EPL451: Data Mining on the Web – Lab 7

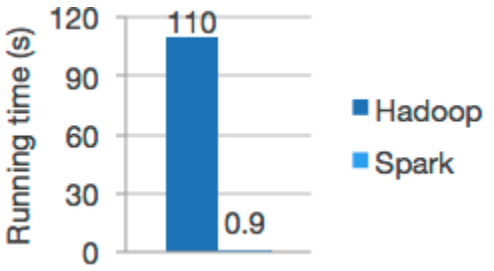
University of Cyprus
Department of
Computer Science

Παύλος Αντωνίου

Γραφείο: Β109, 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)*

Spark vs Hadoop

- 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

Spark vs Hadoop

- Distributed Res. Mgmt & 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 2
  - **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 Spark SQL for SQL and structured data processing, 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 1.6.0 is installed on your Virtual Machine (current versions 2.1.0)
- Start Spark Shell:
  - cd /home/csdeptucy/apache-spark-1.6.0/bin
  - Python Shell:
    - `./pyspark` ⟶
  - 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]`

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 3.4.3 (default, Oct 14 2015 20:33:09)
SparkContext available as sc, HiveContext available as sql
>>>
```

Spark over Python 3.6

- Apache Spark <= v2.1.0 (Mar 2017) not compatible with Python 3.6
- If you installed Python 3.6 in VM => solution:
  - `nano ~/.bashrc`
  - Add in end of the file:
    - # added to force spark use python 3.4 instead of 3.6
    - `export PYSPARK_PYTHON=/usr/bin/python3.4`
  - `source ~/.bashrc`

Spark Essentials: SparkContext

- First thing that a Spark program does is create a ***SparkContext*** object, which tells Spark how to access a cluster
- 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:

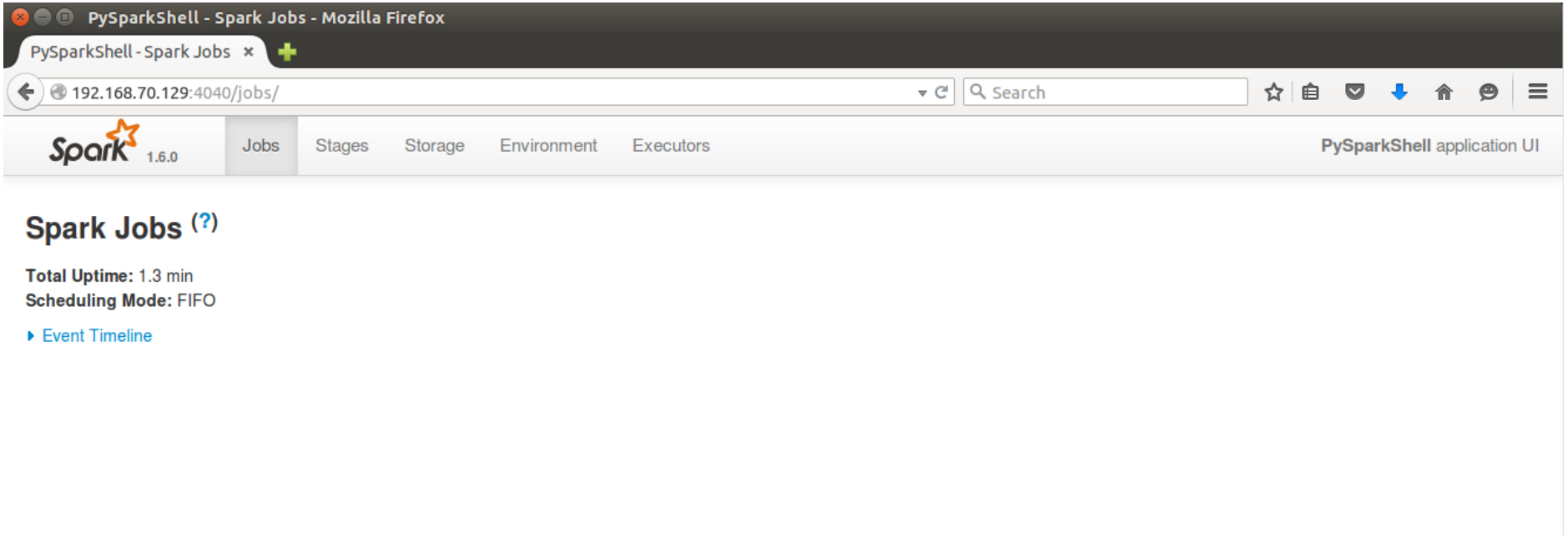
```
scala> sc
res: spark.SparkContext =
spark.SparkContext@470d1f30
```

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

```
>>> sc
<pyspark.context.SparkContext object
at 0xb69fe72c>
```

Spark Web UI

- Each driver program has a web UI, typically on port 4040, that displays information about running tasks, executors, and storage usage
- In browser type `driver-node:4040`
  - In a single node cluster driver-node is localhost



Spark Essentials: Master

- The ***master*** parameter for a ***SparkContext*** 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)
```

- From command line:

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

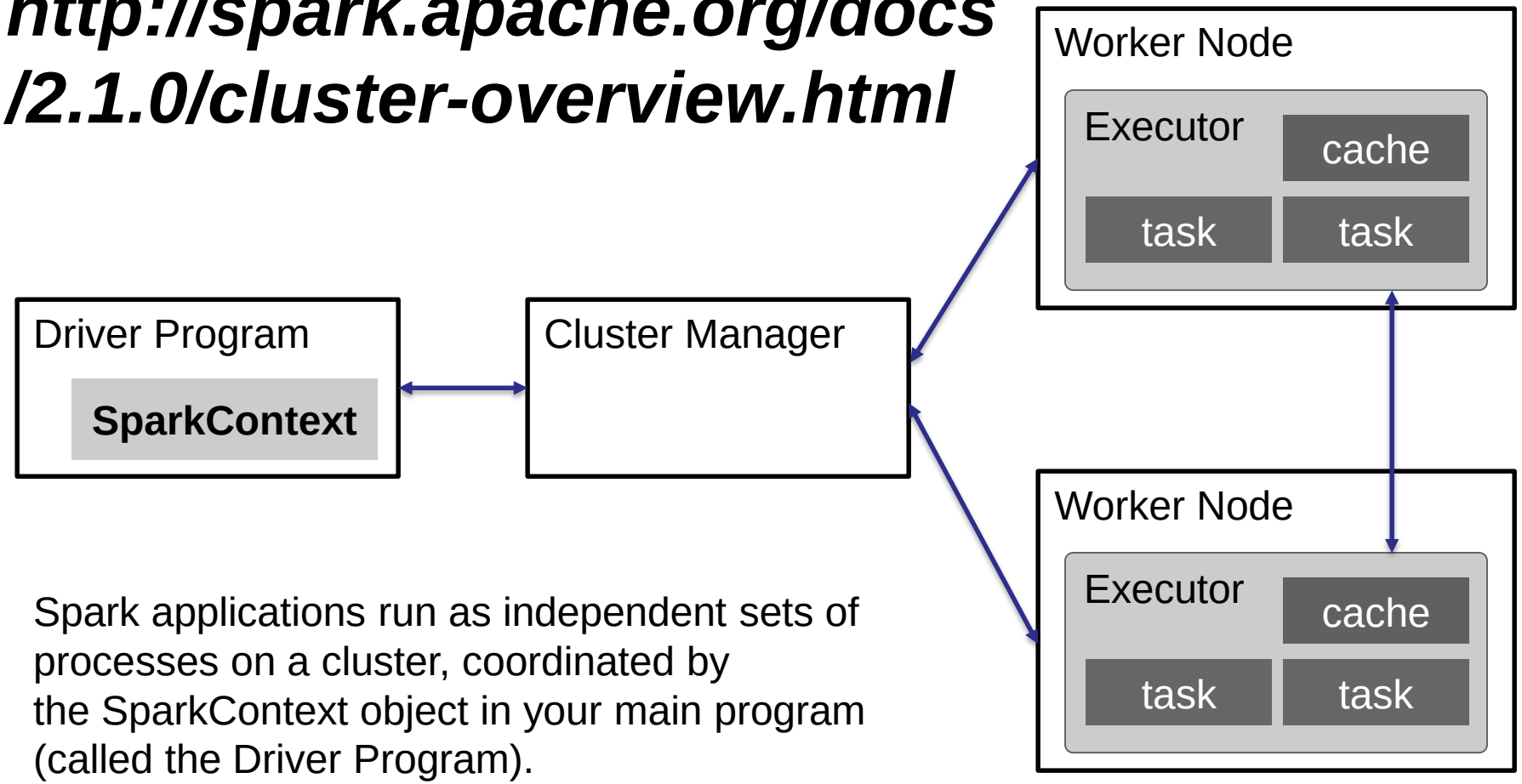
Spark Essentials

- 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*
    - Distributed collection of elements (e.g. files or collections) partitioned across the nodes of the cluster that can be operated on in parallel
  - *shared variables* that can be used in parallel operations

Spark Essentials: Run on cluster

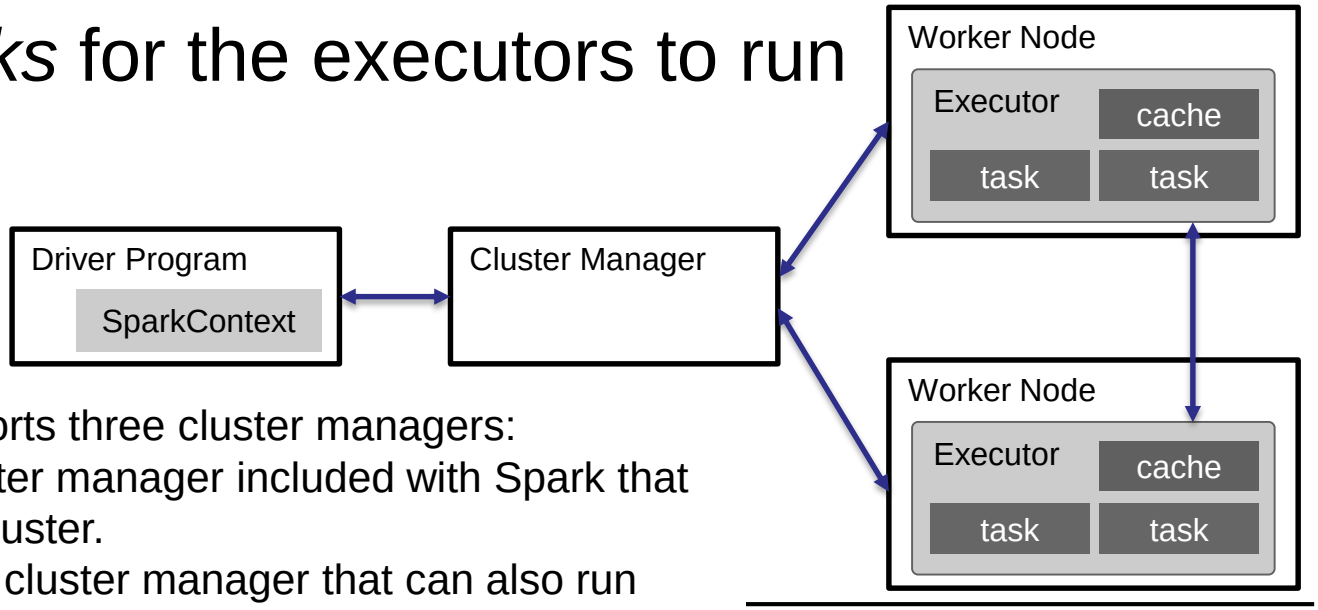
http://spark.apache.org/docs/2.1.0/cluster-overview.html



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 2.

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** – take an existing Scala collection or Python iterable or collection (e.g. lists) and run functions on it in parallel
  - **Hadoop datasets** – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop
- 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

- However, an 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/2.1.0/programming-guide.html#transformations

http://spark.apache.org/docs/2.1.0/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 to select values less than 3 and then collect RDD contents back to driver

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

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*)

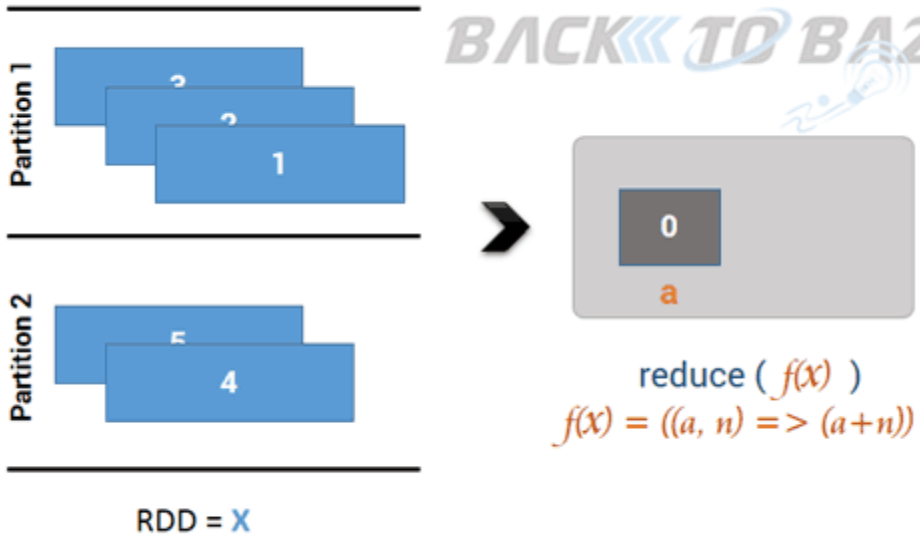
Hands on – pyspark

```
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/csdeptucy/input/unixd
ict.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 b is the current line
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

```
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 b 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 imme
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.

Hands on – pyspark

- 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)
- 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]

Hands on – pyspark

- flatMap() transformation
  - same as map but instead of returning just one element per element returns a sequence per element (which can be empty)
- reduce() action
  - aggregates all elements of RDD using a given function and returns the final result to the driver program
- reduceByKey() action
  - operation on key-value pairs
  - aggregates all values having the same key using a given function
  - returns a distributed dataset

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`
- Modify `kmeans-fleet.py` to cluster fleet data (see Lab5) using different k values
  - Replace `None` with appropriate commands

APPENDIX

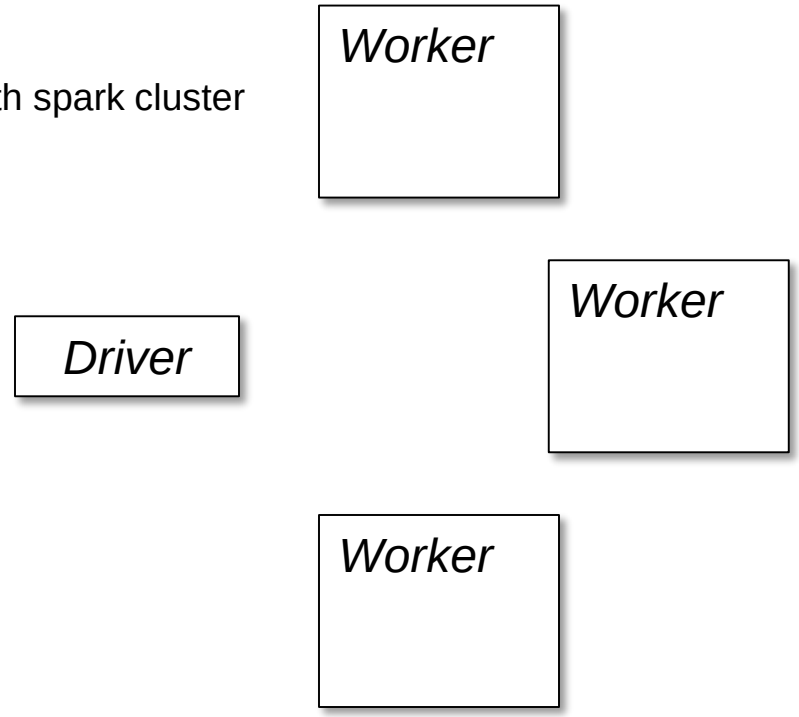
- More examples follow:
  - Log mining
  - K-Means
  - Pagerank
  - Spark SQL
  - Spark streaming

Spark Deconstructed

- We start with Spark running on a cluster… submitting code to be evaluated on it:

We are given the following log file to parse and retrieve statistics

ERROR	php: dying for unknown reasons
WARN	dave, are you angry at me?
ERROR	did mysql just barf?
WARN	xylons approaching
ERROR	mysql cluster: replace with spark cluster

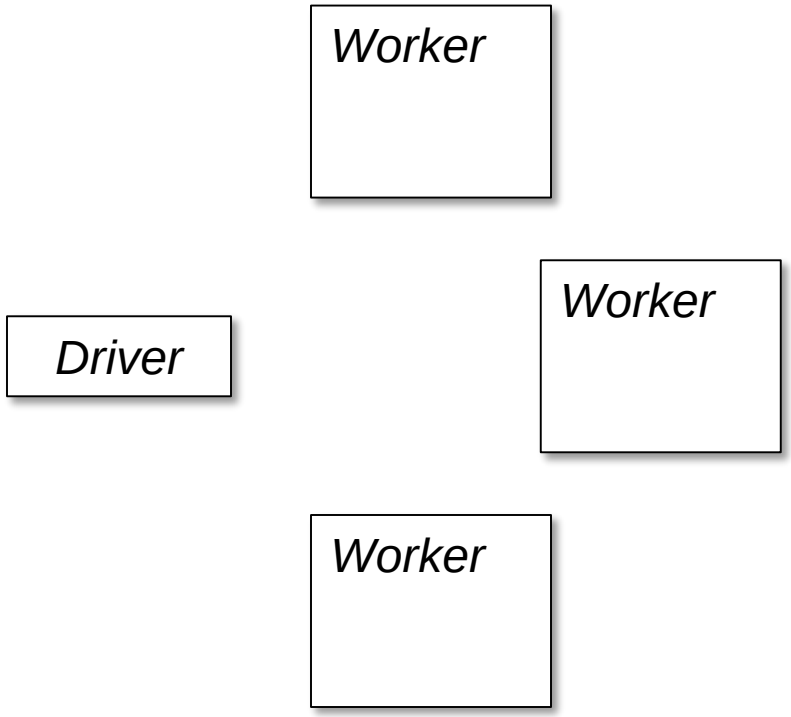


Spark Deconstructed: Log Mining Example

```
# load error messages from a log into memory
# then interactively search for various patterns
from pyspark import SparkContext, SparkConf
sconf = SparkConf().setAppName("LogMining").setMaster("local")
sc = SparkContext(conf=sconf)
# base RDD
file = sc.textFile("hdfs:...")
# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r: r[1])
messages.cache()

# action 1
print("MySQL errors: ", messages.filter(lambda m: "mysql" in
m).count())

# action 2
print("Php errors: ", messages.filter(lambda m: "php" in
m).count())
```

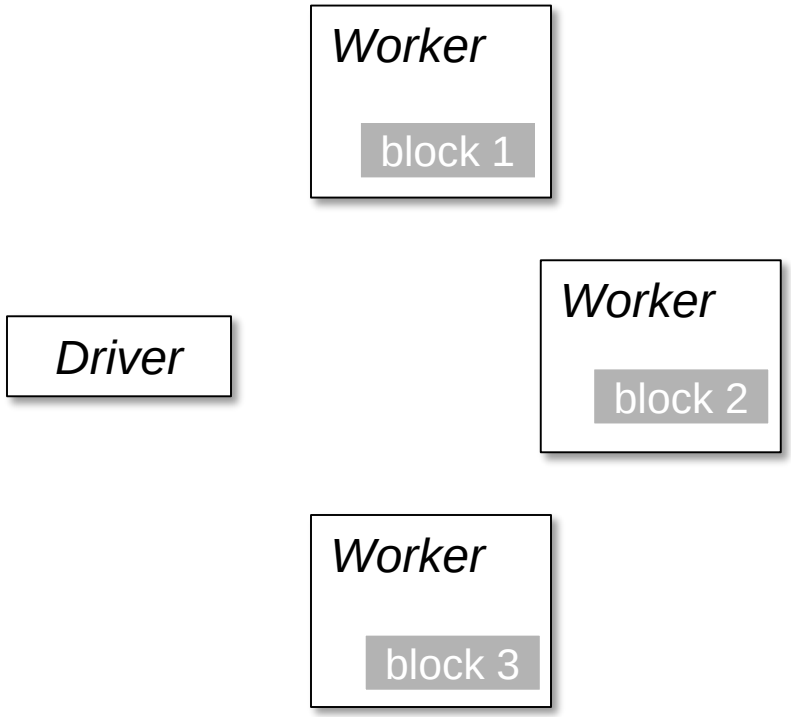
Spark Deconstructed: Log Mining Example

```
# base RDD
file = sc.textFile("hdfs:...")

# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r:
r[1])
messages.cache()
```



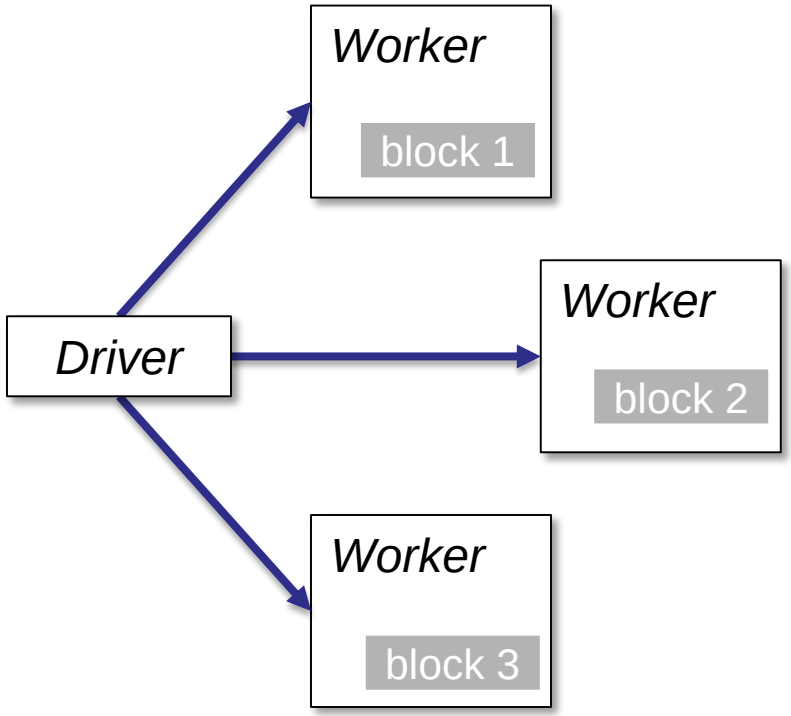
Spark Deconstructed: Log Mining Example

```
# base RDD
file = sc.textFile("hdfs:...")

# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r:
r[1])
messages.cache()
```



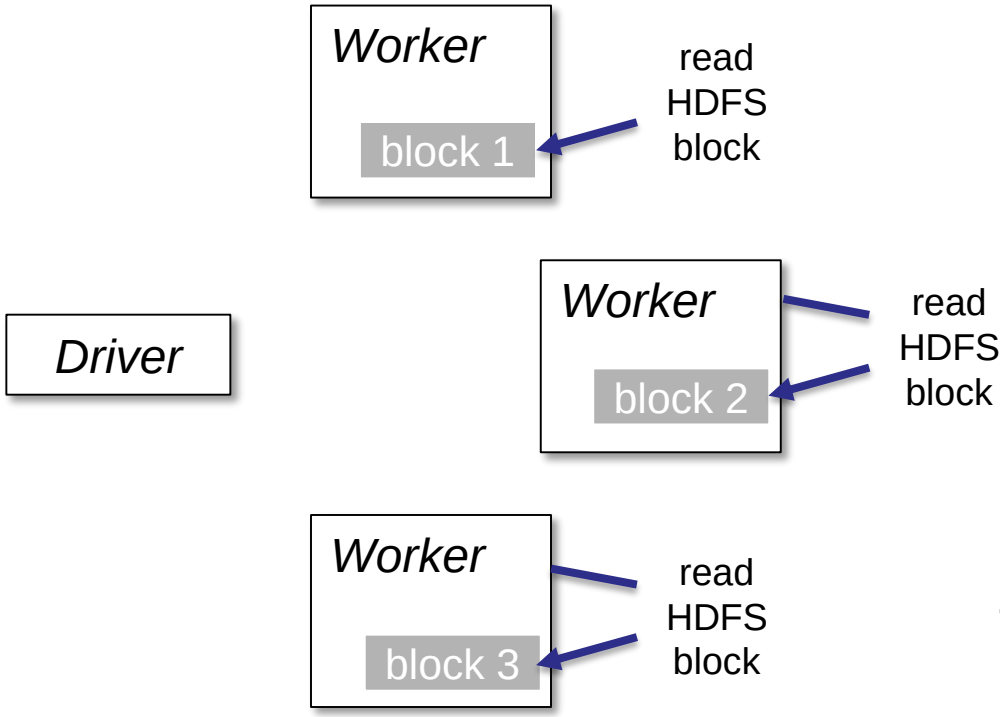
Spark Deconstructed: Log Mining Example

```
# base RDD
file = sc.textFile("hdfs:...")

# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r: r[1])
messages.cache()
```



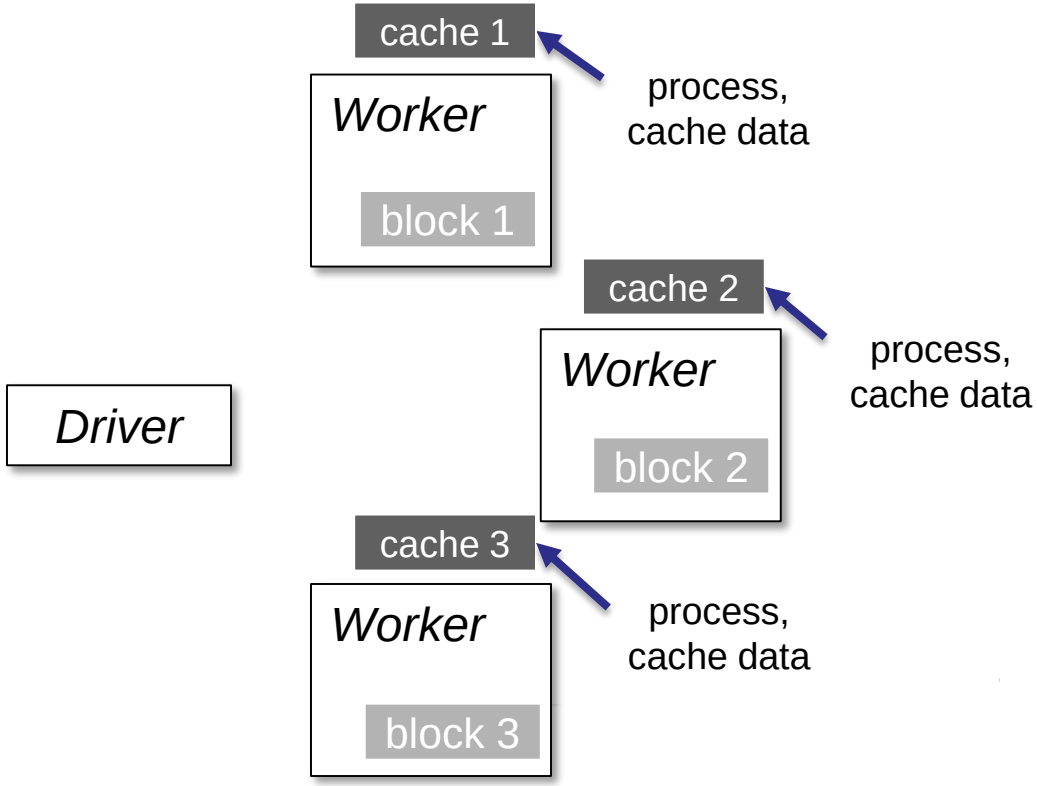
Spark Deconstructed: Log Mining Example

```
# base RDD
file = sc.textFile("hdfs:...")

# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r: r[1])
messages.cache()
```



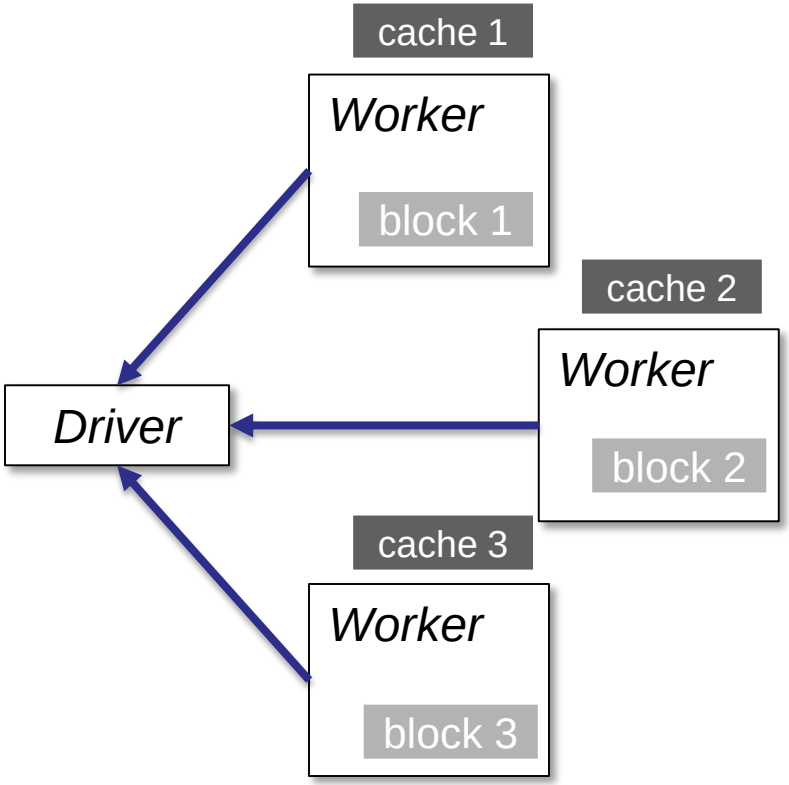
Spark Deconstructed: Log Mining Example

```
# base RDD
file = sc.textFile("hdfs:...")

# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r:
r[1])
messages.cache()
```



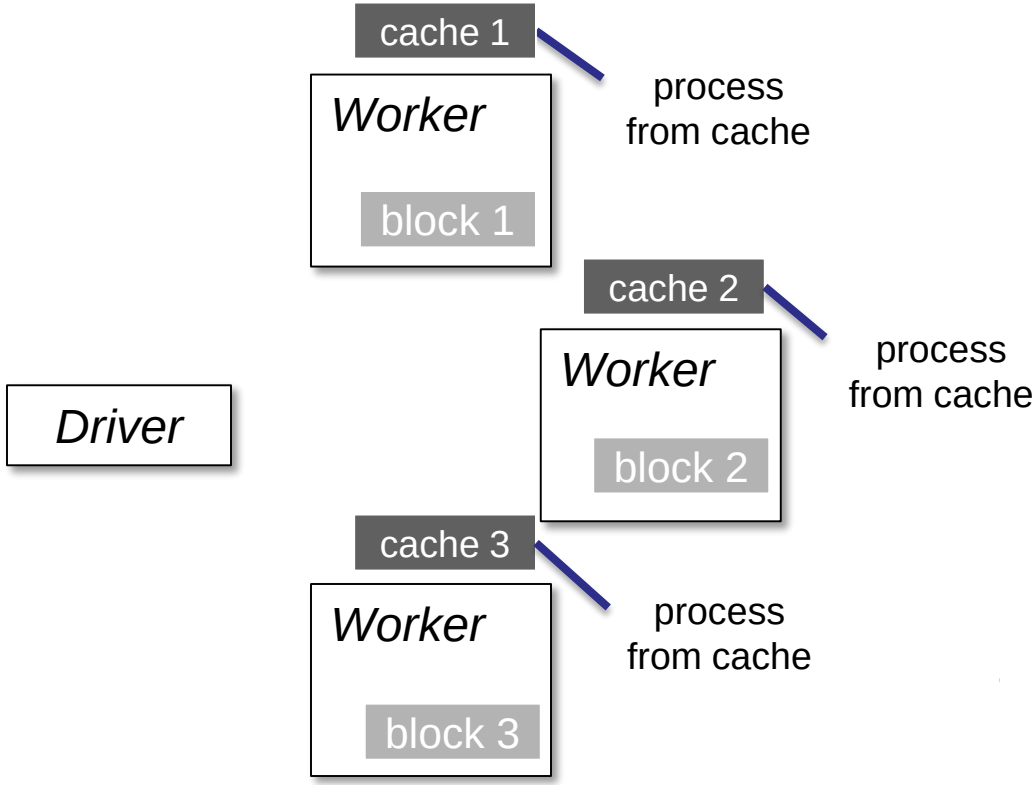
Spark Deconstructed: Log Mining Example

```
# base RDD
file = sc.textFile("hdfs:...")

# transformed RDDs
errors = file.filter(lambda line: line.startswith("ERROR"))
messages = errors.map(lambda e: e.split("\t")).map(lambda r:
r[1])
messages.cache()

# action 1
print("MySQL errors: ",
messages.filter(lambda m:
"mysql" in m).count())
```



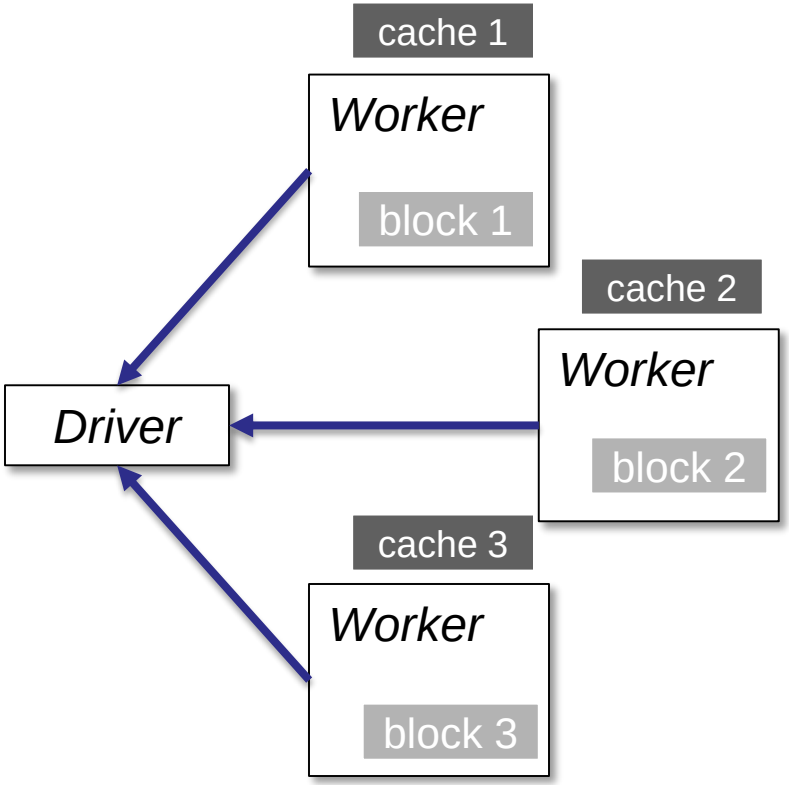
Spark Deconstructed: Log Mining Example

```
# action 2
print("Php errors: ", messages.filter(lambda m: "php" in
m).count())
```



Spark Deconstructed: Log Mining Example

```
# action 2
print("Php errors: ", messages.filter(lambda m: "php" in
m).count())
```



Spark Examples: K-Means

- Next, try using **K-Means** to cluster a set of vector values:

cp data/mllib/kmeans_data.txt .

./bin/run-example SparkKMeans kmeans_data.txt 3 0.01 local

- **Based on the data set:**

```
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
```

- **Please refer to the source code in:**

examples/src/main/scala/org/apache/spark/examples/SparkKMeans.scala

Spark Examples: PageRank

- Next, try using **PageRank** to rank the relationships in a graph:

  cp data/mllib/pagerank_data.txt .
  ./bin/run-example SparkPageRank pagerank_data.txt 10 local

- **Based on the data set:**

```
1 2
1 3
1 4
2 1
3 1
4 1
```

- **Please refer to the source code in:**

examples/src/main/scala/org/apache/spark/examples/SparkPageRank.scala

Spark Examples: Spark SQL

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._
// Define the schema using a case class.
case class Person(name: String, age: Int)
// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt")
.map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")

// SQL statements can be run by using the sql methods provided by
sqlContext.
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND
age <= 19")
// The results of SQL queries are SchemaRDDs and support all the
// normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

Spark Essentials: Broadcast Variables

- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Spark Essentials: *Accumulators*

- Accumulators are variables that can only be "added" to through an *associative* operation
- Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types
- Only the driver program can read an accumulator's value, not the tasks

```
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

```
accum.value
```

driver-side

Spark Examples: Spark Streaming

```
# in one terminal run the NetworkWordCount example in Spark Streaming
# expecting a data stream on the localhost:8080 TCP socket
$ ./bin/run-example org.apache.spark.examples.streaming.NetworkWordCount localhost 8080


# in another terminal use Netcat http://nc110.sourceforge.net/
# to generate a data stream on the localhost:8080 TCP socket
$ nc -lk 8080
hello world
hi there fred
what a nice world there
```

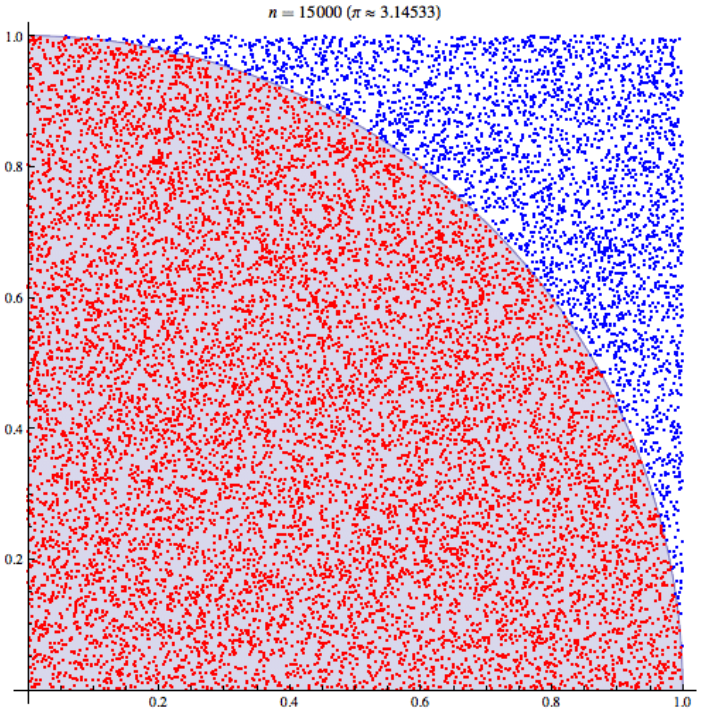
Spark Examples: Spark Streaming

```
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a StreamingContext with a SparkConf configuration
val ssc = new StreamingContext(sparkConf, Seconds(1))

// Create a DStream that will connect to serverIP:serverPort
val lines = ssc.socketTextStream(serverIP, serverPort)

// Split each line into words!
val words = lines.flatMap(_.split(" "))

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print a few of the counts to the console!
wordCounts.print()

ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

Spark Examples: *Estimate PI*

- Now, try using a Monte Carlo method to estimate the value of Pi

./bin/run-example SparkPi 2 local



Spark Examples: *Estimate PI*

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - \ldots)$$

```
import scala.math.random
import org.apache.spark._

/** Computes an approximation to pi */
object SparkPi {
        def main(args: Array[String]) {
                val conf = new SparkConf().setAppName("Spark Pi")
                val spark = new SparkContext(conf)
                val slices = if (args.length > 0) args(0).toInt else 2
                val n = 100000 * slices
                val count = spark.parallelize(1 to n, slices).map { i =>
                        val x = random * 2 - 1
                        val y = random * 2 - 1
                        if (x*x + y*y < 1) 1 else 0
                }.reduce(_ + _)

                println("Pi is roughly " + 4.0 * count / n)
                spark.stop()
        }
}
```

Spark Examples: *Estimate PI*

```
val count = sc.parallelize(1 to n, slices)
```

base RDD

```
  .map { i =>
        val x = random * 2 - 1
        val y = random * 2 - 1
        if (x*x + y*y < 1) 1 else 0
  }
```

transformed RDD

```
  .reduce(_ + _)
```

action