Quick Start

This tutorial provides a quick introduction to using Spark. We will first introduce the API through Spark's interactive shell (in Python or Scala), then show how to write applications in Java, Scala, and Python.

To follow along with this guide, first, download a packaged release of Spark from the Spark website. Since we won't be using HDFS, you can download a package for any version of Hadoop.

Note that, before Spark 2.0, the main programming interface of Spark was the Resilient Distributed Dataset (RDD). After Spark 2.0, RDDs are replaced by Dataset, which is strongly-typed like an RDD, but with richer optimizations under the hood. The RDD interface is still supported, and you can get a more detailed reference at the RDD programming guide. However, we highly recommend you to switch to use Dataset, which has better performance than RDD. See the SQL programming guide to get more information about Dataset.

Interactive Analysis with the Spark Shell

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
```

Or if PySpark is installed with pip in your current environment:

```
pyspark
```

Spark's primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Due to Python's dynamic nature, we don't need the Dataset to be strongly-typed in Python. As a result, all Datasets in Python are Dataset[Row], and we call it `DataFrame` to be consistent with the data frame concept in Pandas and R. Let's make a new DataFrame from the text of the README file in the Spark source directory:

```
>>> textFile = spark.read.text("README.md")
```

You can get values from DataFrame directly, by calling some actions, or transform the DataFrame to get a new one. For more details, please read the *API doc*.

```
>>> textFile.count()  # Number of rows in this DataFrame
126

>>> textFile.first()  # First row in this DataFrame
Row(value=u'# Apache Spark')
```

Now let's transform this DataFrame to a new one. We call `filter` to return a new DataFrame with a subset of the lines in the file.

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
```

We can chain together transformations and actions:

```
>>> textFile.filter(textFile.value.contains("Spark")).count()  # How many lines contain "Spark"?
15
```

More on Dataset Operations

Dataset 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

```
>>> from pyspark.sql.functions import *
>>> textFile.select(size(split(textFile.value, "\s+")).name("numWords")).agg(max(col("numWords"))).collect()
[Row(max(numWords)=15)]
```

This first maps a line to an integer value and aliases it as “numWords”, creating a new DataFrame. `agg` is called on that DataFrame to find the largest word count. The arguments to `select` and `agg` are both *Column*, we can use `df.colName` to get a column from a DataFrame. We can also import pyspark.sql.functions, which provides a lot of convenient functions to build a new Column from an old one.

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

```
>>> wordCounts = textFile.select(explode(split(textFile.value, "\s+")).alias("word")).groupBy("word").count()
```

Here, we use the `explode` function in `select`, to transform a Dataset of lines to a Dataset of words, and then combine `groupBy` and `count` to compute the per-word counts in the file as a DataFrame of 2 columns: “word” and “count”. To collect the word counts in our shell, we can call `collect`:

```
>>> wordCounts.collect()
[Row(word=u'online', count=1), Row(word=u'graphs', count=1), ...]
```

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()
15

>>> linesWithSpark.count()
15
```

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 RDD 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 (pip).

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.sql.SparkSession;
import org.apache.spark.sql.Dataset;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system
    SparkSession spark = SparkSession.builder().appName("Simple Application").getOrCreate();
    Dataset<String> logData = spark.read().textFile(logFile).cache();

    long numAs = logData.filter(s -> s.contains("a")).count();
    long numBs = logData.filter(s -> s.contains("b")).count();

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

    spark.stop();
  }
}
```

This program just counts the number of lines containing ‘a’ and the number containing ‘b’ in the Spark README. Note that you’ll need to replace YOUR_SPARK_HOME with the location where Spark is installed. Unlike the earlier examples with the Spark shell, which initializes its own SparkSession, we initialize a SparkSession as part of the program.

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.

```xml
<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-sql_2.12</artifactId>
      <version>3.3.0</version>
      <scope>provided</scope>
    </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
```

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
```

Other dependency management tools such as Conda and pip can be also used for custom classes or third-party libraries. See also Python Package Management.

Where to Go from Here

Congratulations on running your first Spark application!

- For an in-depth overview of the API, start with the RDD programming guide and the SQL programming guide, or see “Programming Guides” menu for other components.
- For running applications on a cluster, head to the deployment overview.
- Finally, Spark includes several samples in the `examples` directory (Scala, Java, Python, R). You can run them as follows:

```
# For Scala and Java, use run-example:
./bin/run-example SparkPi

# For Python examples, use spark-submit directly:
./bin/spark-submit examples/src/main/python/pi.py

# For R examples, use spark-submit directly:
./bin/spark-submit examples/src/main/r/dataframe.R
```