Spark with Python and Scala Programming

Hieu Nguyen

March 20, 2017

Outline

1. SIS BigData platform
2. Apache Spark
3. PySpark: spark python big data processing
4. Scala: spark scala big data processing
5. Spark SQL and DataFrames

SIS BigData platform

- Our SIS BigData cluster (5 nodes) is now ready to use
- **bigdata0** (coordinator node):
  8 cores CPU, 23.5 GiB Mem, and 88 GiB Disk
- **bigdata1, bigdata2**:
  24 cores CPU, 78.6 GiB Mem, 7.5 TiB Disk
- **bigdata3, bigdata4, bigdata5**:
  24 cores CPU, 62.9 GiB Mem, 4.2 TiB Disk
- The version of Cloudera is CDH 5.10.0
  - 4 services was deployed: HDFS, Hadoop MapReduce 2 (with YARN), Spark, and HBase
  - other services are available: Accumulo, Flume, Hive, Hue, Impala, Isilon, Kafka, Oozie, S3 Connector, Sentry, Solr, and Sqoo

How to Connect to SIS BigData platform

- You need an account on BigData cluster
- **Outside of University network**:
  – log into `shell.sis.uta.fi` (with your basic account)
  (if you are using Windows OS, simply use Tectia-SSH Terminal)

  – Then, `ssh <bigdata-account>@<server>.sis.uta.fi`
- **Inside of University network**:
  – `ssh <bigdata-account>@<server>.sis.uta.fi`

HDFS commands

- `hadoop fs -mkdir /user/<user-name>/foo`
  - creates a directory called "`foo`" in the HDFS home directory of the user `<user-name>`
- `hadoop fs -rm -r /user/<user-name>/foo`
  - removes a directory called "`foo`" in the HDFS home directory of the user `<user-name>`
- `hadoop fs -put foo.txt /user/<user-name>/foo/`
  - copy a file "`foo.txt`" to the HDFS "`foo`" directory
  - similar: `hadoop fs -copyFromLocal foo.txt /user/<user-name>/foo/`
- `hadoop fs -cat /user/<user-name>/foo/foo.txt`
  - sees the content of the file "`foo.txt`" in the HDFS
- `hadoop fs -rm /user/<user-name>/foo/foo.txt`
  - removes a file called "`foo.txt`" in the HDFS
- `hadoop fs -ls /user/<user-name>/foo`
  - does a directory listing for directory "`foo`" in the HDFS

HDFS commands (cont.)

- `hadoop fs -get /user/<user-name>/bar`
  - copy the directory "`bar`" in the user HDFS directory to the local file system
  - similar: `hadoop fs -copyToLocal /user/<user-name>/bar ./`
- `hadoop fs -getmerge /user/<user-name>/bar >> output_merge.txt`
  - does a file merge for the output if Spark job creates multiple output files, then stores the output in the local file system "`output_merge.txt`"

Apache Spark: lightning–fast cluster computing

- a fast and general engine for **large–scale data processing**
- **speed**
  - up to 100x faster than Hadoop MapReduce in memory
  - or 10x faster on disk
- **provides high–level**
  - APIs in: **Java**, **Scala**, **Python**, and **R**
  - tools including:
    – **Spark SQL and DataFrames** for SQL and structured data processing
    – **MLlib** for machine learning
    – **GraphX** for graph processing
    – **Spark Streaming** for stream processing of live datastreams
- **run everywhere**: standalone cluster mode, on EC2, on Hadoop YARN, or on Apache Mesos
- **access diverse data sources**: including HDFS, Cassandra, HBase, Hive, and any Hadoop data source

Apache Spark: components

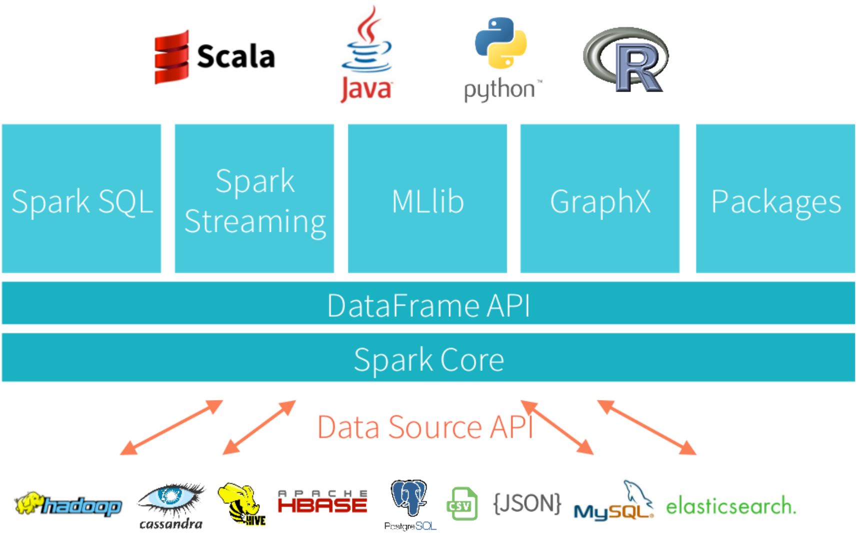


figure from Databricks

Apache Spark: open source ecosystem

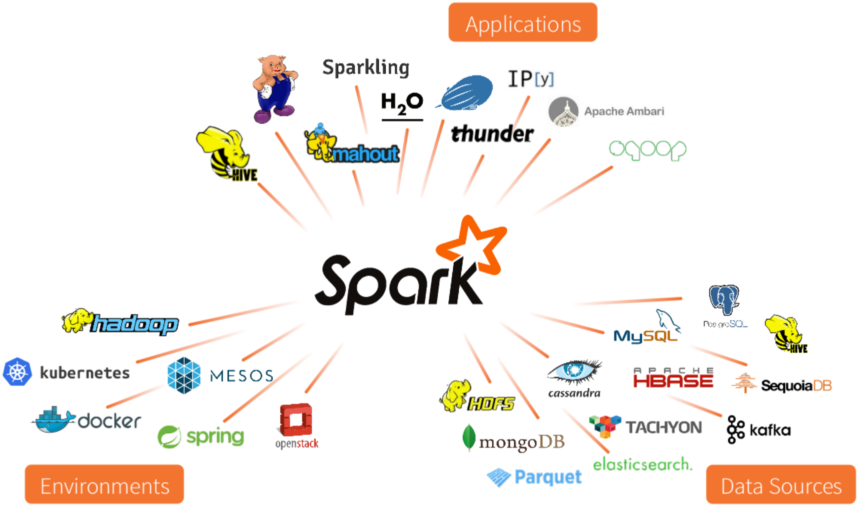


figure from Databricks

Apache Spark: resilient distributed datasets (RDD)

- **collections of objects** spread across a cluster
- stored in **RAM** or on **Disk**
- built through **parallel transformations**
- **automatically** rebuilt on failure
- two types of operations on RDDs: **transformations** and **actions**
- you can **persist (cache)** an RDD
    - by using: `persist()` or `cache()`
    - each node stores any partitions of it
    - computes in memory
    - reuses them in other actions
    - can choose storage level: MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK
    - release by: `unpersist()`

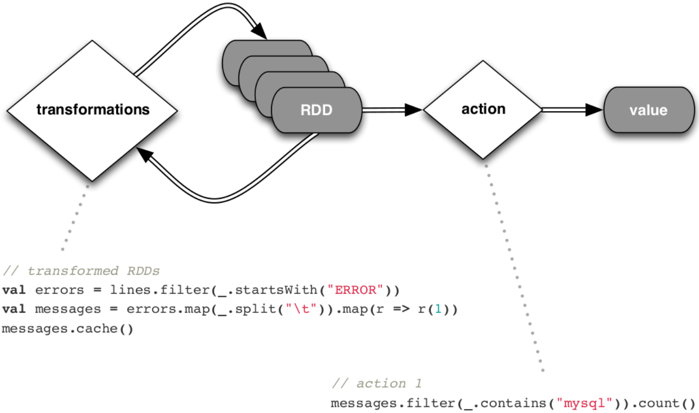
Apache Spark: transformations and actions

- **Spark transformations: are lazy (not executed until an action follows)**
  - `map()`
  - `flatMap()`
  - `reduceByKey()`
  - `filter()`
  - `sample()`
  - `union()`
  - `intersection()`
  - `distinct()`
  - `groupByKey()`
  - `sortByKey()`
  - `join()`
  - `cogroup`

- **Spark actions: time consuming (execution of an action results in all the previously created transformation)**
  - `reduce()`
  - `collect()`
  - `count()`
  - `countByValue()`
  - `first()`
  - `take()`
  - `takeSample()`
  - `foreach()`
  - `saveAsTextFile()`
  - `saveAsHadoop()`

Apache Spark: transformations and actions

- **transformations**: just "remember" the operation to be performed, the dataset to which the operation is to be performed
- **action**: brings back the data from the RDD to the local machine



```
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
```

```
// action 1
messages.filter(_.contains("mysql")).count()
```

figure from Databricks

Apache Spark: SparkContext

- **main entry point** to Spark functionality
- specifies running environment and app name
- uses to create RDDs from many input sources
- creates counters and accumulators
- available in shell (interactive mode) as variable sc
- in your program: you'd make your own

PySpark WordCount: including spark libraries

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext
from pyspark.sql.types import *
from pyspark.sql import Row
import pandas as pd
import matplotlib.pyplot as plt
import re
```

PySpark WordCount: main function

```
if __name__ == ’__main__’:
  //Configure Spark
  APP_NAME = "PySparkExample"
  conf = SparkConf().setAppName(APP_NAME).setMaster("local[1]")
  //conf = SparkConf().setAppName(APP_NAME).setMaster("local[*]")
  //conf = SparkConf().setAppName(APP_NAME).setMaster("yarn-client")
  //conf = SparkConf().setAppName(APP_NAME).setMaster("yarn-master")
  sc = SparkContext(conf=conf)
  //Transforms (load) the input data from HDFS into an RDD
  rddData = sc.textFile("hdfs:///user/hieunguyen/input/wc.txt")
  //Transformed RDD: process line by line, split by space
  wcFM = rddData.flatMap(lambda line:  line.split(" "))
  //Transformed RDD: pass each element (key) by 1 (value)
  wcM = wcFM.map(lambda word:  (word, 1))
  //Transformed RDD: merge key with an associative function
  wcRBK = wcM.reduceByKey(lambda a,b:  a+b)
  //No data is read or processed until after this line
  //Actioned RDD: save the results into HDFS
  wcRBK.saveAsTextFile("hdfs:///user/hieunguyen/wcoutput")
```

PySpark WordCount: run your code with pyspark shell

- Interactive mode: `pyspark`
  - `pyspark --master local[10] --executor-memory 25G --driver-memory 20G --num-executors 12 --executor-cores 2`
  - `pyspark --master local[*] --executor-memory 25G --driver-memory 20G --num-executors 12 --executor-cores 2`
  - `pyspark --master yarn-client --executor-memory 25G --driver-memory 20G --num-executors 12 --executor-cores 2`

  use additional package from third party
  - `pyspark --master yarn-client --packages com.databricks:spark-csv_2.10:1.5.0`

  use additional jar file
  - `pyspark --master yarn-client --jars test.jar`

PySpark WordCount: run your code with spark-submit

- `spark-submit`

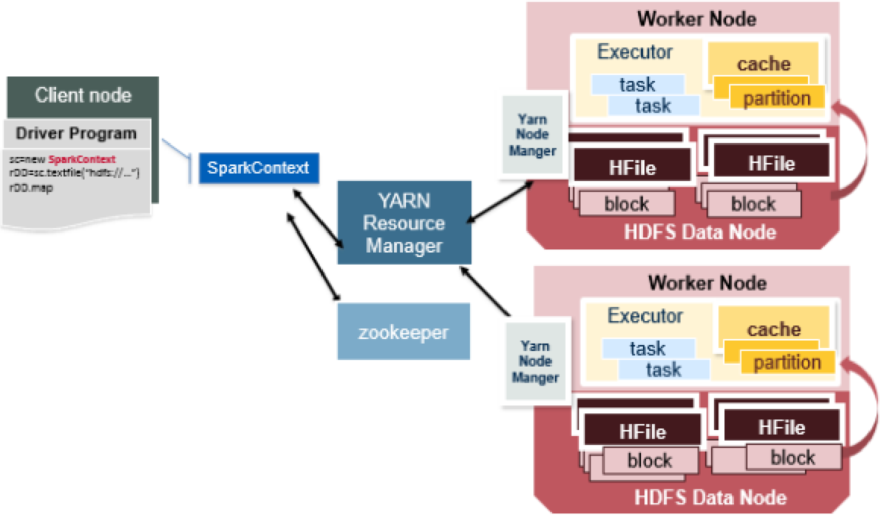
  -
    ```
    spark-submit pysparkExample.py
    ```

  -
    ```
    spark-submit --master yarn-client
    --executor-memory 25G --driver-memory 20G
    --num-executors 12 --executor-cores 2
    pysparkExample.py
    ```

  -
    ```
    spark-submit --master yarn-cluster
    --executor-memory 25G --driver-memory 20G
    --num-executors 12 --executor-cores 2
    pysparkExample.py
    ```
- run your code in background, separated spark output and error

  -
    ```
    nohup spark-submit --master yarn-client
    --executor-memory 25G --driver-memory 20G
    --num-executors 12 --executor-cores 2
    pysparkExample.py > output.out 2>error.err
    ```

PySpark WordCount: how spark job run on a cluster



```
sc=new SparkContext
rDD=sc.textfile("hdfs://...")
rDD.map
```

figure from www.mapr.com

PySpark WordCount: demo

- Demo PySpark WordCount

PySpark: accumulators and user defined functions

- **accumulators**: the global variable that can be shared across tasks
- **UDF**: simple way to add separate functions into Spark that can be used during various transformation stages

PySpark accumulators and UDF: demo

- Demo PySpark accumulators and UDF

Scala WordCount: including spark libraries

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD

import java.io._
```

Scala WordCount: main function

```
object ScalaWordCount {
  def main(args:  Array[String]) {
    val APP_NAME = "ScalaWordCount"

    val conf = new
       SparkConf().setAppName(APP_NAME).setMaster("yarn-client")

    val sc = new SparkContext(conf)

    val rddData =
            sc.textFile("hdfs:///user/hieunguyen/input/wc.txt")

    val wcFM = rddData.flatMap(line => line.split(" "))
    val wcM = wcFM.map(word => (word, 1))
    val wcRBK = wcM.reduceByKey((a, b) => a+b)
    wcRBK.saveAsTextFile("hdfs:///user/hieunguyen/wcoutput")
    }
}
```

Scala WordCount: build source code by using sbt

- Your directory layout should look like this

```
$ find .
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/ScalaWordCount.scala
```

- content of the `simple.sbt` file

```
name := "ScalaWordCount"
version := "1.0"
scalaVersion := "2.10.6"
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.6.0"
)
```

- build by: `sbt package`

Scala WordCount: run your Scala code

- Interactive mode: `spark-shell`

  -
    ```
    spark-shell --master yarn-client
    --executor-memory 25G --driver-memory 20G
    --num-executors 12 --executor-cores 2
    ```
- `spark-submit`

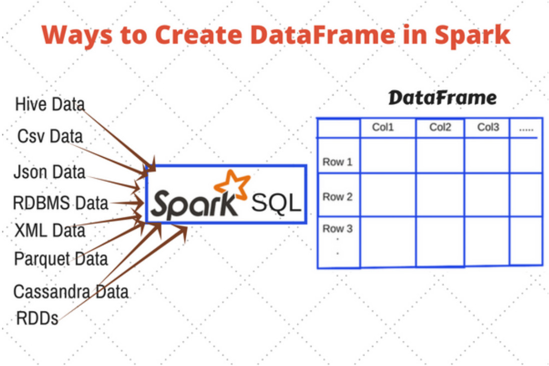
  -
    ```
    spark-submit --class "ScalaWordCount" --master
    yarn-client
    target/scala-2.10/scalawordcount_2.10-1.0.jar
    "seminar/input-wordcount" 1
    ```
  - `"seminar/input-wordcount"` is input file path on HDFS
  - `1` is threshold

Scala WordCount: demo

- Demo Scala WordCount

Spark SQL and DataFrames (DF)

- a DF is a **distributed collection of rows under named columns**
- DF in Spark has the **ability to handle petabytes of data**
- DF has a support for **wide range of data format and sources**
- has API support for different languages like **Python, R, Scala, Java**



Ways to Create DataFrame in Spark

Spark SQL and DataFrames: demo

- Demo Spark SQL and DataFrames

Spark: additional information

- turn an existing collection into an RDD
  - `sc.parallelize(<collection>)`
    i.e., `sc.parallelize([1,2,3])`
- load complete content of the file at once
  - `sc.wholeTextFiles(<file/folder path>)`
- load the file with partitions
  - `sc.wholeTextFiles(<file/folder path>, <number>)`
  - `sc.textFile(<file/folder path>, <number>)`
  - `sc.parallelize(<collection>, <number>)`
- use existing Hadoop input form (only for Java and Scala)
  - `sc.hadoopFile(keyClass, valClass, inputForm, conf`
- key–value pair in Python vs. Scala
  - Python: pair = (a, b), `pair[0] = a, pair[1] = b`
  - Scala: pair = (a, b), `pair._1 = a, pair._2 = b`

References

🌐 Programming Guide
http://spark.apache.org/docs/latest/programming-guide.html .