UNIVERSITY of WASHINGTON

Welcome to BIGDATA 210: Introduction to Data Engineering

Week 2

Week 2

What we will do?

- Questions from last week
- Setting up Sandbox on AWS
- Introductions to Apache Spark
  - What is Apache Spark
  - Zeppelin
- Concepts
  - RDD
  - Typical Program Flow
  - SparkContext
- Hands-On

You can setup your own HDP sandbox image on AWS. Follow these steps:

1. Download the VMWare version of our sandbox from hortonworks.com/sandbox
2. Rename the OVA file to .tar extension, extract it.
3. Create an S3 bucket under your personal AWS account. Remember the name of the bucket!
4. Install EC2 command line tools. Make sure they're in path.
5. cd to the directory you extracted with the tar file.
6. run: `ec2-import-instance -t m3.large Hortonworks_Sandbox_2.X-disk1.vmdk -f VMDK -o <ACCESS_KEY> -w <SECRET_KEY> -b <BUCKET NAME FROM STEP 2> -a x86_64 -p Linux`
7. Wait ...
8. Log into your AWS console, go to EC2, locate the new instance it created (it'll be stopped).
9. Select instance, Action, Create Image. Switch to AMI view.
10. Send AMI image ID to anyone. It creates an EBS volume in your account, don't remove it! It will cost about $5/month.

Apache Spark

What is Apache Spark?

Spark is:

- an open-source software solution that performs rapid calculations on in-memory datasets
- Open Source [Apache hosted & licensed]
- Free to download and use in production
- Developed by a community of developers
- Spark supports using well known languages such as:
  - Scala, Python, R, Java
- Spark SQL: Seamlessly mix SQL queries with Spark programs
- Spark on YARN enables deep integration with Hadoop and other YARN enabled workloads in the enterprise

Why Spark

- Elegant Developer APIs: Data Frames/SQL, Machine Learning, Graph algorithms and streaming
  - Scala, Python, Java and R
  - Single environment for importing, transforming, and exporting data
- In-memory computation model
  - Effective for iterative computations
- High level API
  - Allows users to focus on the business logic and not internals
- Promotes code reuse: APIs and data types are similar for batch and streaming

Why Spark ...

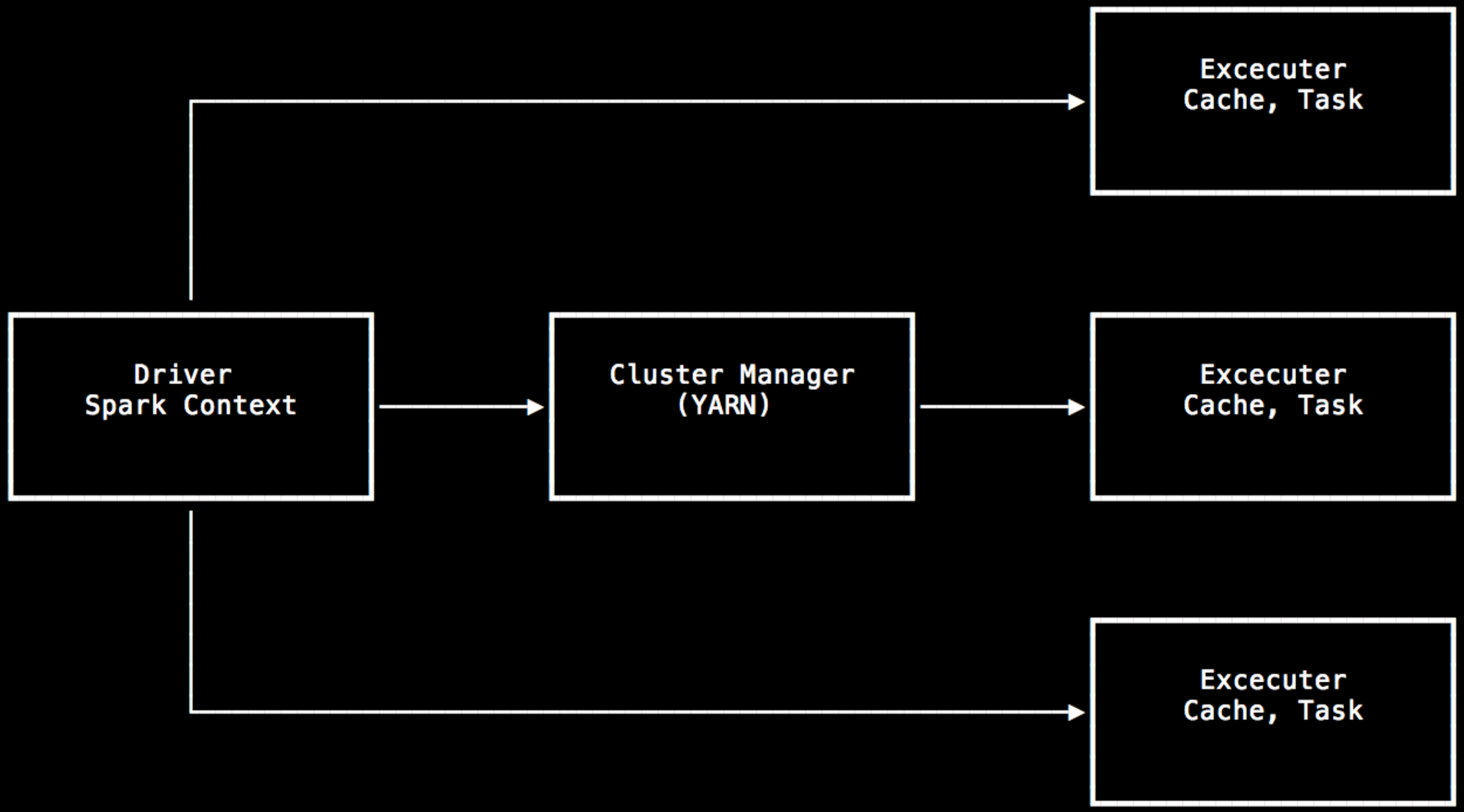
- Supports wide variety of workloads
  - MLlib for Data Scientists
  - Spark SQL for Data Analysts
  - Spark Streaming for micro batch use cases
  - Spark Core, SQL, Streaming, Mllib, and GraphX for Data Processing Applications
- Integrated fully with Hadoop and also an open source tool
  - Native integration with Hive, HDFS and any Hadoop FileSystem implementation
- Faster then MapReduce*

Why Spark ...

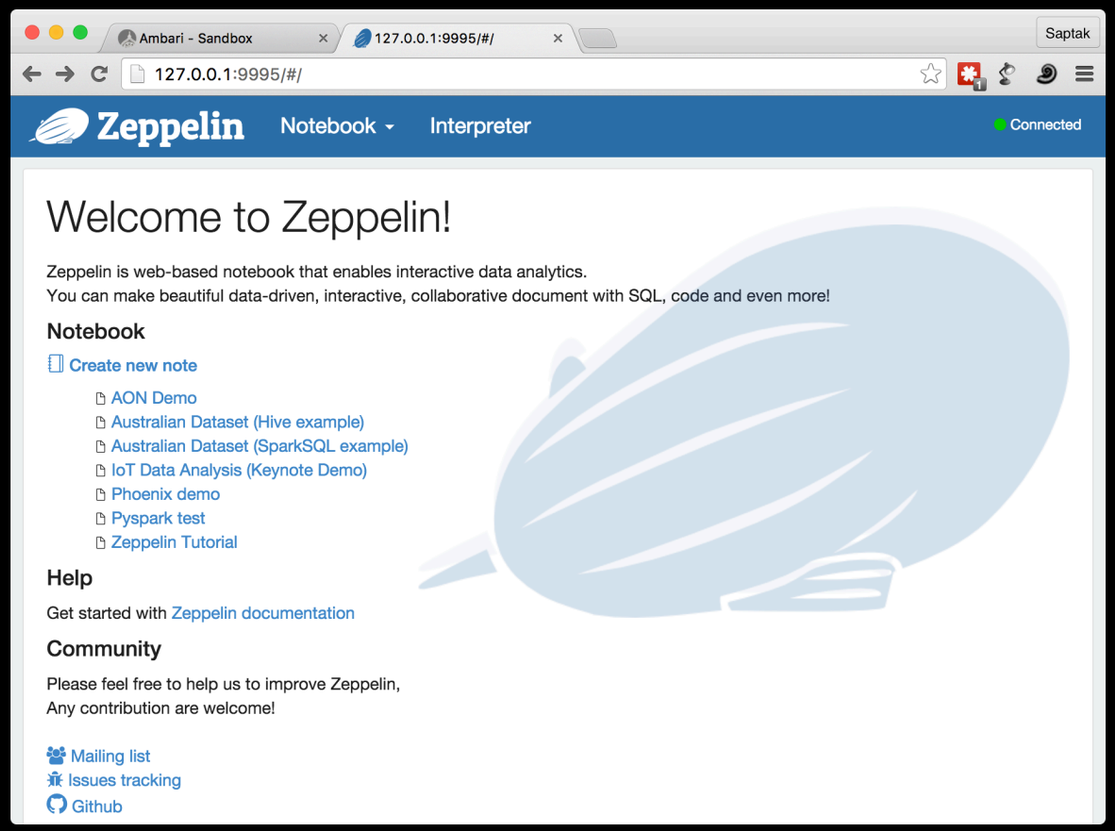
- Spark as query federation engine
  - Bring data from multiple sources to join/query in Spark
- Use multiple Spark libraries together
  - Common to see Core, ML & Sql used together
- Use Spark with various Hadoop ecosystem projects
  - Use Hive and Spark together
  - Use Hbase and Spark together

Why is Spark faster?

- Caching data to memory can avoid extra reads from disk
- Scheduling of tasks from 15-20s to 15-20ms
- Resources are dedicated the entire life of the application
- Can link multiple maps and reduces together without having to write intermediate data to HDFS
- Every reduce doesn't require a map


Excecuter
Cache, Task
Driver
Spark Context
Cluster Manager
(YARN)
Excecuter
Cache, Task
Excecuter
Cache, Task

Launch Zeppelin from your browser



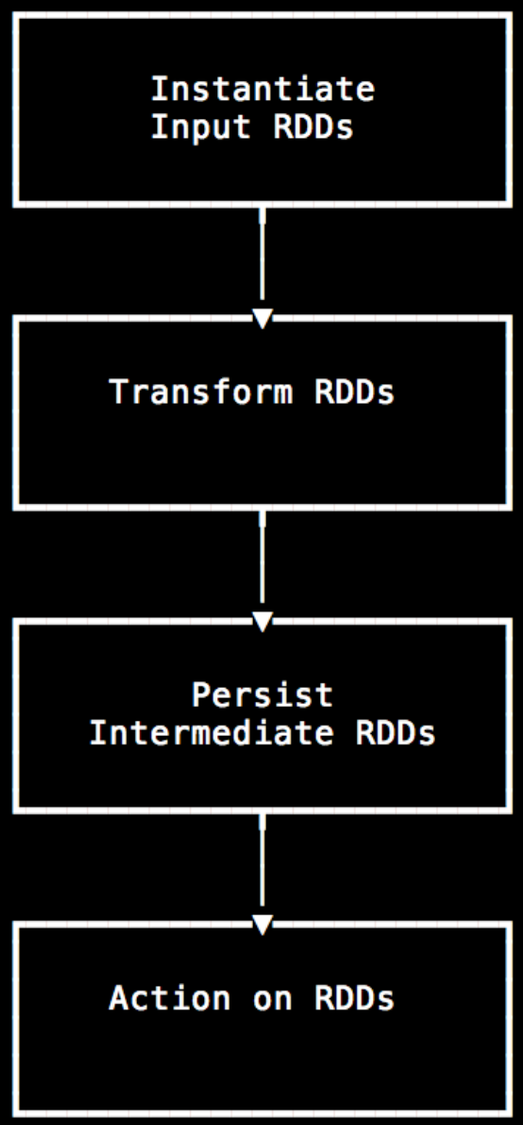
Zeppelin is at port 9995

Concepts

RDD

- Resilient
- Distributed
- Dataset

Program Flow



SparkContext

- Created by your driver Program
- Responsible for making your RDD's resilient and Distributed
- Instantiates RDDs
- The Spark shell creates a "sc" object for you

Creating RDDs

```
myRDD = parallelize([1,2,3,4])
myRDD = sc.textFile("hdfs:///tmp/shakepeare.txt")
#                   file://, s3n://
```

```
hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT name, age from users")
```

Creating RDDs

- Can also create from:
  - JDBC
  - Cassandra
  - HBase
  - Elasticsearch
  - JSON, CSV, sequence files, object files, ORC, Parquet, Avro

Passing Functions to Spark

Many RDD methods accept a function as a parameter

```
myRdd.map(lambda x: x*x)
```

is the same things as

```
def sqrN(x):
  return x*x

myRdd.map(sqrN)
```

RDD Transformations

- map
- flatmap
- filter
- distinct
- sample
- union, intersection, subtract, cartesian

Map example

```
myRdd = sc. parallelize([1,2,3,4])
myRdd.map(lambda x: x+x)
```

→ Results: 1, 2, 6, 8

Transformations are lazily evaluated

foreach()

Foreach is a very useful action. Usually done for side effects, such as saving data to a database, or updating a shared variable

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.foreach(lambda x: print(x))
```

```
1
2
3
4
```

RDD Actions

- first()
- collect()*
- count
- countByValue
- take(n)
- top
- reduce
- aggregate

* Make sure you only call this on small datasets or risk crashing your driver!

saveAsTextFile(path): write the RDD out to a file

Reduce

```
sum = rdd.reduce(lambda x, y: x + y)
```

Aggregate

```
sumCount = nums.aggregate(
                (0, 0),
                (lambda acc, value: (acc[0] + value, acc[1] + 1)),
                (lambda acc1 , acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])))

return sumCount[0] / float(sumCount[1])
```

Imports

```
from pyspark import SparkConf, SparkContext
import collections
```

Setup the Context

```
conf = SparkConf().setMaster("local").setAppName("foo")
sc = SparkContext(conf = conf)
```

First RDD

```
lines = sc.textFile("hdfs:///tmp/shakepeare.txt")
```

Extract the Data

```
ratings = lines.map(lambda x: x.split()[2])
```

Aggregate

```
result = ratings.countByValue()
```

Sort and Print the results

```
sortedResults = collections.OrderedDict(sorted(result.items()))

for key, value in sortedResults.iteritems():
  print "%s %i" % (key, value)
```

Run the Program

```
spark-submit ratings-counter.py
```

Key, Value RDD

```
#list of key, value RDDs where the value is 1.
totalsByAge = rdd.map(lambda x: (x, 1))
```

With Key, Value RDDs, we can:

- reduceByKey()
- groupByKey()
- sortByKey()
- keys(), values()

```
rdd.reduceByKey(lambda x, y: x + y)
```

Set operations on Key, Value RDDs

With two lists of key, value RDDs, we can:

- join
- rightOuterJoin
- leftOuterJoin
- cogroup
- subtractByKey

Mapping just values

When mapping just values in a Key, Value RDD, it is more efficient to use:

- mapValues()
- flatMapValues()

as it maintains the original partitioning.

Persistence

```
result = input.map(lambda x: x * x)
result.persist().is_cached
print(result.count())
print(result.collect().mkString(","))
```

Hands-On

Download the Data

```
#remove existing copies of dataset from HDFS
hadoop fs -rm  /tmp/kddcup.data_10_percent.gz

wget http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_percent.gz \
-O /tmp/kddcup.data_10_percent.gz
```

Load data in HDFS

```
hadoop fs -put /tmp/kddcup.data_10_percent.gz /tmp
hadoop fs -ls -h /tmp/kddcup.data_10_percent.gz

rm  /tmp/kddcup.data_10_percent.gz
```

First RDD

```
input_file = "hdfs:///tmp/kddcup.data_10_percent.gz"

raw_rdd = sc.textFile(input_file)
```

Retrieving version and configuration information

```
sc.version
sc.getConf.get("spark.home")
System.getenv().get("PYTHONPATH")
System.getenv().get("SPARK_HOME")
```

Count the number of rows in the dataset

```
print raw_rdd.count()
```

Inspect what the data looks like

```
print raw_rdd.take(5)
```

Spreading data across the cluster

```
a = range(100)

data = sc.parallelize(a)
```

Filtering lines in the data

```
normal_raw_rdd = raw_rdd.filter(lambda x: 'normal.' in x)
```

Count the filtered RDD

```
normal_count = normal_raw_rdd.count()

print normal_count
```

Importing local libraries

```
from pprint import pprint

csv_rdd = raw_rdd.map(lambda x: x.split(","))

head_rows = csv_rdd.take(5)

pprint(head_rows[0])
```

Using non-lambda functions in transformation

```
def parse_interaction(line):
    elems = line.split(",")
    tag = elems[41]
    return (tag, elems)

key_csv_rdd = raw_rdd.map(parse_interaction)
head_rows = key_csv_rdd.take(5)
pprint(head_rows[0]
```

Using collect() on the Spark driver

```
all_raw_rdd = raw_rdd.collect()
```

Questions