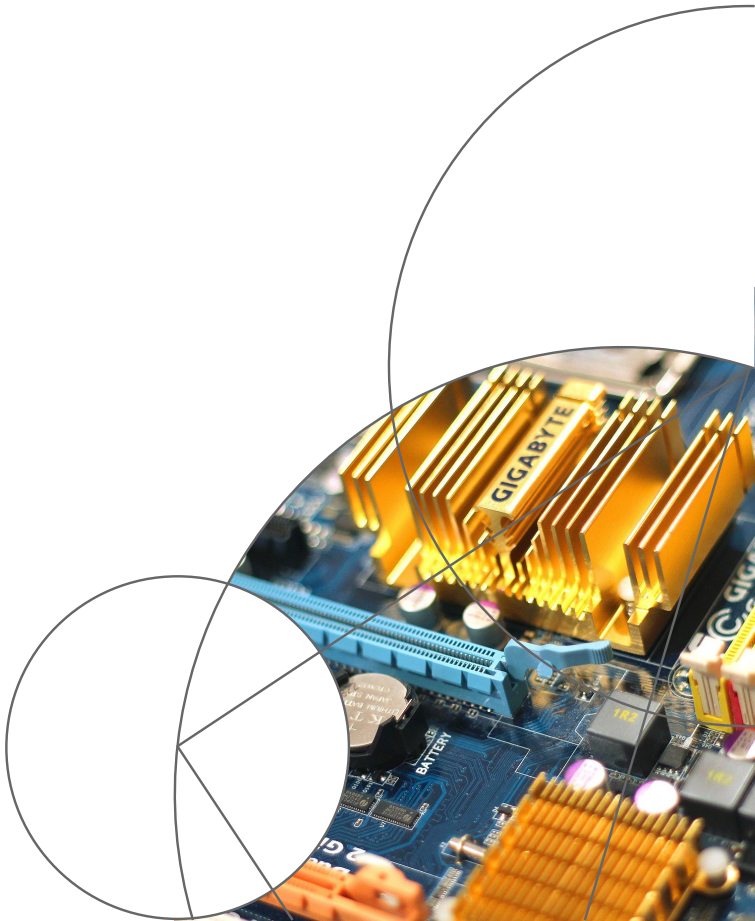
L11 – Apache Spark

Large-Scale Data Analysis

Fabian Gieseke

Image Group
Department of Computer Science
University of Copenhagen

Universitetsparken 1, Room 1-1-N110
fabian.gieseke@di.ku.dk



28 May 2019

Outline

Outline

Shortcomings of Hadoop?

Shortcomings of Hadoop?

Hadoop Worfklow

Hadoop (HDFS and YARN) offer a simplified API and easy-to-use parallel and fault tolerant access to big data. However, the worfklow is kind of incremental:

1. Intermediate results (e.g., output of mappers) are stored on disk (slow!).
2. Between each map and reduce step, Hadoop sorts/shuffles the data via the network (might be very slow!).
3. Multiple jobs can be exected in a chain, but this introduces some kind of "barrier" (all jobs are executed independently from each other!).
4. ...

Thus: Fault tolerant computations, but not suited for all tasks. For instance, not suited if many iterations have to be performed!

Shortcomings of Hadoop?

Hadoop Worfklow

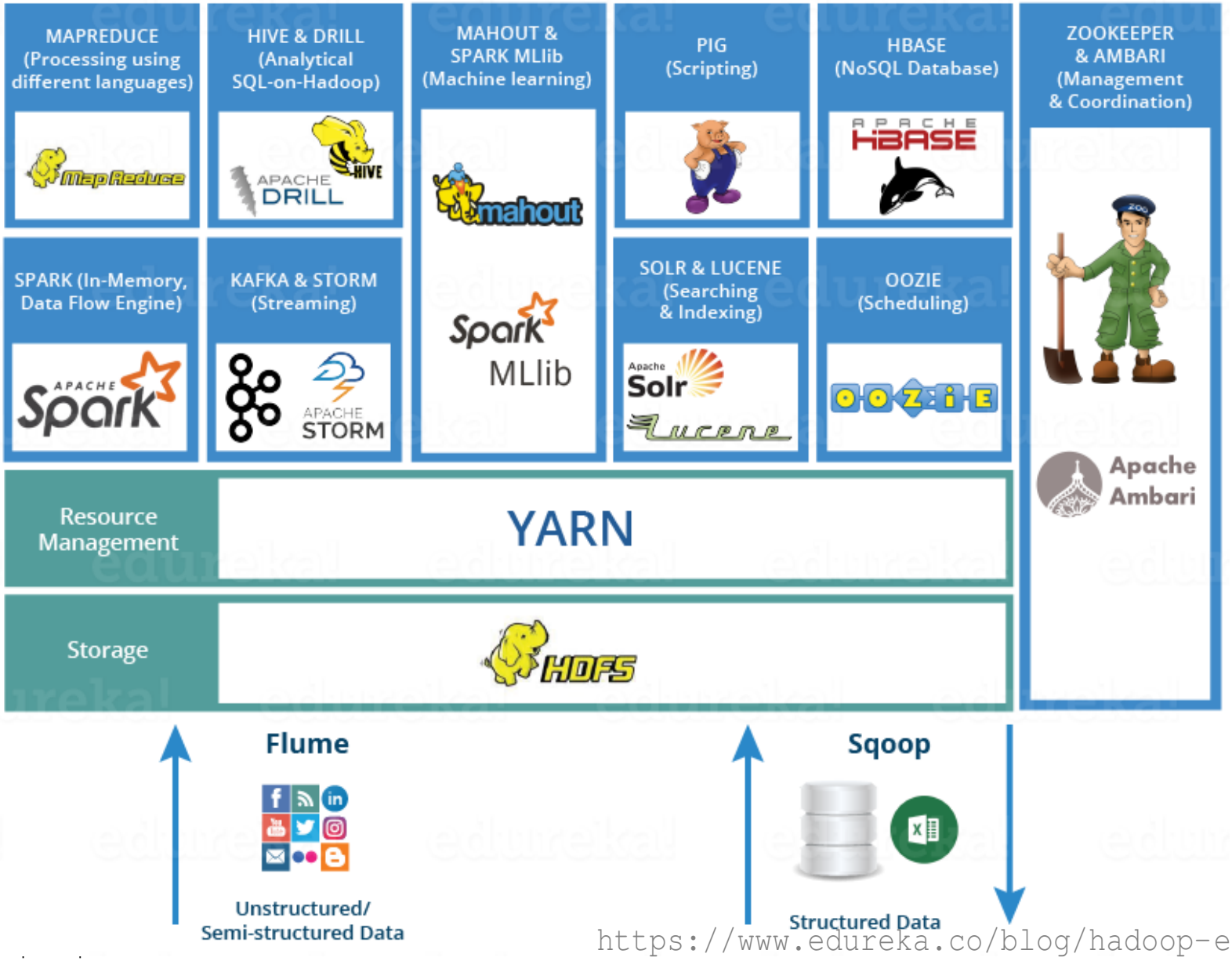
Hadoop (HDFS and YARN) offer a simplified API and easy-to-use parallel and fault tolerant access to big data. However, the worfklow is kind of incremental:

1. Intermediate results (e.g., output of mappers) are stored on disk (slow!).
2. Between each map and reduce step, Hadoop sorts/shuffles the data via the network (might be very slow!).
3. Multiple jobs can be exected in a chain, but this introduces some kind of "barrier" (all jobs are executed independently from each other!).
4. ...

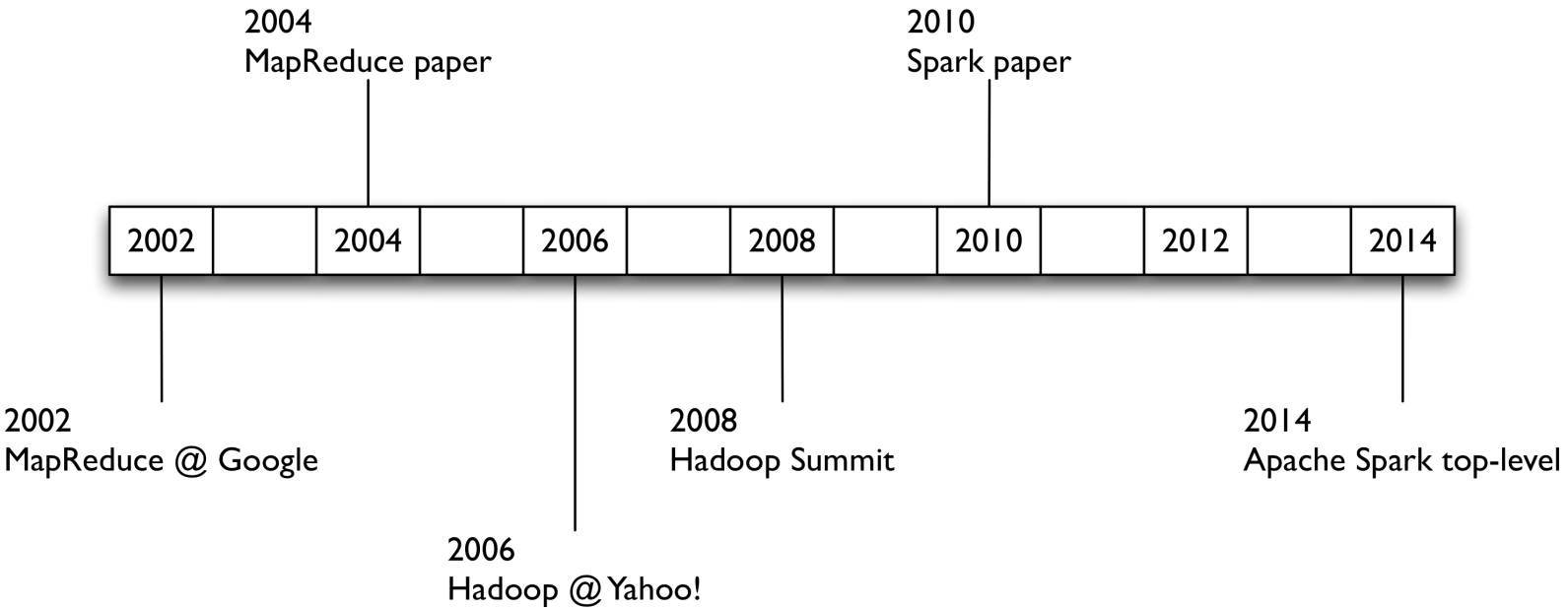
Thus: Fault tolerant computations, but not suited for all tasks. For instance, not suited if many iterations have to be performed!

Hence: Many specialized systems/workarounds/software packages . . .

Hadoop Ecosystem



Spark History



Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Apache Spark

Lightning-fast cluster computing

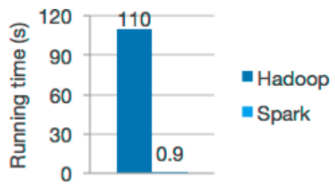
Download | Libraries | Documentation | Examples | Community | Developers | Apache Software Foundation

Apache Spark™ is a fast and general engine for large-scale data processing.

Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

Ease of Use

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

Generality

Latest News

Spark 2.1.1 released (May 02, 2017)

Spark Summit (June 5-7th, 2017, San Francisco) agenda posted (Mar 31, 2017)

Spark Summit East (Feb 7-9th, 2017, Boston) agenda posted (Jan 04, 2017)

Spark 2.1.0 released (Dec 28, 2016)

Archive

Download Spark

Built-in Libraries:

SQL and DataFrames
Spark Streaming
MLlib (machine learning)
GraphX (graph)

Third-Party Projects

http://spark.apache.org/

Apache Spark

Lightning-fast cluster computing

Download | Libraries | Documentation | Examples | Community | Developers | Apache Software Foundation

Apache Spark™ is a fast and general engine for large-scale data processing.

Latest News

Spark 2.1.1 released (May 02, 2017)

Spark Summit (June 5-7th, 2017, San Francisco) agenda posted (Mar 31, 2017)

Spark Summit East (Feb 7-9th, 2017, Boston) agenda posted (Jan 04, 2017)

Spark 2.1.0 released (Dec 28, 2016)

Archive

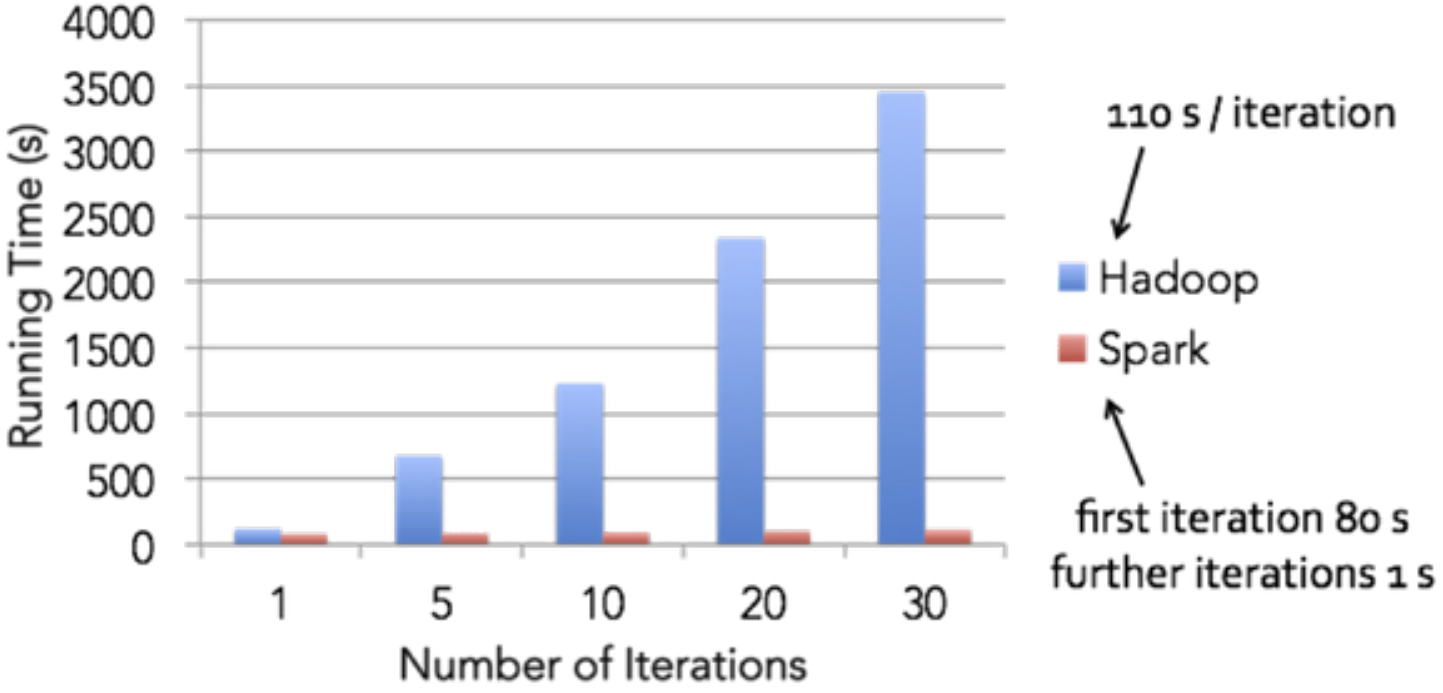
Download Spark

Key Concepts

1. Spark combines: Batch, interactive, real-time processing
2. Interfaces: Scala, Java, Python, R, . . .
3. High-level of abstraction: Simplified big data programming (you'll see in a second!)
4. Keep data in main memory of cluster (i.e., in the nodes' memories)
5. Optimized data flow . . .
6. MapReduce just one of the supported techniques . . .
7. . . .

http://spark.apache.org/

What's new in Spark?



Key Idea

Keep the data in main memory of workers if needed. Operations are consecutive transformations. In case an error occurs, one simply reloads the data and applies the chain of functions again (faul tolerant). This results in big speed-ups, especially for iterative schemes . . .

Especially machine learning!

Demo: PySpark

```
lsda@lsdabox:~$ ./.start_dfs_yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /srv/hadoop-2.7.7/logs/hadoop-lsda-namenode-lsdabox.out
localhost: starting datanode, logging to /srv/hadoop-2.7.7/logs/hadoop-lsda-datanode-lsdabox.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /srv/hadoop-2.7.7/logs/hadoop-lsda-secondarynamenode-lsdabox.o
ut
starting yarn daemons
starting resourcemanager, logging to /srv/hadoop-2.7.7/logs/yarn-lsda-resourcemanager-lsdabox.out
localhost: starting nodemanager, logging to /srv/hadoop-2.7.7/logs/yarn-lsda-nodemanager-lsdabox.out
lsda@lsdabox:~$ source .venvs/lsda/bin/activate
(lsda) lsda@lsdabox:~$ pyspark
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
19/05/25 11:13:50 WARN util.Utils: Your hostname, lsdabox resolves to a loopback address: 127.0.1.1; using 10.
0.2.15 instead (on interface enp0s3)
19/05/25 11:13:50 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
19/05/25 11:13:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.1
      /_/

Using Python version 3.6.7 (default, Oct 22 2018 11:32:17)
SparkSession available as 'spark'.
>>> text = sc.textFile("hdfs:///user/lsda/lsda.txt")
>>> words = text.flatMap(lambda line: line.split(" "))
>>> pairs = words.map(lambda word: (word, 1))
>>> counts = pairs.reduceByKey(lambda a, b: a + b)
>>> counts.collect()
[('love', 2), ('Hadoop!', 1), ('cannot', 1), ('', 6), ('live', 1), ('this', 1), ('count', 2), ('Of', 1), ('cou
rse,', 1), ('these', 1), ('own.', 1), ('But', 1), ('why', 1), ('do', 3), ('use', 2), ('fancy', 1), ('reducer',
 1), ('files', 1), ('let', 1), ('them', 1), ('work.', 1), ('Just', 1), ('have', 1), ('figure', 1), ('out', 1),
 ('just', 1), ('of', 1), ('times:', 1), ('Hadoop,', 2), ('Hadoop.', 1), ('I', 6), ('Really,', 1), ('without',
1), ('Hadoop', 2), ('anymore.', 1), ('In', 1), ('particular,', 1), ('word', 2), ('example...', 1), ('could', 1
), ('all', 2), ('words', 1), ('on', 1), ('my', 1), ('should', 1), ('this?', 1), ('can', 1), ('the', 3), ('mapp
er', 1), ('and', 2), ('hard', 1), ('to', 2), ('how', 1), ('it!', 1), ("Let's", 1), ('a', 1), ('couple', 1), ('
Wonderful!', 1)]
>>>
```

Spark's Key Ingredient: RDDs

Resilient Distributed Dataset (RDD)

An RDD is an immutable and distributed collection of data objects. Similarly to HDFS files, an RDD is split into multiple partitions that are stored on different nodes of your cluster.

1. Immutable collection of objects, distributed over cluster.
   (user can control the partioning and storage of RDDs in memory/on disk)
2. Transformations/Actions: Parallel operators to manipulate RDDs.
3. Fault tolerance: Automatic rebuilt of intermediate results in case of failure.
4. ...

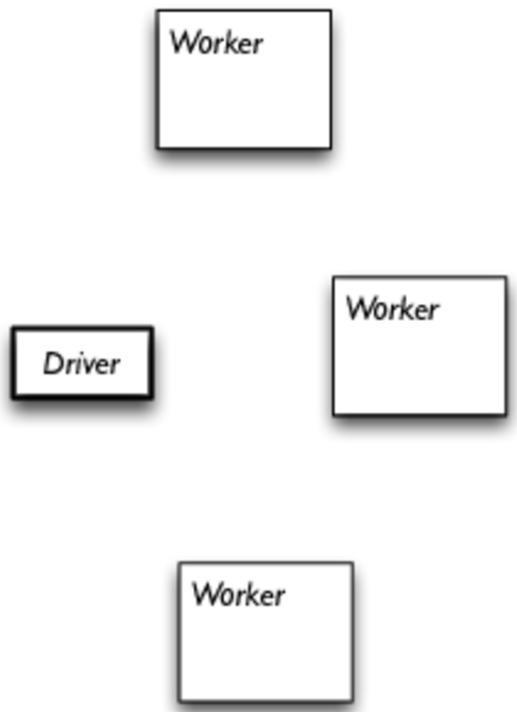
Example: Log Mining

```
# load error messages from a log into memory
# then interactively search for patterns

# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

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

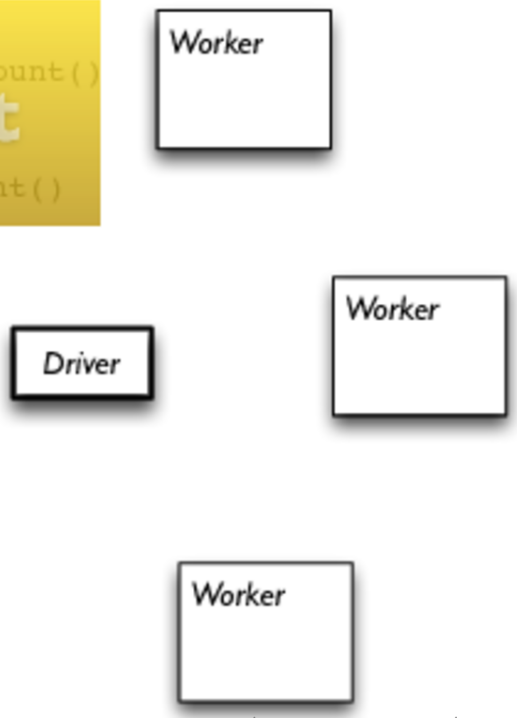


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

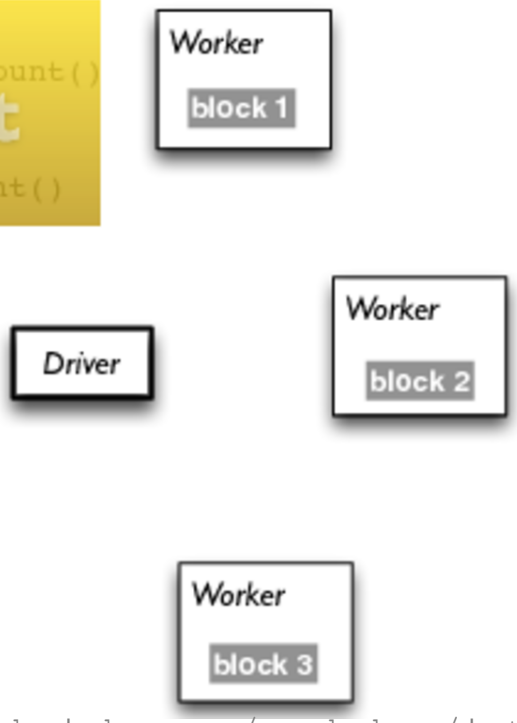


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

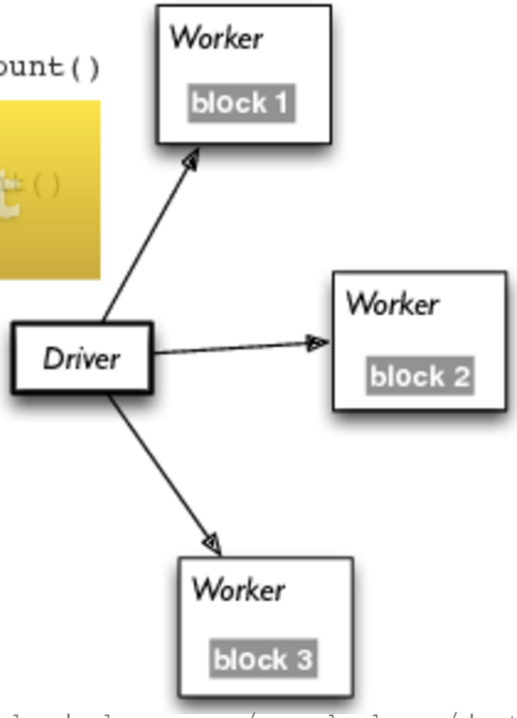


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

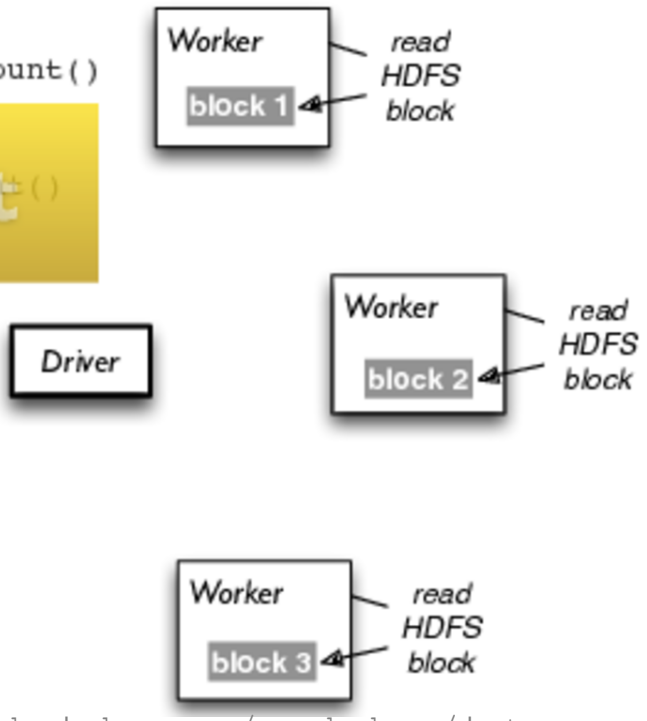


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

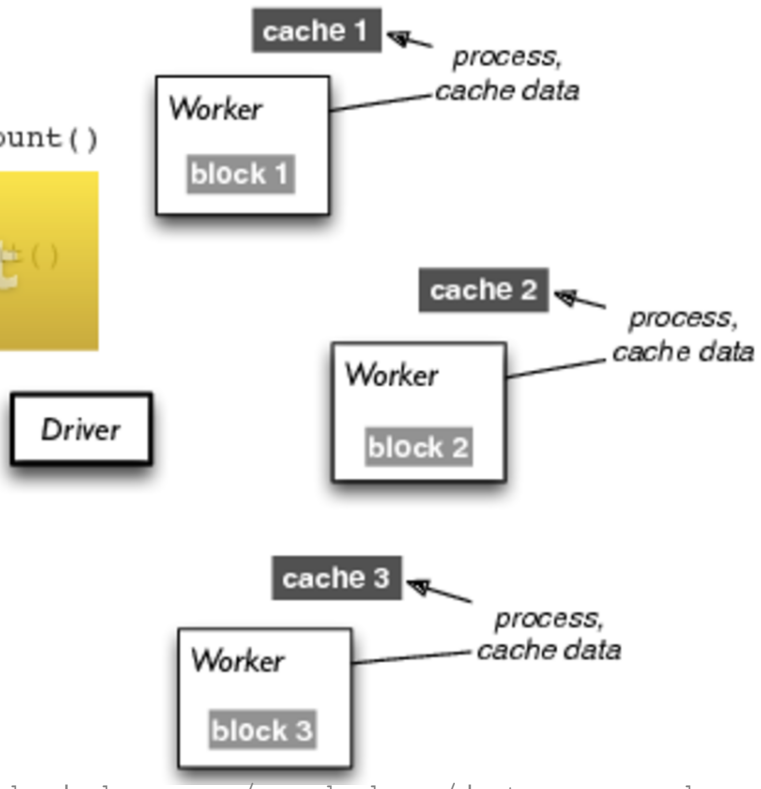


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

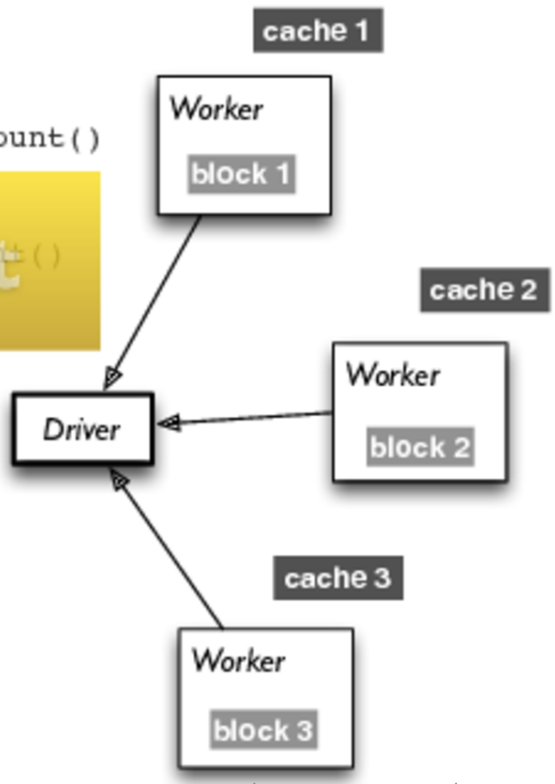


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part



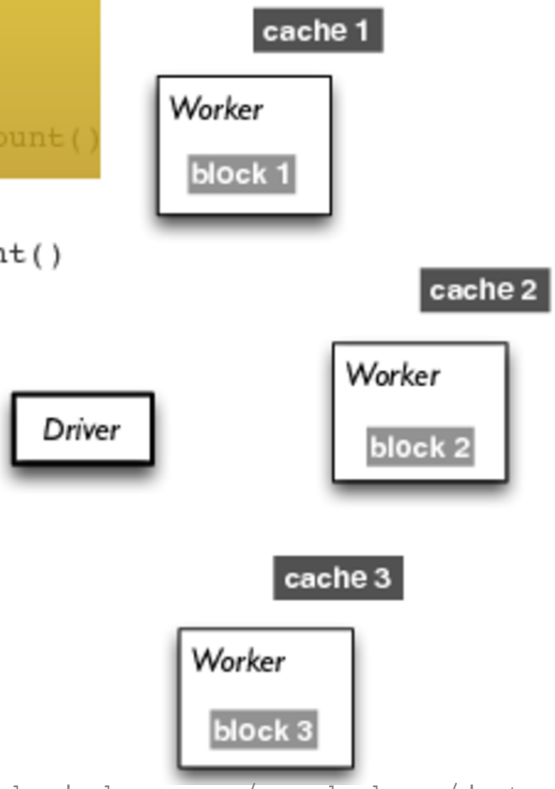
Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
  .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

discussing the other part

```
# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```



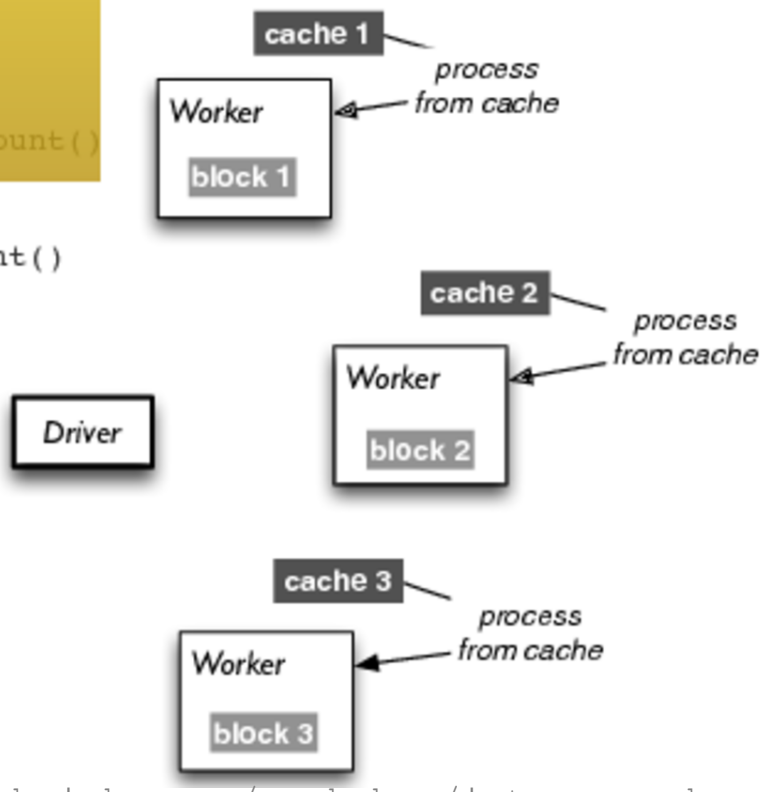
Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
  .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

discussing the other part

```
# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```



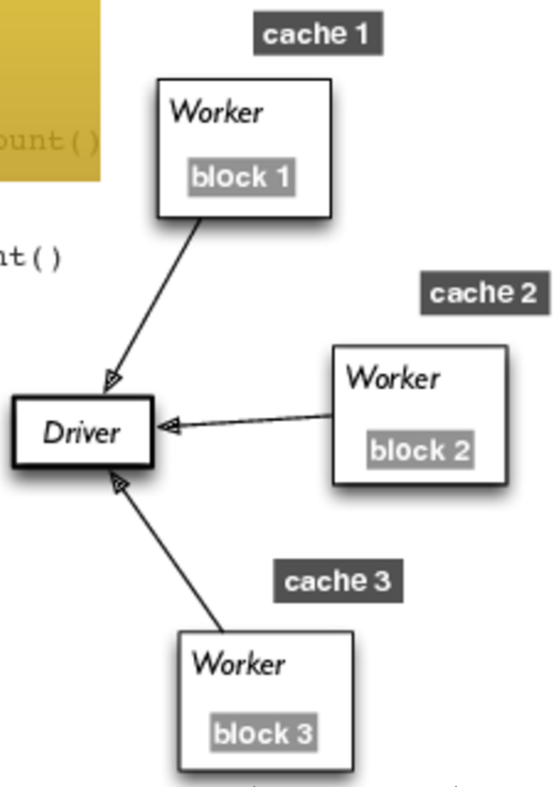
Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
  .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

discussing the other part

```
# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

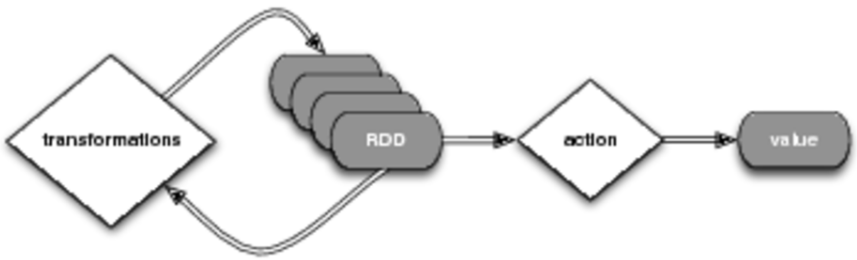


Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining

Looking at the RDD transformations and actions from another perspective…

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```



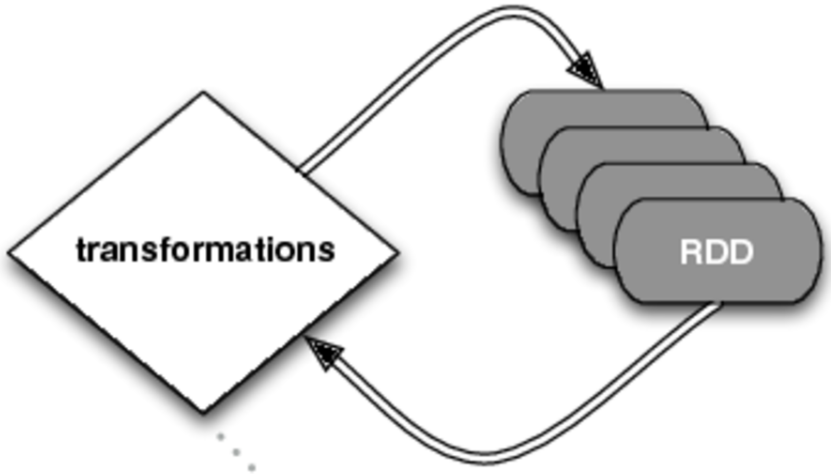
Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Example: Log Mining


```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
 .map(lambda x: x.split("\t"))
```

Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

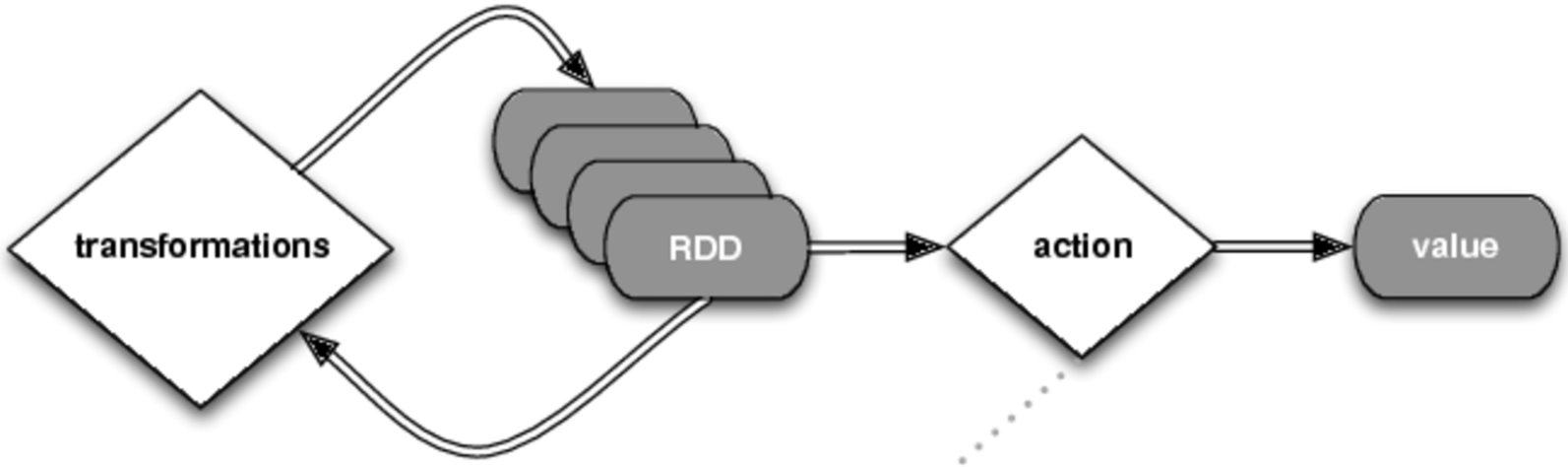
Example: Log Mining




```
# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()
```

Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

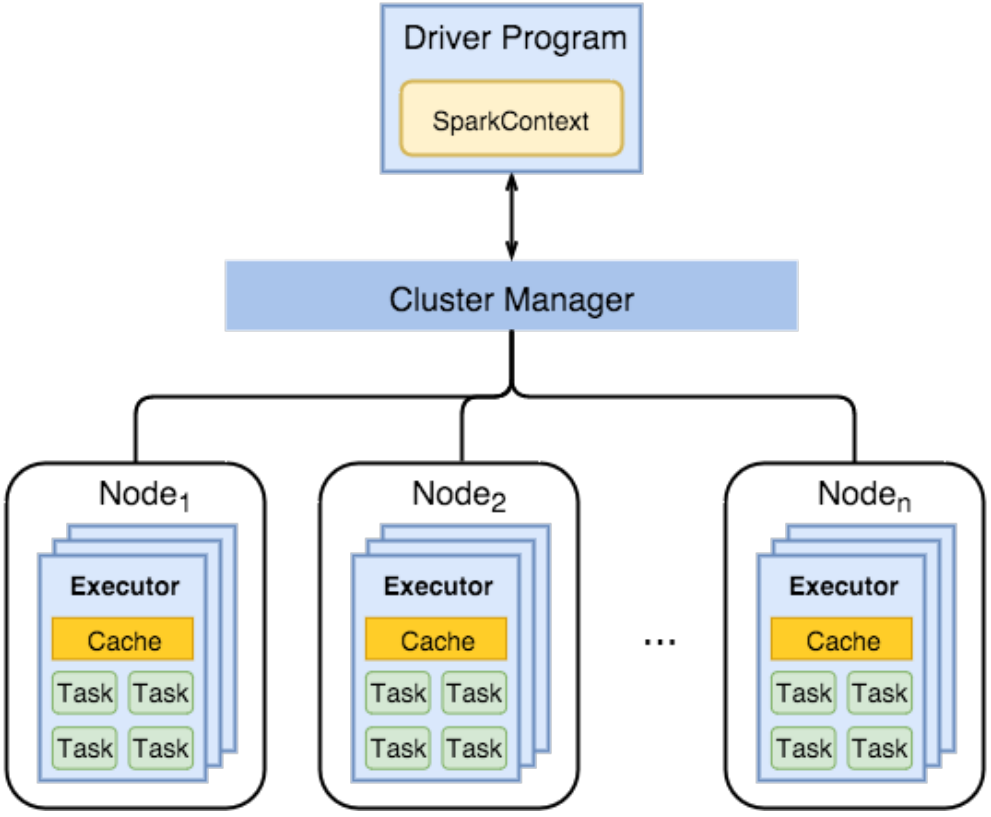
Example: Log Mining




```
# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

Paco Nathan, http://training.databricks.com/workshop/intro_spark.pdf

Distributed Execution



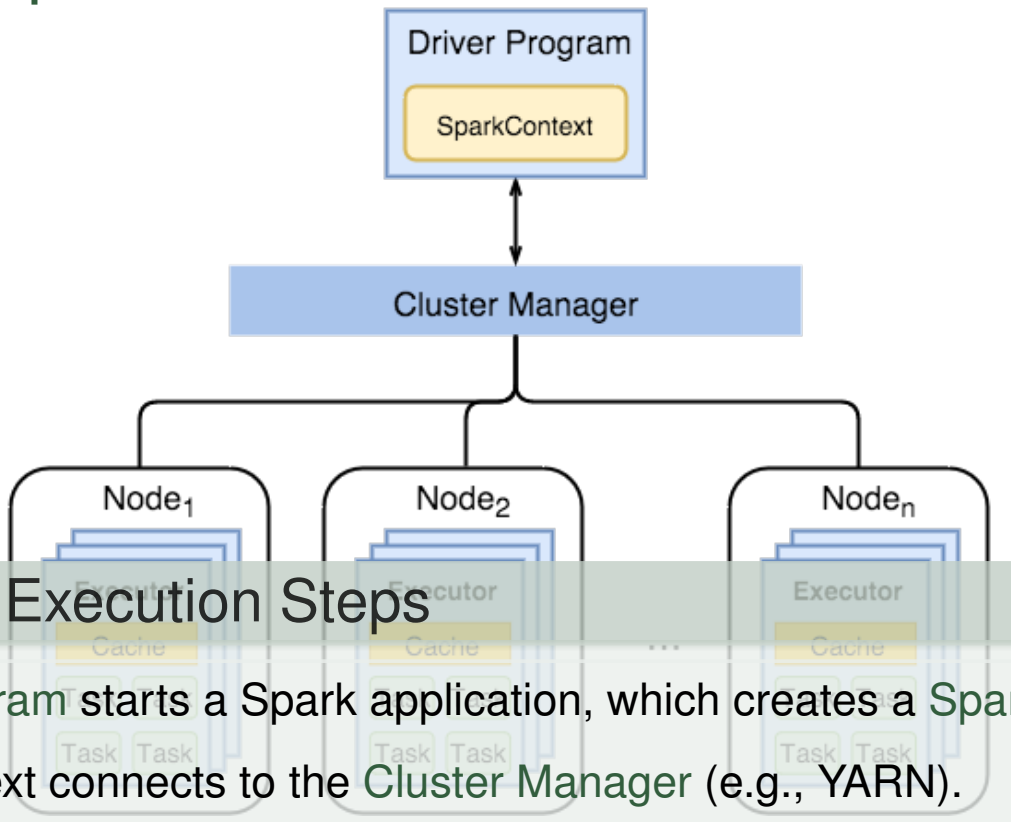
Execution of Jobs

1. Driver Program: The driver program runs on the node you are interacting with.
2. Executors: The worker nodes that execute the job (in a distributed fashion).

Communication via cluster manager (e.g., YARN).

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

Spark Application



Spark Program: Execution Steps

1. The Driver Program starts a Spark application, which creates a SparkContext object.
2. The SparkContext connects to the Cluster Manager (e.g., YARN).
3. The Cluster Manager identifies Executors (threads) on the various nodes of the cluster, which can run the computations (and store the data).
4. The Driver Program sends code to the Executors.
5. The SparkContext sends tasks to the Executors.
6. The Executors perform computations . . .

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

SparkContext and PySpark

```
lsda@lsdabox:~$ ./.start_dfs_yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /srv/hadoop-2.7.7/logs/hadoop-lsda-namenode-lsdabox.out
localhost: starting datanode, logging to /srv/hadoop-2.7.7/logs/hadoop-lsda-datanode-lsdabox.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /srv/hadoop-2.7.7/logs/hadoop-lsda-secondarynamenode-lsdabox.o
ut
starting yarn daemons
starting resourcemanager, logging to /srv/hadoop-2.7.7/logs/yarn-lsda-resourcemanager-lsdabox.out
localhost: starting nodemanager, logging to /srv/hadoop-2.7.7/logs/yarn-lsda-nodemanager-lsdabox.out
lsda@lsdabox:~$ source .venvs/lsda/bin/activate
(lsda) lsda@lsdabox:~$ pyspark
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
19/05/25 11:13:50 WARN util.Utils: Your hostname, lsdabox resolves to a loopback address: 127.0.1.1; using 10.
0.2.15 instead (on interface enp0s3)
19/05/25 11:13:50 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
19/05/25 11:13:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.1
      /_/

Using Python version 3.6.7 (default, Oct 22 2018 11:32:17)
Sp
>>
>>
>>
>>
>>
>>> counts.collect()
[
rs
 1),
 ('just', 1), ('of', 1), ('times:', 1), ('Hadoop,', 2), ('Hadoop.', 1), ('I', 6), ('Really,', 1), ('without',
1), ('Hadoop', 2), ('anymore.', 1), ('In', 1), ('particular,', 1), ('word', 2), ('example...', 1), ('could', 1
), ('all', 2
er', 1), ('a
Wonderful!', 1)]
>>>
```

Interactive Spark

1. PySpark: The SparkContext is generated automatically (variable `sc`).
2. You can use this object to create new Spark things (e.g., RDDs, . . .).

Running a Spark Application

Word Count in Spark

```
1  from pyspark import SparkConf
2  from pyspark import SparkContext
3
4  def word_count(sc):
5
6      text = sc.textFile("hdfs:///user/lsda/lsda.txt")
7      words = text.flatMap(lambda line: line.split(" "))
8      pairs = words.map(lambda word: (word, 1))
9      counts = pairs.reduceByKey(lambda a, b: a + b)
10     counts.saveAsTextFile("hdfs:///user/lsda/counts")
11
12 if __name__ == "__main__":
13
14     conf = SparkConf().setAppName("Word Count")
15     # run job on "local mode" (pseudo-cluster), use as many
16     # threads as the number of processors available for JVM.
17     conf = conf.setMaster("local[*]")
18     sc = SparkContext(conf=conf)
19
20     word_count(sc)
```

Start Hadoop. Afterwards: `spark-submit word_count.py`.

Setting the Spark Master

1. local: Run Spark locally with only one worker thread
2. local[k]: Run Spark locally with k worker threads
3. local[*]: Run Spark locally with as many threads as available processors
4. spark://HOST:PORT: Connect to a Spark standalone cluster
   (e.g., a Spark server on AWS/Azure)
5. yarn: Run Spark using YARN
6. ...

On your VM: `spark-submit --master yarn --deploy-mode client word_count.py`

Outline

Spark's Key Ingredient: RDDs

Resilient Distributed Dataset (RDD)

An RDD is an immutable and distributed collection of data objects. Similarly to HDFS files, an RDD is split into multiple partitions that are stored on different nodes of your cluster.

1. Immutable collection of objects, distributed over cluster.
   (user can control the partioning and storage of RDDs in memory/on disk)
2. Transformations/Actions: Parallel operators to manipulate RDDs.
3. Fault tolerance: Automatic rebuilt of intermediate results in case of failure.
4. ...

Creating RDDs

Various ways to create RDDs

1. Parallelizing a collection in the driver program
2. Loading data from an external data source
   1. Local file system
   2. HDFS
   3. S3
   4. ...
3. Transforming an existing RDD ...

```
1  # (1) parallelize collection via driver program
2  lines = sc.parallelize(["when", "nothing is", \
3                          "going right", "go left"])
4
5  # (2) load from external sources
6  passwd = sc.textFile("file:///etc/passwd")
7  air = sc.textFile("hdfs:///user/lsda/airline_data/2016_1.csv")
8  air_all = sc.textFile("hdfs:///user/lsda/airline_data/*.csv")
```

Main Operators

Transformations and Actions

1. Transformations: A transformation returns a new RDD. Transformation are applied in a lazy fashion, which means that the result is not computed immediately but only once it is needed!
2. Actions: Computes some result based on an RDD. Actions are eager, meaning that they are applied immediately

Laziness and eagerness are key concepts in Spark's execution workflow!

```
1  # load data
2  text = sc.textFile("hdfs:///user/lsda/lsda.txt")
3
4  # apply transformations
5  words = text.flatMap(lambda line: line.split(" "))
6  words_hadoop = words.filter(lambda w: "hadoop" in w.lower())
7  words_love = words.filter(lambda w: "love" in w.lower())
8  words_both = words_hadoop.union(words_love)
9
10 # apply action
11 words_both.count()
```

Transformations and Actions

https://spark.apache.org/docs/latest/programming-guide.html

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average)

Transformations and Actions

https://spark.apache.org/docs/latest/programming-guide.html

groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.

Transformations and Actions

https://spark.apache.org/docs/latest/programming-guide.html

pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

Transformations and Actions

https://spark.apache.org/docs/latest/programming-guide.html

takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

The Spark RDD API also exposes asynchronous versions of some actions, like `foreachAsync` for `foreach`, which immediately return a `FutureAction` to the caller instead of blocking on completion of the action. This can be used to manage or wait for the asynchronous execution of the action.

Shuffle operations

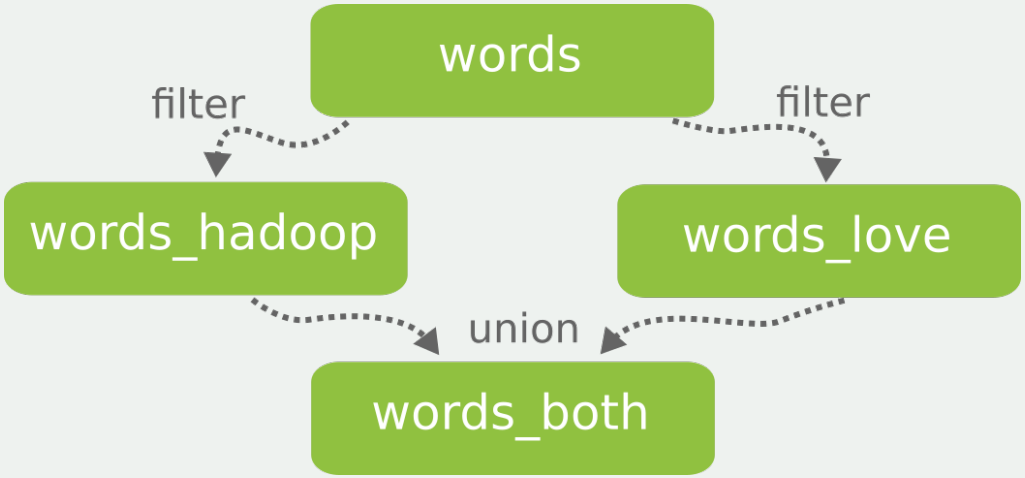
Certain operations within Spark trigger an event known as the shuffle. The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

Background

Transformations and Actions

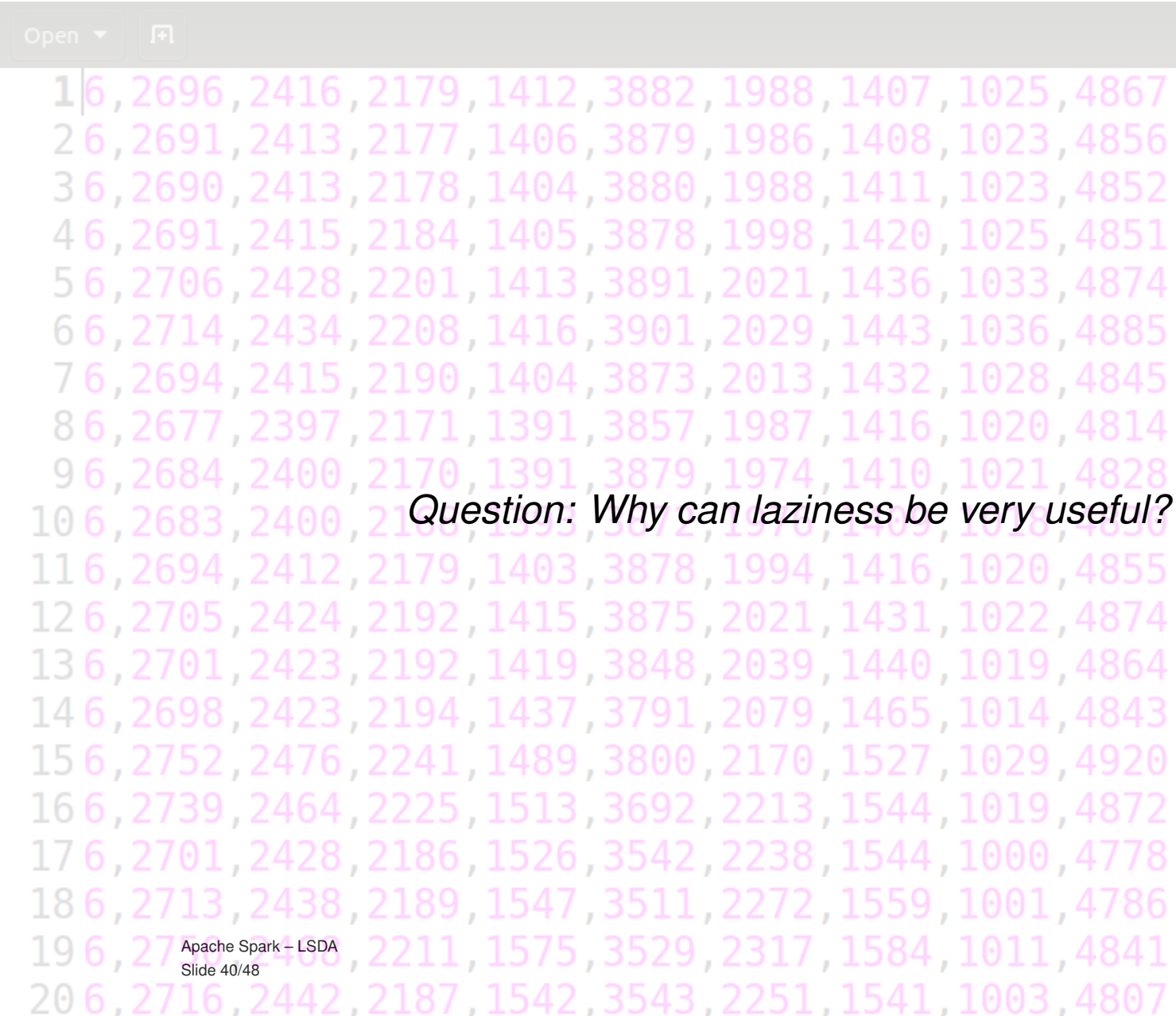
Lineage Graph

The lineage graph keeps track of the dependencies between the different RDDs. Spark can use this information to recompute an RDD in case of failure.




```
1  # load data
2  text = sc.textFile("hdfs:///user/lsda/lsda.txt")
3
4  # apply transformations
5  words = text.flatMap(lambda line: line.split(" "))
6  words_hadoop = words.filter(lambda w: "hadoop" in w.lower())
7  words_love = words.filter(lambda w: "love" in w.lower())
8  words_both = words_hadoop.union(words_love)
```

Lazy Transformations



Question: Why can laziness be very useful?

Lazy Transformations

Being lazy can pay off . . .

```
1  # load Landsat test data from HDFS
2  test = sc.textFile("hdfs:///user/lsda/landsat_test.csv")
3
4  # apply transformation
5  pts = test.map(lambda line: [int(p) for p in line.split(",")])
6
7  # get first 10 points
8  pts.take(10)
```

Important: The application of the transformation `map` is deferred until the action `take` is executed. Spark does not parse all the nine million lines. As soon as the `take` action got enough data, Spark can stop. This can save a lot of runtime!

Spark optimizes the chain of operations!

Persistence (Caching)

https://spark.apache.org/docs/latest/programming-guide.html

RDD Persistence

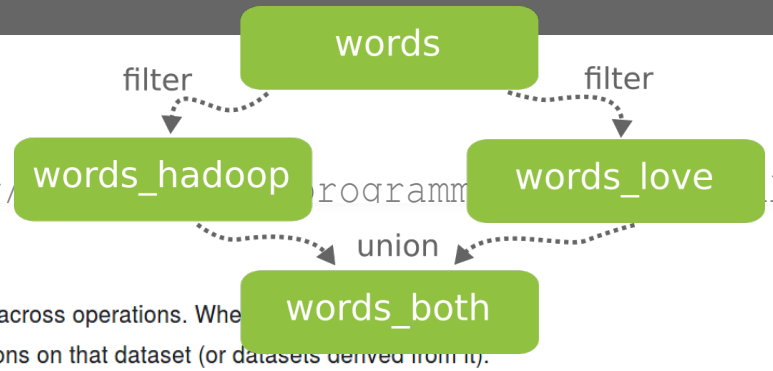
One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

You can mark an RDD to be persisted using the `persist()` or `cache()` methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

In addition, each persisted RDD can be stored using a different *storage level*, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes. These levels are set by passing a `StorageLevel` object (Scala, Java, Python) to `persist()`. The `cache()` method is a shorthand for using the default storage level, which is `StorageLevel.MEMORY_ONLY` (store deserialized objects in memory). The full set of storage levels is:

Storage Level	Meaning
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 (Java and Scala)	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 (Java and Scala)	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.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Persistence (Caching)



https://spark.apache.org/ programm l

RDD Persistence

One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations. Whe
each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it).
This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

You can mark an RDD to be persisted using the `persist()` or `cache()` methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

in addition, each persisted RDD can be stored using a different *storage level*, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes. These levels are set by passing a `StorageLevel` object (Scala, Java, Python) to `persist()`. The `cache()` method is a shorthand for using the default storage level, which is `StorageLevel.MEMORY_ONLY` (store deserialized objects in memory). The full set of storage levels is:

Storage Level	Meaning
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 (Java and Scala)	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 (Java and Scala)	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.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Caching

```
1  # load data
2  text = sc.textFile("hdfs://user/lsda/lsda.txt")
3
4  # apply transformations
5  words = text.flatMap(lambda line: line.split(" "))
6  words_hadoop = text.filter(lambda w: "hadoop" in w.lower())
7  words_love = text.filter(lambda w: "love" in w.lower())
8  words_both = words_hadoop.union(words_love)
9  words_both.cache() # or .persist()
10
11 # apply actions
12 words_both.count()
13 words_both.take(10) # 2nd action, basically for free!!!
```

Outline

Accumulators

Example

```
1  accum = sc.accumulator(0)
2
3  myrdd = sc.parallelize([1, 2, 3, 4])
4  myrdd.foreach(lambda x: accum.add(x))
5
6  # equals 10
7  accum.value
```

"Accumulators are variables that are only 'added' to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types."

https://spark.apache.org/docs/latest/rdd-programming-guide.html

Broadcasting

Example

```
1  # broadcasted via the driver
2  cool_stuff = sc.broadcast([1, 2, 3])
3
4  # can be accessed by each worker via the
5  # 'value' attribute (read-only)
6  cool_stuff.value
```

"Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost."

https://spark.apache.org/docs/latest/rdd-programming-guide.html

Example I: K-Means & Apache Spark

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

K-Means for the Iris dataset on Apache Spark (based on https://github.com/apache/spark/blob/master/examples/src/main/python/kmeans.py)

In [1]:
```
import numpy as np
```

In [2]:
```
# NOTE: Start the distributed file system first
# read input data from HDFS
lines = sc.textFile("hdfs:///user/lsda/iris.data")
```

In [3]:
```
def parseVector(line):
    """ Parses an input line and generates a
    vector (numpy array) containing the points.
    """

    # last entry is the label (not used for K-Means)
    return np.array([float(x) for x in line.split(',')[:-1]])
```

In [4]:
```
def closestPoint(p):
    """ Gets a new point p computes the
    closest cluster index for p given the
    broadcasted centers
    """

    # get broadcasted centers
    centers = centers_bc.value

    bestIndex = 0
    closest = float("+inf")

    for i in range(len(centers)):

        tempDist = np.sum((p - centers[i]) ** 2)

        if tempDist < closest:
            closest = tempDist
            bestIndex = i

    return bestIndex
```

In [5]:
```
# important: filter bad lines beforehand!
lines_filtered = lines.filter(lambda line: len(line.split(" ")) == 5)
```

Example II: Logistic Regression & Apache Spark

Algorithm 1: Logistic regression

Input: data $\{(\boldsymbol{x}_1, y_1), \ldots, \boldsymbol{x}_N, y_N)\} \subseteq (\mathbb{R}^n \times \{-1, 1\})^N$, learning rate η

Output: weights of linear hypothesis $h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$

1 initialize $\boldsymbol{w}$

2 **repeat**

`// gradient of negative log-likelihood over` N

3 $\boldsymbol{g} \leftarrow -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \boldsymbol{x}_n}{1+e^{y_n \boldsymbol{w}^\mathsf{T} \boldsymbol{x}_n}}$

`// model parameter update`

4 $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$

5 **until** *stopping criterion is met*

See Christian's slides & Abu-Mostafa *et al.*, Learning From Data, 2012.

Example II: Logistic Regression & Apache Spark

Algorithm 1: Logistic regression

Input: data $\{(\boldsymbol{x}_1, y_1), \ldots, \boldsymbol{x}_N, y_N)\} \subseteq (\mathbb{R}^n \times \{-1, 1\})^N$, learning rate η

Output: weights of linear hypothesis $h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$

1 initialize $\boldsymbol{w}$

2 **repeat**

`// gradient of negative log-likelihood over` N

3 $\boldsymbol{g} \leftarrow -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \boldsymbol{x}_n}{1+e^{y_n \boldsymbol{w}^\mathsf{T} \boldsymbol{x}_n}}$

`// model parameter update`

4 $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$

5 **until** *stopping criterion is met*

Question: How would you implement this in Spark for huge *N*?

See Christian's slides & Abu-Mostafa *et al.*, Learning From Data, 2012.

Outline

Summary & Outlook

Today

- Apache Spark
- Resilient Distributed Datasets (RDDs)
- Shared Variables

Outlook

- Next week: XGBoost (FG)
- Next week: Distributed Data Analysis with Apache Spark (FG)

Further Reading & Watching

- White. *Hadoop: The Definitive Guide*, 4th Edition, 2015.
- Karau, Konwinski, Wendell, Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*, 1st Edition, 2015.
- Bengfort and Kim. *Data Analytics with Hadoop: An Introduction for Data Scientists*, 1st Edition, 2016.