SPARK

Introduction to Information Retrieval

CS 150

Donald J. Patterson

Spark™ Lightning-fast cluster computing

databricks™ making big data simple

Matei Zaharia

Assistant Professor
Douglas T. Ross Career Development Professor of Software Technology
CSAIL, EECS, MIT

I'm an assistant professor at MIT CSAIL, where I work on computer systems and big data as part of the PDOS and bigdata@CSAIL groups. I'm also co-founder and CTO of Databricks, the big data company commercializing Apache Spark.

You can contact me at matei@mit.edu or find me in Stata Center office 32G-996.

Teaching

I'm currently teaching 6.S897: Large-Scale Systems, a graduate seminar on clouds and big data.

Projects

Publications

Talks

Code

HISTORY

- Started at the Berkeley AMPLab in 2009 as a research project
- open-source in 2010
- Submitted to the Apache Foundation in 2013
- Version 1.5.2 was released on 11/9/15
- One of the top open source projects today
  - https://github.com/apache/spark

SPARK

CLAIMS

- Cluster computing framework
- Supports general execution graphs
- Supports multiple languages
  - Java, Scala, Python, R
- Supports multiple storage types
  - HDFS, SQL, text files
- Supports libraries
  - MLib for machine learning
  - GraphX for graph processing
  - Streaming

SPARK

CLAIMS

- Runs on Amazon EC2
- Runs as a standalone installation
- Runs on Apache Mesos
- Runs on Hadoop YARN

SPARK

CLAIMS

- Hadoop integration
- Interactive Shell
- Analytic Suite for large-scale graph processing
- MapReduce is just one data flow supported
- RDD
  - Resilient Distributed Dataset

Basics

Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python. Start it by running the following in the Spark directory:

Scala **Python**

```
./bin/pyspark
```

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs. Let's make a new RDD from the text of the README file in the Spark source directory:

```
>>> textFile = sc.textFile("README.md")
```

RDDs have *actions*, which return values, and *transformations*, which return pointers to new RDDs. Let's start with a few actions:

```
>>> textFile.count() # Number of items in this RDD
126

>>> textFile.first() # First item in this RDD
u'# Apache Spark'
```

Now let's use a transformation. We will use the `filter` transformation to return a new RDD with a subset of the items in the file.

```
>>> linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

We can chain together transformations and actions:

```
>>> textFile.filter(lambda line: "Spark" in line).count() # How many lines contain "Spark"?
15
```

More on RDD Operations

RDD actions and transformations can be used for more complex computations. Let's say we want to find the line with the most words:

Scala | **Python**

```
>>> textFile.map(lambda line: len(line.split())).reduce(lambda a, b: a if (a > b) else b)
15
```

This first maps a line to an integer value, creating a new RDD. `reduce` is called on that RDD to find the largest line count. The arguments to `map` and `reduce` are Python anonymous functions (lambdas), but we can also pass any top-level Python function we want. For example, we'll define a `max` function to make this code easier to understand:

```
>>> def max(a, b):
...     if a > b:
...         return a
...     else:
...         return b
...

>>> textFile.map(lambda line: len(line.split())).reduce(max)
15
```

One common data flow pattern is MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily:

```
>>> wordCounts = textFile.flatMap(lambda line: line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)
```

Here, we combined the `flatMap`, `map`, and `reduceByKey` transformations to compute the per-word counts in the file as an RDD of (string, int) pairs. To collect the word counts in our shell, we can use the `collect` action:

```
>>> wordCounts.collect()
[(u'and', 9), (u'A', 1), (u'webpage', 1), (u'README', 1), (u'Note', 1), (u'"local"', 1), (u'variable', 1), ...]
```

SPARK

Caching

Spark also supports pulling data sets into a cluster-wide in-memory cache. This is very useful when data is accessed repeatedly, such as when querying a small “hot” dataset or when running an iterative algorithm like PageRank. As a simple example, let’s mark our `linesWithSpark` dataset to be cached:

Scala **Python**

```
>>> linesWithSpark.cache()

>>> linesWithSpark.count()
19

>>> linesWithSpark.count()
19
```

It may seem silly to use Spark to explore and cache a 100-line text file. The interesting part is that these same functions can be used on very large data sets, even when they are striped across tens or hundreds of nodes. You can also do this interactively by connecting `bin/pyspark` to a cluster, as described in the programming guide.

Self-Contained Applications

Suppose we wish to write a self-contained application using the Spark API. We will walk through a simple application in Scala (with sbt), Java (with Maven), and Python.

Scala | Java | Python

This example will use Maven to compile an application JAR, but any similar build system will work.

We'll create a very simple Spark application, `SimpleApp.java`:

```java
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system
    SparkConf conf = new SparkConf().setAppName("Simple Application");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("a"); }
    }).count();

    long numBs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("b"); }
    }).count();

    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
  }
}
```

SPARK

To build the program, we also write a Maven `pom.xml` file that lists Spark as a dependency. Note that Spark artifacts are tagged with a Scala version.

```
<project>
  <groupId>edu.berkeley</groupId>
  <artifactId>simple-project</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.5.2</version>
    </dependency>
  </dependencies>
</project>
```

We lay out these files according to the canonical Maven directory structure:

```
$ find .
./pom.xml
./src
./src/main
./src/main/java
./src/main/java/SimpleApp.java
```

SPARK

To build the program, we also write a Maven `pom.xml` file that lists Spark as a dependency. Note that Spark artifacts are tagged with a Scala version.

```
<project>
  <groupId>edu.berkeley</groupId>
  <artifactId>simple-project</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.5.2</version>
    </dependency>
  </dependencies>
</project>
```

We lay out these files according to the canonical Maven directory structure:

```
$ find .
./pom.xml
./src
./src/main
./src/main/java
./src/main/java/SimpleApp.java
```

SPARK

Now, we can package the application using Maven and execute it with `./bin/spark-submit`.

```
# Package a JAR containing your application
$ mvn package
...
[INFO] Building jar: {..}/{..}/target/simple-project-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/simple-project-1.0.jar
...
Lines with a: 46, Lines with b: 23
```

Spark

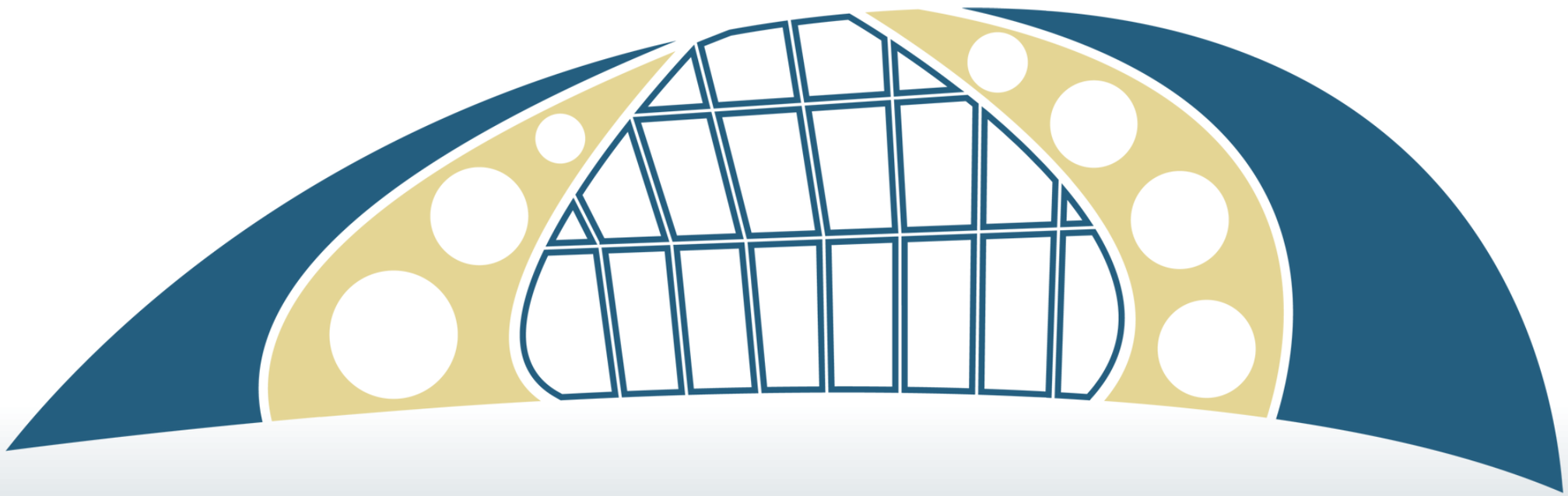
A framework for iterative and interactive cluster computing

Matei Zaharia, Mosharaf Chowdhury, Justin Ma,
Michael Franklin, Scott Shenker, Ion Stoica

amplab
UC Berkeley

SPARK


Strata+
Hadoop
WORLD
New Directions for
Spark in 2015
Matei Zaharia
Databricks
#StrataHadoop
strataconf.com



WESTMONT INSPIRED

COMPUTING LAB