Introduction to Apache Spark

SMU, 2019

Nikita Neveditsin
nikita.neveditsin@smu.ca

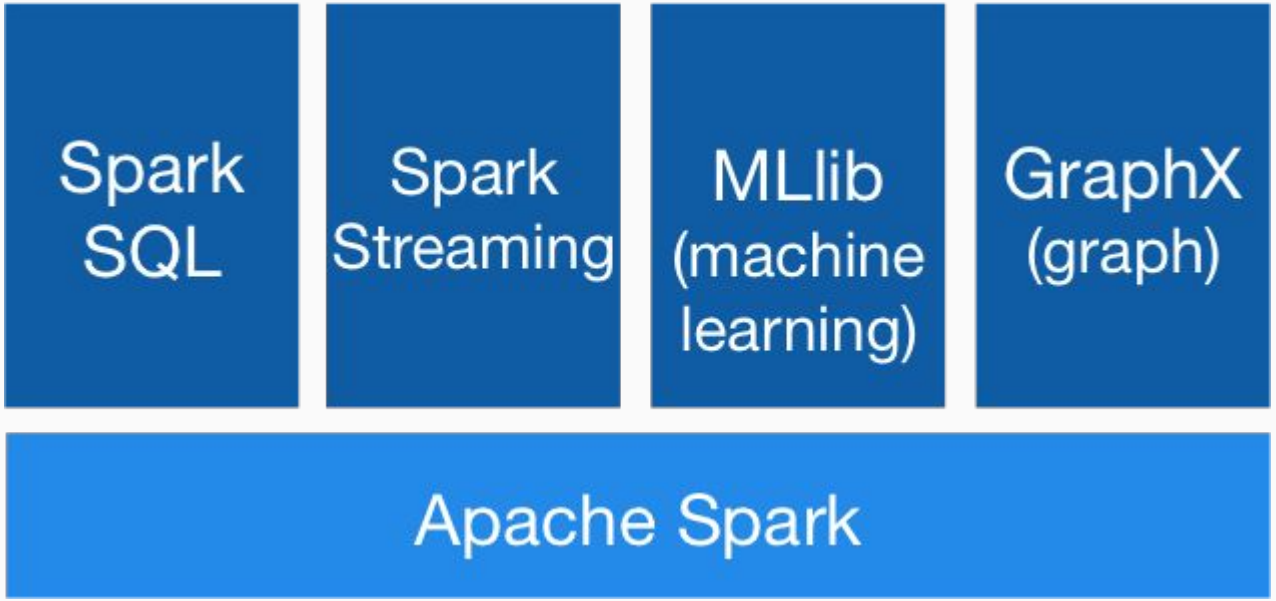
What is Spark?

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Originally developed at the University of California, Berkeley in 2014

- Used for Big Data analytics
- Works with Scala, Java, R, Python
- Can interact with Hadoop components

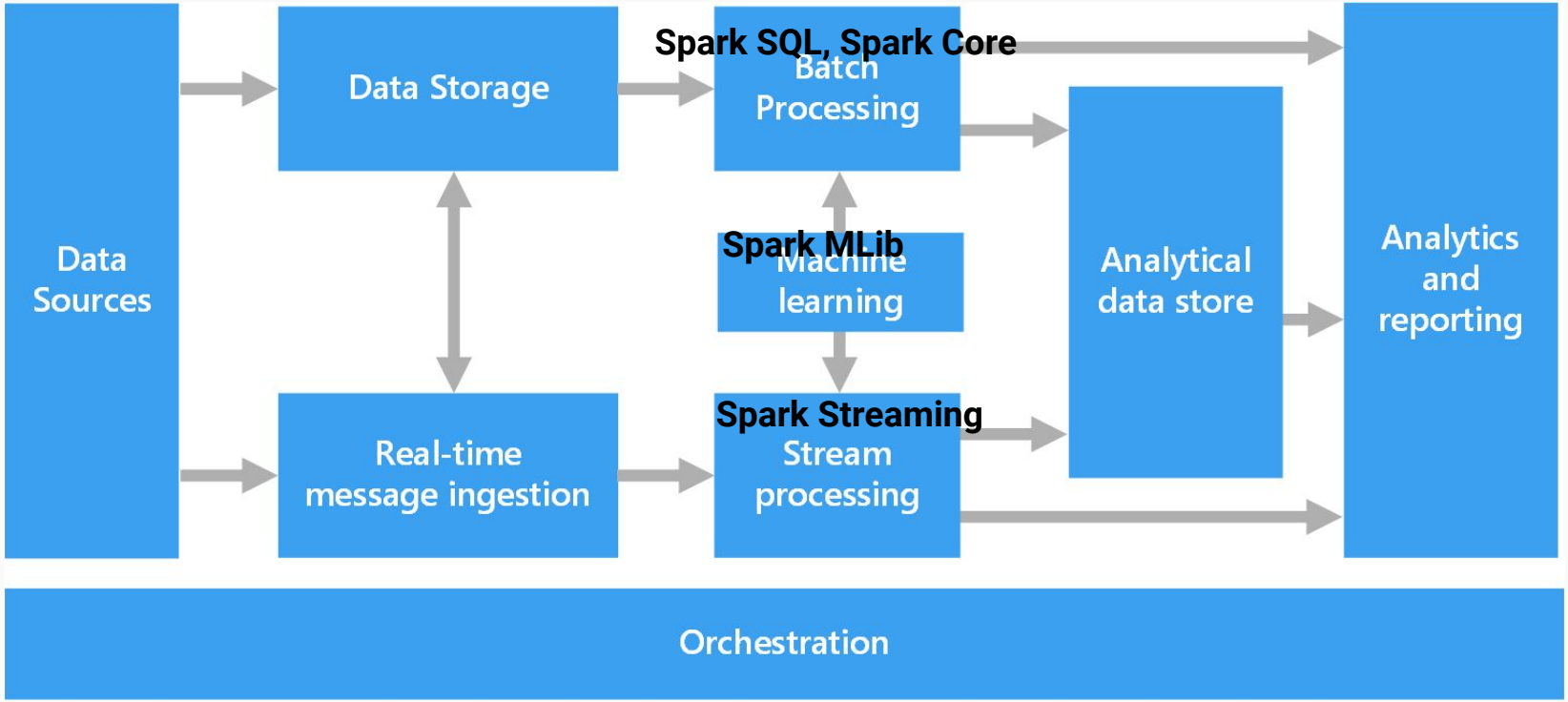
[link]

Spark components



[link]

Spark components (cont-d)

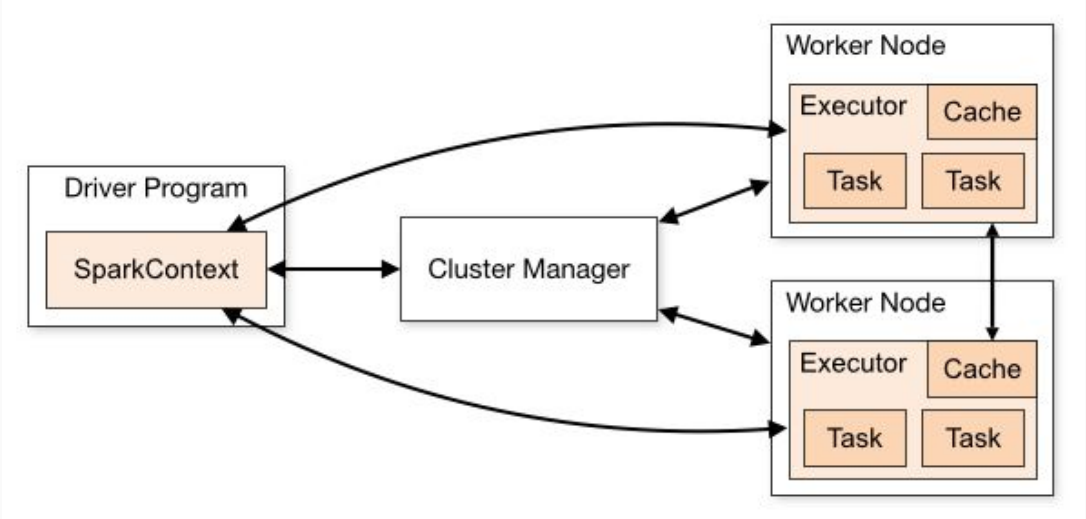


[link]

Spark: Master/Slave architecture (again)

Spark is running in a distributed cluster environment:

- **Client** submits a *Spark application* to Driver
- **Driver** is a process where your application runs. It sends tasks to Executors and schedules them
- **Cluster manager** is responsible for resource allocation/management (CPU/RAM/Storage)
- **Executors** do the actual computations by running tasks and store RDDs in their RAM

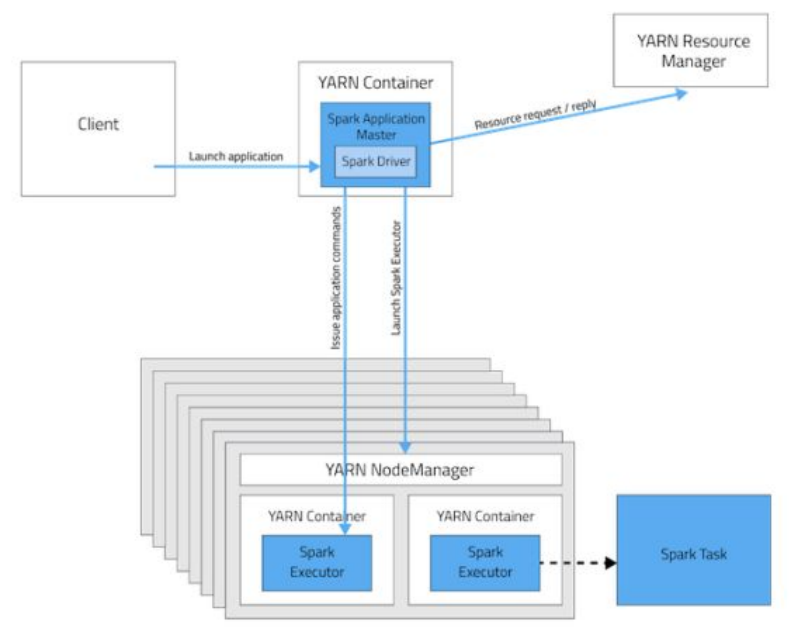


[link]

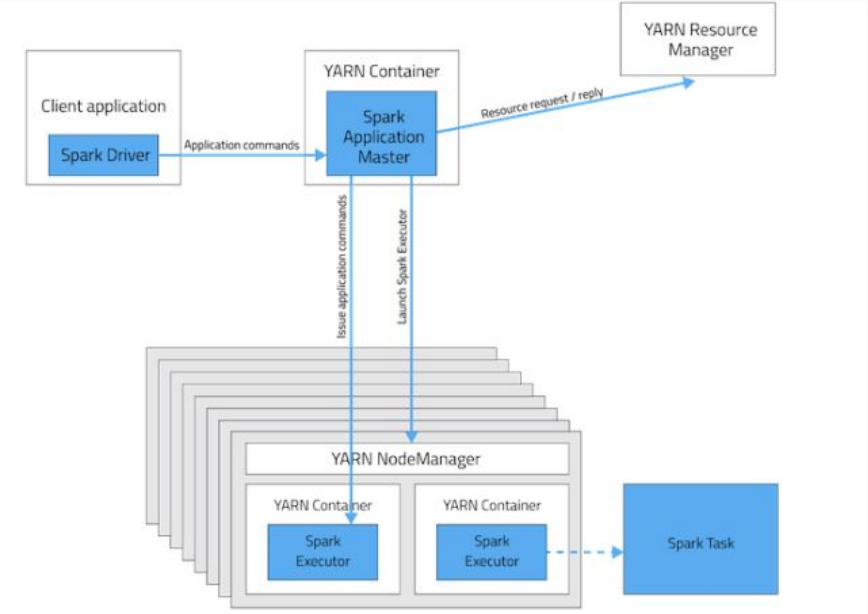
Spark: Cluster management options

- **Standalone** – a simple cluster manager included with Spark that makes it easy to set up a cluster: good choice if you don't need integration with Hadoop (Hive, Pig, HDFS)
- **Hadoop YARN** – the resource manager in Hadoop 2: choose if you need to integrate Spark applications with Hadoop (or already have a Hadoop cluster)
- **Apache Mesos** – a general cluster manager that can also run Hadoop MapReduce and service applications (alternative to YARN, may run faster as it's written in C++ vs YARN which is written in Java)
- **Kubernetes** – an open-source system for automating deployment, scaling, and management of containerized application (usually used when other applications run on the same physical machines)

Spark: running on top of YARN



Cluster deployment mode



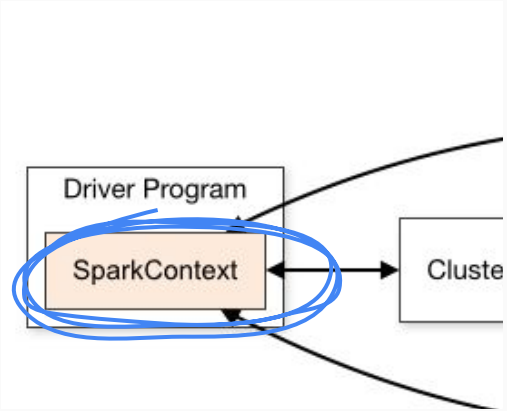
Client deployment mode

[link]

Spark Context

SparkContext was the entry gate of Apache Spark functionality prior to Spark 2.0. The most important step of any Spark driver application is to generate SparkContext. It allows your Spark Application to access Spark Cluster with the help of Resource Manager

To create SparkContext, first **SparkConf** should be made. The SparkConf has a configuration parameter that our Spark driver application will pass to SparkContext. Some of these parameter defines properties of Spark driver application. While some are used by Spark to allocate resources on the cluster, like the number, memory size, and cores used by executor running on the worker nodes [l]

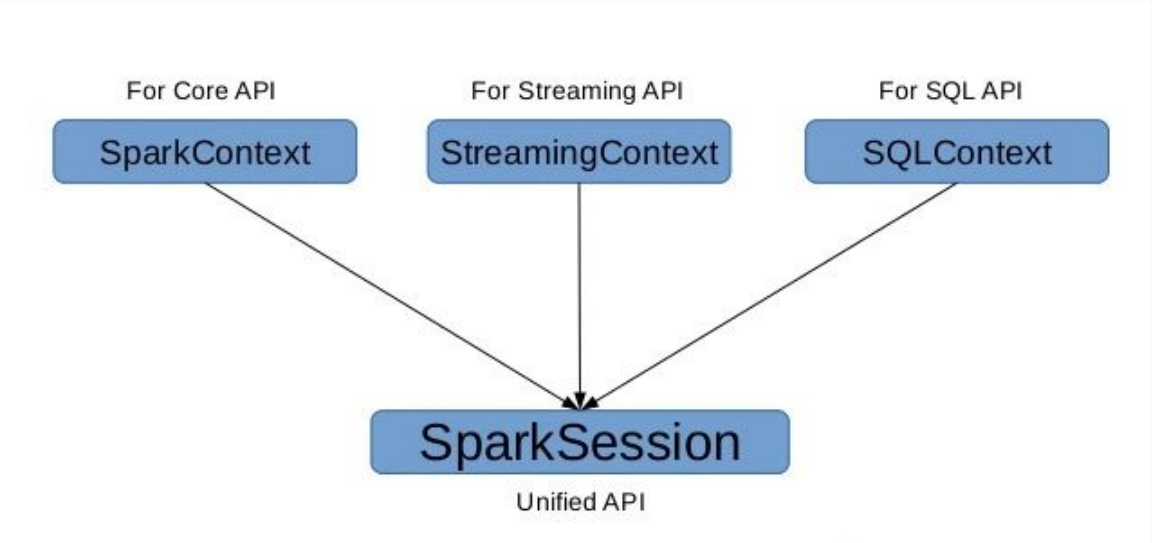


[link]

Spark Session

SparkSession is the entry point from Spark 2.0: prior to 2.0, there was SparkContext, SQLContext, and StreamingContext. SparkSession unifies these three interfaces

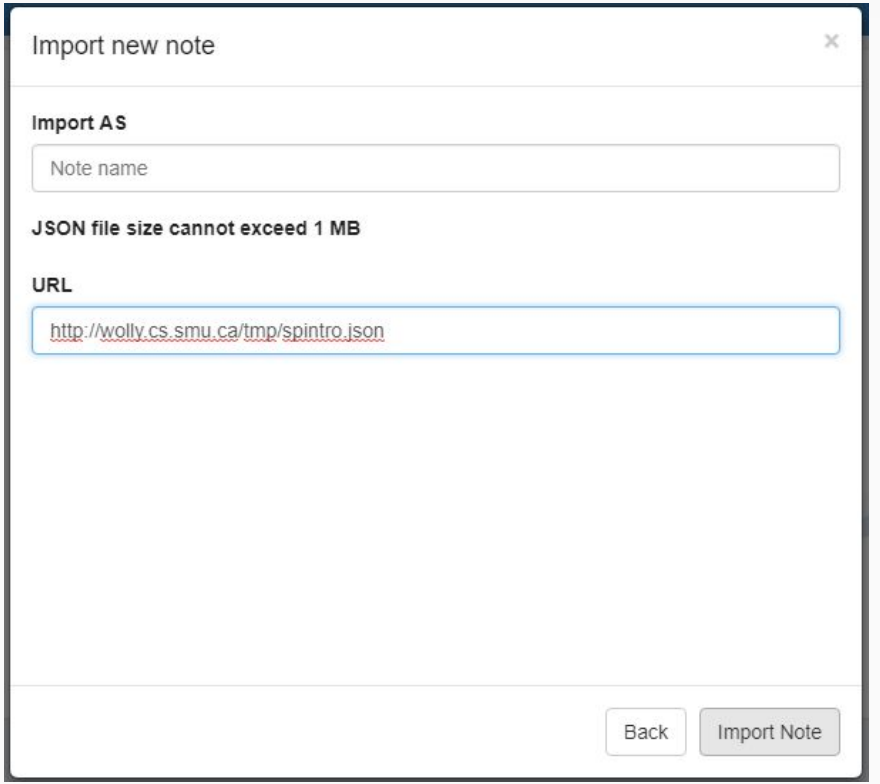
Zeppelin Notebook provides SparkSession with **spark** object.



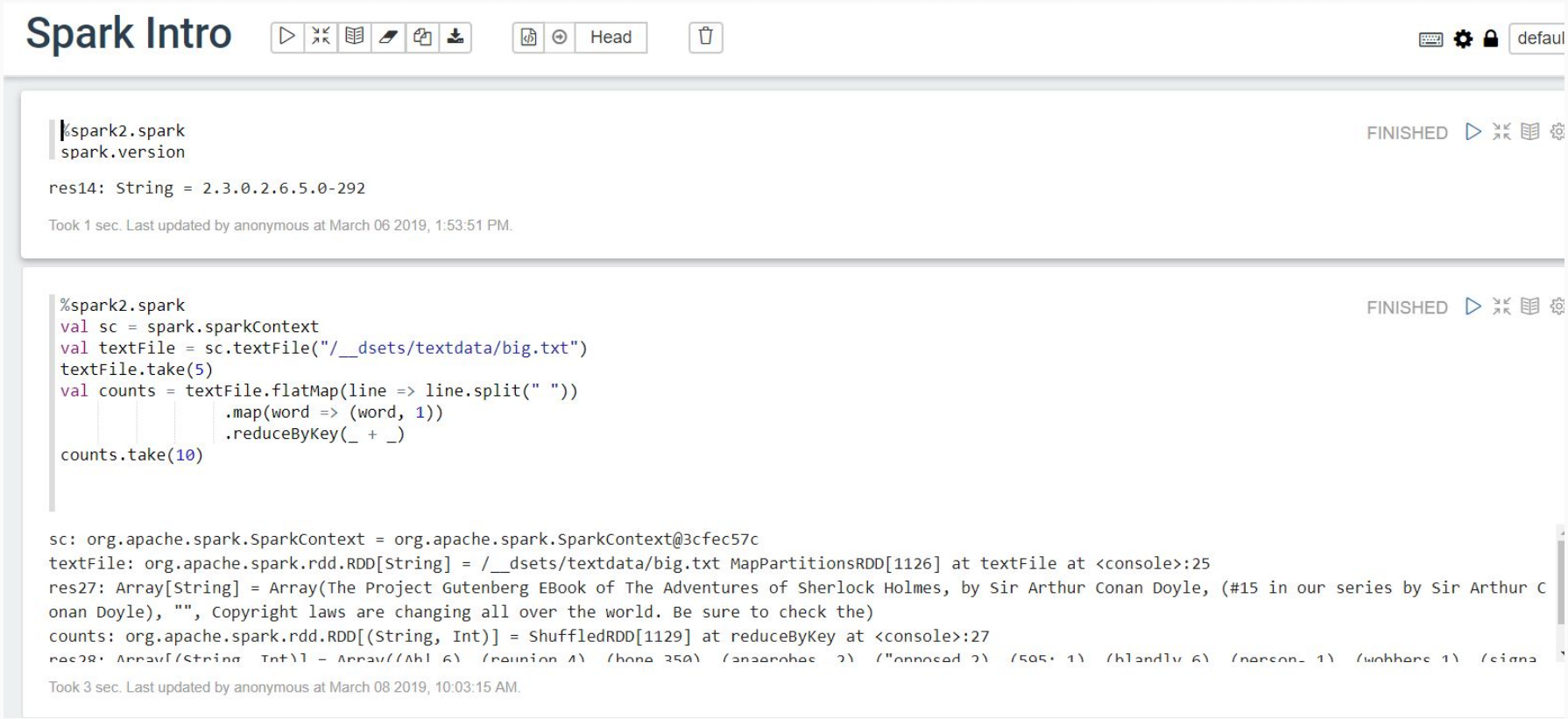
[link]

First Spark program: word count

- Open Zeppelin Notebook (http://sandbox-hdp.hortonworks.com:9995)
- Import Note -> Add from URL http://wolly.cs.smu.ca/tmp/spintro.json
- Run first 2 cells



First Spark program: word count (cont-d)



First Spark program: word count (cont-d)

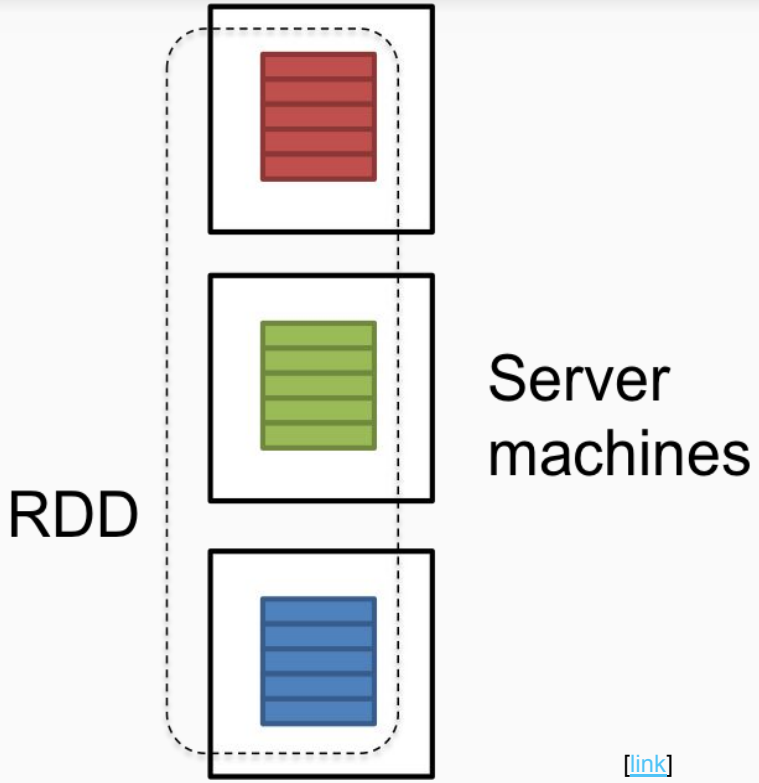
- **spark** is a pre-defined object that stores SparkSession
- we are getting **SparkContext** from **SparkSession**
- Using **textFile** method of SparkSession, we are reading a text file from HDFS into **RDD**
- Then, we are performing map and reduce transformations and getting a new RDD with name **counts**
- take(10) method returns the first 10 elements of the resulting RDD
- The code is written in Scala programming language

What is RDD (Resilient Distributed Dataset)?

- **Resilient** - immutable and fault tolerant
- **Distributed** - distributed across cluster nodes
- **Dataset** - dataset

It's a core abstraction of Spark. From user perspective you can work with RDDs as with **collections of objects**

Java 8 Streams have very similar semantics from user's perspective



[link]

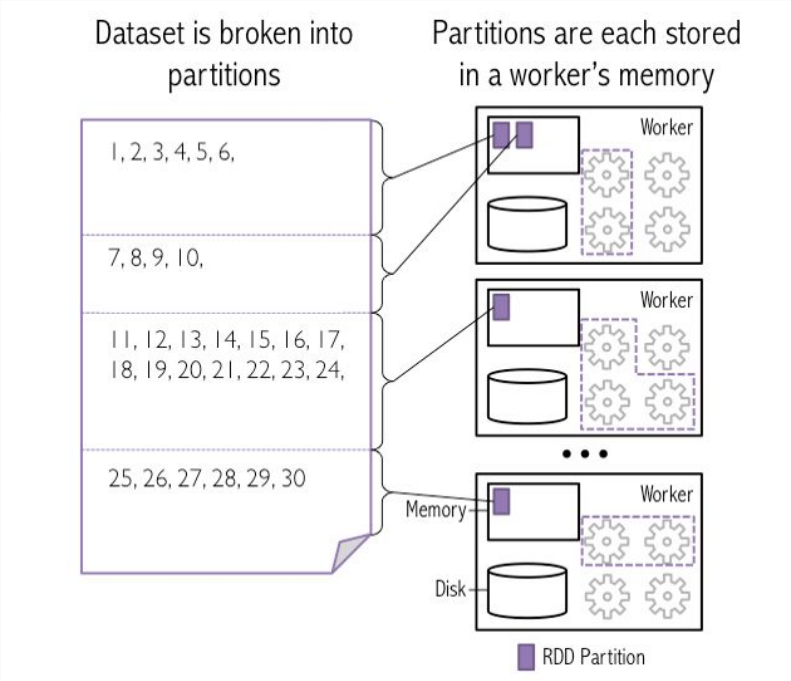
What is RDD? (cont-d)

Main properties:

- Stored **in memory** (however, can use HDD in case of RAM shortage)
- Divided by **Partitions**
- Supports two types of operations: **Transformations** and **Actions**
- Has a **Lineage** (log of transformations used to build a dataset) - for fault tolerance
- **DAG** is used for computations

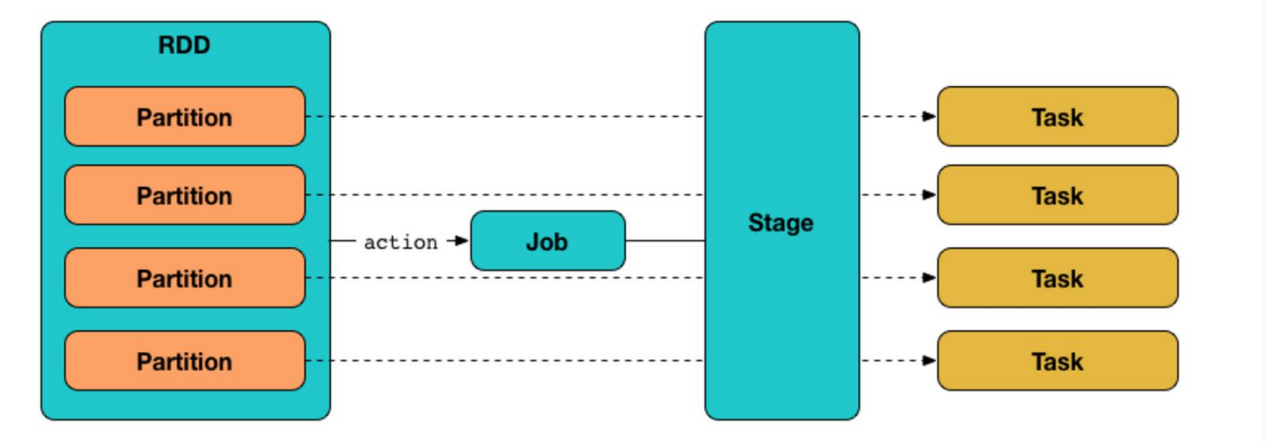
RDD Partitions

- RDD holds references to partitions
- Each partition is a subset of the dataset
- Partitions are stored on cluster nodes (one node can store multiple partitions)
- Each partition is stored by default in RAM



[link]

RDD Partitions (cont-d)



[link]

There is a **task** created for each partition, so number of partitions should generally be equal to at least 1-4 times number of CPU cores/threads in your Spark cluster for the best performance. Initial partition size depends on file size (if we work with HDFS) and on *default parallelism*

RDD Partitions: practice

- Run the 3rd cell in your Zeppelin notebook:

```
%spark2.spark
sc.defaultParallelism
textFile.partitions.size
counts.partitions.size
val rep_counts = counts.repartition(8)
rep_counts.partitions.size
textFile.partitions(1)
counts.partitions(1)
```

```
res172: Int = 2
res173: Int = 2
res174: Int = 2
rep_counts: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[118] at repartition at <console>:25
res175: Int = 8
res176: org.apache.spark.Partition = org.apache.spark.rdd.HadoopPartition@101c
res177: org.apache.spark.Partition = org.apache.spark.rdd.ShuffledRDDPartition@1
```

Took 1 sec. Last updated by anonymous at February 21 2019, 12:05:29 PM. (outdated)

RDD Partitions: practice (cont-d)

- On the current machine, default parallelism is 2, so our textFile RDD is created with 2 partitions: that's a minimum number of partitions with this setting
- We can use repartition method to change number of partitions
- textFile RDD's partitions have type "HadoopPartition" as they were created from HDFS file
- counts RDD's partitions have type "ShuffledRDDPartition" as a result of transformations

RDD Partitions: practice (cont-d)

- Run the next 2 cells to create a **huge.txt** file which is 10 times bigger than big.txt and see that this time Spark distributed it among 5 partitions

```
%sh
hadoop fs -get /__dsets/textdata/big.txt
for i in {1..100}
do
  cat big.txt >> huge.txt
done
hadoop fs -put huge.txt /__dsets/textdata/
rm huge.txt
hadoop fs -du -h /__dsets/textdata/
```

```
6.2 M    /__dsets/textdata/big.txt
618.8 M  /__dsets/textdata/huge.txt
1.4 M    /__dsets/textdata/out
1.4 M    /__dsets/textdata/out.txt
```

Took 12 sec. Last updated by anonymous at February 21 2019, 12:34:32 PM.

```
val hugeTxt = sc.textFile("/__dsets/textdata/huge.txt")
hugeTxt.partitions.size
```

```
hugeTxt: org.apache.spark.rdd.RDD[String] = /__dsets/textdata/huge.txt MapPartitionsRDD[124] at textFile at <console>:25
res180: Int = 5
```

Took 0 sec. Last updated by anonymous at February 21 2019, 12:34:36 PM.

Exercise

Try to run word count in the next cell for the **huge.txt** file and take 10 first elements of the RDD with:

- Default number of partitions
- 8 partitions
- 256 partitions

- How many partitions are there by default?
- Is there any difference in time when running with different number of partitions? Why?
- Can you distribute data among less number of partitions?

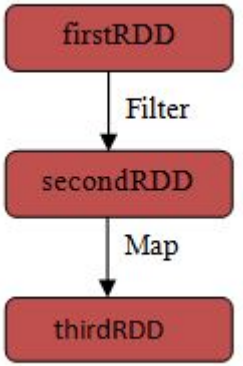
RDD Lineage

- Each RDD keeps track of its parent RDD - it allows Spark to improve fault tolerance - if one of the RDDs is lost, it can be restored from its parent(s)
- toDebugString method can be used to print lineage
- Lineage is also a *logical execution plan*

```
hugeCounts.toDebugString

res211: String =
(128) ShuffledRDD[224] at reduceByKey at <console>:27 []
  +-(128) MapPartitionsRDD[223] at map at <console>:26 []
      |   MapPartitionsRDD[222] at flatMap at <console>:25 []
      |   /__dsets/textdata/huge.txt MapPartitionsRDD[190] at textFile at <console>:25 []
      |   /__dsets/textdata/huge.txt HadoopRDD[189] at textFile at <console>:25 []

Took 36 sec. Last updated by anonymous at February 21 2019, 1:08:02 PM.
```



[link]

Transformations vs Actions

We can perform two types of operations on RDD: **transformations** and **actions**

- Transformations are based on **lazy evaluations**. They **declare** how data should be *transformed*. The *result* of transformation is *another RDD*. Transformations are *intermediate* operations.
- Actions are actual computations. They **apply all transformations** on RDD and perform the “action” to *get results*. Result is either written on the file system or returned to Driver. The result of action is an actual value/file/collection/etc. Actions are *terminal* operations.

Transformations vs Actions (cont-d)

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

[link]

Exercise

Find transformations and actions in the word count example code:

```
%spark2.spark
val sc = spark.sparkContext
val textFile = sc.textFile("/__dsets/textdata/big.txt")
textFile.take(5)
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)
counts.take(10)
```

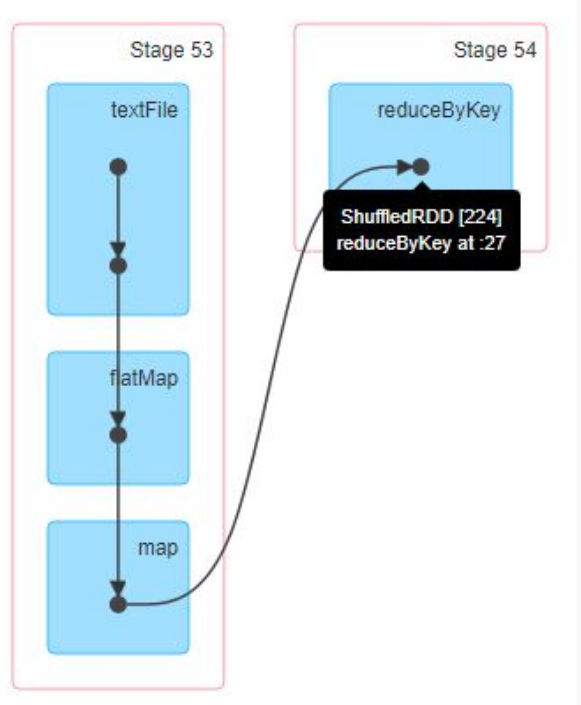
Took 3 sec. Last updated by anonymous at March 08 2019, 10:03:15 AM.

Why transformations are evaluated **lazily**? Maybe it would be better to make all transformations terminal operations?

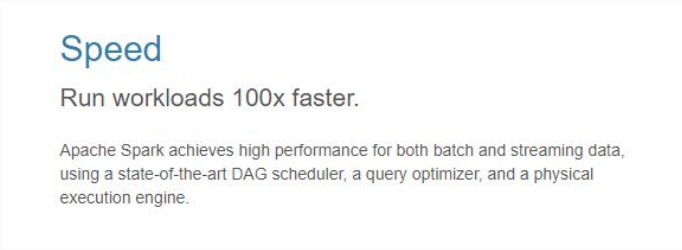
Spark: DAG (Direct Acyclic Graph)

- Spark uses DAG model for computations (as *Apache Tez* in Hadoop)
- DAG of Apache Spark is a set of Vertices and Edges. Vertices are RDDs, Edges are operations
- You may go to http://sandbox-hdp.hortonworks.com:4040 and see DAGs of all your jobs
- Spark uses DAGScheduler to transform a logical execution plan (lineage) to physical execution plan (set of stages)

  Note that Stage is a set of ***parallel tasks*** (one task per partition)



Spark vs Hadoop



[link]

Not Really: it's 100x faster than MapReduce, but not Tez

- Generally, both Spark and Hadoop may **co-exist**: Spark uses YARN and HDFS, can work with Hive and HBase
- Spark is better suitable for **interactive queries**
- Spark mostly replaced Hadoop in ETL (most users switch from Pig to Spark)
- Hadoop is good for batch jobs
- Machine learning: Spark MLib is significantly faster than Apache Mahout for Hadoop
- Spark is suitable for Streaming

Scala

- Scala is a general-purpose programming language providing support for functional programming and a *strong static type system*
- Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine
- Scala provides language interoperability with Java, so that libraries written in either language may be referenced directly in Scala or Java code [l]
- **Spark is written in Scala**
- Although you may use Python or R with Spark, Scala provides some additional benefits

[link]

Scala (cont-d)

Basic syntax features: **var** vs **val**

```
val x = 1 //constant
var y = 1 //variable
y = 5
x = 5
```

```
x: Int = 1
y: Int = 1
y: Int = 5
<console>:25: error: reassignment to val
       x = 5
         ^
```

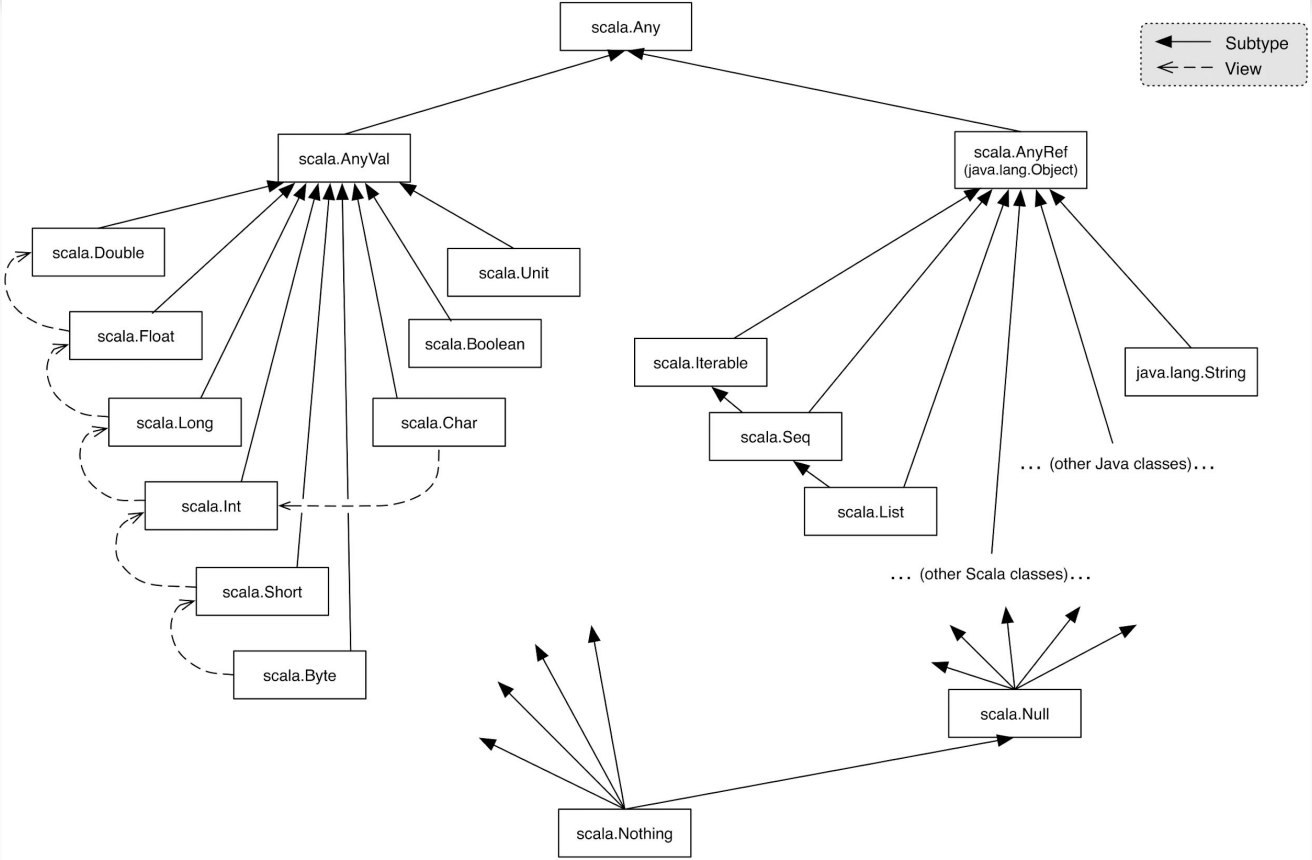
Scala (cont-d)

Basic syntax features: **explicit type declaration**

```
var x:Int = 1
var y:Double = 2.0
print(x+y)
```

```
x: Int = 1
y: Double = 2.0
3.0
```

Scala (cont-d)



[link]

Scala (cont-d)

Basic syntax features: **if else**

```
var flag = true
var flag2 = false
if (flag && !flag2) {
    println("OK" + " " + flag)
} else {
    print("NOK")
}
```

```
flag: Boolean = true
flag2: Boolean = false
OK true
```

Scala (cont-d)

Basic syntax features: **for loops**

```
for (a <- 1 to 3){
    println(a)
}
```

```
1
2
3
```

```
var res = for { a <- 1 to 10 if a % 2 == 0 } yield a*a
//same as filter + map
```

```
res: scala.collection.immutable.IndexedSeq[Int] = Vector(4, 16, 36, 64, 100)
```

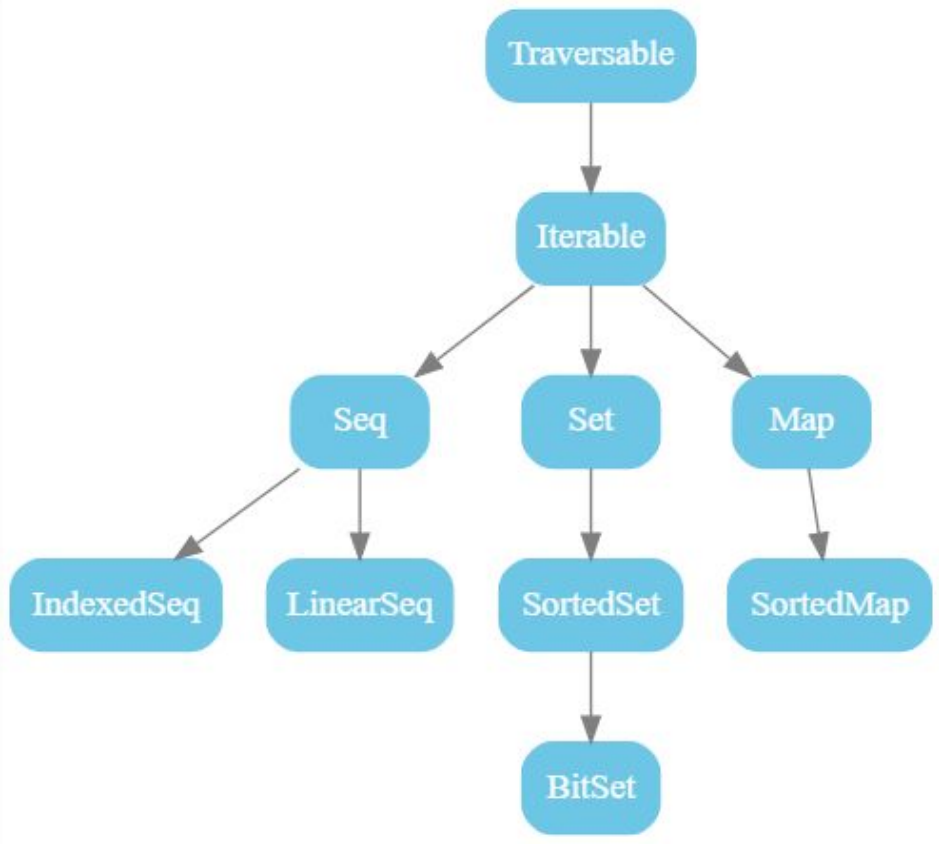
Scala (cont-d)

Basic syntax features: **functions**

```
def concat( s1:String, s2:String, sep:String = " " ) : String = {
    return s1+sep+s2
}
print(concat("Hello", "World"))
```

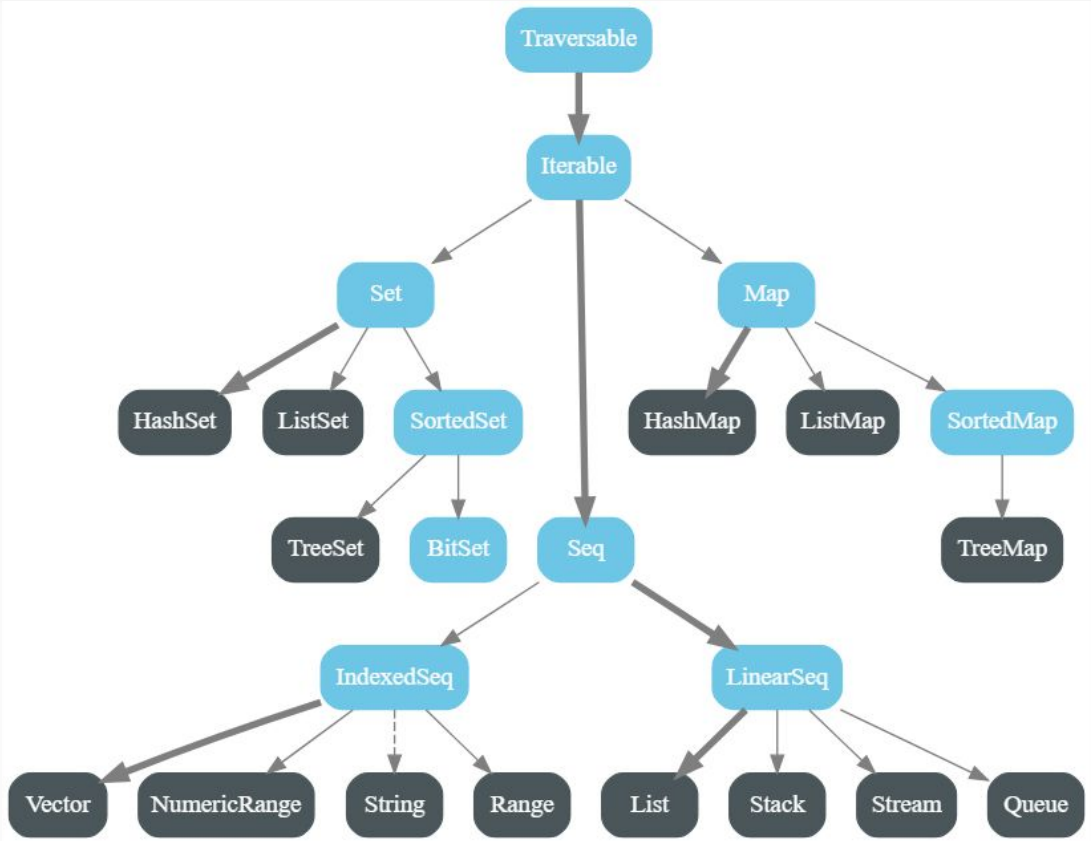
```
concat: (s1: String, s2: String, sep: String)String
Hello World
```

Scala: collections



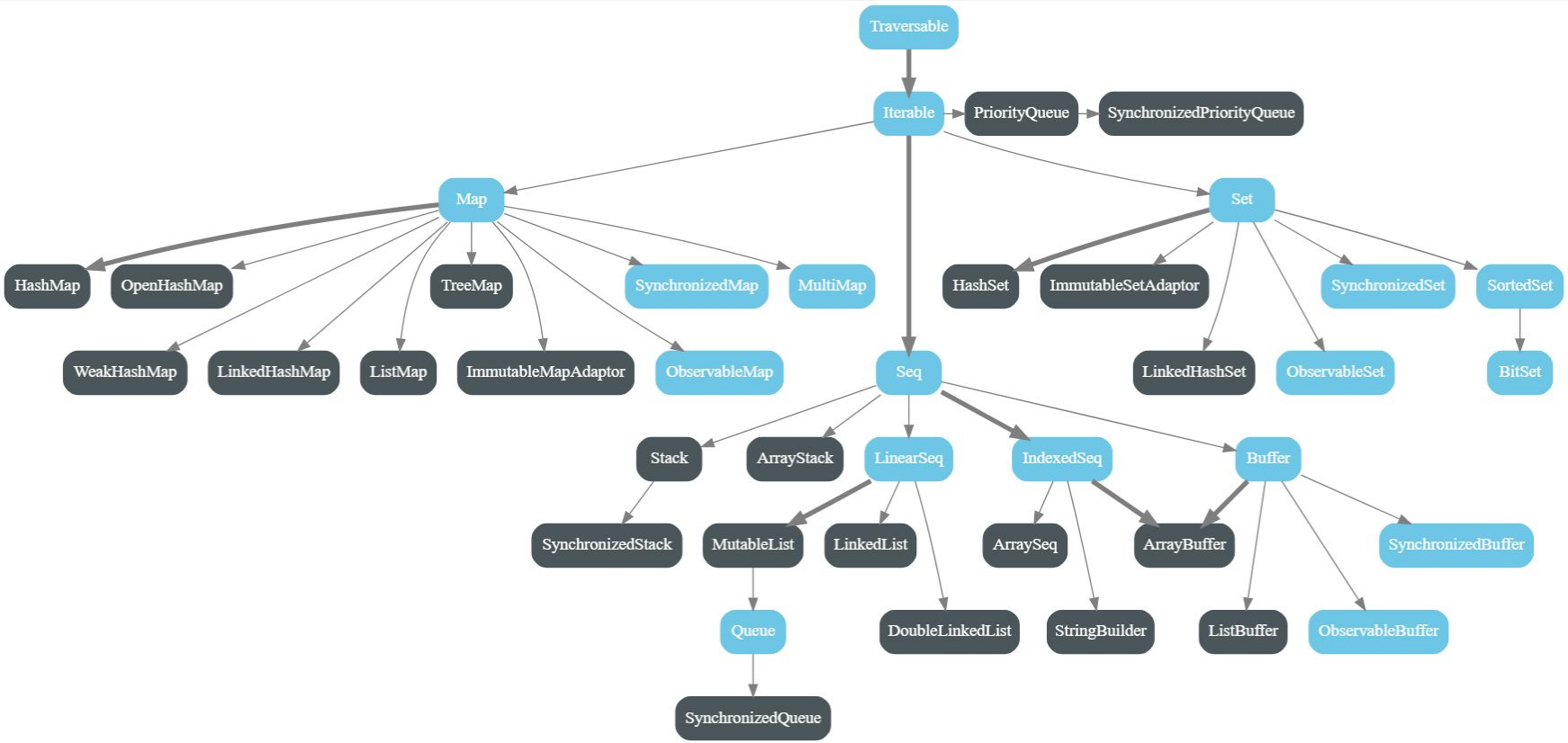
[link]

Scala: collections: immutable



[link]

Scala: collections: mutable



[link]

Scala: collections: syntax

```
val tuple = (10, "hi", 30)
println(tuple._1 + tuple._2)
val m1 = Map("x" -> 24, "y" -> 25, "z" -> 26)
println(m1("x"))
val lst1 = List(10, 20, 30)
//lst1(0) = 100 //will give us an error
println(lst1(0))
val arr1 = Array(10, 20, 30)
arr1(0) = 100
println(arr1(0))
val rang = (1 to 5)
println(rang(0))
```

```
tuple: (Int, String, Int) = (10,hi,30)
10hi
m1: scala.collection.immutable.Map[String,Int] = Map(x -> 24, y -> 25, z -> 26)
24
lst1: List[Int] = List(10, 20, 30)
10
arr1: Array[Int] = Array(10, 20, 30)
100
rang: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
1
```

Scala: collections: syntax (mutable)

```
import scala.collection.mutable
val map1 = Map("x" -> 24, "y" -> 25, "z" -> 26)
map1("a") = 100
map1("x") = 0
println(map1)
val mlist = ListBuffer(10, 20, 30)
mlist += 100
print(mlist)
```

```
import scala.collection.mutable
map1: scala.collection.mutable.Map[String,Int] = Map(z -> 26, y -> 25, x -> 24)
Map(z -> 26, y -> 25, a -> 100, x -> 0)
mlist: scala.collection.mutable.ListBuffer[Int] = ListBuffer(10, 20, 30)
res373: mlist.type = ListBuffer(10, 20, 30, 100)
ListBuffer(10, 20, 30, 100)
```

Scala: collections to RDD

You can convert a Scala collection to Spark RDD with **parallelize** method of SparkContext object:

```
val pt = sc.parallelize(lst1)
pt.partitions.size
pt.collect()
```

```
pt: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[538] at parallelize at <console>:64
res483: Int = 2
res484: Array[Int] = Array(10, 20, 30)
```

Scala: functional programming

```
val mlist = ListBuffer(10, 20, 30, 100)
mlist.
    filter(a => a < 100).
    map(a => a*a).
    foreach(println(_))
```

```
mlist: scala.collection.mutable.ListBuffer[Int] = ListBuffer(10, 20, 30, 100)
100
400
900
```

Scala: functional programming (cont-d)

Don’t confuse **Scala**’s filter/map/reduce/etc. with **Spark** RDD’s methods with the same name

```
val mlist = ListBuffer(10, 20, 30, 100)
val sumsq = mlist.
    filter(a => a < 100).
    map(a => a*a).
    reduce(_ + _)
print(sumsq)
```

```
mlist: scala.collection.mutable.ListBuffer[Int] = ListBuffer(10, 20, 30, 100)
sumsq: Int = 1400
1400
```

Scala: functional programming (cont-d)

Don't confuse **Scala**'s filter/map/reduce/etc. with **Spark** RDD's methods with the same name. **Note that _ symbol is used as a placeholder for parameters**

```
val mlist = ListBuffer(10, 20, 30, 100)
val sumsq = mlist.
    filter(a => a < 100).
    map(a => a*a).
    reduce(_ + _)
print(sumsq)
```

```
mlist: scala.collection.mutable.ListBuffer[Int] = ListBuffer(10, 20, 30, 100)
sumsq: Int = 1400
1400
```

Scala: functional programming (cont-d)

Same code without a placeholder

```
val mlist = ListBuffer(10, 20, 30, 100)
val sumsq = mlist.
    filter(a => a < 100).
    map(a => a*a).
    reduce((a, b) => a + b)
print(sumsq)
```

```
mlist: scala.collection.mutable.ListBuffer[Int] = ListBuffer(10, 20, 30, 100)
sumsq: Int = 1400
1400
```

Scala: functional programming (cont-d)

Regex example

```
val words = List("frog", "hog", "spark", "program")
words.
    filter(w => w matches "^.+og$").
    foreach(println(_))
```

```
words: List[String] = List(frog, hog, spark, program)
frog
hog
```

Exercise

Modify our example word count program:

- You should print the top 20 most used words (hint: use **sortBy** method to sort key-value pairs by value)
- The words should contain letters only (no dashes, underscores, etc.)
- Length of the words should be at least 2 letters
- The following words should be ignored:
  "the","of","and","to","in","that","was","his","he","with","is","as","had","it","by","for","at","not","The","be", "on", "or", "an", "my", "in", "so", "if", "but", "has", "had", "have", "were", "from", "this", "when", "been", "into", "will", "very"," then", "are", "there", "would", "only", "she", "their", "what", "his", "her"

Hint: you may use subtract method to subtract one RDD from another

PySpark

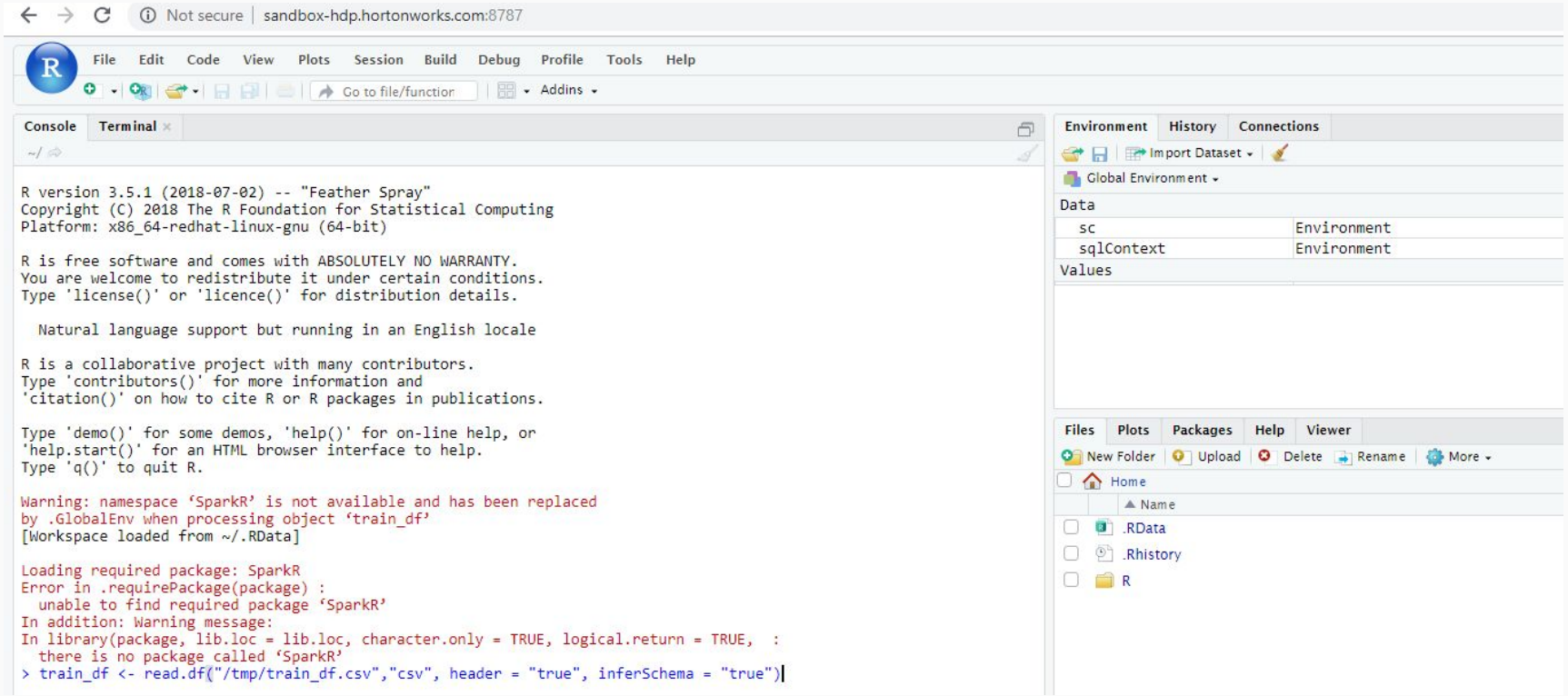
If you don’t like Scala, you can work with Spark from **Python**:

```
%spark2.pyspark
sc = spark.sparkContext
text_file = sc.textFile("/__dsets/textdata/big.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.take(10)
##counts.saveAsTextFile("/__dsets/textdata/out")
```

```
[(u'', 69285), (u'gag', 1), (u'worn,"', 1), (u'"Fool,', 1), (u'Dartmouth,', 1), (u'34.--Tuberculous', 1), (u'"F
vice--the', 1)]
```

SparkR

Or from **R**:



Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

[link]

Spark Streaming (cont-d)

Spark Streaming divides input data into **batches** (also called microbatches) which are then processed by the Spark engine

[link]

Spark Streaming (cont-d)

Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. Internally, a DStream is represented as a sequence of RDDs.

Any operation applied on a DStream translates to operations on the underlying RDDs

[link]

Try it yourself exercise

- Connect to web console of your sandbox instance: http://sandbox-hdp.hortonworks.com:4200/
- wget http://wolly.cs.smu.ca/tmp/sender.py
- wget http://wolly.cs.smu.ca/tmp/big.txt

Sender is a demo program that emulates behaviour of streaming data source. It listens on port 12345. When some client is connected to the port, it starts sending data (pieces of a text file) by chunks with interval 1 second.

Run:

python sender.py big.txt 200

Where big.txt is a text file to read, 200 is size of chunks (you can set your own)

```
 GNU nano 2.5.3          File: sender.py

import socket, time, sys


f=open(sys.argv[1], "r")
offset = int(sys.argv[2])
content = f.read()
s = socket.socket()
port = 12345
s.bind(('', port))
s.listen(5)
p, addr = s.accept()
print "Client connected from", addr

idx = 0
leng = len(content)
while True:
    time.sleep(1)

    if idx+offset > leng:
        p.send(content[idx:])
        idx = 0

    p.send(content[idx: idx+offset])
    idx += offset

p.close()
```

Try it yourself exercise (cont-d)

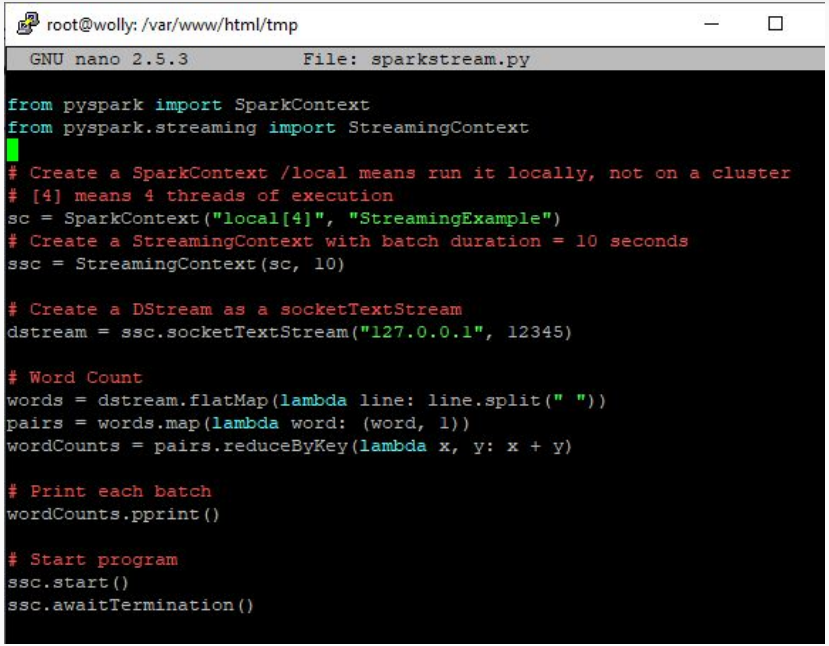
In the same console and the same folder:

wget http://wolly.cs.smu.ca/tmp/sparkstream.py

It's a sample WordCount program that creates a dstream and reads data from a socket.

How to use:

spark-submit sparkstream.py



Try it yourself exercise (cont-d)

Not secure | sandbox-hdp.hortonworks.com:4200

```
19/02/26 18:32:40 INFO DAGScheduler: waiting: Set(ResultStage 16)
19/02/26 18:32:40 INFO DAGScheduler: failed: Set()
19/02/26 18:32:40 INFO DAGScheduler: Submitting ResultStage 16 (PythonRDD[44] at RDD at PythonRDD.scala:48), which has no missing parents
19/02/26 18:32:40 INFO MemoryStore: Block broadcast_12 stored as values in memory (estimated size 7.0 KB, free 366.1 MB)
19/02/26 18:32:40 INFO MemoryStore: Block broadcast_12_piece0 stored as bytes in memory (estimated size 4.0 KB, free 366.1 MB)
19/02/26 18:32:40 INFO BlockManagerInfo: Added broadcast_12_piece0 in memory on sandbox-hdp.hortonworks.com:36311 (size: 4.0 KB, free: 366.2 MB)
19/02/26 18:32:40 INFO SparkContext: Created broadcast 12 from broadcast at DAGScheduler.scala:1039
19/02/26 18:32:40 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 16 (PythonRDD[44] at RDD at PythonRDD.scala:48) (first 15 tasks are for partit
19/02/26 18:32:40 INFO TaskSchedulerImpl: Adding task set 16.0 with 1 tasks
19/02/26 18:32:40 INFO TaskSetManager: Starting task 0.0 in stage 16.0 (TID 33, localhost, executor driver, partition 0, ANY, 7649 bytes)
19/02/26 18:32:40 INFO Executor: Running task 0.0 in stage 16.0 (TID 33)
19/02/26 18:32:40 INFO ShuffleBlockFetcherIterator: Getting 1 non-empty blocks out of 1 blocks
19/02/26 18:32:40 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 1 ms
19/02/26 18:32:40 INFO PythonRunner: Times: total = 46, boot = -19959, init = 20003, finish = 2
19/02/26 18:32:40 INFO PythonRunner: Times: total = 52, boot = -19969, init = 20021, finish = 0
19/02/26 18:32:40 INFO Executor: Finished task 0.0 in stage 16.0 (TID 33). 1785 bytes result sent to driver
19/02/26 18:32:40 INFO TaskSetManager: Finished task 0.0 in stage 16.0 (TID 33) in 74 ms on localhost (executor driver) (1/1)
19/02/26 18:32:40 INFO TaskSchedulerImpl: Removed TaskSet 16.0, whose tasks have all completed, from pool
19/02/26 18:32:40 INFO DAGScheduler: ResultStage 16 (runJob at PythonRDD.scala:141) finished in 0.100 s
19/02/26 18:32:40 INFO DAGScheduler: Job 8 finished: runJob at PythonRDD.scala:141, took 0.202717 s
-------------------------------------------
Time: 2019-02-26 18:32:40
-------------------------------------------
(u'', 1)
(u'interests', 1)
(u'old', 1)
(u'some', 1)
(u'activity,', 1)
(u'From', 1)
(u'occupied', 1)
(u'week', 2)
(u'these', 1)
(u'from', 2)
...

19/02/26 18:32:40 INFO JobScheduler: Finished job streaming job 1551205960000 ms.0 from job set of time 1551205960000 ms
19/02/26 18:32:40 INFO JobScheduler: Total delay: 0.289 s for time 1551205960000 ms (execution: 0.224 s)
```

Try it yourself exercise: try to save as TextFiles

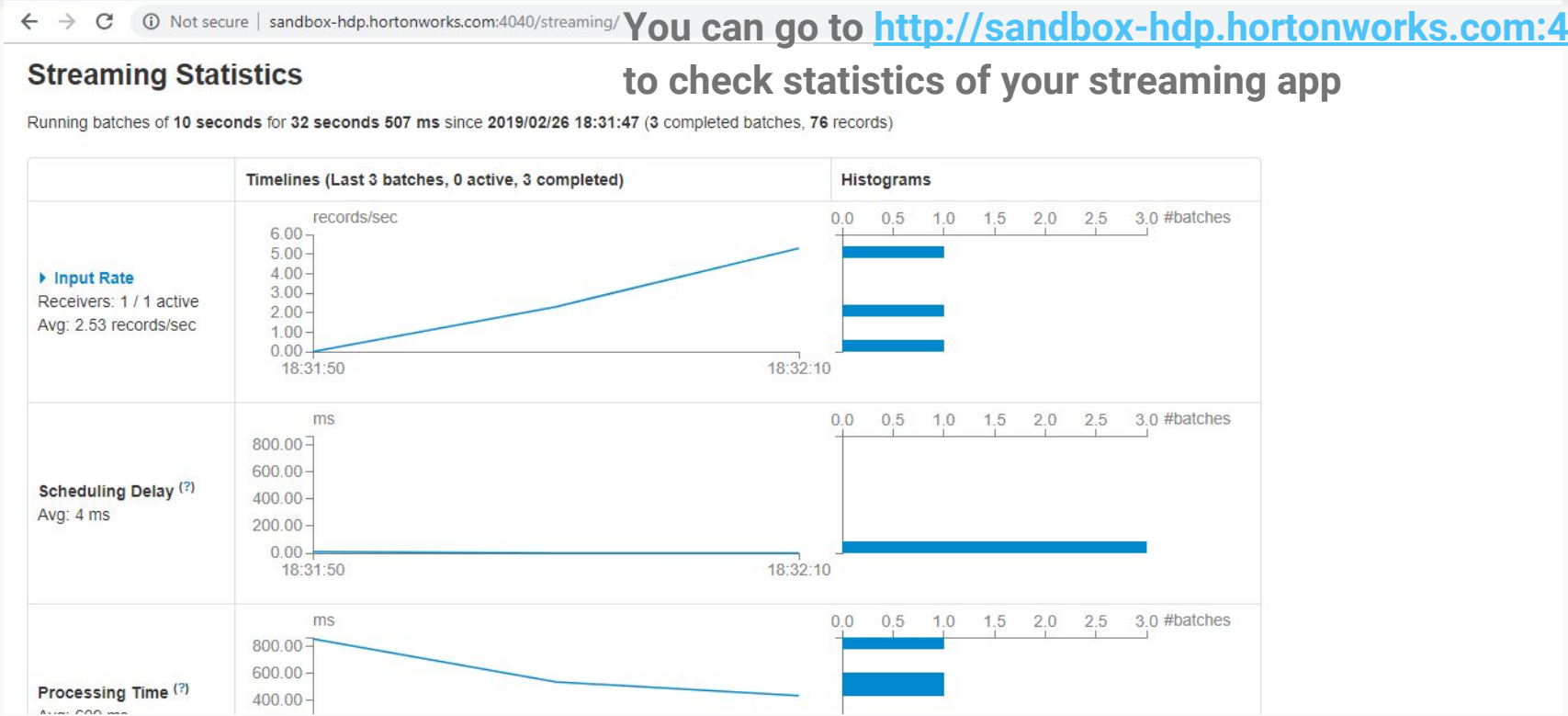
Output Operations on DStreams

Output operations allow DStream's data to be pushed out to external systems like a database or a file systems. Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs). Currently, the following output operations are defined:

Output Operation	Meaning
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as `SequenceFiles` of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

[link]

Try it yourself exercise (cont-d)



You can go to http://sandbox-hdp.hortonworks.com:4040/ to check statistics of your streaming app