines.flatMap(lambda line: line.split(" "))
   .map(lambda word => (word, 1))
   .reduceByKey(lambda x, y: x + y)
```

Basic Example: Word Count (Spark & Scala)

```scala
>val lines = sc.textFile("hamlet.txt")
>val counts = lines.flatMap(_.split(" "))
  .map((_, 1))
  .reduceByKey(_ + _)
```

Spark RDD (Resilient Distributed Dataset)

- Immutable, partitioned collections of objects spread across a cluster, stored in RAM or on Disk
- Built through lazy parallel transformations
- **Fault tolerance** – automatically built at failure

```scala
var myRDD = sc.sequenceFile("hdfs://...")
```

- We can apply **Transformations** or **Actions** on RDD
Spark Cluster

- **Driver program** – The process running the main() function of the application and creating the SparkContext
- **Cluster manager** – External service for acquiring resources on the cluster (e.g. standalone, Mesos, YARN)
- **Worker node** - Any node that can run application code in the cluster
- **Executor** – A process launched for an application on a worker node
Spark Scheduler

- **Task** - A unit of work that will be sent to one executor
- **Job** - A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action
- **Stage** - Each job gets divided into smaller sets of tasks called *stages* that depend on each other
Scala

- **Spark** was originally written in Scala
  - Java and Python API were added later

- **Scala**: high-level language for the JVM
  - Object oriented
  - Functional programming
  - Immutable
  - Inspired by criticism of the shortcomings of Java

- Static types
  - Comparable in speed to Java
  - Type inference saves us from having to write explicit types most of the time

- Interoperates with Java
  - Can use any Java class
  - Can be called from Java code
Scala vs. Java

Declaring variables:
```scala
var x: Int = 7
var x = 7 // type inferred
val y = "hi" // read-only
```

Java equivalent:
```java
int x = 7;
final String y = "hi";
```

Functions:
```scala
def square(x: Int): Int = x*x

def announce(text: String) =
{
    println(text)
}
```

Java equivalent:
```java
int square(int x) {
    return x*x;
}

void announce(String text) {
    System.out.println(text);
}
```
Spark & Scala: Creating RDD

# Turn a Python collection into an RDD
>sc.parallelize([1, 2, 3])

# Turn a Scala collection into an RDD
>sc.parallelize(List(1, 2, 3))

# Load text file from local FS, HDFS, or S3 or SoftLayer object store
>sc.textFile("file.txt")
>sc.textFile("directory/**.txt")
>sc.textFile("hdfs://namenode:9000/path/file")
>sc.textFile("swift://ContainerName.spark/ObjectName")
Spark & Scala: Basic Transformations

```scala
>val nums = sc.parallelize(List(1, 2, 3))

// Pass each element through a function
>val squares = nums.map(x: x*x)  // {1, 4, 9}

// Keep elements passing a predicate
>val even = squares.filter(x => x % 2 == 0)  // {4}

// Map each element to zero or more others
>nums.flatMap(x => 0.to(x))
///=> {0, 1, 0, 1, 2, 0, 1, 2, 3}
```
Spark & Scala: Basic Actions

```scala
> val nums = sc.parallelize(List(1, 2, 3))

// Retrieve RDD contents as a local collection
> nums.collect()  //=> List(1, 2, 3)

// Return first K elements
> nums.take(2)  //=> List(1, 2)

// Count number of elements
> nums.count()  //=> 3

// Merge elements with an associative function
> nums.reduce{case (x, y) => x + y}  //=> 6

// Write elements to a text file
> nums.saveAsTextFile("hdfs://file.txt")
```

Spark & Scala: Key-Value Operations

```scala
> pets = sc.parallelize(
  List(("cat", 1), ("dog", 1), ("cat", 2)))
> pets.reduceByKey(_ + _)
  //=> ((cat, 3), (dog, 1))
> pets.groupByKey() //=> {(cat, [1, 2]), (dog, [1])}
> pets.sortByKey() //=> [(cat, 1), (cat, 2), (dog, 1)]
```
Example: Spark Core API

Goal:

Find number of distinct names per "first letter".

<table>
<thead>
<tr>
<th>Ahir</th>
<th>Pat</th>
<th>Andy</th>
</tr>
</thead>
</table>
Example: Spark Core API

Goal:

Find number of distinct names per "first letter".

Example: Spark Core API

Goal: Find number of distinct names per “first letter”

```scala
sc.textFile("hdfs:/names")
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```

```
res0 = [(A, 2), (P, 1)]
```
Example: Spark Core API

Better implementation:

```scala
sc.textFile("hdfs:/names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), 1))
  .reduceByKey(_ + _)
  .collect()
```

Original:

```scala
sc.textFile("hdfs:/names")
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues { names => names.toSet.size }
  .collect()
```
Example: PageRank

How to implement PageRank algorithm using Map/Reduce?

```scala
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
  .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

PageRank Algorithm

- Start each page with a rank of 1
- On each iteration:
  A. \( \text{contrib} = \frac{\text{curRank}}{\text{neighbors}} \)
  B. \( \text{curRank} = 0.15 + 0.85 \sum_{\text{neighbors}} \text{contrib} \)

Spark Platform

Spark Platform

Spark RDD API
Spark Platform: GraphX

```python
graph = Graph(vertexRDD, edgeRDD)
graph.connectedComponents() # returns a new RDD
```
Spark Platform: GraphX
Example: PageRank

PageRank is implemented using **Pregel** graph processing

```scala
// get people with top-k pageranks
def findTopPageRank(allPeople: RDD[String], links: RDD[(String, String, Double)], k: Int) = {
  val versRDD = allPeople.map(p => (uid(p), p))
  val edgesRDD = links.map{(l, r, score) => Edge(uid(l), uid(r), score)}

  val g = Graph(versRDD, edgesRDD).cache
  val ranks = g.pageRank(0.001)

  ranks.vertices.top(k)(Ordering.by(_.2)).map(p => (fromUid(p._1), p._2))
}
Spark Platform: MLLib

model = LogisticRegressionWithSGD.train(trainRDD)
dataRDD.map(point => model.predict(point))

Spark Platform: MLLib
Example: K-Means Clustering

Goal:
Segment tweets into clusters by geolocation using Spark MLLib K-means clustering

Spark Platform: MLlib
Example: K-Means Clustering

To run the k-means algorithm in Spark, we need to first read the csv file

```scala
val sc = new SparkContext("local[4]", "kmeans")
val data = sc.textFile(arg)
val parsedData = data.map {
  line =>
    Vectors.dense(line.split(',').slice(0, 2).map(_.toDouble))
}

Then we can run the spark kmeans algorithm:

```scala
val iterationCount = 100
val clusterCount = 10
val model = KMeans.train(parsedData, clusterCount, iterationCount)
```

From the model we can get the cluster centers and group the tweets by cluster:

```scala
val clusterCenters = model.clusterCenters map (_.toArray)
val cost = model.computeCost(parsedData)
printIn("Cost: "+ cost)
val tweetsByGroup = data .map (_.split(',').slice(0, 2).map(_.toDouble)) .groupBy(rdd => model.predict(Vectors.dense(rdd))) .collect()
sc.stop()
```

Spark Platform: MLLib
Example: K-Means Clustering

[Map image showing k-means clustering with numerical labels for each cluster.]

Spark Platform: Streaming

dstream = spark.networkInputStream()
dstream.countByWindow(Seconds(30))
Spark Platform: Streaming Example

```scala
// popular topics of last 3 minutes
def latestTopics(docDStream: DStream[String], k: Int) = {
  val topicDStream = docDStream.flatMap(d => getTopics(d).map(t => (t, 1)))
  val topicCountDStream = topicDStream.reduceByKeyAndWindow(
    \[(x, y) => x + y \],
    \[(x, y) => x - y \],
    Minutes(3), // window size
    Minutes(1)  // batch size
  )
  val sortedTopics = topicCountDStream.transform(rdd => rdd.sortBy(_.2, false))
  sortedTopics
}
```
Spark Platform: SQL and DataFrames

Spark Platform: SQL

```
rdd = sql("select * from rdd1 where age > 10")
```
Spark Platform: SQL and DataFrames Example

Using RDDs

data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \.reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \.map(lambda x: [x[0], x[1][0] / x[1][1]]) \.collect()

Using DataFrames

sqlCtx.table("people") \.groupBy("name") \.agg("name", avg("age")) \.collect()
Machine Learning Pipeline with Spark ML

```scala
// create pipeline
tok = Tokenizer(in="text", out="words")
tf = HashingTF(in="words", out="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tok, tf, lr])

// train pipeline
df = sqlCtx.table("training")
model = pipeline.fit(df)

// make predictions
df = sqlCtx.read.json("/path/to/test")
model.transform(df)
  .select("id", "text", "prediction")
```

Combined Analytics of Data

Analyze tabular data with SQL
Analyze graph data using GraphX graph analytics engine
Use same machine learning Infrastructure
Use same solution for streaming data

Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, Michael Franklin, and Ion Stoica, “GRAPHX: UNIFIED GRAPH ANALYTICS ON SPARK”, spark summit July 2014