Processing Big Data

Introduction

- How to efficiently process large amount of data?
  - Use many machines
  - Use many cores
- How to efficiently use many machine/cores
  - Use a suitable programming model
- What is a programming model
  - A way to write some program,
  - with access to a limited set of functions,
    - provided by libraries or frameworks
  - and an environment to execute it.

MapReduce

The MapReduce programming model

- Popularized by a paper from Google : **MapReduce: simplified data processing on large clusters (2008)**
- Simple model with 2 basic operations
  - Map
  - Reduce
- Assume data are structured as (key,value)
- Apply successive map and reduce operations
  - Not necessarily limited to 1 map and 1 reduce in theory

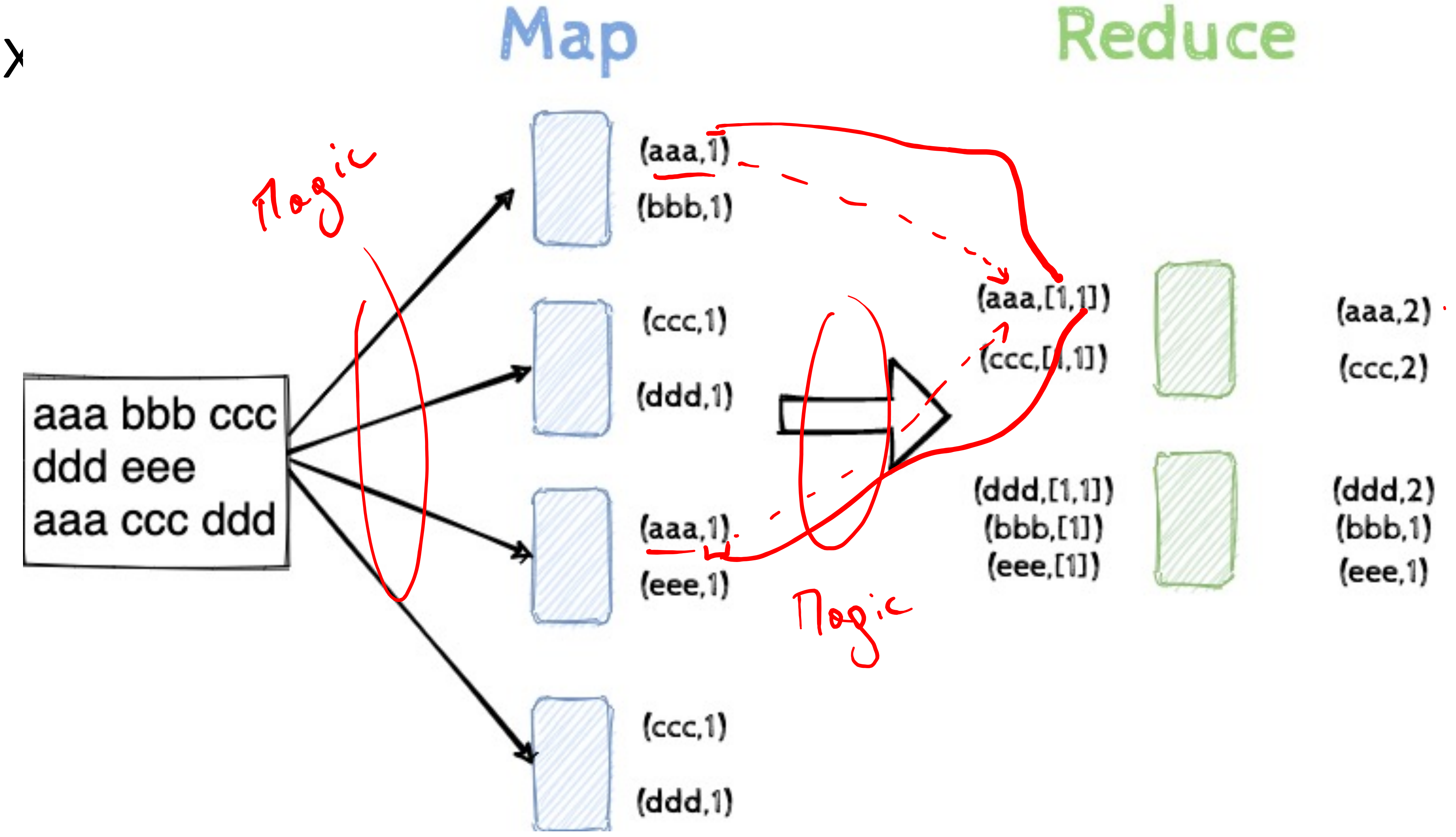
Map and Reduce

- Map and reduce are functions with defined input-output
- Map :
  - Uses a single (key, value) to produce multiple pairs
  - Input : (key, value) or (key,_) (_, value)
  - Output : One or many (key, value) pairs
- Reduce :
  - Gets all values associated with a given key and produce new pairs
  - Input : (key, [value1, value2, …, valueN])
  - Output : One or many (key, value) pairs

Example

- Word count
  - Take a text, produce a list of (word, Nb occurences)
- Map function :
  - Assume each word appears only once
  - Use word as key and add value 1
  - Input : (word, _)
  - Output : (word, 1)
- Reduce function :
  - Sum all ‘1’ for a given key (word)
  - Input : (word, [1,1,1,1,1,1])
  - Output : (word, 6)

Ex

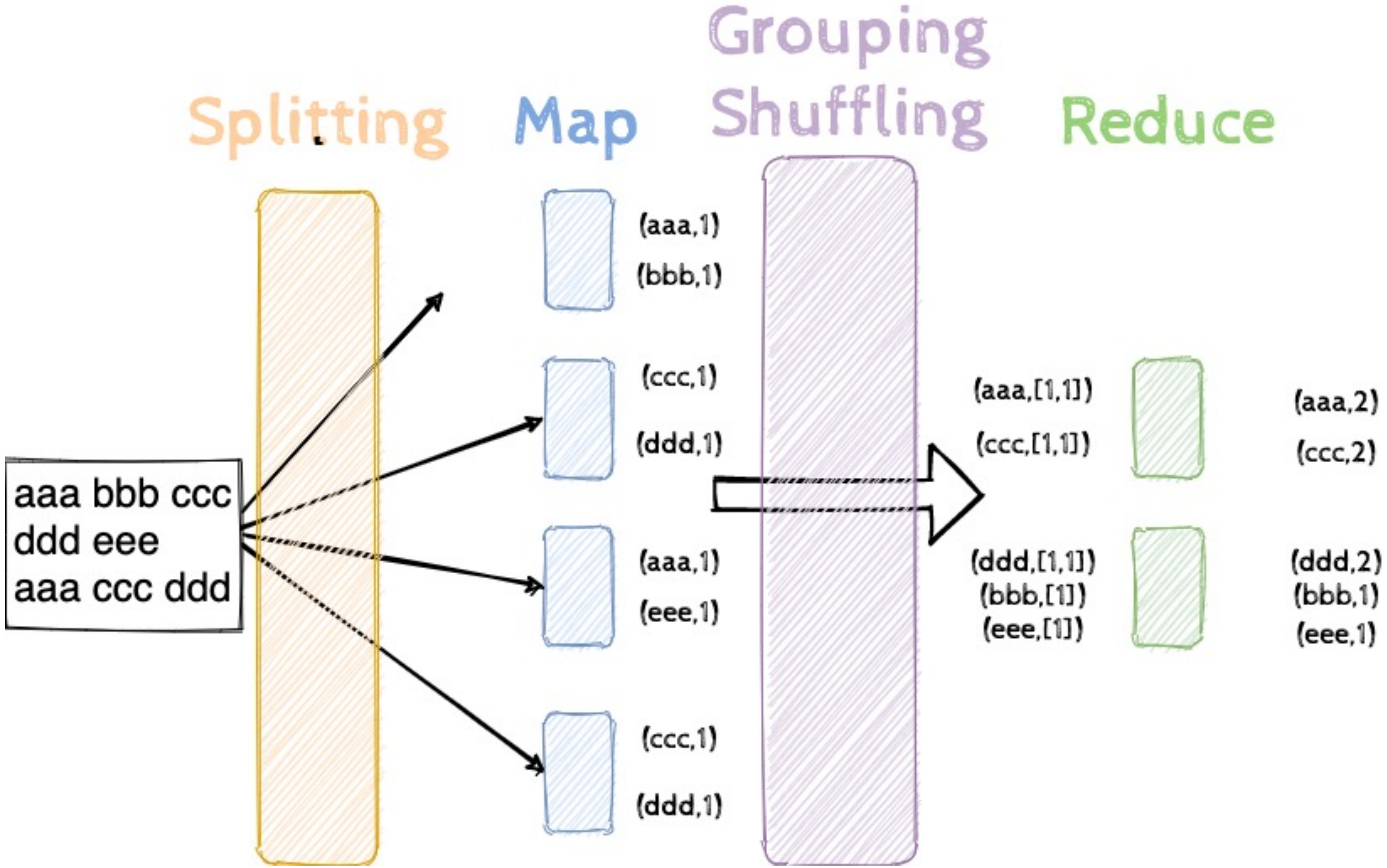

Map
Reduce
aaa bbb ccc
ddd eee
aaa ccc ddd
Magic
(aaa,1)
(bbb,1)
(ccc,1)
(ddd,1)
(aaa,1)
(eee,1)
(ccc,1)
(ddd,1)
Magic
(aaa,[1,1])
(ccc,[1,1])
(ddd,[1,1])
(bbb,[1])
(eee,[1])
(aaa,2)
(ccc,2)
(ddd,2)
(bbb,1)
(eee,1)

Benefits of MapReduce

- A lot of real world problems can be expressed as MapReduce
- Maps and reduce functions can be executed in parallel
  - Map works on independent data
  - Reduce works on a single key
- Reduce can have inner parallelism
  - If complex, can reduce with multiple threads

Implementations questions

- How is the input split into individual pairs for mappers?
- How are output of map grouped by key and sent to the correct reduce ?
- How is final result written ?
- All these are answered by a framework
  - All follow the same model but implementation may vary
- Basically, a MapReduce application has 4 phases
  - Splitting, Mapping, Grouping/Shuffling, Reducing

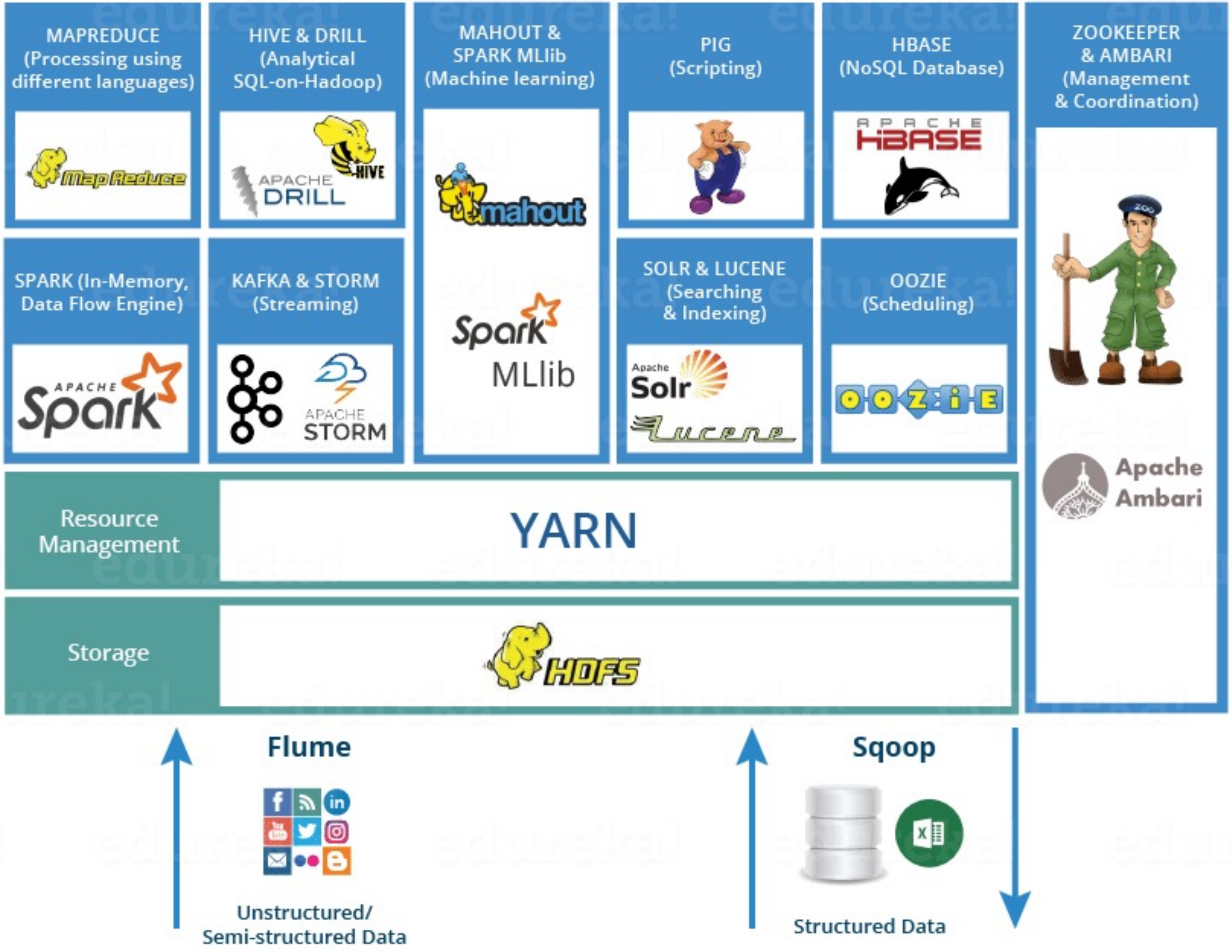

Grouping
Splitting
Map
Shuffling
Reduce
aaa bbb ccc
ddd eee
aaa ccc ddd
(aaa,1)
(bbb,1)
(ccc,1)
(ddd,1)
(aaa,1)
(eee,1)
(ccc,1)
(ddd,1)
(aaa,[1,1])
(ccc,[1,1])
(ddd,[1,1])
(bbb,[1])
(eee,[1])
(aaa,2)
(ccc,2)
(ddd,2)
(bbb,1)
(eee,1)

MapReduce

The Hadoop Framework

Introduction

- Hadoop is an open source implementation of Google MapReduce
- Provides full stack of services/frameworks
  - Not only MapReduce since version 2
- Most important components
  - HDFS
  - YARN
  - MapReduce



https://www.edureka.co/blog/hadoop-ecosystem

YARN

- Yet Another Resource Negotiator
- Manages resources (nodes)
  - Knows the nodes available
  - Can provide nodes to an application
- YARN doesn’t know or care about applications
  - So it can be used by any kind of application
- Main benefits
  - Nodes can be shared between user/applications
  - New applications/frameworks can use it and avoid managing resources
- Yarn is used outside of Hadoop

Writing Hadoop MapReduce Programs

- A MapReduce program is called a Job
- Main language is Java
  - But can use any executable or script
- Map function implemented in a Mapper
- Reduce function implemented in a Reducer
- Default implementation
  - Splitting
    - If input is text file, keys are whole line
  - Shuffling
    - Based on hash value of key

Mapper example

```
#!/usr/bin/env python
"""mapper.py"""

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

→ separator   key, value

https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

Reducer Example

```python
#!/usr/bin/env python
"""reducer.py"""

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

separator

https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

Execution

- A Job is submitted to a Hadoop Cluster
  - hadoop command
- All files have to be in HDFS
  - All paths relative to HDFS
- Number of mapper instances
  - Automatically decided based on the size (blocks) of input
- Number of reducers
  - Computed, can be set manually
  - Ideal value depends on the output of mappers

Execution - 2

- Mappers/Reducers are executed close to data if possible
  - Use replication factor
- Result of a Job is written to HDFS
  - In a directory
- Output directory contains 1 file per reducer instance
  - Named *part-000xxx*
- Jobs cannot overwrite existing files
  - Remember to remove previous results before new execution

```
Error Launching job : Output directory hdfs://localhost:9000/result.txt already exists
```

Hadoop Streaming

- A simple tool to use almost anything as map and reduce functions
  - Part of the standard Hadoop distribution
- Mappers and Reducers can be any exec or script
  - Works with Python
- Mappers
  - Read from files on HDFS
  - Write to standard output as text with tab as (key value) separator
- Reducers
  - Read from standard input, assume tab as separator
  - Get (key value) pairs (!)
  - Write to standard output, automatically saved to file

Hadoop Streaming

- Limitations
  - No real grouping/shuffling
  - Reduce receives multiple (key,value) pairs
  - Relies on files and STDIN/STDOUT for data transfer
- Example :
  - hadoop jar hadoop-streaming-3.1.4.jar -input /sample-text-file.txt -output /results -mapper mapper.py -reducer reducer.py

HDFS

HDFS

Beyond MapReduce

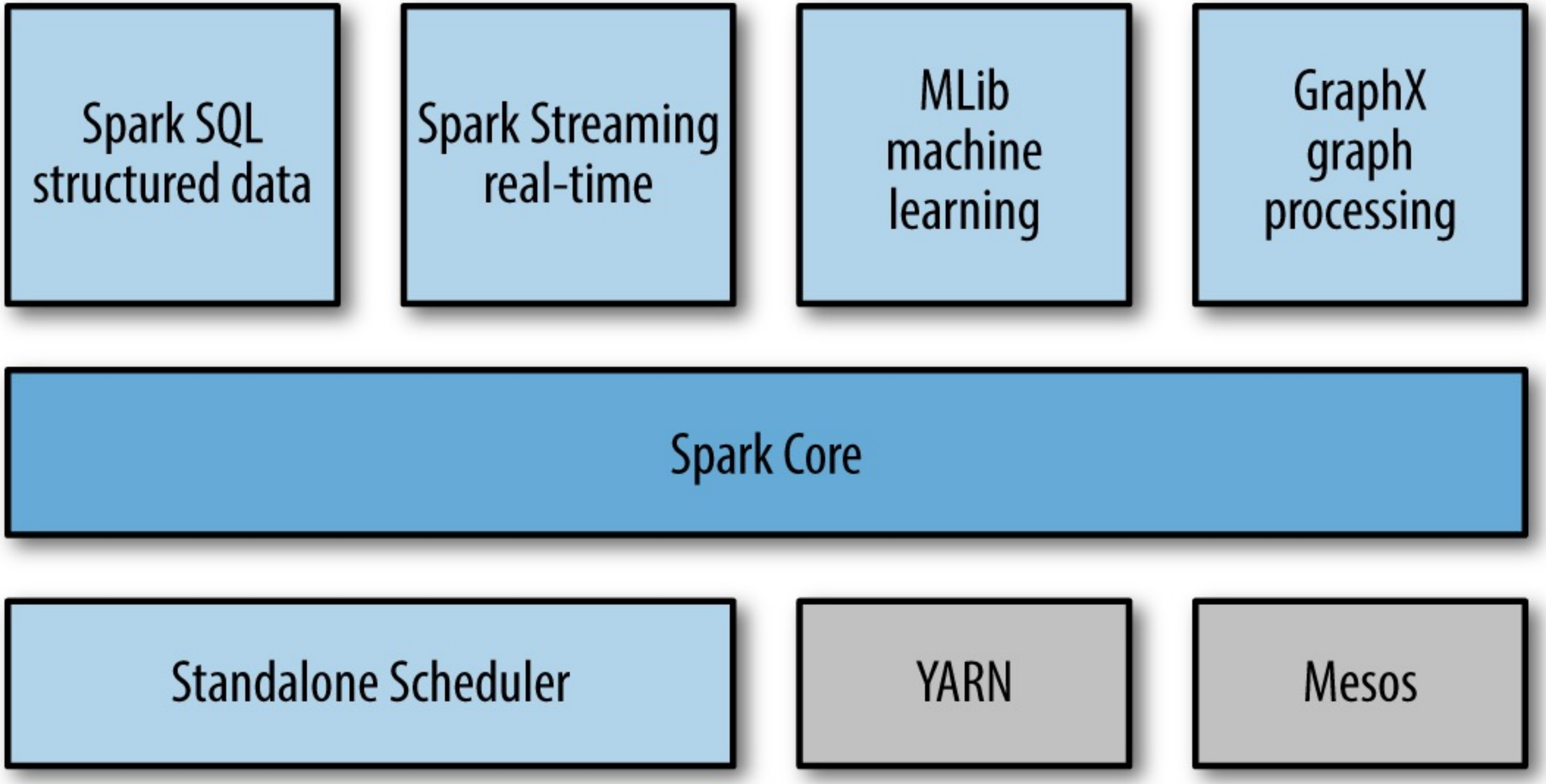
The Spark Framework

Limits of MapReduce & Hadoop

- Only 2 primitives
- No easy support for iterative applications
- Shuffling/grouping phase can be costly
- Hard to have map and reduces phases in parallel
  - Must wait for mappers to finish
- Heavy use of disk for intermediate results
  - Very slow

Apache Spark

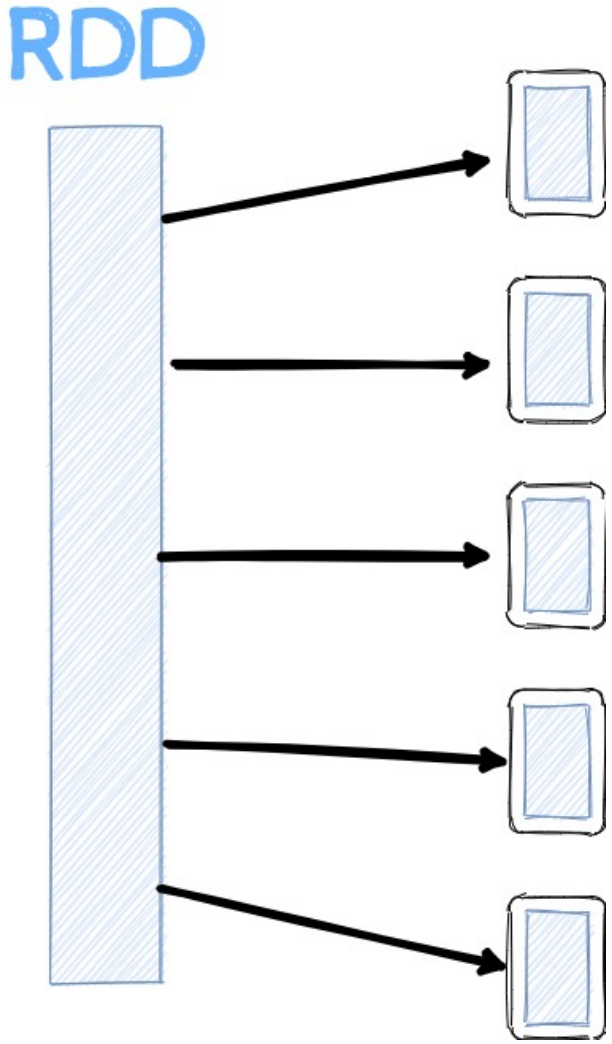
- Addresses limitations of Hadoop
- Spark
  - In-Memory computation
  - Workflows with cycles
  - Supports MapReduce like operations
  - Multi languages support : Scala, Java, Python, R
- https://spark.apache.org/



https://medium.com/analytics-vidhya/apache-spark-primer-ca1a6d060fc8

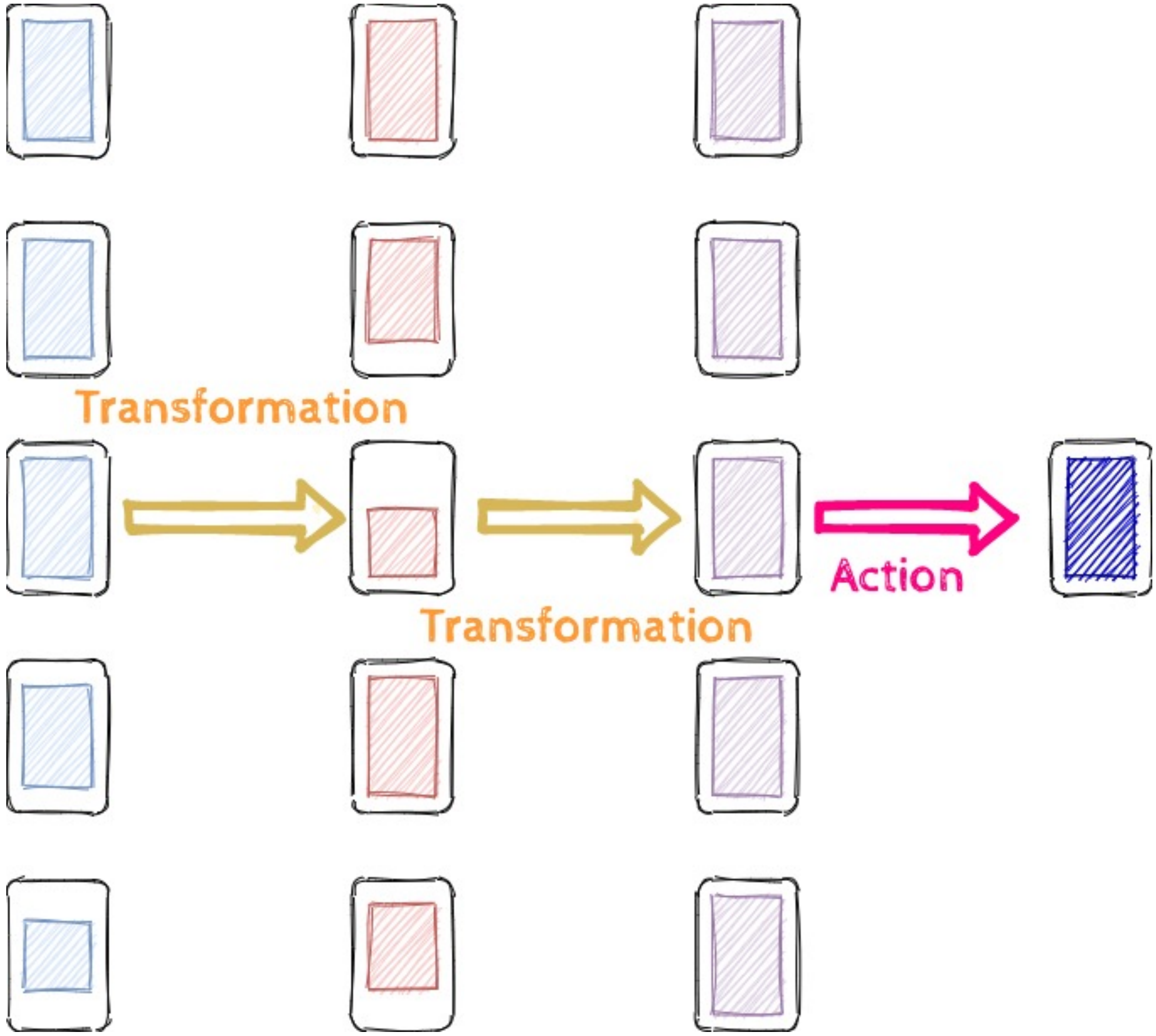
Resilient Distributed Datasets

- RDDs
  - Array-like data structure
  - Mostly in-memory
- Partitioned
  - An RDD is divided into blocks
  - Blocks are located on different machines
- Fault tolerant
  - Spark will remember operations on RDDs
  - Replay them in case of failure
- Immutable ← very important !
  - You CANNOT modify an RDD
  - You can only create a new one
    - Beware of memory usage



Resilient Distributed Datasets

- RDDs are created through **transformations**
  - Of raw data or another RDD
  - Example : map, filter, reduceByKey, groupBy…
- RDDs support **actions**
  - Action take an RDD and returns something NOT an RDD
  - Example : collect, count, reduce, save…
- Transformations are lazy !!!
  - Nothing will happen until you perform an action
  - Why ?


Transformation
Transformation
Action

Example : Word Count in Python

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1))  \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

2 values of the same key

Conclusion

- Hadoop and Spark are complex frameworks
  - Relies on other frameworks (Yarn…)
  - Used by higher level frameworks (ML…)
- Many other exists
  - Some are specialized on specific data
- How to choose one?
  - Performance
  - Language supported
  - Ease of use