# Intro to Apache Spark

# What is Spark?

- Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.
  - Written in Scala. Supports Scala, Java, Python, and R.
- It is streamlined for fast real time as well as batch performance.
- It can be used in a Spark cluster and coexist with a HDFS cluster using HDFS for input and output.
- The MapReduce concept is useful in understanding how Spark functions work!

# Spark Installation and Setup

- Download latest version from `http://spark.apache.org/downloads.html` by selecting the package type of "Pre-built for Hadoop 3 and later." The download is a compressed tarball, called `spark-3.4.1-bin-hadoop3.tgz`
- Create a top-level folder and unpack inside it as follows:

```
cd ~
mkdir spark-install
cd spark-install
tar xzvf ../Downloads/spark-3.4.1-bin-hadoop3.tgz
ln -s  spark-3.4.1-bin-hadoop3 spark
```

- Then add Spark programs to your path as follows (all in one line):

```
echo "export PATH=~/spark-install/spark/bin:~/spark-
    install/spark/sbin/:$PATH" >> ~/.bashrc
source ~/.bashrc
```

- Test it by starting one of the shells. For example, try `spark-shell` (for Scala-based shell) or try `pyspark` for Python-based shell. Use `Ctrl-d` to terminate the shell.

# Interactive Data Analysis(1)

- Start the interactive spark shell:

```
spark-shell
```

- Let's make a new Dataset from the text of an input file:

```
scala> val textFile = spark.read.textFile("word-count/input/
    Alice-in-Wonderland.txt")
textFile: org.apache.spark.sql.Dataset[String] = [value:
    string]
```

- We can get values directly from the Dataset, such as:

```
scala> textFile.count()
res0: Long = 3840

scala>textFile.first()
res1: String = ***This is the Project Gutenberg Etext of
    Alice in Wonderland***

scala> textFile.tail(4)
res2: Array[String] = Array(remembering her own child-life,
    and the happy summer days., "", "
          THE END", ?)
```

# Interactive Data Analysis (2)

- Now, let's use a transformation on the Dataset:

```
scala> val linesWithAlice = textFile.filter(line => line.
    contains("Alice"))
linesWithAlice: org.apache.spark.sql.Dataset[String] = [
    value: string]

scala> linesWithAlice.count()
res4: Long = 393
```

- We can chain together transformations and actions:

```
scala> val linesWithAlice = textFile.filter(line => line.
    contains("Alice")).count()
linesWithAlice: Long = 393
```

# Interactive Data Analysis (3)

- Dataset actions and transformation can be used for more complex analysis. Let's find the length of the longest line.

```
scala> textFile.map(line => line.split(" ").size).reduce((a,
     b) => if (a > b) a else b)
res5: Int = 56

scala> import java.lang.Math
import java.lang.Math

scala>  textFile.map(line => line.split(" ").size).reduce((a
    , b) => Math.max(a, b))
res6: Int = 56
```

- Note that we can import and use any Java library method in Scala.
- We can use Spark to implement the MapReduce pattern directly.

```
scala> val wordCounts = textFile.flatMap(line => line.split(
    " ")).groupByKey(identity).count()
wordCounts: org.apache.spark.sql.Dataset[(String, Long)] = [
    key: string, count(1): bigint]

scala> wordCounts.collect()
res7: Array[(String, Long)] = Array((By,4), (Aside,1), (
    those,10), (flashed,1), (some,47), (still,12), ...
```

# Interactive Data Analysis (4)

- We can pull data into a cluster-wide in-memory cache. This will speed up performance for a datset that is being used repeatedly.

```
scala> linesWithAlice.cache()
res10: linesWithAlice.type = [value: string]

scala> linesWithAlice.count()
res11: Long = 393
```

- These same functions can be used on very large data sets, even when they are striped across tens or hundreds of nodes. We can also do this interactively by connecting bin/spark-shell to a cluster, as we will see later.
- Note that a Dataset by itself isn't parallel/distributed by default. We will see later how to work with distributed data.

# First Standalone Example (Java)

- Examine the first simple app example under `CS535-resources/examples/Spark` in the simpleapp folder.
- Examine the code directly here: SimpleApp.java
- We can export the jar file from Eclipse (or other IDEs). Create a standard Java project and include all Spark jar files from `~/spark-install/spark/jars` as *External JAR*s for the project.
- Now, we submit the jar file to run on Spark as follows:

```
spark-submit --class "SimpleApp" --master local[4]
    simpleapp.jar
```

- Note that we are running Spark locally on our system. That is denoted by the `local[4]` option. The number in square brackets it is the number of threads to use.
- To redirect the verbose messages from Spark, redirect as follows:

```
spark-submit --class "SimpleApp" --master local[4]
    simpleapp.jar 2> log
```

# First Standalone Example (Python)

- Examine the first Python example in the `CS535-resources/examples/Spark` in the simpleapp-python folder.
- Examine the code directly here: SimpleApp.py
- Now, we run the python script on Spark as follows:

```
spark-submit --master local[4] SimpleApp.py
```

- Note that we are running Spark locally on our system. That is denoted by the `local[4]` option. The number in square brackets it is the number of threads to use.
- To redirect the verbose messages from Spark, redirect as follows:

```
spark-submit --master local[4] SimpleApp.py 2> log
```

# Installing Spark+Python ecosystem

- Here is a high level overview of the installation steps
  - Install Python
  - Install Java JDK
  - Install Jupyter Notebook
  - Install Apache Spark
    - Download the spark tgz file and unpack it (similar to Hadoop)
    - Set the paths for your login to the spark commands can be found from the command line.
- Use this PySpark Tutorial to walk you through the steps.

# Interactive Spark with Python

- Simply start the Python Spark shell with the the command `pyspark`.
- Using Pyspark from Jupyter lab is the better way to go. Open the simpleapp-jupyter example in Jupyter and follow along.

# Operation of a Spark Application

- A Spark application consists of a driver program that launches various parallel operations on the cluster. It creates distributed datasets on the cluster, and then applies operations to them.
- Driver programs access Spark via a `SparkContext` object, which represents a connection to a computing cluster. We use this to create Resilient Distributed Datasets (RDDs) on which we can then apply various parallel operations.
- The driver program manages a number of nodes called executors. These are used to parallelize the operations.
- Spark programs can be written via interactive shells (for Scala and Python, for example) or they can be standalone (for Scala, Python, Java etc). For production environments, standalone would be the norm.

# Spark Data Representation

- Spark supports three data representations: Resilient Distributed Dataset (RDD), Dataframe, Dataset.
- A *RDD* is a collection of objects that stores data partitioned across the nodes of the cluster, enabling parallel processing. This is the fundamental underlying data structure in Spark.
- A *Dataframe* is also a distributed collection organized into named columns. Dataframe are recommenede because of ease of usage.
- A *Dataset* is an extension of *Dataframe* with benefits of both *RDD* and *Dataset*. Supported in Scala and Java but not supported in Python (as it needs a statically typed language)