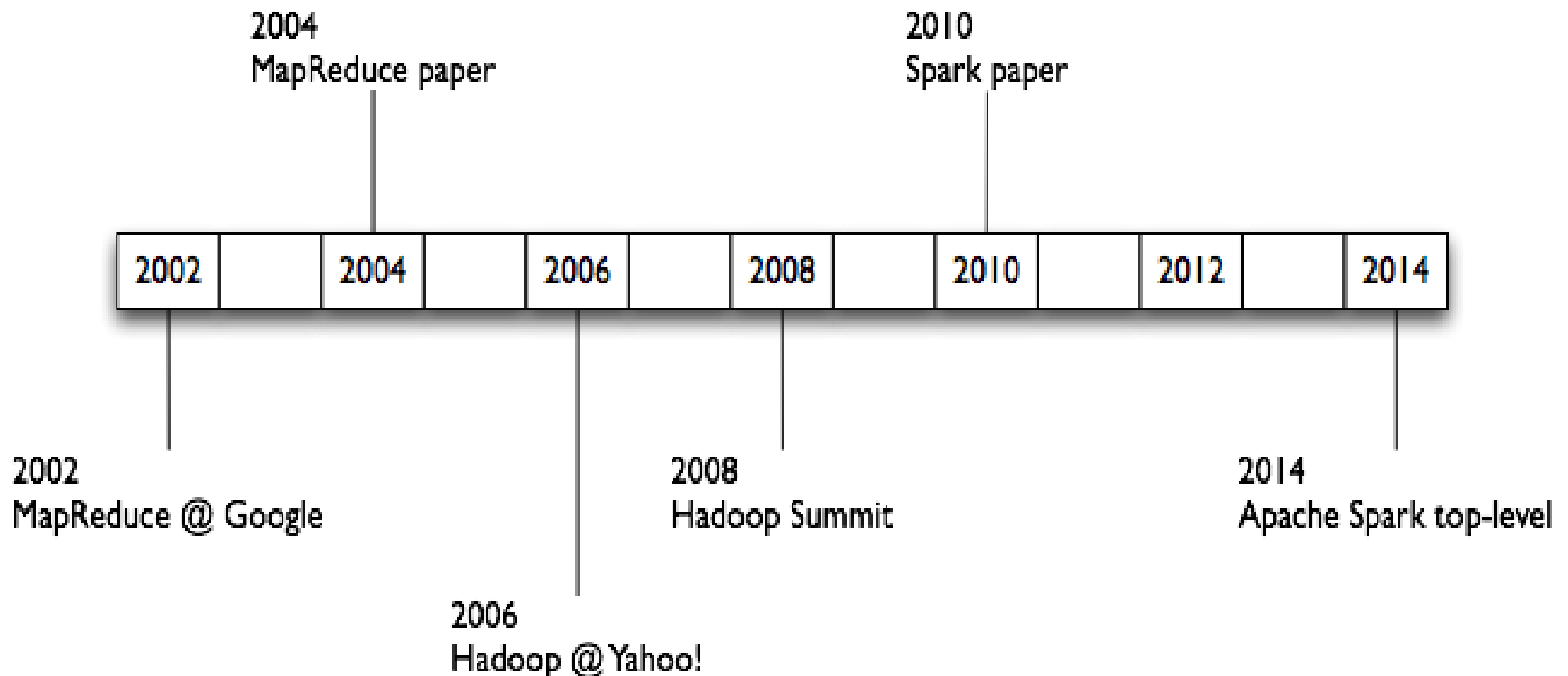# Introduction to Spark

Spark

- Apache Spark™ is a fast and general-purpose engine for large-scale data processing
- Spark aims at achieving the following goals in the Big data context
  - Generality: diverse workloads, operators, job sizes
  - Low latency: sub-second
  - Fault tolerance: faults are the norm, not the exception
  - Simplicity: often comes from generality

Spark History

- Originally developed at the University of California - Berkeley's AMPLab

2004
MapReduce paper

2010
Spark paper

| 2002 | | 2004 | | 2006 | | 2008 | | 2010 | | 2012 | | 2014 |

2002
MapReduce @ Google

2008
Hadoop Summit

2014
Apache Spark top-level

2006
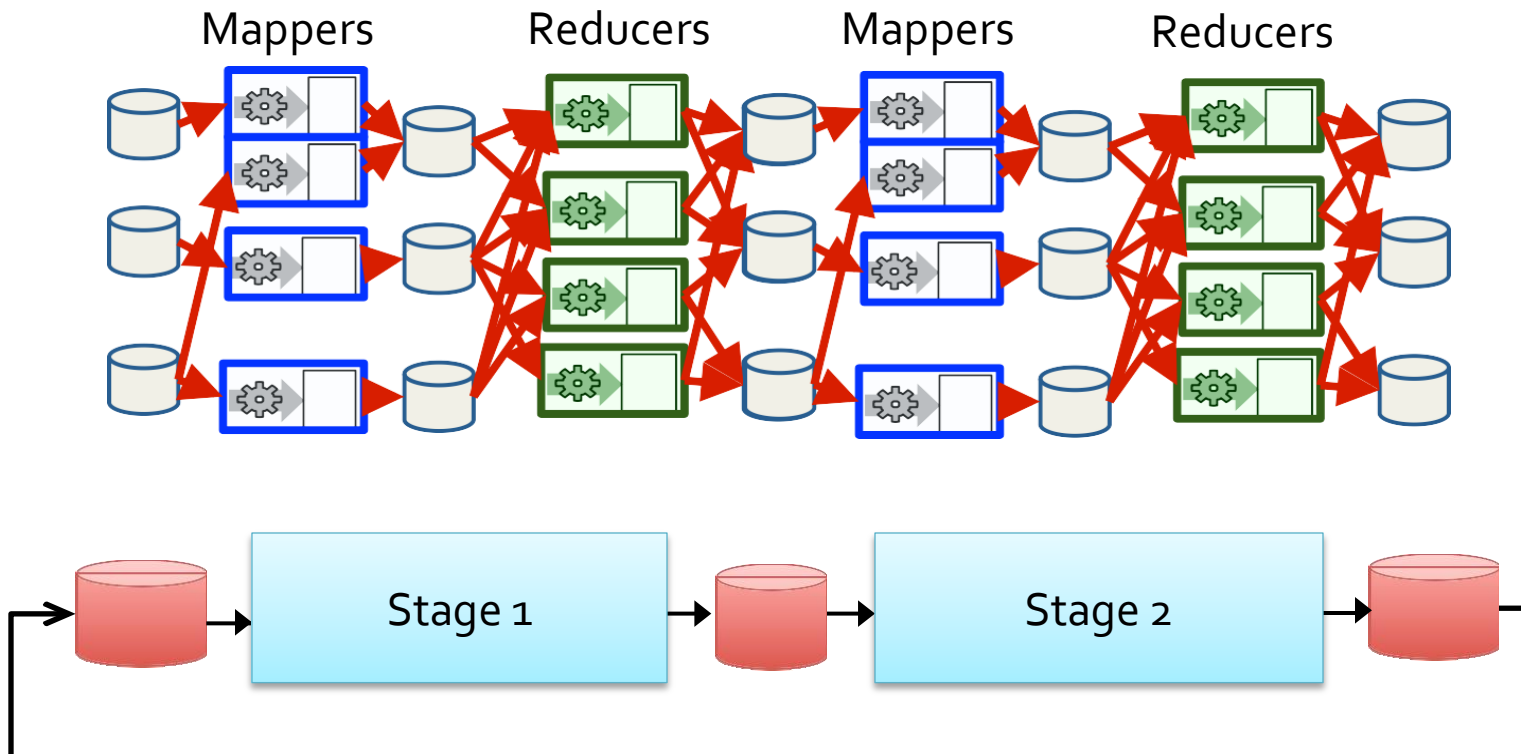Hadoop @ Yahoo!

Spark: Motivations

MapReduce and Iterative Jobs

- Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage
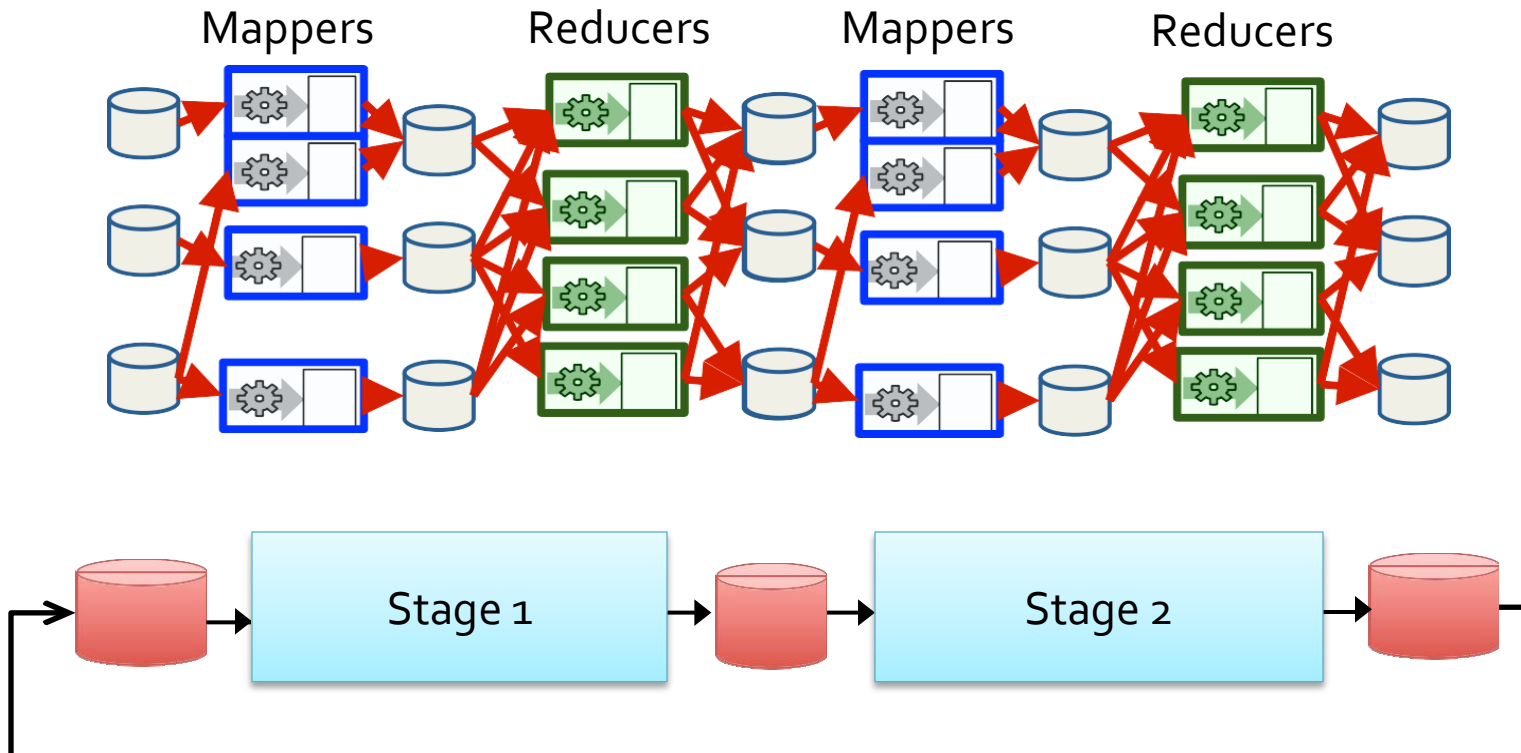
MapReduce and Iterative Jobs

- Disk I/O is very slow (even if it is local I/O)

Apache Spark: Motivation and Opportunity

- Motivation
  - Using MapReduce for complex **iterative jobs** or **multiple jobs on the same data** involves lots of disk I/O
- Opportunity
  - The **cost** of **main memory decreased**
    - Hence, large main memories are available in each server
- Solution
  - Keep **more data in main memory**
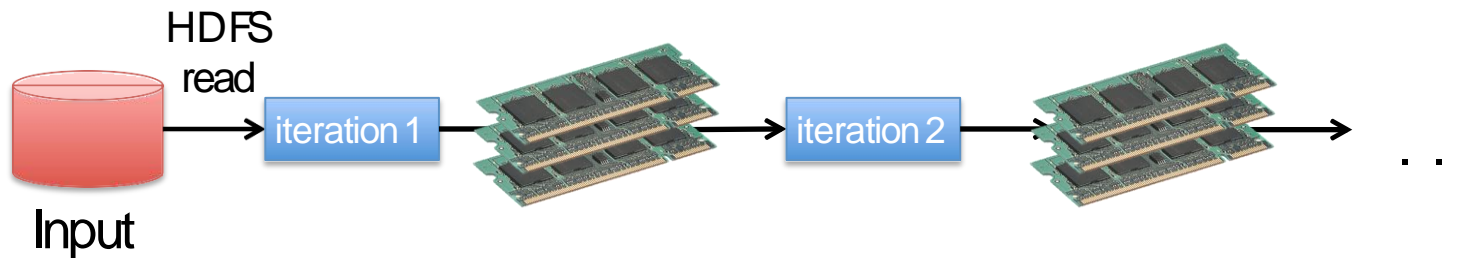   - Basic idea of Spark

From MapReduce to Spark

- MapReduce: Iterative job
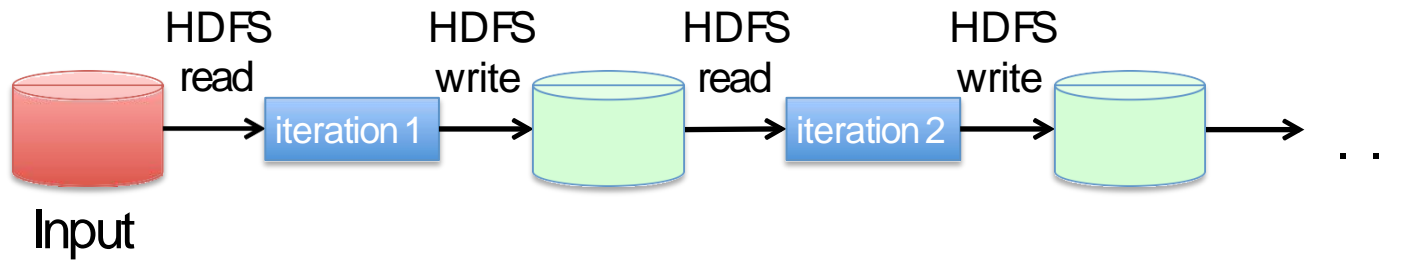
From MapReduce to Spark

- Spark: Iterative job

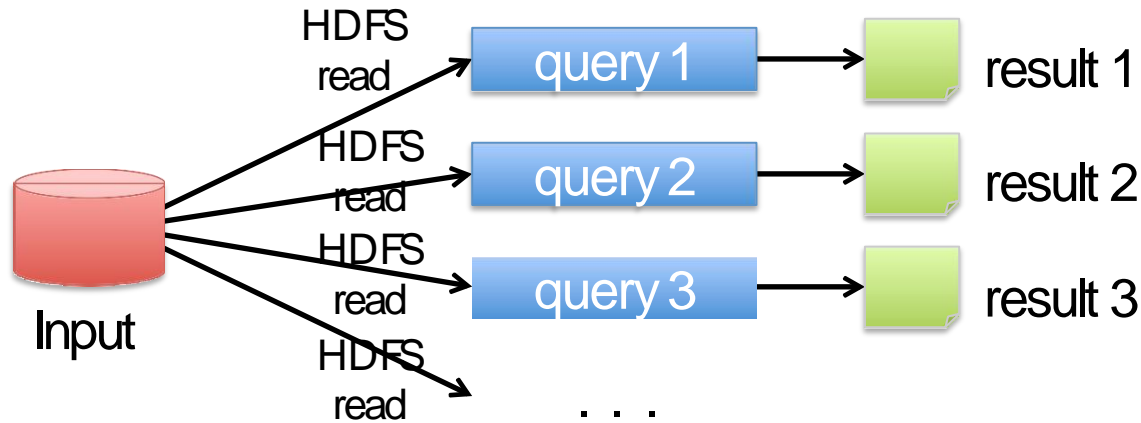- Data are shared between the iterations by using the main memory
  - Or at least part of them
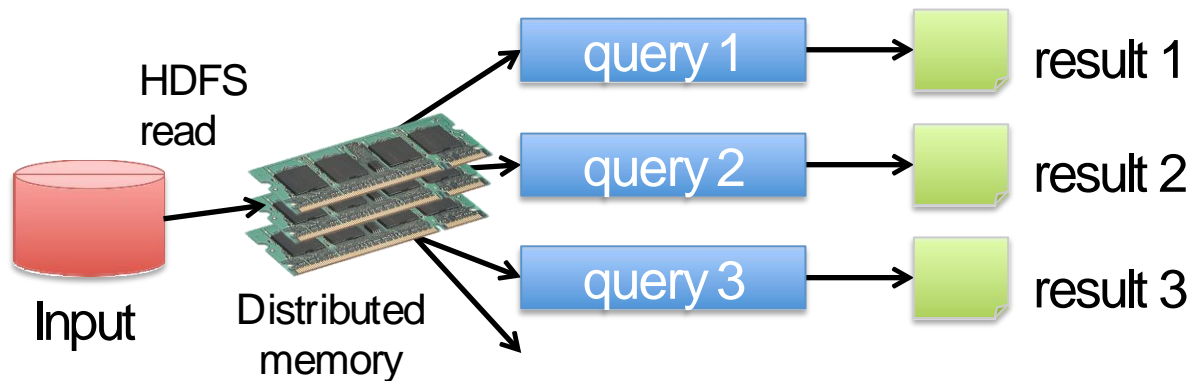- 10 to 100 times faster than disk

From MapReduce to Spark

- MapReduce: Multiple analyses of the same data

From MapReduce to Spark

- Spark: Multiple analyses of the same data

- Data are read only once from HDFS and stored in main memory
  - Split of the data across the main memory of each server

Spark: Resilient Distributed Data sets (RDDs)

- Data are represented as Resilient Distributed Datasets (RDDs)

  - Partitioned/Distributed collections of objects spread across the nodes of a cluster

  - Stored in main memory (when it is possible) or on local disk

- Spark programs are written in terms of operations on resilient distributed data sets

Spark: Resilient Distributed Data sets (RDDs)

- RDDs are built and manipulated through a set of parallel
  - Transformations
    - map, filter, join, …
  - Actions
    - count, collect, save, …
- RDDs are automatically rebuilt on machine failure

Spark Computing Framework

- Provides a programming abstraction (based on RDDs) and transparent mechanisms to execute code in parallel on RDDs
  - Hides complexities of fault-tolerance and slow machines
  - Manages scheduling and synchronization of the jobs

MapReduce vs Spark

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc…
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, Python, and R

MapReduce vs Spark

- Lower overhead for starting jobs
- Less expensive shuffles

In-Memory RDDs Can Make a Big Difference

- Two iterative Machine Learning algorithms:
  - K-means Clustering
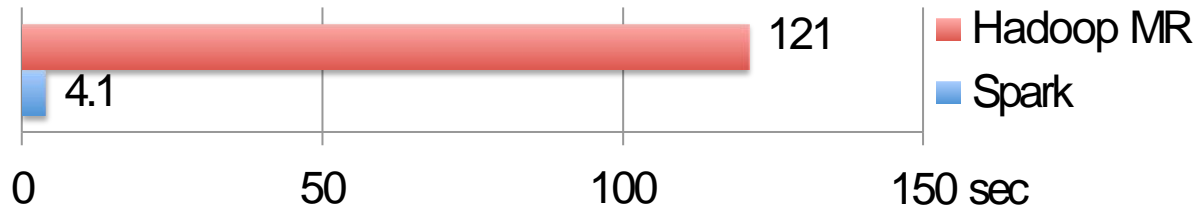
  - Logistic Regression

Petabyte Sort Challenge

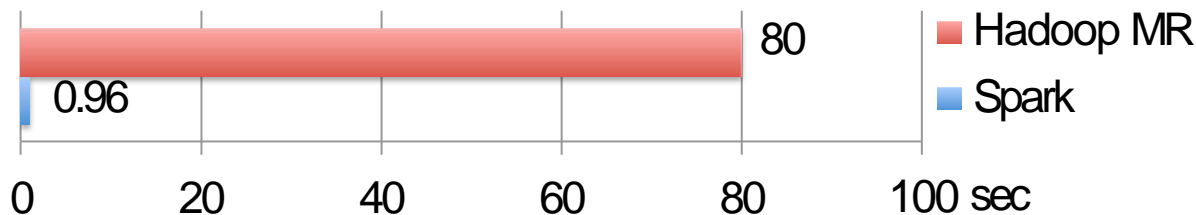|	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Daytona Gray 100 TB sort benchmark record (tied for 1st place)

Spark: Main components

Spark Components
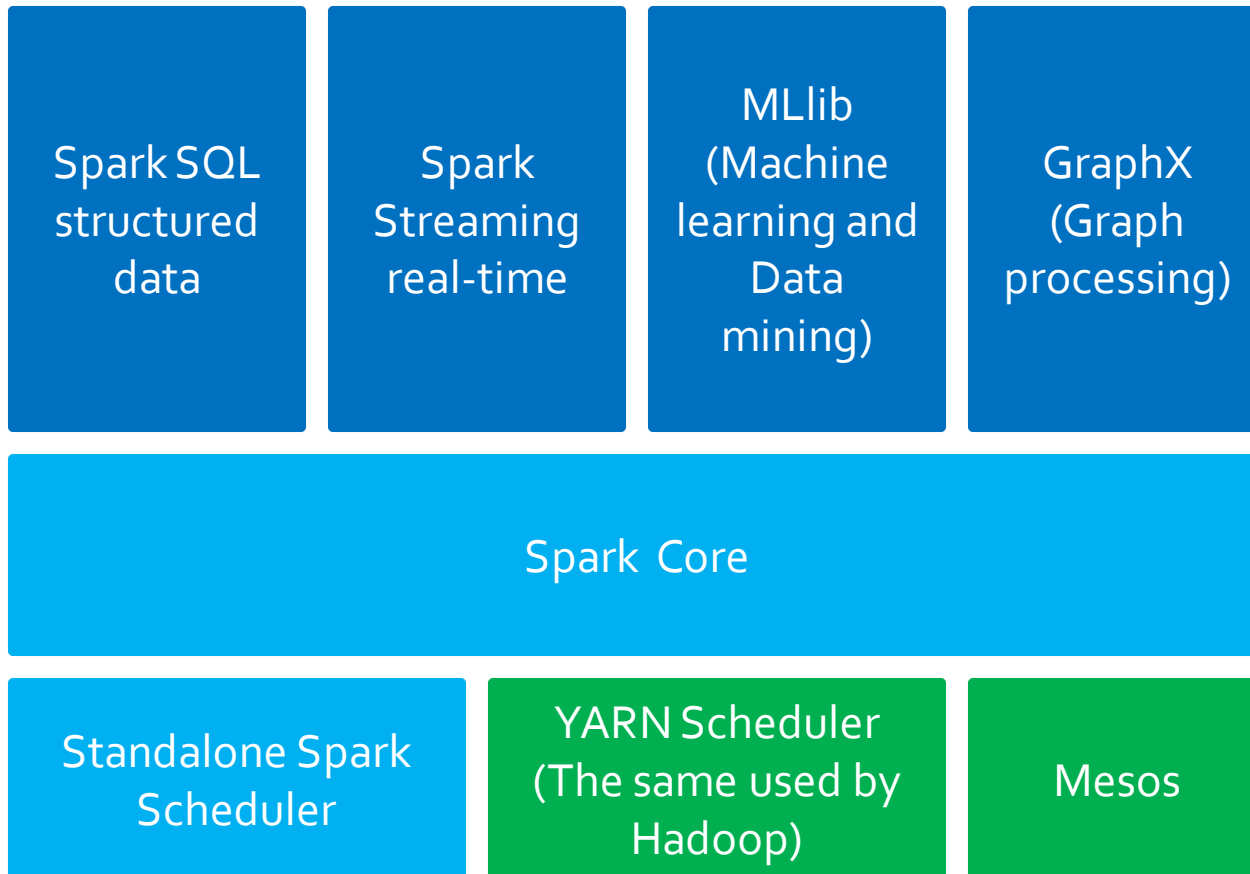
Spark Components

- Spark is based on a basic component (the Spark Core component) that is exploited by all the high-level data analytics components

  - This solution provides a more uniform and efficient solution with respect to Hadoop where many non-integrated tools are available

- When the efficiency of the core component is increased also the efficiency of the other high-level components increases

Spark Components

- Spark Core
  - Contains the basic functionalities of Spark exploited by all components
    - Task scheduling
    - Memory management
    - Fault recovery
    - ...
  - Provides the APIs that are used to create RDDs and applies transformations and actions on them

Spark Components

- Spark SQL structured data
  - This component is used to interact with structured datasets by means of the SQL language or specific querying APIs
    - Based on Datasets
  - It supports also
    - Hive Query Language (HQL)
  - It interacts with many data sources
    - Hive Tables, Parquet, Json, ..
  - It exploits a query optimizer engine

Spark Components

- Spark Streaming real-time
  - It is used to process live streams of data in real-time
  - The APIs of the Streaming real-time components operated on RDDs and are similar to the ones used to process standard RDDs associated with "static" data sources

Spark Components

- MLlib
  - It is a machine learning/data mining library
  - It can be used to apply the parallel versions of some machine learning/data mining algorithms
    - Data preprocessing and dimensional reduction
    - Classification algorithms
    - Clustering algorithms
    - Itemset mining
    - ....

Spark Components

- GraphX
  - A graph processing library
  - Provides algorithms for manipulating graphs
    - Subgraph searching
    - PageRank
    - ....
  - The Python version is not available
- GraphFrames
  - A graph library based on DataFrames and Python

Spark Schedulers

- Spark can exploit many schedulers to execute its applications
  - Hadoop YARN
    - Standard scheduler of Hadoop
  - Mesos cluster
    - Another popular scheduler
  - Standalone Spark Scheduler
    - A simple cluster scheduler included in Spark

Spark Basic Concepts

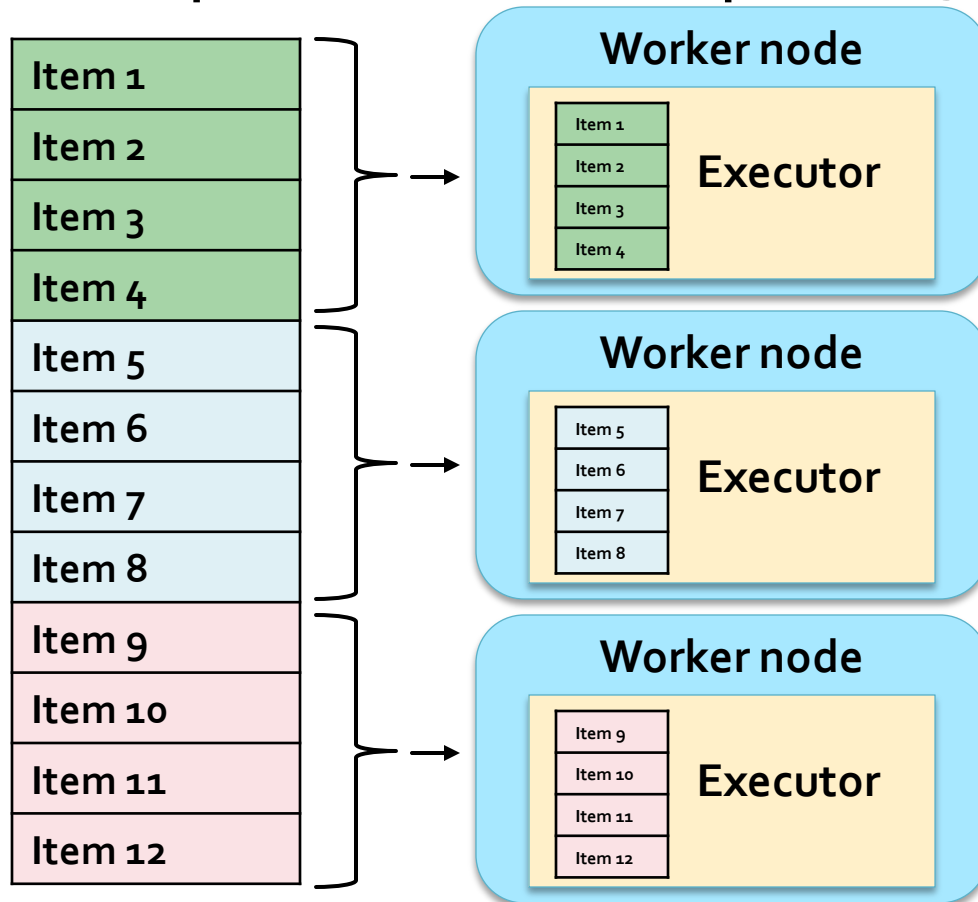# Resilient Distributed Data sets (RDDs)

- RDDs are the primary abstraction in Spark
- RDDs are distributed collections of objects spread across the nodes of a clusters
  - They are split in partitions
  - Each node of the cluster that is running an application contains at least one partition of the RDD(s) that is (are) defined in the application

Resilient Distributed Data sets (RDDs)

- RDDs
  - Are stored in the main memory of the executors running in the nodes of the cluster (when it is possible) or in the local disk of the nodes if there is not enough main memory
  - Allow executing in parallel the code invoked on them
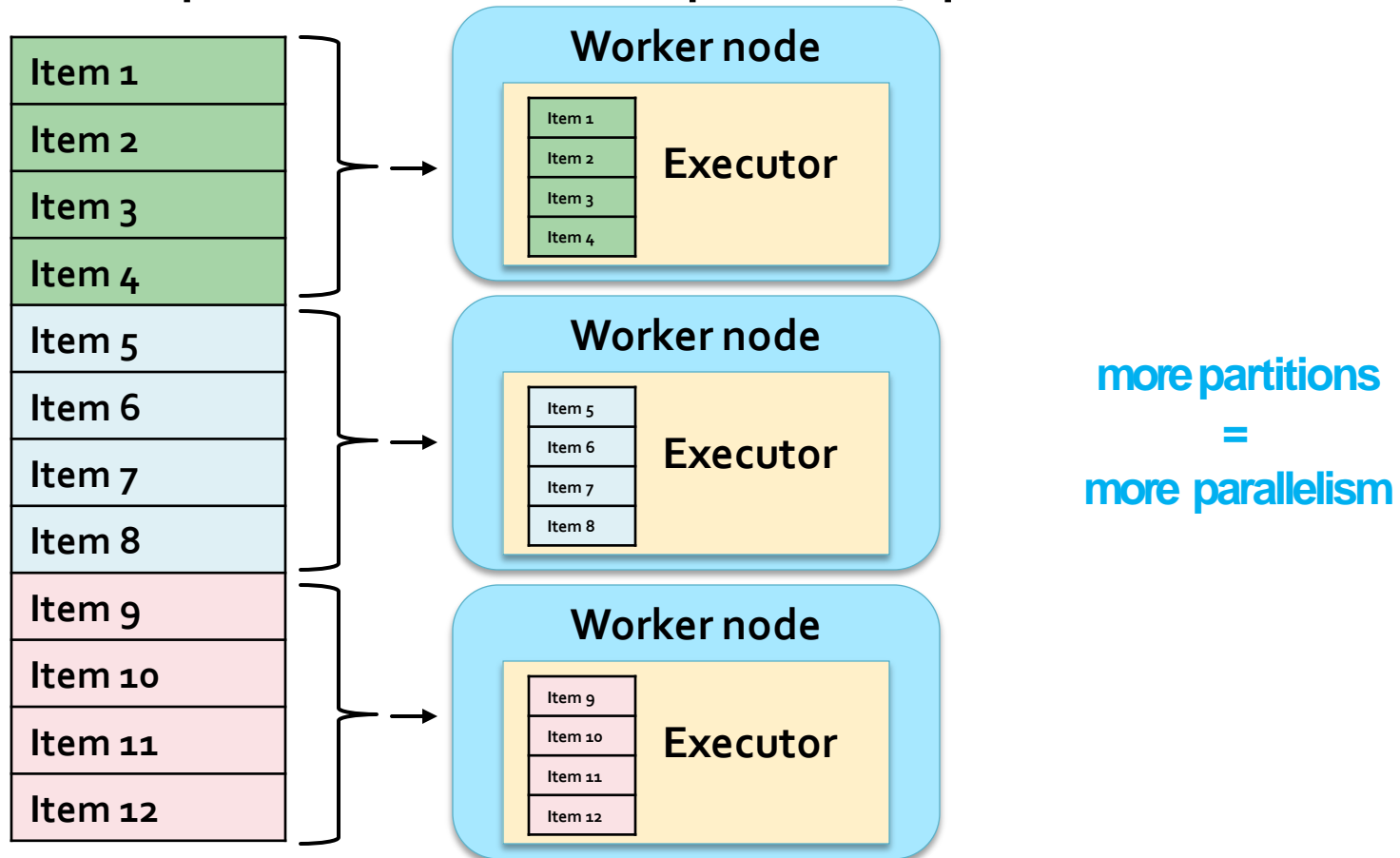   - Each executor of a worker node runs the specified code on its partition of the RDD

Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions

Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions

Resilient Distributed Data sets (RDDs)

- RDDs
  - Are immutable once constructed
    - i.e., the content of an RDD cannot be modified
- Spark tracks lineage information to efficiently recompute lost data (due to failures of some executors)
    - i.e., for each RDD, Spark knows how it has been constructed and can rebuilt it if a failure occurs
    - This information is represented by means of a DAG (Direct Acyclic Graph) connecting input data and RDDs

Resilient Distributed Data sets (RDDs)

- RDDs can be created
  - by parallelizing existing collections of the hosting programming language (e.g., collections and lists of Scala, Java, Pyhton, or R)
    - In this case the number of partition is specified by the user
  - from (large) files stored in HDFS
    - In this case there is one partition per HDFS block
  - from files stored in many traditional file systems or databases
  - by transforming an existing RDDs
    - The number of partitions depends on the type of transformation

Resilient Distributed Data sets (RDDs)

- Spark programs are written in terms of operations on resilient distributed data sets
  - Transformations
    - map, filter, join, …
  - Actions
    - count, collect, save, …

Spark Framework

- Spark
  - Manages scheduling and synchronization of the jobs
  - Manages the split of RDDs in partitions and allocates RDDs' partitions in the nodes of the cluster
  - Hides complexities of fault-tolerance and slow machines
    - RDDs are automatically rebuilt in case of machine failures

Spark Programs

Supported languages

- Spark supports many programming languages
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX)
  - Java
  - Python
  - R

Supported languages

- Spark supports many programming languages
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Sparl SQL, Spark Streaming, MLlib, GraphX)
  - Java
  - **Python ← We will use Python**
  - R

Structure of Spark programs

- The Driver program
  - Contains the main method
  - "Defines" the workflow of the application
  - Accesses Spark through the **SparkContext** object
    - The SparkContext object represents a connection to the cluster
  - Defines Resilient Distributed Datasets (RDDs) that are "allocated" in the nodes of the cluster
  - Invokes parallel operations on RDDs

Structure of Spark programs

- The Driver program defines
  - Local variables
    - The standard variables of the Python programs
  - RDDs
    - Distributed "variables" stored in the nodes of the cluster
  - The SparkContext object allows
    - Creating RDDs
    - "Submitting" executors (processes) that execute in parallel specific operations on RDDs
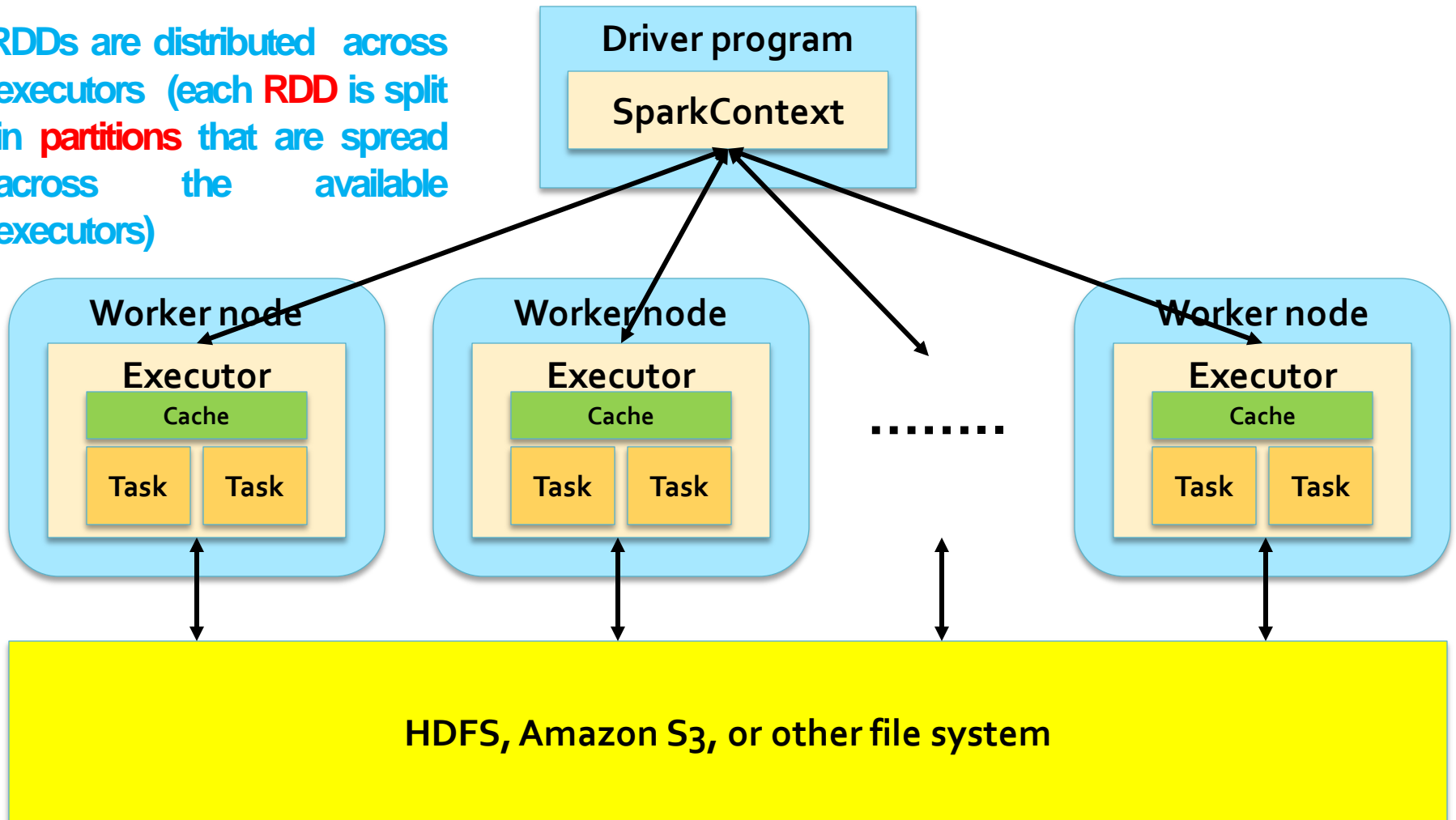   - Transformations and Actions

Structure of Spark programs

- The worker nodes of the cluster are used to run your application by means of **executors**
- Each executor runs on its partition of the RDD(s) the operations that are specified in the driver

Distributed execution of Spark
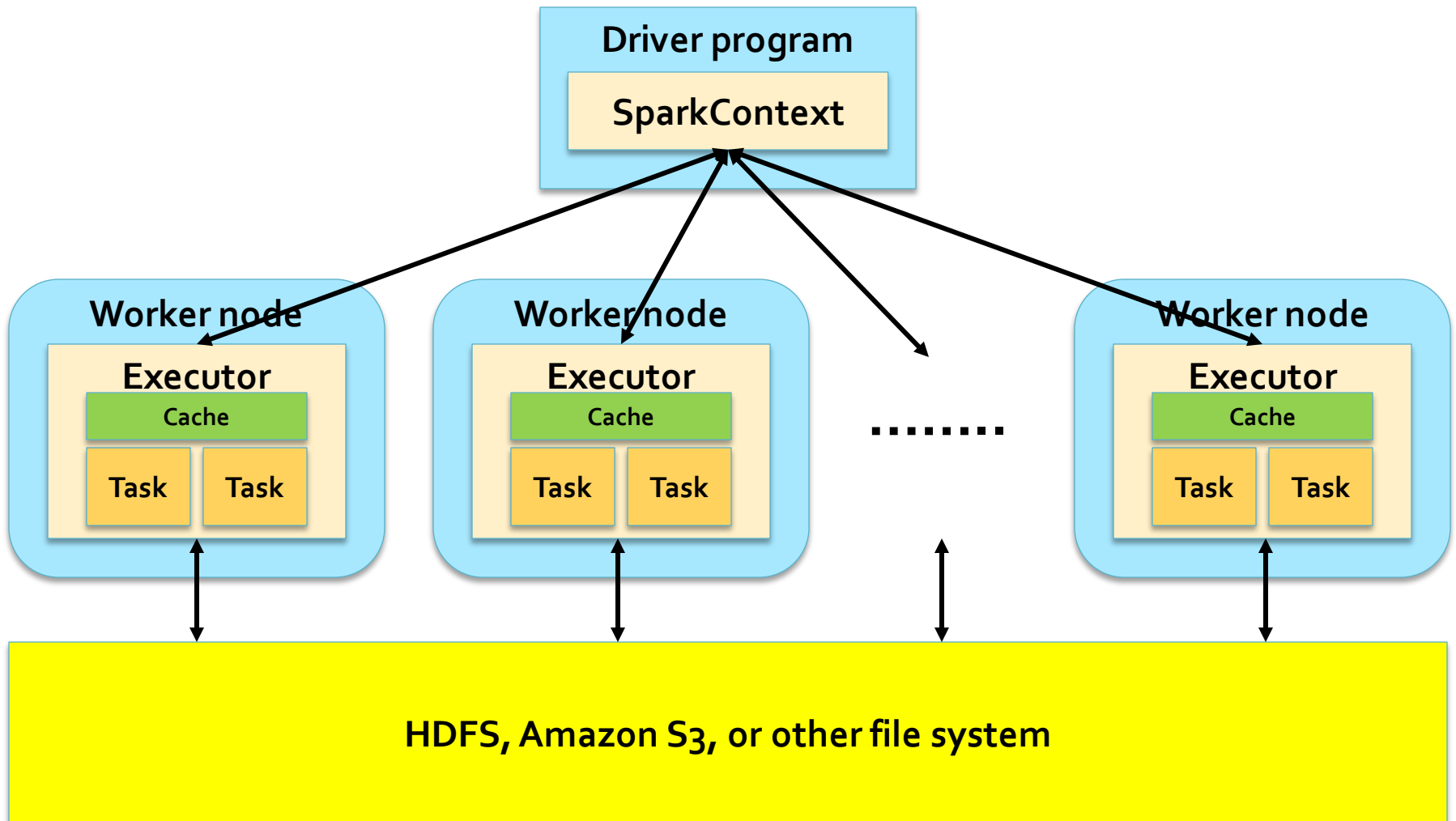
Distributed execution of Spark

RDDs are distributed across executors (each **RDD** is split in **partitions** that are spread across the available executors)

Driver program

SparkContext

Worker node

Executor

Cache

Task | Task

Worker node

Executor

Cache

Task | Task

....

Worker node

Executor

Cache

Task | Task

HDFS, Amazon S3, or other file system

Local execution of Spark

- Spark programs can also be executed locally
  - Local threads are used to parallelize the execution of the application on RDDs on a single PC
    - Local threads can be seen are "pseudo-worker" nodes
  - It is useful to develop and test the applications before deploying them on the cluster
  - A local scheduler is launched to run Spark programs locally

Local execution of Spark
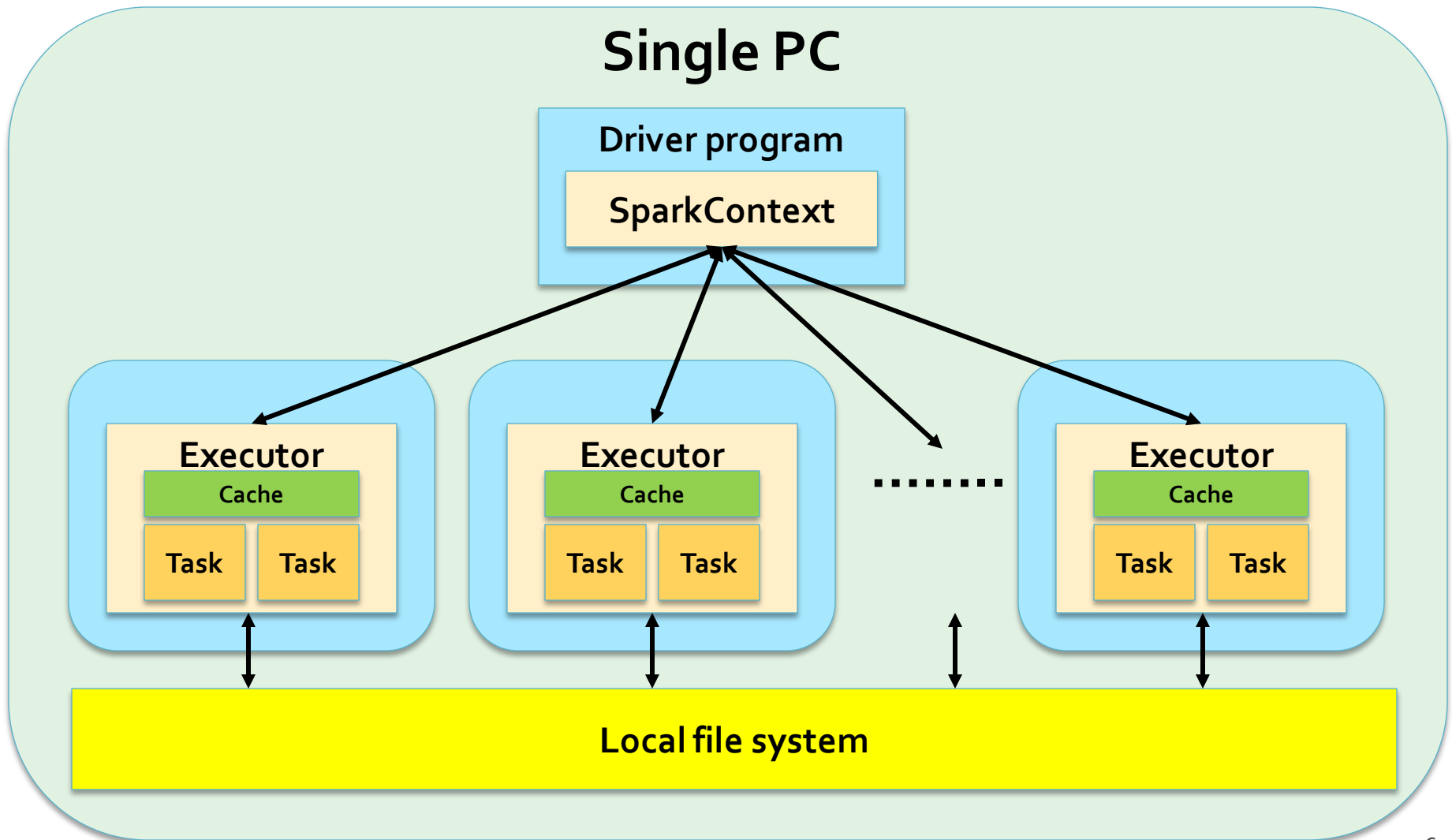
Spark official terminology

- Application
  - User program built on Spark
  - It consists of a driver program and executors on the cluster
- Driver program
  - The process running the main() function of the application and creating the SparkContext

Based on http://spark.apache.org/docs/latest/cluster-overview.html

Spark official terminology

- Cluster manager
  - An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- Deploy mode
  - Distinguishes where the driver process runs
    - In "cluster" mode, the framework launches the driver inside of the cluster
    - In "client" mode, the submitter launches the driver outside of the cluster
- Worker node
  - Any node of the cluster that can run application code in the cluster

Spark official terminology

- Executor
  - A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them
  - Each application has its own executors
- Task
  - A unit of work that will be sent to one executor
- Job
  - A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect)

Spark official terminology

- Stage
  - Each **job** gets **divided into** smaller sets of tasks called **stages**
  - The output of one stage is the input of the next stage(s)
    - Except the stages that compute (part of) the final result (i.e., the stages without output edges in the graph representing the workflow of the application)
      - The outputs of those stages is stored in HDFS or a database
  - The **shuffle operation** is always executed **between** two **stages**
    - Data must be grouped/repartitioned based on a grouping criteria that is different with respect to the one used in the previous stage
    - Similar to the shuffle operation between the map and the reduce phases in MapReduce
    - **Shuffle** is a **heavy operation**

Spark Programs: Examples

Spark program: Count line

- Count the number of lines of the input file
  - The name of the file is set to "myfile.txt"
- Print the results on the standard output

Spark program: Count line

```python
from pyspark import SparkConf, SparkContext

if __name__ == "__main__":

    #Create a configuration object and
    #set the name of the application
    conf = SparkConf().setAppName("Spark Line Count")

    # Create a Spark Context object
    sc = SparkContext(conf=conf)

    # Store the path of the input file in inputfile
    inputFile= "myfile.txt"
```

Spark program: Count line

```
# Build an RDD of Strings from the input textual file
# Each element of the RDD is a line of the input file
linesRDD = sc.textFile(inputFile)

# Count the number of lines in the input file
# Store the returned value in the local variable numLines
numLines = linesRDD.count()
# Print the output in the standard output
print("NumLines:", numLines)

# Close the Spark Context object
sc.stop()
```

Spark program: Count line

```python
from pyspark import SparkConf, SparkContext

if __name__ == "__main__":

    #Create a configuration object and
    #set the name of the application
    conf = SparkConf().setApp

    # Create a Spark Context
    sc = SparkContext(conf=conf)
    # Store the path of the input file in inputfile
    inputFile = "myfile.txt"
```

Local Python variables.
They are allocated in the main memory
of the same process instancing
the Driver

Spark program: Count line

```
# Build an RDD of Strings from the input textual file
# Each element of the RDD is a line of the input file
linesRDD = sc.textFile(inputFile)


# Count the number of lines
# Store the returned value in the local variable numLines
numLines = linesRDD.count()
# Print the output in the standard output
print("NumLines:", numLines)


# Close the Spark Context
sc.stop()
```

RDD.
It is allocated/stored in the main memory or in the local disk of the executors of the worker nodes

Local Python variable.
It is allocated in the main memory of the same process instancing the Driver

Spark program: Count line

- Local variables
  - Can be used to store only "small" objects/data
    - The maximum size is equal to the main memory of the process associated with the Driver
- RDDs
  - Are used to store "big/large" collections of objects/data in the nodes of the cluster
    - In the main memory of the worker nodes, when it is possible
    - In the local disks of the worker nodes, when it is necessary

Spark program: Word Count

- Word Count implemented by means of Spark
  - The name of the input file is specified by using a command line parameter (i.e., argv[1])
  - The output of the application (i.e., the pairs (word, num. of occurrences) is stored in an output folder (i.e., argv[2])
- **Note:** Do not worry about details

Spark program: Word Count

```python
from pyspark import SparkConf, SparkContext
import sys

if __name__ == "__main__":
    """

    Word count example
    """

    inputFile= sys.argv[1]
    outputPath = sys.argv[2]

    #Create a configuration object and
    #set the name of the application
    conf = SparkConf().setAppName("Spark Word Count")

    # Create a Spark Context object
    sc = SparkContext(conf=conf)
```

Spark program: Word Count

```
# Build an RDD of Strings from the input textual file
# Each element of the RDD is a line of the input file
lines = sc.textFile(inputFile)

# Split/transform the content of lines in a
# list of words and store them in the words RDD
words = lines.flatMap(lambda line: line.split(sep=' '))

#Map/transform each word in the words RDD
#to a pair/tuple (word,1) and store the result in the words_one RDD
words_one = words.map(lambda word: (word, 1))
```

Spark program: Word Count

```
# Count the num. of occurrences of each word.
# Reduce by key the pairs of the words_one RDD and store
# the result (the list of pairs (word, num. of occurrences)
# in the counts RDD
counts = words_one.reduceByKey(lambda c1, c2: c1 + c2)

# Store the result in the output folder
counts.saveAsTextFile(outputPath)

# Close/Stop the Spark Context object
sc.stop()
```