EPL660: Information Retrieval and Search Engines – Lab 9

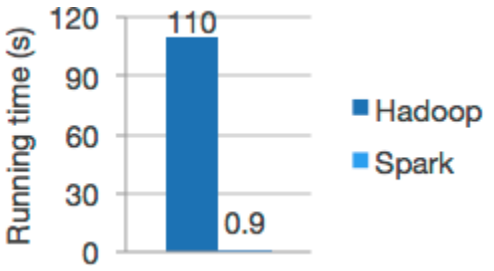
University of Cyprus
Department of Computer Science

Παύλος Αντωνίου

Γραφείο: B109, ΘΕΕ01

Introduction to Apache Spark

- Fast and general **engine for large-scale data processing** on clusters
  - claim to run programs up to 100x faster than Hadoop MapReduce for in-memory analytics, or 10x faster on disk.



- Developed in the AMPLab at UC Berkeley
  - Started in 2009
- Open-sourced in 2010 under a BSD license
- Proven scalability to over 8000 nodes in production

Spark vs Hadoop

- **Spark** is an in-memory distributed processing engine

- **Hadoop** is a framework for distributed storage (**HDFS**) and distributed resource management and job scheduling (**YARN**) and distributed processing (**Map/Reduce**)

- *Spark can run with (by default) or without Hadoop components (HDFS/YARN)*

Storage in Spark

- Distributed Storage Options:
  - Local filesystem (non distributed)
  - **Hadoop HDFS** – Great fit for batch (offline) jobs.
  - **Amazon S3** – For batch jobs. Commercial.
  - **Apache Cassandra** (DB) – Perfect for streaming data analysis (time series) and an overkill for batch jobs.
  - **Apache HBase** (DB)
  - **MongoDB** (DB)

- Cassandra vs Hbase vs MongoDB: http://db-engines.com/en/system/Cassandra%3BHBase%3BMongoDB

Job Scheduling in Spark

- Distributed Resource Management & Job Scheduling Options:
  - **Standalone**: simple cluster manager included with Spark that makes it easy to set up a cluster
  - **Hadoop YARN**: the resource manager in Hadoop
  - **Apache Mesos**: a general cluster manager that can also run Hadoop MapReduce and service applications
- Hadoop vs Spark

Key points about Spark

- Runs on both Windows and UNIX-like systems
- Provides high-level APIs in Java, Scala, Python and R
- Supports rich set of higher-level tools including SQL and Dataframes for SQL and structured data processing (e.g. json files), Mllib for machine learning, GraphX for graph processing, and Spark Streaming for stream processing of live data streams (e.g. sources: TCP/IP sockets, Twitter…)

Running Apache Spark

- Apache Spark 3.0.0 is installed on your Virtual Machine
- Start Spark Shell:
  - cd /usr/local/spark/bin
  - Python Shell:
    - `./pyspark` ⟶

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.0
      /_/

Using Python version 3.7.8 (default, Jun 29 2020 05:46:05)
SparkSession available as 'spark'.
```

  - Scala Shell
    - `./spark-shell`
  - R Shell
    - `./sparkR`
  - Submit an application written in a file
    - `./spark-submit --master local[2] SimpleApp.py`
  - Run ready-made examples
    - `./run-example <class> [params]`

Spark Essentials: SparkContext

- First thing that a Spark program does in order to access a cluster is to create a ***SparkContext*** object
- In the shell for either Scala or Python, this is the ***sc*** variable, which is created automatically
- In your programs, you must use a constructor to instantiate a new ***SparkContext***
- Then in turn ***SparkContext*** gets used to create other variables

Hands on – sparks-shell / pyspark

- From the “scala>” REPL prompt, type:

```
scala> sc
res0: org.apache.spark.SparkContext =
org.apache.spark.SparkContext@10b87ff6
```

- From the python “>>>” interpreter, type:

```
>>> sc
<SparkContext master=local[*]
appName=PySparkShell>
```

Spark Essentials: SparkSession

- From Spark 2.0 onwards, SparkSession is introduced
- In your programs, you must use a SparkSession.builder to instantiate a new ***`SparkSession`*** object
- In the shell for either Scala or Python, this is the ***`spark`*** variable, which is created automatically
- No need to create ***`SparkContext`***, since ***`SparkSession`*** encapsulates the same functionalities

Hands on – sparks-shell / pyspark

- From the “scala>” REPL prompt, type:

```
scala> spark
res0:
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@49c8
3262
```

- From the python “>>>” interpreter, type:

```
>>> spark
<pyspark.sql.session.SparkSession
object at 0x7f5e68293be0>
```

Spark Essentials: Master

- The ***master*** parameter determines which cluster to use

e.g. `./spark-submit --master local[2] SimpleApp.py`

master	description
local	run Spark locally with one worker thread (no parallelism)
local[K]	run Spark locally with K worker threads (ideally set to # cores)
spark://HOST:PORT	connect to a Spark standalone cluster manager; PORT depends on config (7077 by default)
mesos://HOST:PORT	connect to a Mesos cluster manager; PORT depends on config (5050 by default)

Hands on - change master param

- Through a program (via SparkContext object)
  - To create a SparkContext object you first need to build a SparkConf object that contains information about your application.
  - Scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
val conf = new SparkConf().setAppName("MyApp").setMaster("local[4]")
val sc = new SparkContext(conf)
```

  - Python

```
from pyspark import SparkContext, SparkConf
sconf = SparkConf().setAppName("MyApp").setMaster("local[4]")
sc = SparkContext(conf=sconf)
```

- Command line (initiate shell with 4 worker threads):

```
./bin/spark-shell --master local[4]
./bin/pyspark --master local[4]
```

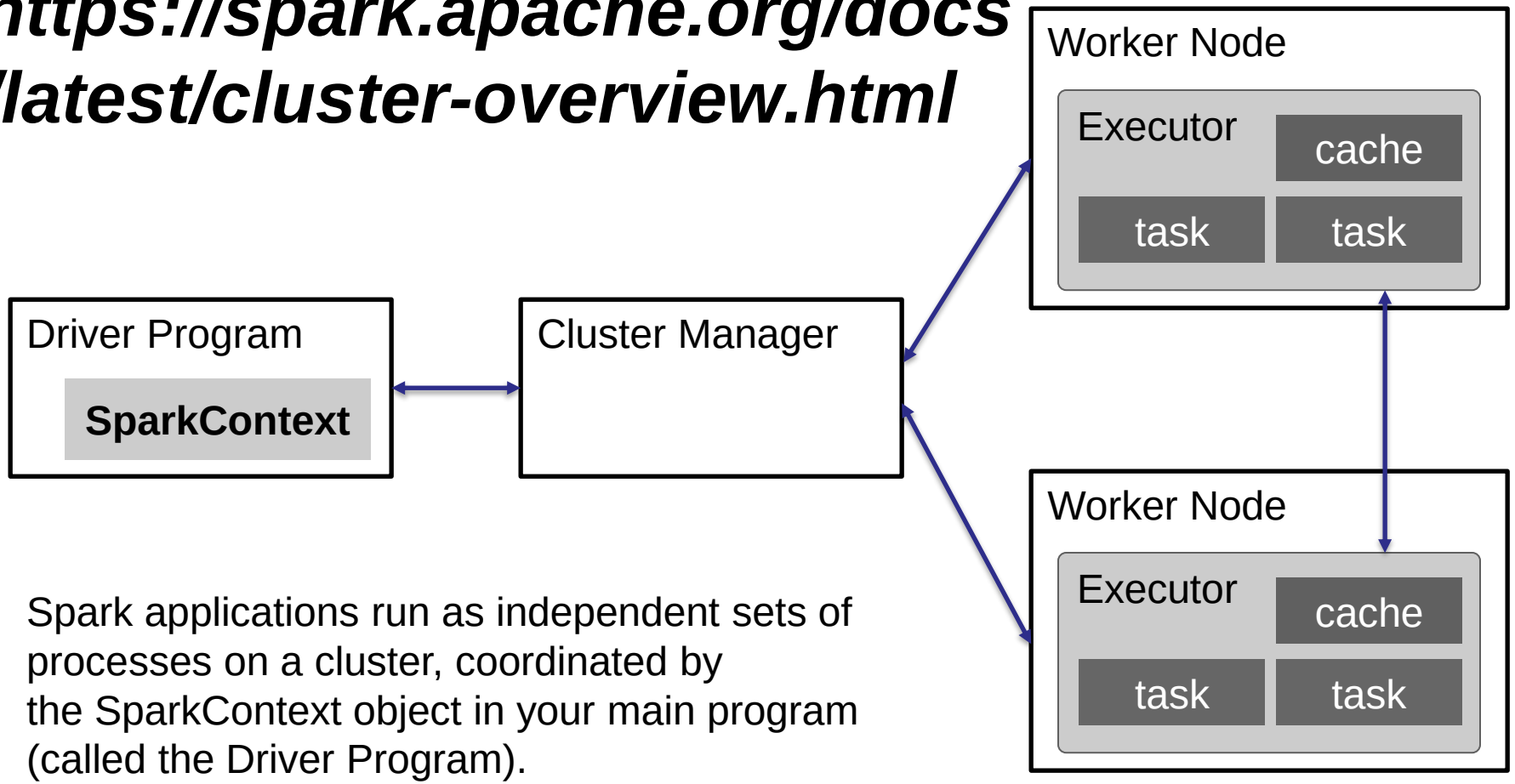
Spark Essentials

- Spark application consists of a *driver program* that runs the user's main function and executes various *parallel operations* on a cluster

- Main abstractions on a cluster:
  - *resilient distributed dataset* (RDD)*s*
    - Read-only distributed collection of elements (e.g. lists, text) that can be run in parallel (parallel processing) on many devices. Each record in the RDD can be divided into logical parts and then executed on different nodes of the cluster.
  - *shared variables* that can be used in parallel operations
  - Find more here

Spark Essentials: Run on cluster

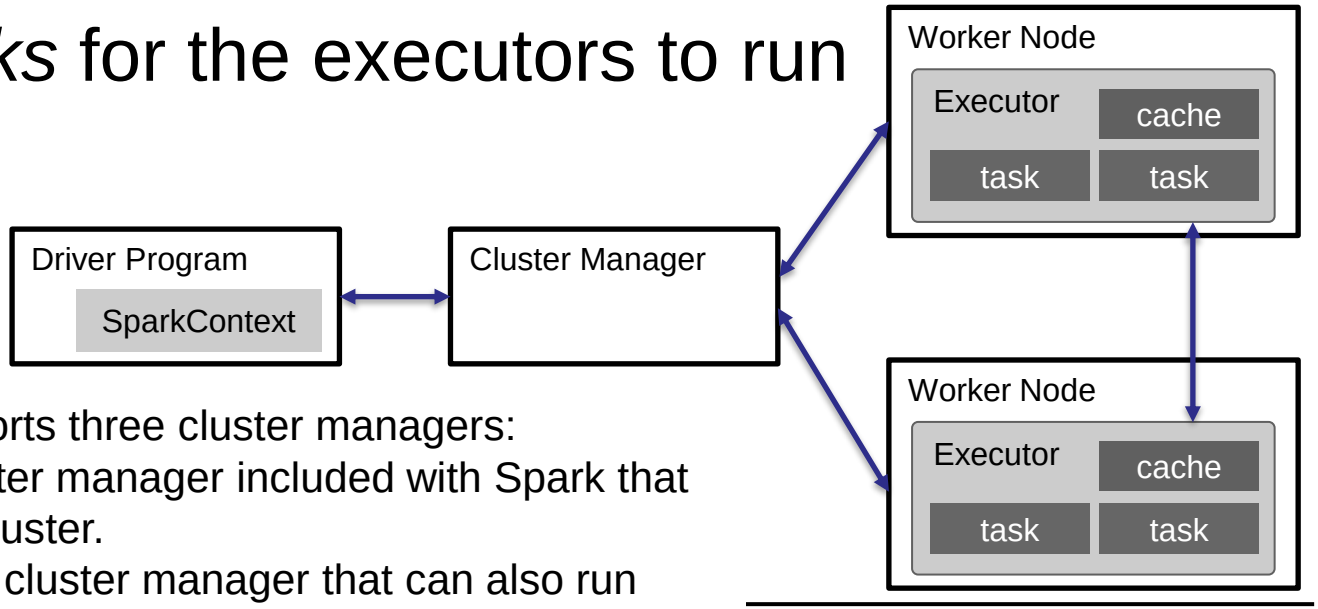
https://spark.apache.org/docs/latest/cluster-overview.html



Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the Driver Program).

Spark Essentials: Run on clusters

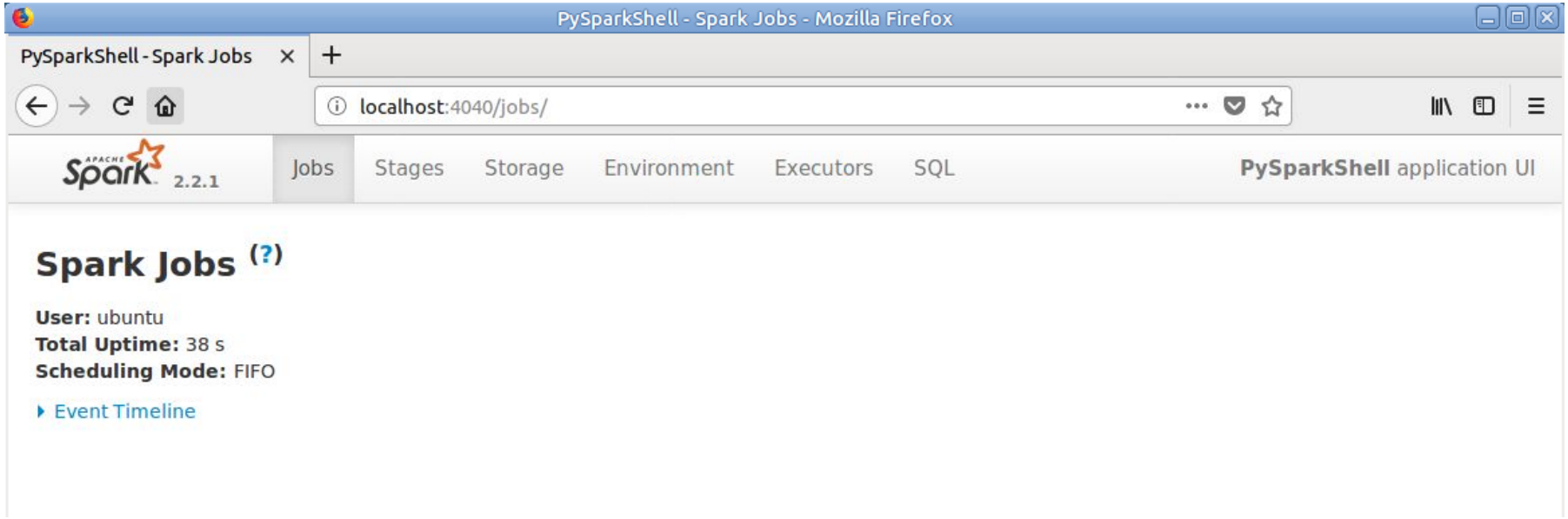
1. connects to a *cluster manager* which allocates resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



The system currently supports three cluster managers:
Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
Hadoop YARN – the resource manager in Hadoop.

App Monitoring: Spark Web UI

- Every SparkContext launches a Web UI, typically on port 4040, to display information about running tasks, executors, and storage usage
- In browser type `driver-node-ipaddr:4040`
  - In a single node cluster driver-node-ipaddr is localhost



Spark Essentials: RDD

- **R**esilient **D**istributed **D**atasets (RDDs) **distributed collection of elements** that can be operated on in parallel
- There are currently two types:
  - **parallelized collections** – created by Scala collections or Python iterables or collections (e.g. lists)
  - **Hadoop datasets** – created by files (text files, sequence files, InputFormat files) stored in HDFS
- RDD objects are immutable
  - Primarily for high-speed gains

Spark Essentials: RDD

- There exist two types of *operations* on RDDs: **transformations** and **actions**
- transformations are lazy: do not compute their results immediately
  - example transformations: map(), filter()
- transformations only computed when an action requires a result to be returned to driver
  - transformed RDDs recomputed each time an action runs on them
  - Example actions: reduce(), collect(), count()

Spark Essentials: RDD

- RDD can be persisted into storage in memory or disk

- Transformations create a new RDD dataset from an existing one

Spark Essentials: Transformations & Actions

- A full list with all the supported transformations and actions can be found on the following links

http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Hands on – parallelized collections

- From the python ">>>" interpreter, let's create a collection (list) holding the numbers 1 to 5:

```
data = [1, 2, 3, 4, 5]
```

- then create an RDD (parallelized collection) based on that data that can be operated on in parallel:

```
distData = sc.parallelize(data)
```

- finally use a *filter* transformation to select values less than 3 and then action *collect* RDD contents back to driver

```
distData.filter(lambda s: s<3).collect()
```

  - Filter transformation returns a new dataset formed by selecting those elements of the source on which *function* returns true

Hands on – parallelized collections

- Lambda: anonymous functions on runtime
  - Normal function definition: def f (x): return x**2
    - call: f(8)
  - Anonymous function: g = lambda x: x**2
    - call: g(8)

Spark Essentials: RDD

- Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Cassandra, Hypertable, HBase, etc.

- Spark supports text files, SequenceFiles, and any other Hadoop InputFormat, and can also take a directory or a glob (e.g. /data/201404*)

Hands on – pyspark

```
from pyspark import SparkContext, SparkConf
sconf = SparkConf().setAppName("SimpleApp").setMaster("local")
sc = SparkContext(conf=sconf)
# file is a list of lines from a file located on HDFS
file =
sc.textFile("hdfs://localhost:54310/user/ubuntu/input/unixdict
.txt")
# lineLength is the result of a map transformation. Function
len() is applied to each element of file list (i.e. each line)
Not immediately computed, due to laziness.
lineLengths = file.map(lambda line: len(line))
# reduce is an action. At this point Spark breaks the
computation into tasks to run on separate machines; each
machine runs both its part of the map and a local reduction,
returning only its answer to the driver program.
# a is the previous aggregate result and n is the current item
totalLength = lineLengths.reduce(lambda a, n: a + n)
print("The result is : ",totalLength)
```

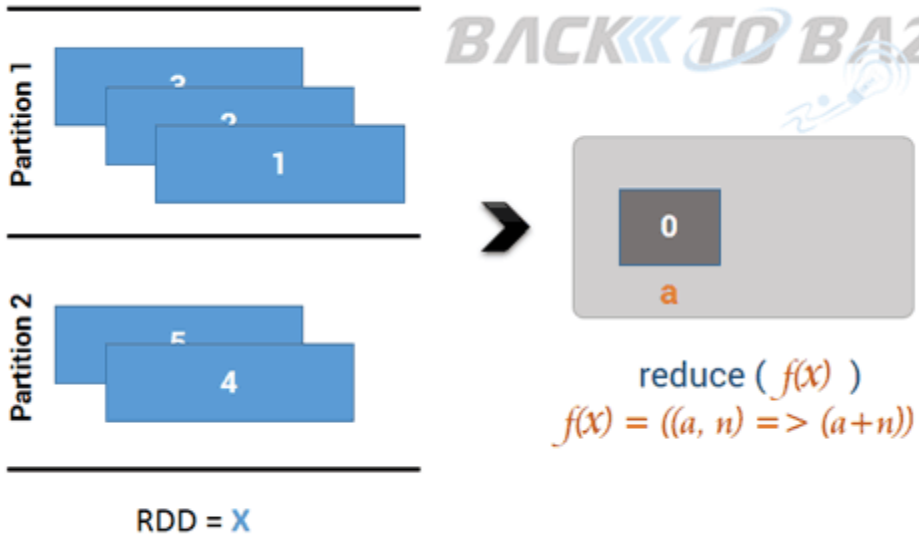
RUN APP on SPARK USING: ./spark-submit SimpleApp.py

Hands on – pyspark

- map() transformation
  - applies the given function on every element of the RDD => returns new RDD representing the results
  - x = [1, 2, 3, 4, 5] # python list
  - par_x = sc.parallelize(x) # distributed RDD
  - result = par_x.map(lambda i : i**2) # new RDD
  - print(result.collect()) → [1, 4, 9, 16, 25]
- reduce() action
  - aggregates all elements of RDD using a given function and returns the final result to the driver program

Hands on – pyspark

```
from pyspark import SparkContext, SparkConf
sconf = SparkConf().setAppName("SimpleApp").setMaster("local")
sc = SparkContext(conf=sconf)
# file is a list of lines from a file located on HDFS
fil
sc.                                                          ıixd
ict
# l                                                          ɔn
ler
Not
lir
# r
com
mac                                                          ı,
ret
# a is the previous aggregate result and n is the current line
totalLength = lineLengths.reduce(lambda a, n: a + n)
print("The result is : ",totalLength)
```



RUN APP on SPARK USING: ./spark-submit --master local SimpleApp.py

Hands on – pyspark

```
from pyspark import SparkContext, SparkConf
sconf = SparkConf().setAppName("SimpleApp").setMaster("local")
sc = SparkContext(conf=sconf)
# file is a list of lines from a file located on HDFS
file =
sc.textFile("hdfs://localhost:54310/user/csdeptucy/input/unixd
ict.txt")
# lineLength is the result of a map transformation. Function
len() i
Not imm
lineLen
# reduc
computa                                                    ch
machine                                                    ction,
returni
# a is the previous aggregate result and b is the current line
totalLength = lineLengths.reduce(lambda a, n: a + n)
print("The result is : ",totalLength)
```

If we also wanted to use `lineLengths` again later, we could add (before reduce):
`lineLengths.persist()` OR
`lineLengths.cache()`
which would cause lineLengths to be saved in memory after the first time it is computed.

RUN APP on SPARK USING: ./spark-submit --master local SimpleApp.py

Spark Essentials: Persistence

- Spark can persist (or cache) an RDD dataset in memory across operations
  - cache() : use only default storage level MEMORY_ONLY
  - persist() : specify storage level (see next slide)
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

Spark Essentials: Persistence

transformation	description
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Spark Essentials: Persistence

- **How to choose Persistence:**
  - If your RDDs fit comfortably with the default storage level (**MEMORY_ONLY**), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
  - If not, try using **MEMORY_ONLY_SER** and selecting a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access.
  - Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, re-computing a partition may be as fast as reading it from disk.
  - Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). All the storage levels provide full fault tolerance by re-computing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to re-compute a lost partition.

More on transformation & actions

- flatMap() transformation
  - same as map but instead of returning just one element per element returns a sequence per element (which can be empty) – flattens the results
- reduceByKey() transformation
  - operation on key-value pairs
    - in Python, key-value pairs can be implemented as tuples
  - aggregates all values having the same key using a given function
  - returns a distributed dataset (RDD)

RDD Word Count Example

- Now lets run word count example:

```
file = sc.textFile("pg4300.txt")
words = file.flatMap(lambda line: line.split(" "))
            .map(lambda word: (word, 1))
            .reduceByKey(lambda a, b: a + b)
words.saveAsTextFile(“output") # output folder on local fs
```

- of course *pg4300.txt* could be located in HDFS as well:

```
file =
sc.textFile("hdfs://localhost:54310/user/csdeptucy/input/pg4
300.txt")
words = file.flatMap(lambda line: line.split(" "))
            .map(lambda word: (word, 1))
            .reduceByKey(lambda a, b: a + b)
words.saveAsTextFile("hdfs://...")
```

Machine Learning on Spark

- **MLlib** is Apache Spark's scalable machine learning library
- Write applications quickly in Java, Scala, Python, and R
- Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk
- Runs on any Hadoop data source (e.g. HDFS, HBase, or local files), making it easy to plug into Hadoop workflows

What is MLlib?

- Classification: logistic regression, linear support vector machine (SVM), naive Bayes …
- Clustering: K-means, Gaussian mixtures …
- Regression: generalized linear regression, survival regression,...
- Decomposition: singular value decomposition (SVD), principal component analysis (PCA)
- Decision trees, random forests, and gradient-boosted trees
- Recommendation: alternating least squares (ALS)
  - based on collaborative filtering
- Topic modeling: latent dirichlet allocation (LDA)
- Frequent itemsets, association rules, and sequential pattern mining

Why MLlib?

- scikit-learn?
  - Python library
  - Not Distributed (single-machine) => Not Scalable, Runs slow on large dataset
  - Algorithms:
    - Classification: SVM, nearest neighbors, random forest, …
    - Clustering: k-means, spectral clustering, …
    - Regression: support vector regression (SVR), ridge regression, Lasso, logistic regression, …
    - Decomposition: PCA, non-negative matrix factorization (NMF), independent component analysis (ICA), …
- Mahout?
  - Java/Scala library
  - Distributed => Scalable
  - Core algorithms on top of Hadoop Map/Reduce => Runs slow on large datasets
  - Can run on Spark
  - Algorithms
    - Classification: logistic regression, naive Bayes, random forest,
    - Clustering: k-means, fuzzy k-means, …
    - Recommendation: ALS, …
    - Decomposition: PCA, SVD, randomized SVD, …

K-means (python)

- Study the file `kmeans-example.py`
  - to cluster a set of vector values
  - Input file: /usr/local/spark/data/mllib/kmeans-data.txt

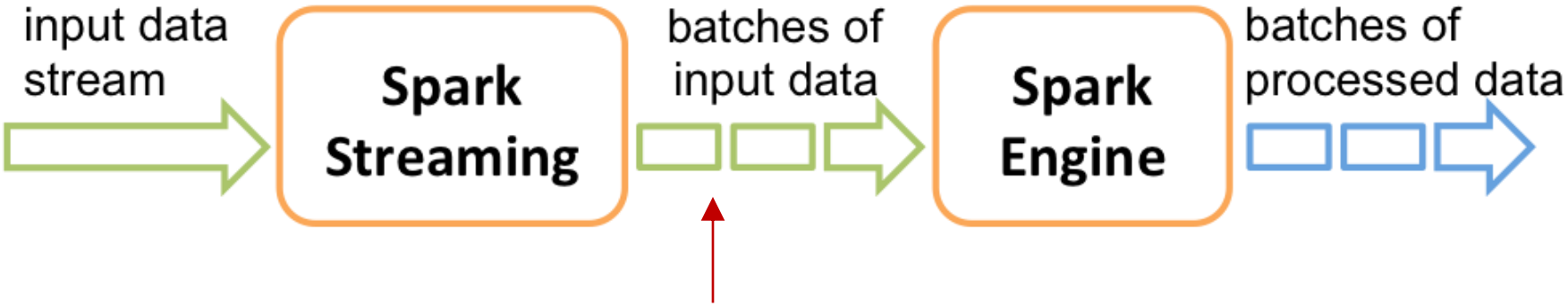
```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

What is Spark Streaming?

- Enables scalable, high-throughput, fault-tolerant **stream processing of live data streams**
- Data ingestion from many sources like Kafka, Kinesis, TCP sockets, Twitter, …
- Data processing using complex algorithms expressed with functions like map, reduce, join, …
- Processed data can be pushed out to filesystems, databases, and live dashboards

How Spark Streaming works?

- Internally Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



DStream: a continuous stream of data.
DStream is represented as a sequence of RDDs.

Spark Examples: Spark Streaming

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
# Create a local StreamingContext with two working threads and batch
interval of 1 second
sconf = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sconf, 1)
# Create a DStream to connect to hostname:port, like localhost:9999
# lines DStream represents the stream of data that will be received
from the data server. Each record in this DStream is a line of text.
lines = ssc.socketTextStream("localhost", 9999)
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
# Print the first ten elements of each RDD generated in this DStream
to the console
wordCounts.pprint()
ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

See next slides

Detailed explanation here.

Spark Examples: Spark Streaming

```
# Firstly, in one terminal run Netcat (http://nc110.sourceforge.net)
# to generate a data stream on the localhost:9999 TCP socket
$ nc -lk 9999
hello world
hi there fred
what a nice world there

# In another terminal run the NetworkWordCount example
# expecting a data stream on the localhost:9999 TCP socket
$ /usr/local/spark/bin/spark-submit network-word-count.py
```

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999

hello world



...
```

Scala Java Python

```
# TERMINAL 2: RUNNING network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
...
-------------------------------------------
Time: 2014-10-14 15:25:21
-------------------------------------------
(hello,1)
(world,1)
...
```

Spark Examples: Spark Streaming

- Input DStreams represent the stream of input data received from streaming sources
- `lines` was an input DStream as it represented the stream of data received from the netcat server
- Every input DStream (except file stream) is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing.
- Spark worker/executor is a long-running task occupying one of the cores allocated to the Spark Streaming application.
  - Important: Spark Streaming application needs to be allocated enough cores (or threads, if running locally) to process received data, as well as to run the receiver(s).
  - When running a Spark Streaming program locally, do not use "local" or "local[1]" as the master URL. Either of these means that only one thread will be used for running tasks locally. If you are using an input DStream based on a receiver (e.g. sockets, Kafka, etc.), then the single thread will be used to run the receiver, leaving no thread for processing the received data.

https://spark.apache.org/docs/latest/streaming-programming-guide.html#input-dstreams-and-receivers