Data Analytics using

APACHE Spark™

Unified engine for large-scale data analytics

Sukeshini
Big Data Analytics & Machine Learning Team
C-DAC Bengaluru
sukeshini@cdac.in

Introduction to Spark

What is Apache Spark?

"Apache Spark™ is a multi-language engine for executing Data Engineering, Data Science, and Machine Learning on single-node machine or clusters."

Latest version – 3.2.1, Released on Jan 26, 2022

Spark Motivation

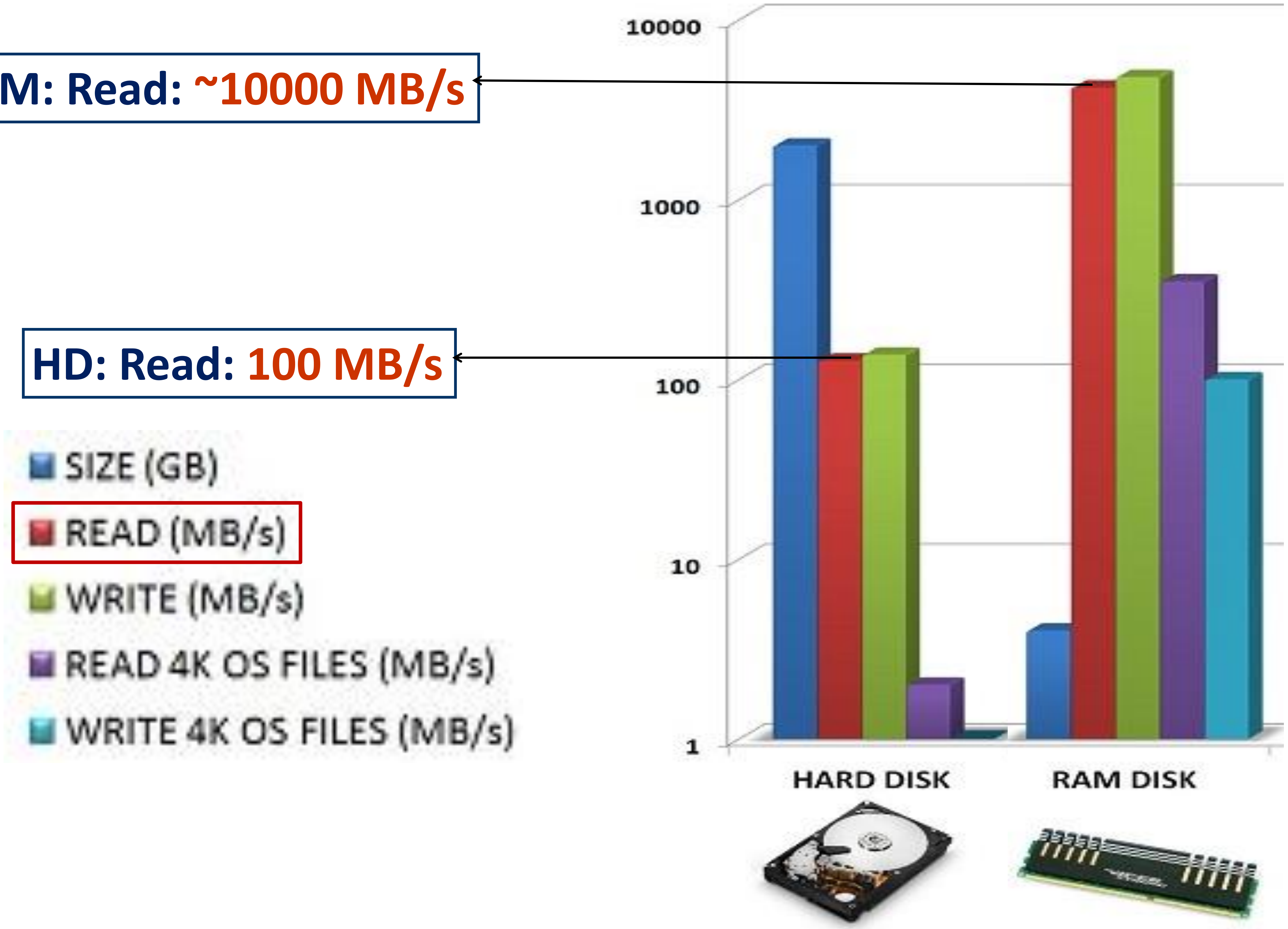
Disk IO is very slow…

Cost of Memory has come down…

Opportunity:

More "Memory"
Keep More Data "in Memory"

Spark

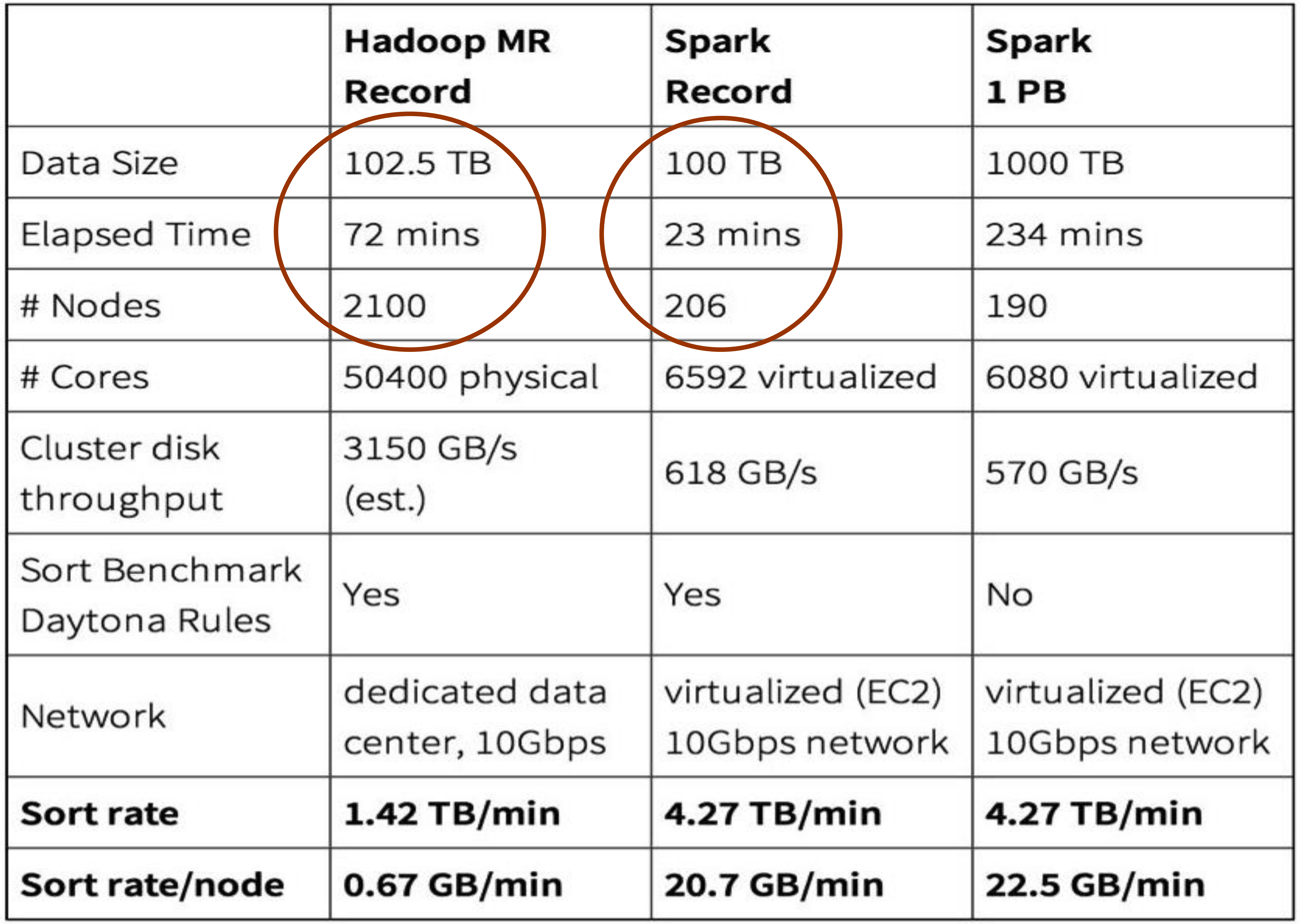


RAM: Read: ~10000 MB/s

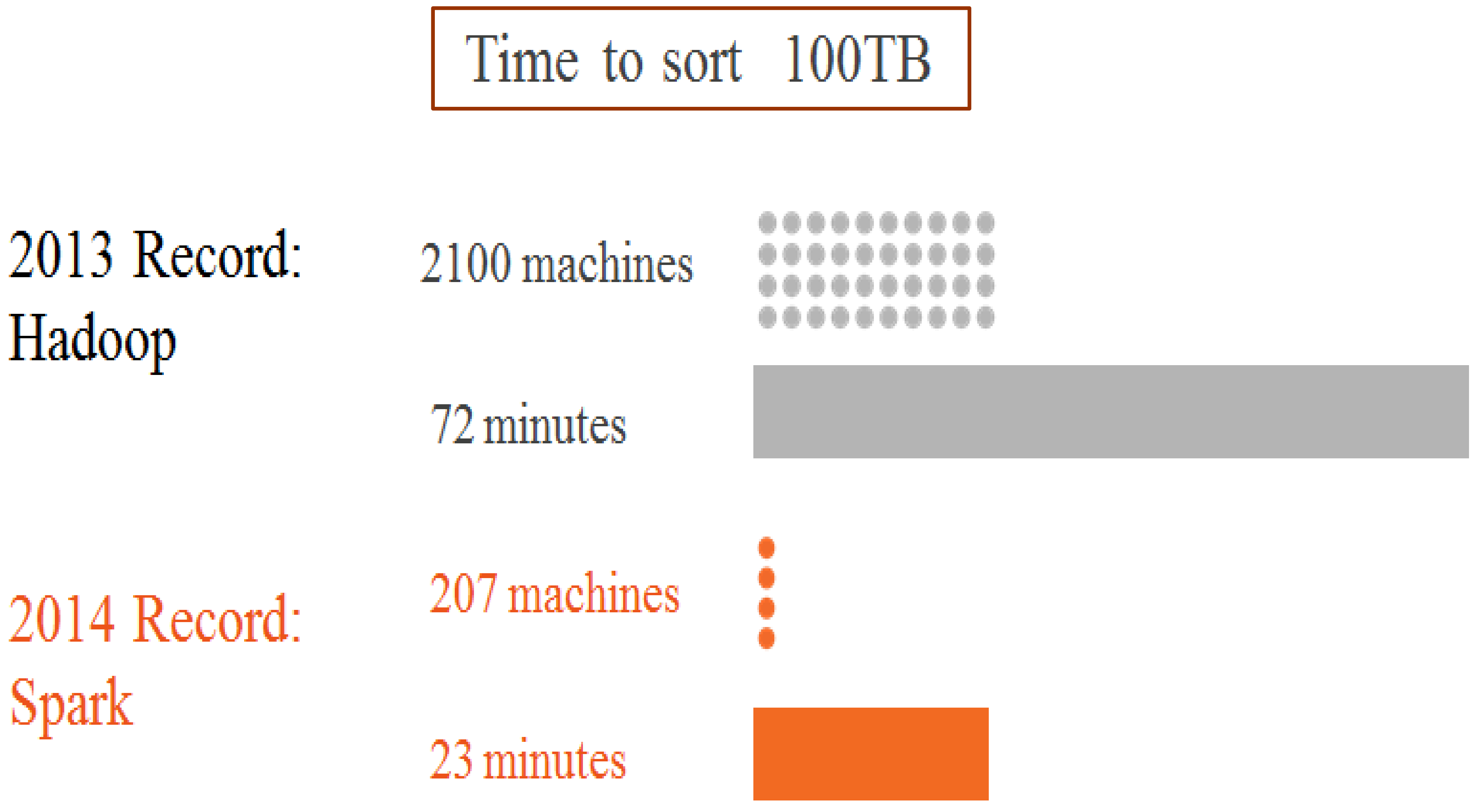
HD: Read: 100 MB/s

Ref: http://www.storagereview.com/images/ramdiskarticle1.jpg

Spark sets a new record in Petabyte Sort

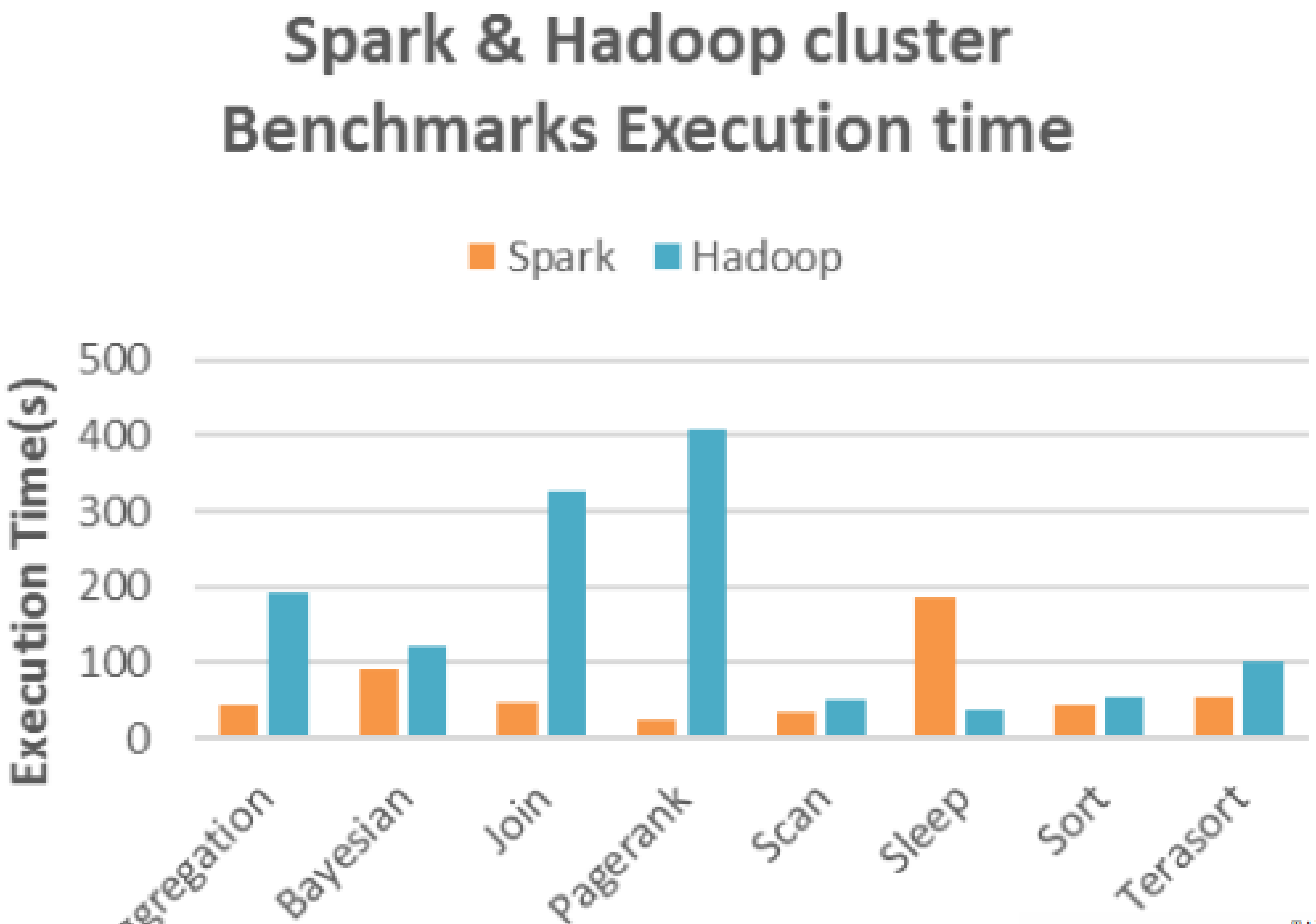


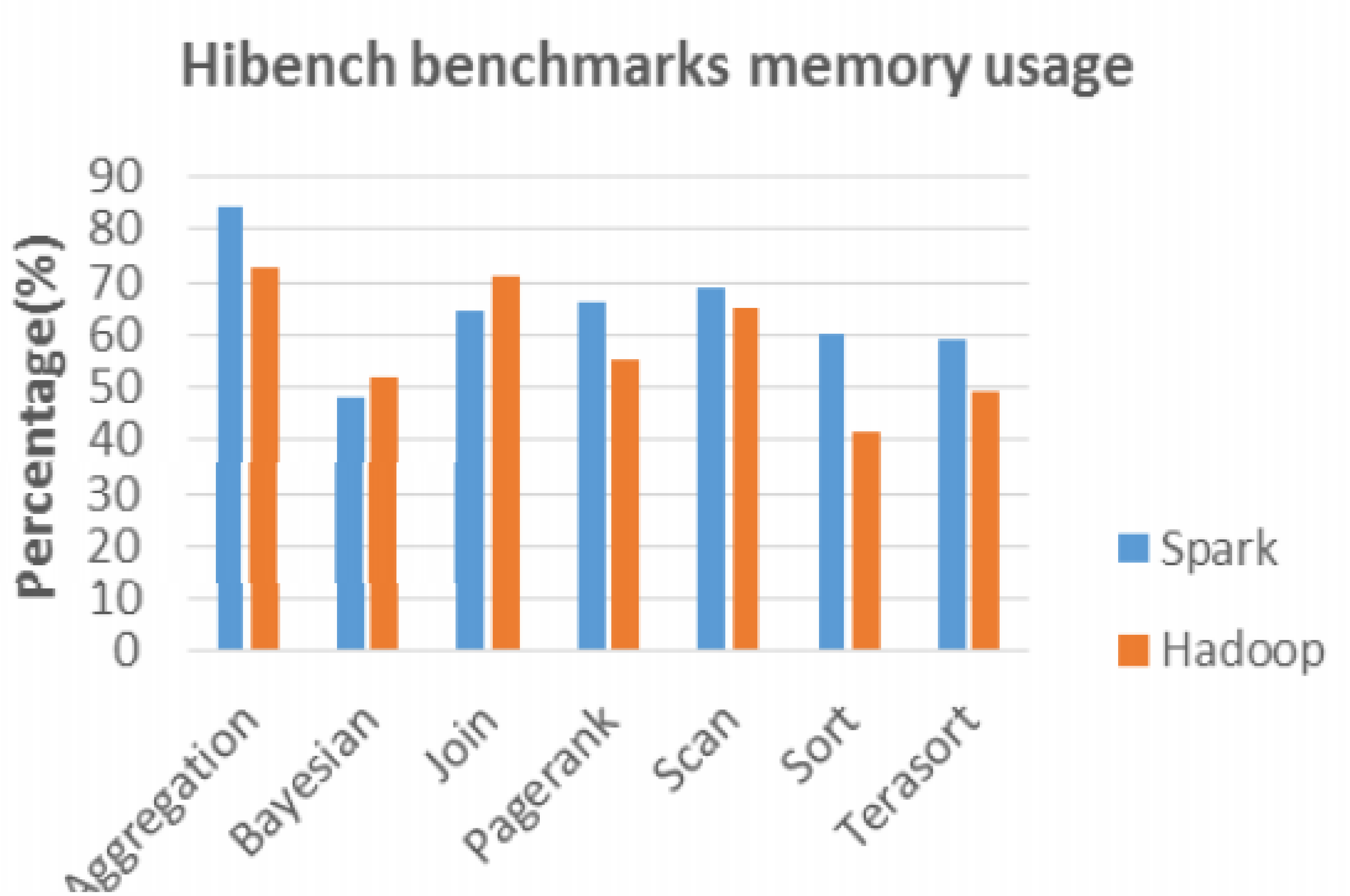
	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated 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**



Ref: https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Other Benchmarks


Spark & Hadoop cluster Benchmarks Execution time


Hibench benchmarks memory usage

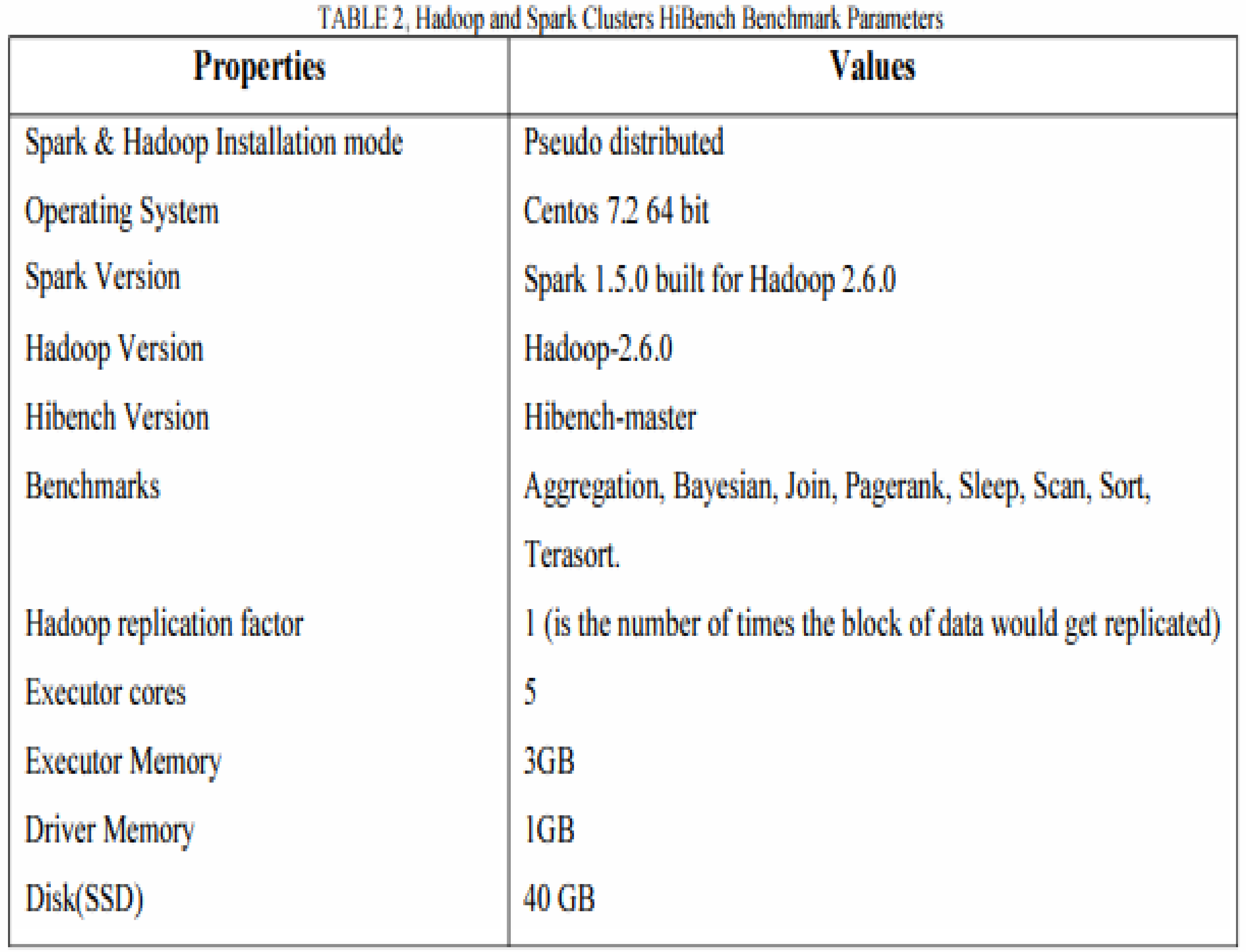
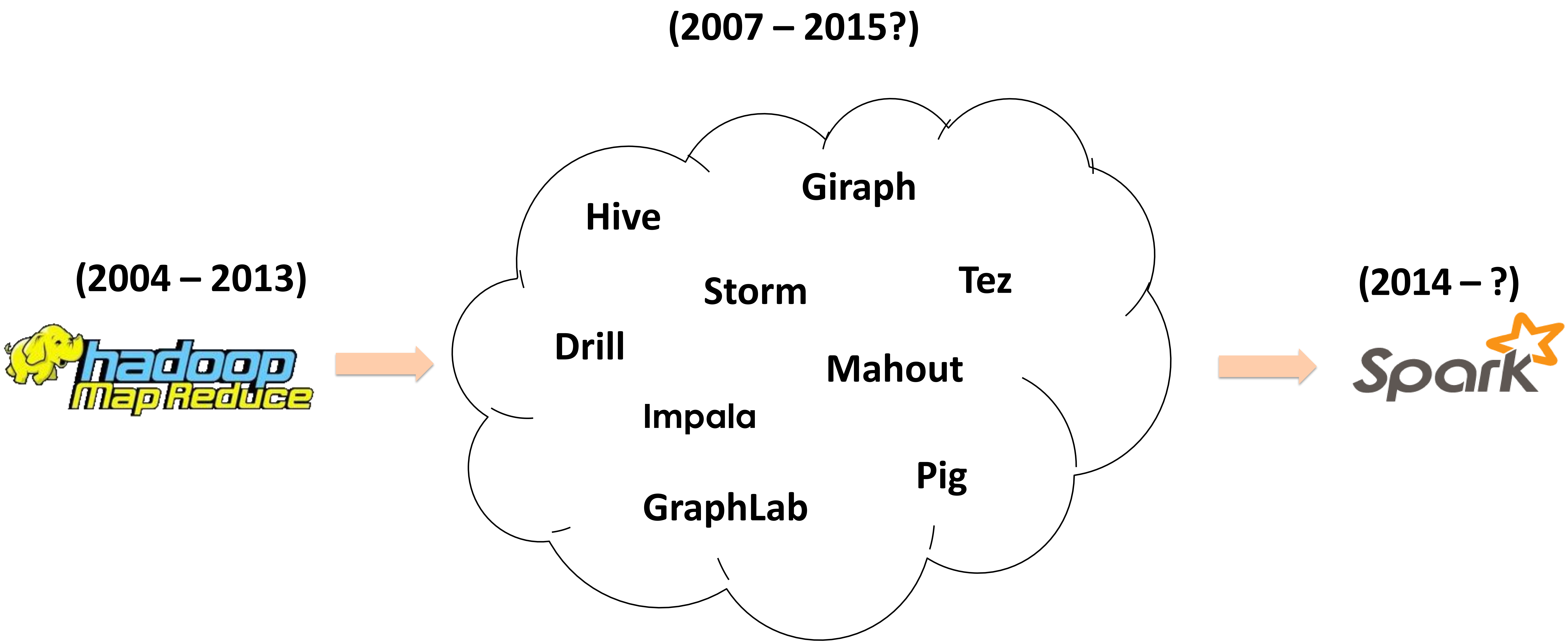


TABLE 2, Hadoop and Spark Clusters HiBench Benchmark Parameters

Properties	Values
Spark & Hadoop Installation mode	Pseudo distributed
Operating System	Centos 7.2 64 bit
Spark Version	Spark 1.5.0 built for Hadoop 2.6.0
Hadoop Version	Hadoop-2.6.0
Hibench Version	Hibench-master
Benchmarks	Aggregation, Bayesian, Join, Pagerank, Sleep, Scan, Sort, Terasort.
Hadoop replication factor	1 (is the number of times the block of data would get replicated)
Executor cores	5
Executor Memory	3GB
Driver Memory	1GB
Disk(SSD)	40 GB

Ref:
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7847709

Evolution



(2004 – 2013)

(2007 – 2015?)

Hive

Giraph

Storm

Tez

Drill

Mahout

Impala

Pig

GraphLab

(2014 – ?)

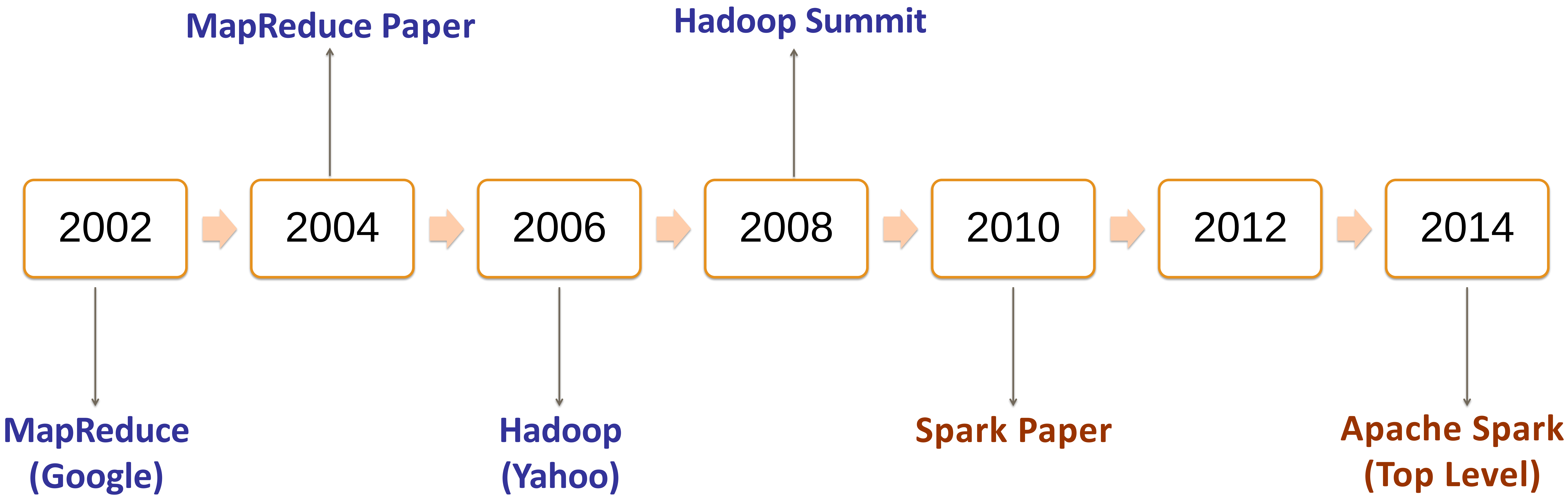
General Batch Processing

Specialized Systems

(Iterative, Interactive, ML, Streaming, Graph, SQL, etc)

General Unified Engine

Evolution…



2002 → 2004 → 2006 → 2008 → 2010 → 2012 → 2014

2002: MapReduce (Google)

2004: MapReduce Paper

2006: Hadoop (Yahoo)

2008: Hadoop Summit

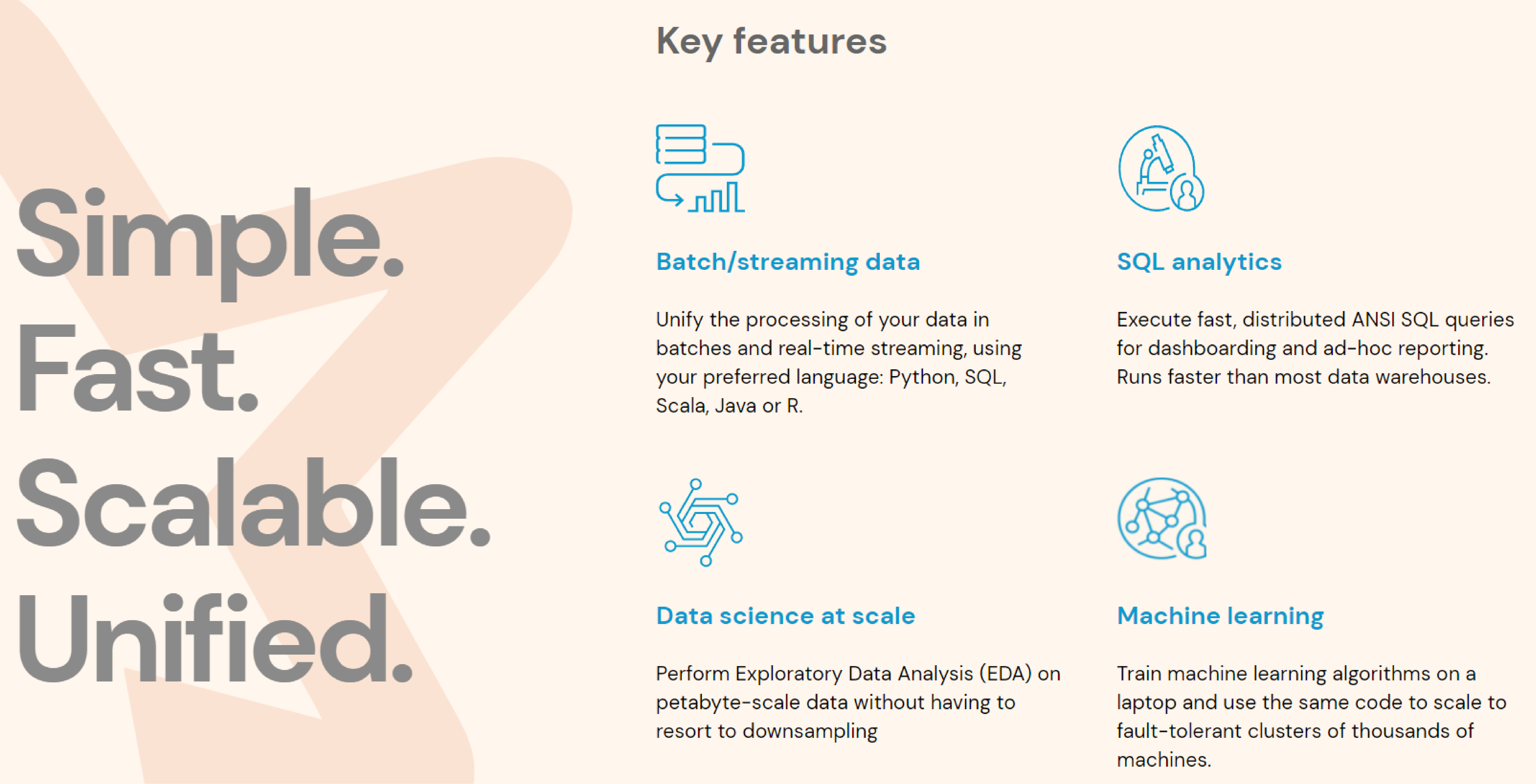
2010: Spark Paper

2014: Apache Spark (Top Level)

Spark

- Developed in 2009 at amplab UC BERKELEY
- Open sourced in 2010
- Spark is one of the largest open source communities in Big Data

spark.apache.org



Simple.
Fast.
Scalable.
Unified.

Key features

Batch/streaming data

Unify the processing of your data in batches and real-time streaming, using your preferred language: Python, SQL, Scala, Java or R.

SQL analytics

Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. Runs faster than most data warehouses.

Data science at scale

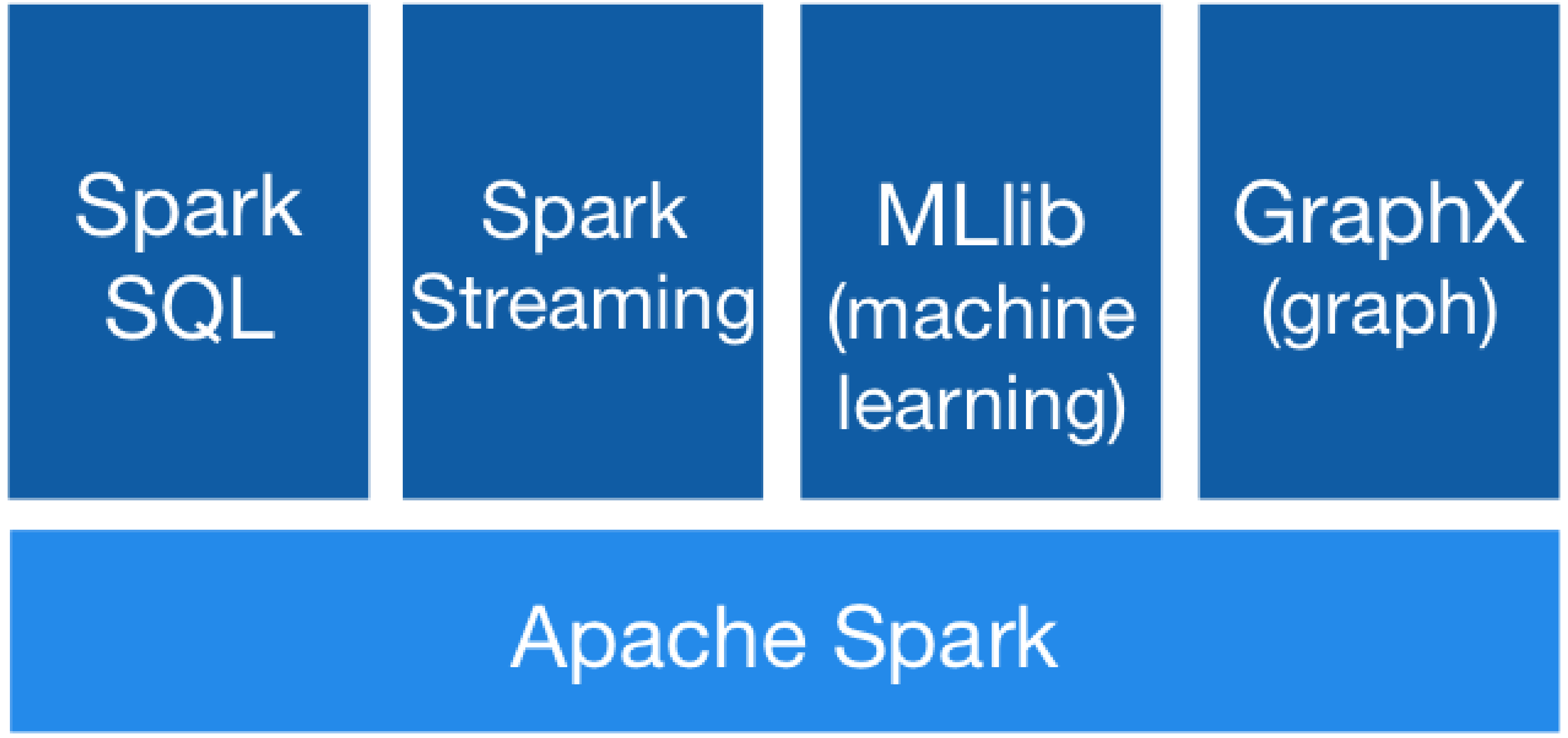
Perform Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling

Machine learning

Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines.

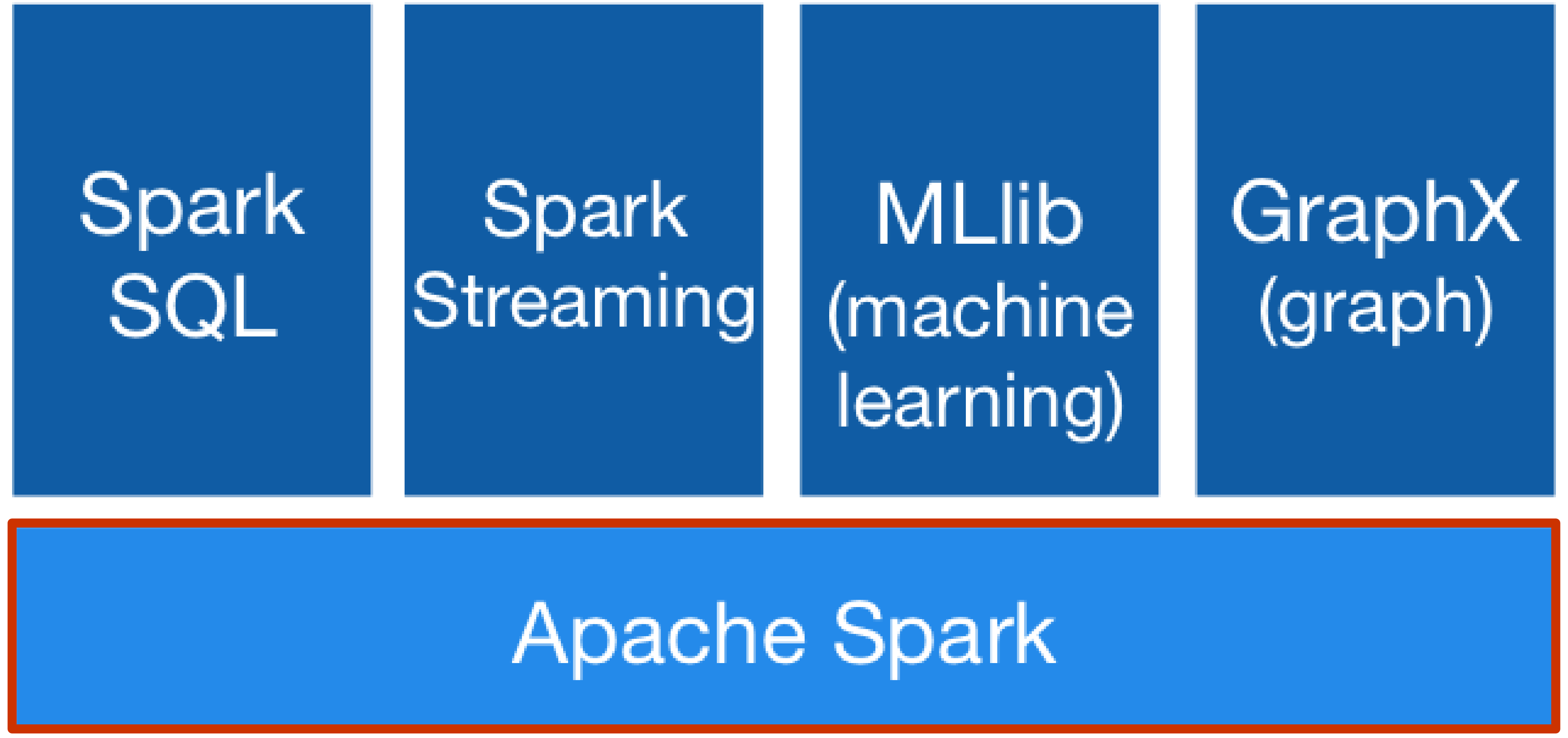
Spark Components

- **Spark Core**
  - Provides distributed Task Dispatching, Scheduling, and basic I/O functionalities.
- **Spark SQL**
  - Lets you query structured data as a distributed dataset (RDD) in Spark
- **Spark Streaming**
  - Allows scalable stream processing on micro-batches of data
- **MLlib**
  - Scalable Machine Learning library
- **Graphx**
  - Allows custom Iterative Graph Algorithms using Pregel API



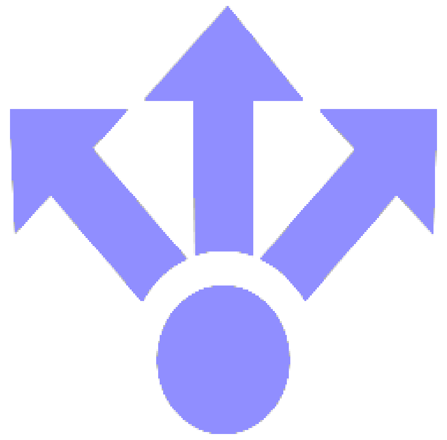
Spark Core

Provides distributed Task Dispatching, Scheduling, and basic I/O functionalities.



Scheduling

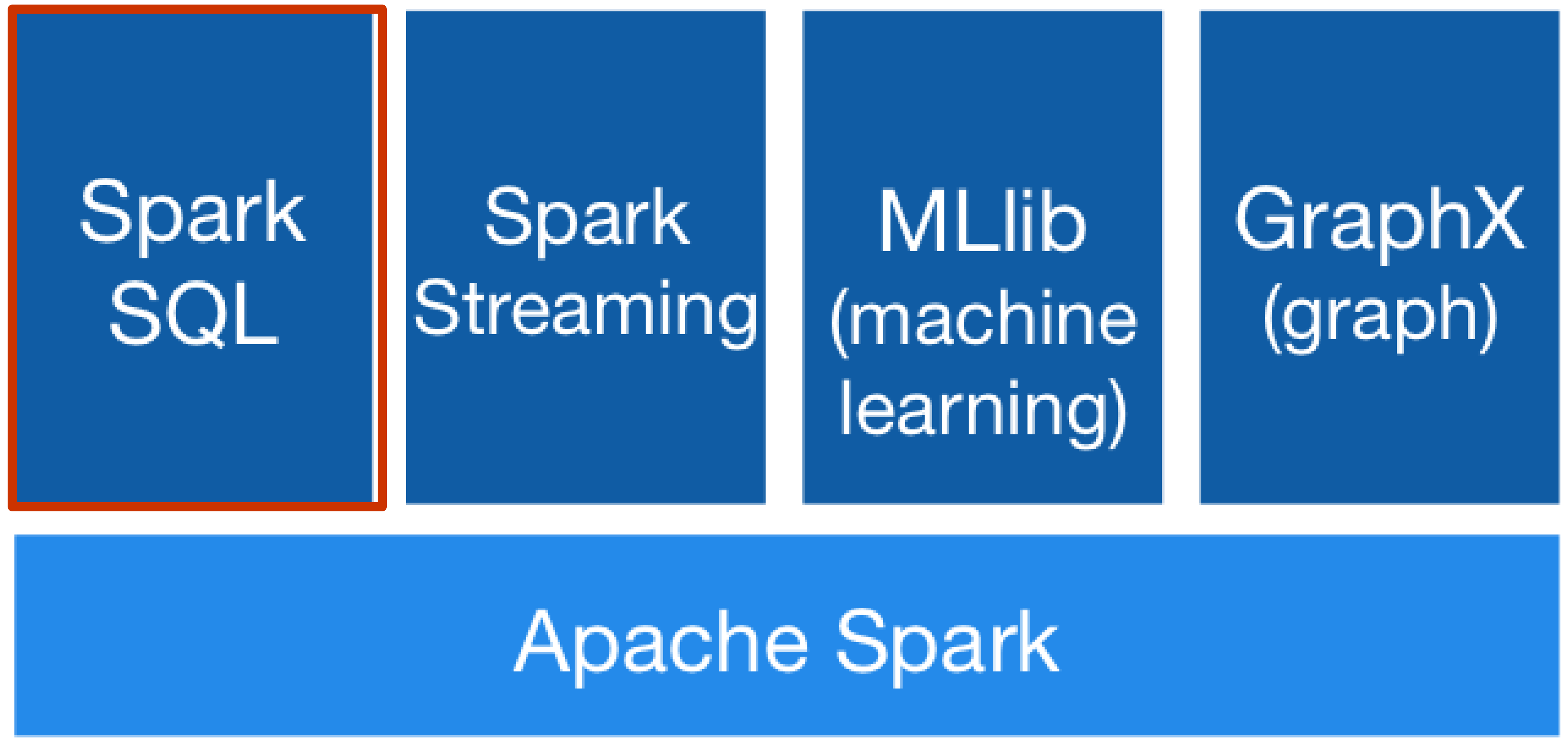
Monitoring



Distribution

SparkSQL

- Lets you query structured data inside Spark programs, using SQL/ DataFrame API.
- Usable in Java, Scala, Python and R.
- Runs SQL/ HiveQL queries

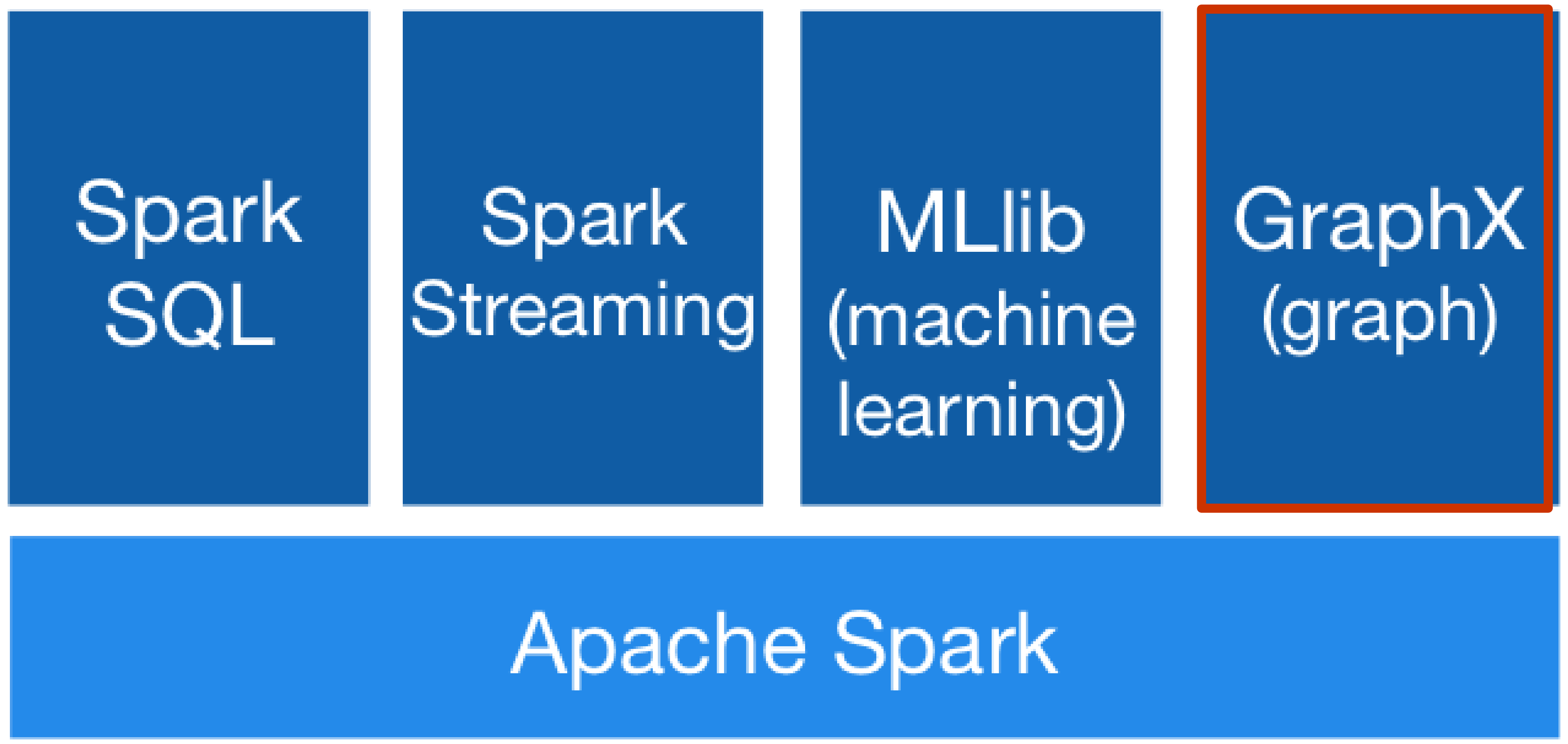


Features

- **Integrated:** Mix Spark programs with RDDs (Python, Scala and Java) and Spark SQL queries.
- **Unified Data Access:** Query from different data sources such as Hive tables, JSON.
- **Hive Compatibility:** Runs modified hive queries on existing warehouses.
- **DB Connectivity:** JDBC and ODBC
- **Performance and Scalability:** Scales to thousands of nodes

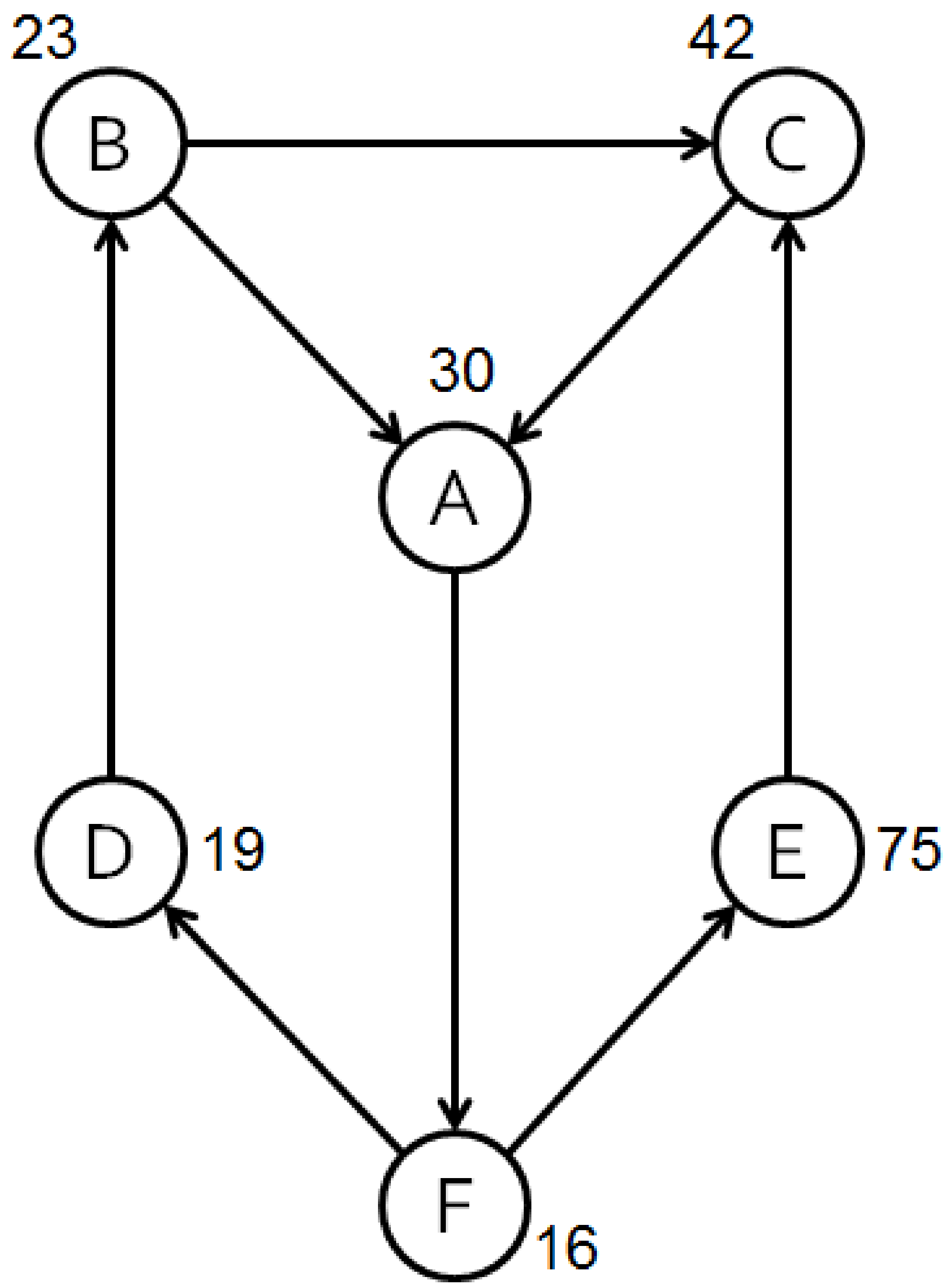
GraphX

- Spark API for graph-parallel computation.
- Introducing the **Resilient Distributed Property Graph:** a directed multi-graph with properties attached to each vertex and edge.
- GraphX includes a **collection of graph algorithms** and builders to simplify graph analytics tasks.



Example: Find the oldest follower of each user in a network

```
val oldestFollowerAge = graph.mrTriplets(
    e => (e.dst.id, e.src.age),//Map
    (a,b)=> max(a, b) //Reduce
  )
.vertices
```



Spark's Languages

Scala

Python

Java

R

Spark Packages

https://spark-packages.org
Paused
R
SparkPackages
Feedback
Register a package
Login
Find a package
All (451)
Core (14)
Data Sources (52)
Machine Learning (93)
Streaming (57)
Graph (20)
PySpark (18)
Applications (16)
Deployment (12)
Examples (26)
Tools (33)

A community index of third-party packages for Apache Spark.

Showing packages 1 - 50 out of 451

Next >

spark-als

Another, hopefully better, implementation of ALS on Spark (already merged into MLlib)

@mengxr / Latest release: 0.1.0 (2014-11-27) / BSD 3-Clause / ★★★★☆ (1)

3 ml 2 mllib 2 recommendation

mllib-grid-search

Spark Packages is a community site hosting modules that are not part of Apache Spark. Your use of and access to this site is subject to the terms of use.

Hadoop or Spark?

Hadoop MapReduce

2004 - First MapReduce Paper

Simplified Data processing on large clusters

(Jeffrey Dean & Sanjay Ghemawat, OSDI, 2004, Google, Inc.)

Map-Reduce is a high-level programming model and implementation for "large-scale parallel data processing".

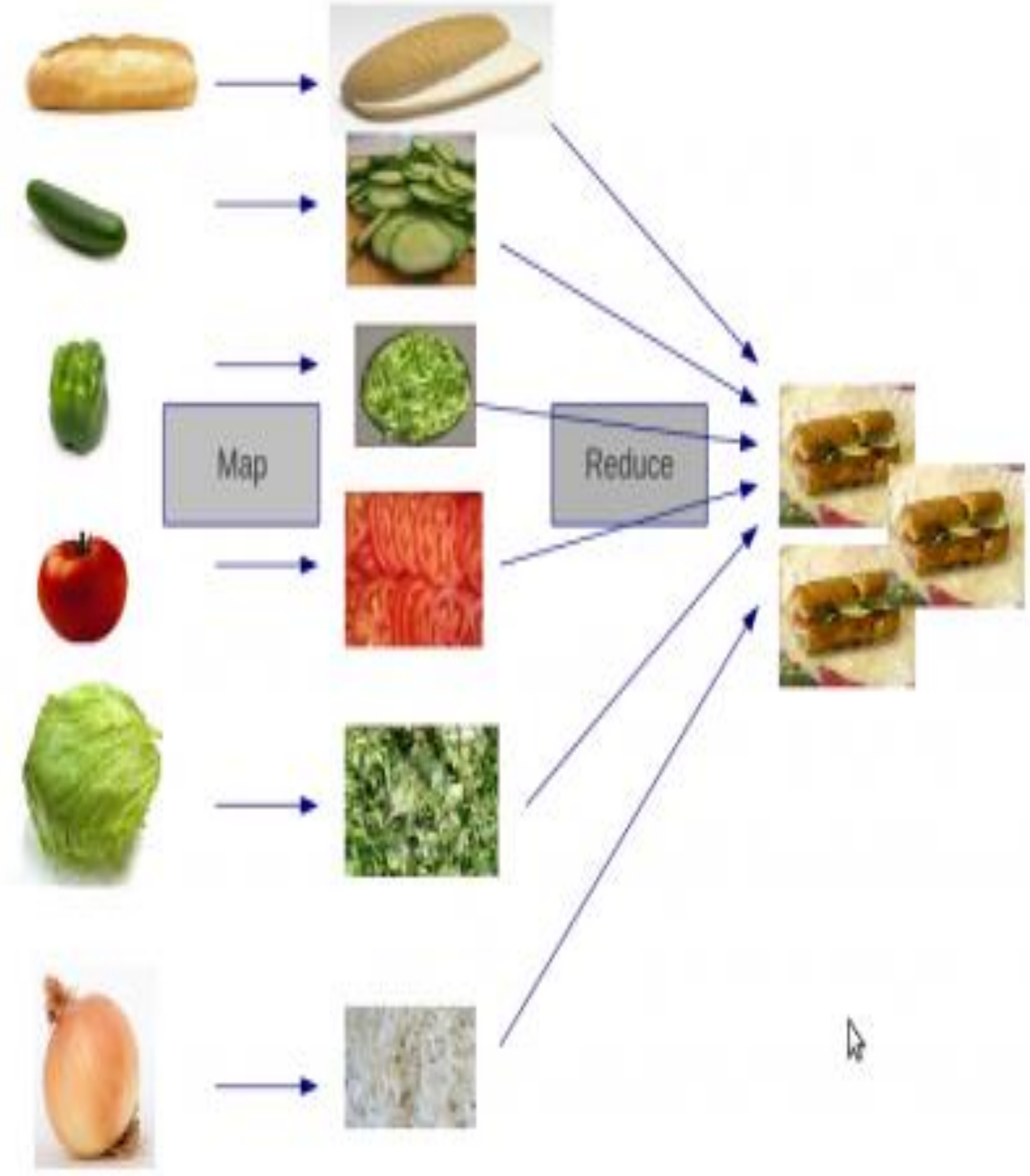


Image Source: http://tm.durusau.net/?cat=84

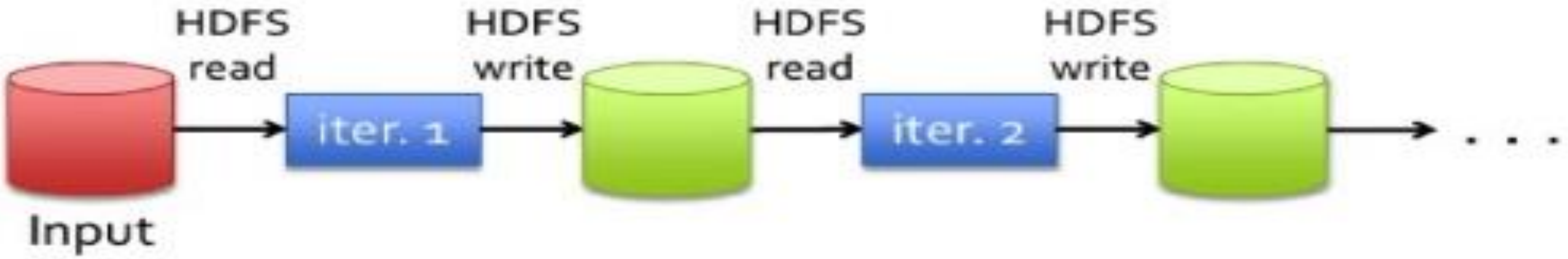
Spark vs. Hadoop

Spark is a better candidate for iterative jobs

Each stage passes through the hard drives

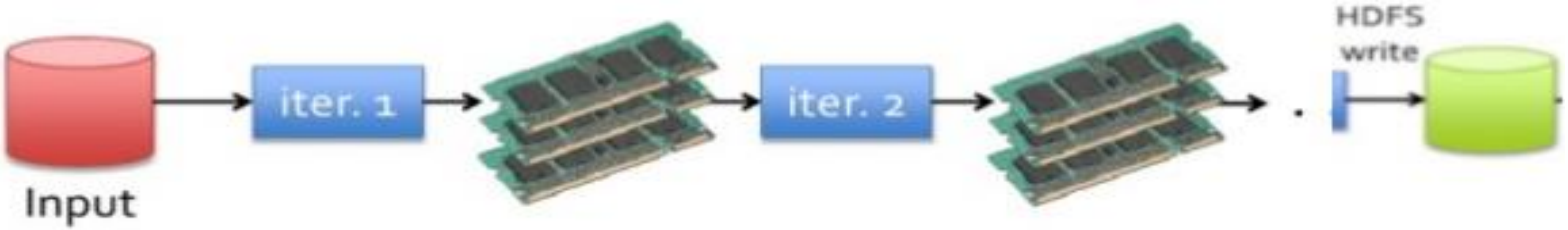
(MapReduce iterative jobs involve a lot of disk I/O for each repetition…Very slow)

Hadoop MapReduce



Use Memory instead of Disk…10 to 100x faster

Apache Spark



Spark vs. Hadoop MR…

MapReduce Limitations

- Inefficient if you are repeating similar searches again and again.
- Reduce operations do not take place until all the Maps are complete.
- Designing and Programming using MR is difficult.
- Performance bottlenecks due to heavy disk I/O
- Unable to accommodate the Batch size of real time use cases.

“MR doesn’t compose well for large applications”

Therefore, building specialized systems as workarounds…..

Spark vs. Hadoop MR…

Spark's Advantages

- Generalized patterns
  - Unified engine for many use cases
- Lazy evaluation of the lineage graph
  - Reduces wait states, better pipelining
- In-Memory processing
  - Uses large memory spaces
- Lower overheads
  - Starts a job quickly
- Less expensive shuffles

Is Spark replacing Hadoop?

Enemies or Frenemies ??

Can work together even though there are fundamental differences.

Image Credits: www.cartoonbucket.com

Migrating from Hadoop to Spark - Tools

Application Type	Hadoop or Other	Spark
Distributed Storage	HDFS	No DFS (has to use 3rd party file system)
Batch Processing	Hadoop MR	Spark RDDs & DAGs
Querying/ ETL	Hive, PIG, HBase	Spark SQL (Supports NoSQL data stores), Spork (Pig on Spark)
Stream Processing or Real time processing	Hadoop Streaming, Storm	Spark Streaming
Machine Learning	Mahout (Almost migrated to Spark)	MLlib or Mahout
Interactive Exploration	-NA-	Interactive shells for Scala & Python

Hadoop vs. Spark

Infrastructure Type

- Hadoop is essentially a distributed data infrastructure: It distributes massive data collections across multiple nodes within a cluster of commodity server,
- Where as Spark is data-processing framework.

Performance

- Spark performs better when all the data fits in the memory, especially on dedicated clusters;
- Hadoop MapReduce is designed for data that doesn't fit in the memory and it can run well alongside other services.

Compatibility to Data Sources/ Types

- Spark's compatibility to data types and data sources is the same as Hadoop MapReduce.

Hadoop vs. Spark

Batch processing with huge data

- Hadoop MapReduce is great for batch processing if the data size is huge where as spark needs more resources to compete with Hadoop.

Fault Tolerance

- Both have good failure tolerance. Hadoop MapReduce is slightly more tolerant.
- But, Hadoop fault tolerance comes at the cost of computational performance.
- In Spark, we have to re-compute using Lineage information.

Security

- Spark security is still in its infancy;
- Hadoop MapReduce has more security features.

Do I need to learn Hadoop first to learn Apache Spark?

No, you don't need to learn Hadoop to learn Spark.

Spark is an independent framework.

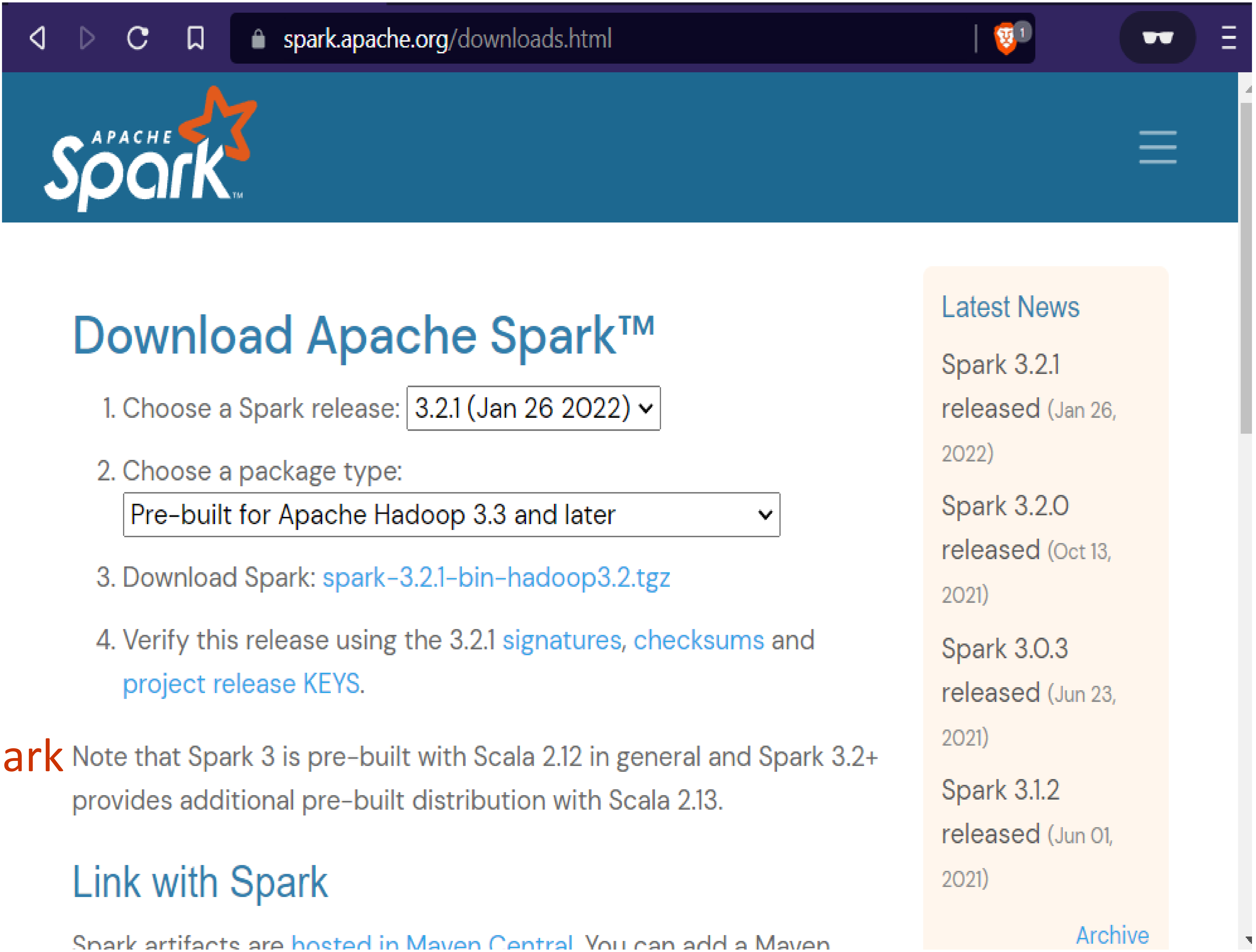
If you want to use HDFS or YARN with Spark, better start with Hadoop.

Installing Apache Spark

Installing Spark

- ❑ Using pre-built binaries (Along with Hadoop)
- ❑ Using Source
- ❑ Using pypi
- ❑ PySpark is also available in pypi.

  To install just run: $ pip install pyspark



Download Apache Spark™

1. Choose a Spark release: 3.2.1 (Jan 26 2022)
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later
3. Download Spark: spark-3.2.1-bin-hadoop3.2.tgz
4. Verify this release using the 3.2.1 signatures, checksums and project release KEYS.

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.

Link with Spark

Spark artifacts are hosted in Maven Central. You can add a Maven

Latest News

Spark 3.2.1 released (Jan 26, 2022)

Spark 3.2.0 released (Oct 13, 2021)

Spark 3.0.3 released (Jun 23, 2021)

Spark 3.1.2 released (Jun 01, 2021)

Archive

Interactive Shells

Python

```
$ pyspark --master local[4]
$ pyspark --master local[*] (Use the threads equal to #cores)
$ pyspark --master local[4]  code.py
$ pyspark --master spark://IPAddressOfMaster:7077 code.py
```

Launching Applications

```
# Run application locally on 8 cores
./bin/spark-submit --master local[8] Python_code.py 100
# Arguments to python program
```

Launching Applications…

# Run on a Spark standalone cluster in client deploy mode

```
./bin/spark-submit \
--master spark://207.184.161.138:7077 \
--executor-memory 20G \
--total-executor-cores 100 \
/path/to/examples.jar \
1000
```

--executor-memory → Amount of memory to use per executor process

--total-executor-cores → The number of cores to use on each executor.

Launching Applications…

Other Examples

```
# Run application locally on 8 cores
$ spark-submit --class org.apache.spark.examples.SparkPi \
  --master local[8] \
   /path/to/examples.jar  100
```

```
# Run a Python application on a Spark Standalone cluster
$ spark-submit --master spark://207.184.161.138:7077 \
examples/src/main/python/pi.py   100
```

Spark Execution Model

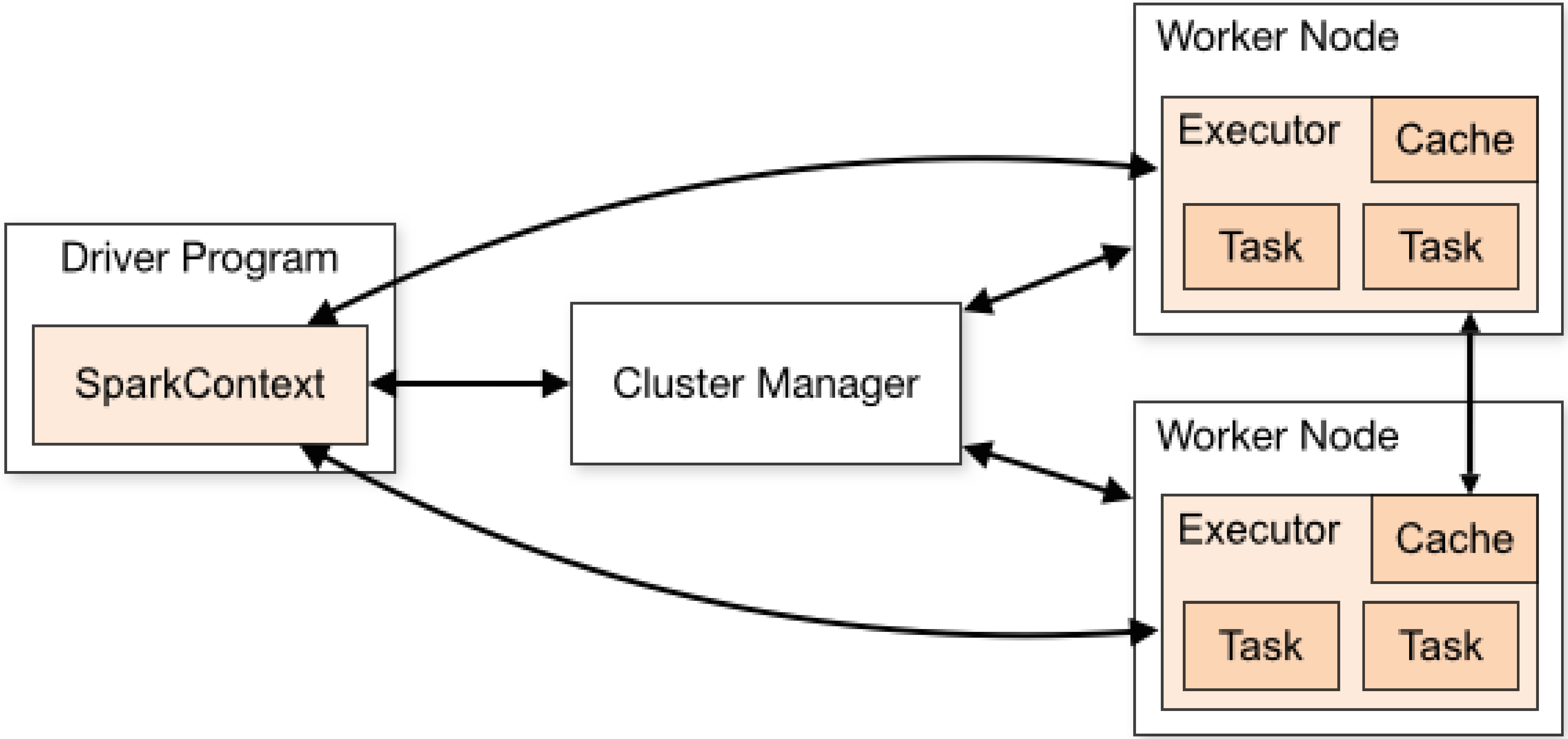
Spark Driver & Workers… (with YARN)

Credits: http://0x0fff.com/spark-architecture/

Client Node
Driver JVM
Spark Context
Controls spark.driver.memory RAM
Master Node
Worker Node 1
YARN Node Manager
HDFS Datanode
Node Memory Pool
(yarn.nodemanager.resource.memory-mb)
Executor JVM #1
Task #1
Requires spark.task.cpus
Task #2
Requires spark.task.cpus
…
Controls spark.executor.cores CPU cores and spark.executor.memory RAM
Executor JVM #2
Task #1
Requires spark.task.cpus
Task #2
Requires spark.task.cpus
…
Controls spark.executor.cores CPU cores and spark.executor.memory RAM
…
…
Worker Node N
YARN Node Manager
HDFS Datanode
Node Memory Pool
(yarn.nodemanager.resource.memory-mb)
Executor JVM #M-1
Task #1
Requires spark.task.cpus
Task #2
Requires spark.task.cpus
…
Controls spark.executor.cores CPU cores and spark.executor.memory RAM
Executor JVM #M
Task #1
Requires spark.task.cpus
Task #2
Requires spark.task.cpus
…
Controls spark.executor.cores CPU cores and spark.executor.memory RAM
…

Spark Program Execution

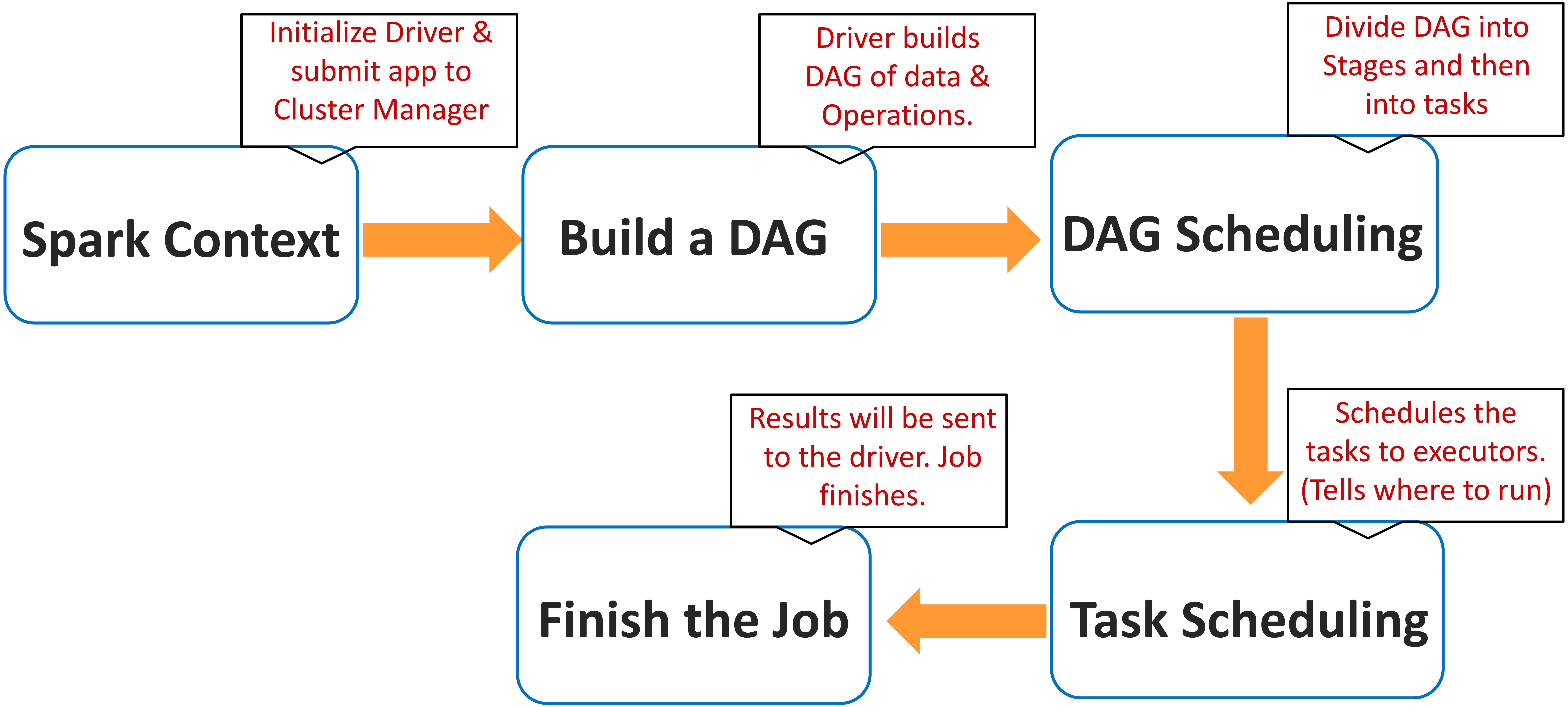
Spark program has 2 components: Driver Component & Worker Component



Driver Component runs on: Master Node (or) Local Threads

Worker program runs on: Worker Nodes (or) Local Threads

Spark Program Execution



Spark Context

Initialize Driver & submit app to Cluster Manager

Build a DAG

Driver builds DAG of data & Operations.

DAG Scheduling

Divide DAG into Stages and then into tasks

Task Scheduling

Schedules the tasks to executors. (Tells where to run)

Finish the Job

Results will be sent to the driver. Job finishes.

Initializing Spark - Spark Context

- A Spark program first creates a **SparkContext** object
- Tells Spark how and where to access a cluster
- PySpark shell automatically create the “sc” variable

```
val conf = new
SparkConf().setAppName(appName).setMaster(master)
new SparkContext(conf)
```

Scala

```
SparkConf conf = new
SparkConf().setAppName(app).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

Java

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

Python

Spark Jargons

Application	**User program** built on Spark. Consists of a driver program and executors on the cluster.
Application Jar	A **jar containing the user's Spark application**. In some cases users will want to create an **"uber jar"** containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver Program	The **process running the main part** of the application and creating the SparkContext.
Cluster Manager	**Acquiring & scheduling resources** on the cluster (e.g. standalone manager, Mesos, YARN)
Worker Node	Any node that can run application code in the cluster, **Which have the executors**
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.
Job	A parallel computation **consisting of multiple tasks** that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each **job gets divided into smaller sets of tasks called stages** that depend on each other (similar to the map and reduce stages in MapReduce); You can see this in driver log.
Task	A **unit of work** that will be **sent to one executor**
DAG	DAG stands for **Directed Acyclic Graph,** in the present context its **a DAG of operators.**

Resilient Distributed Datasets (RDDs)

What is RDD?

Resilient Distributed Dataset (RDD) is the primary data abstraction in Apache Spark and the core of Spark.

- Spark Core Concept → **R**esilient **D**istributed **D**ata
- The primary abstraction in Spark
- It is a **Partitioned fault-tolerant collection of data elements.**
- Can be **operated on in parallel**.

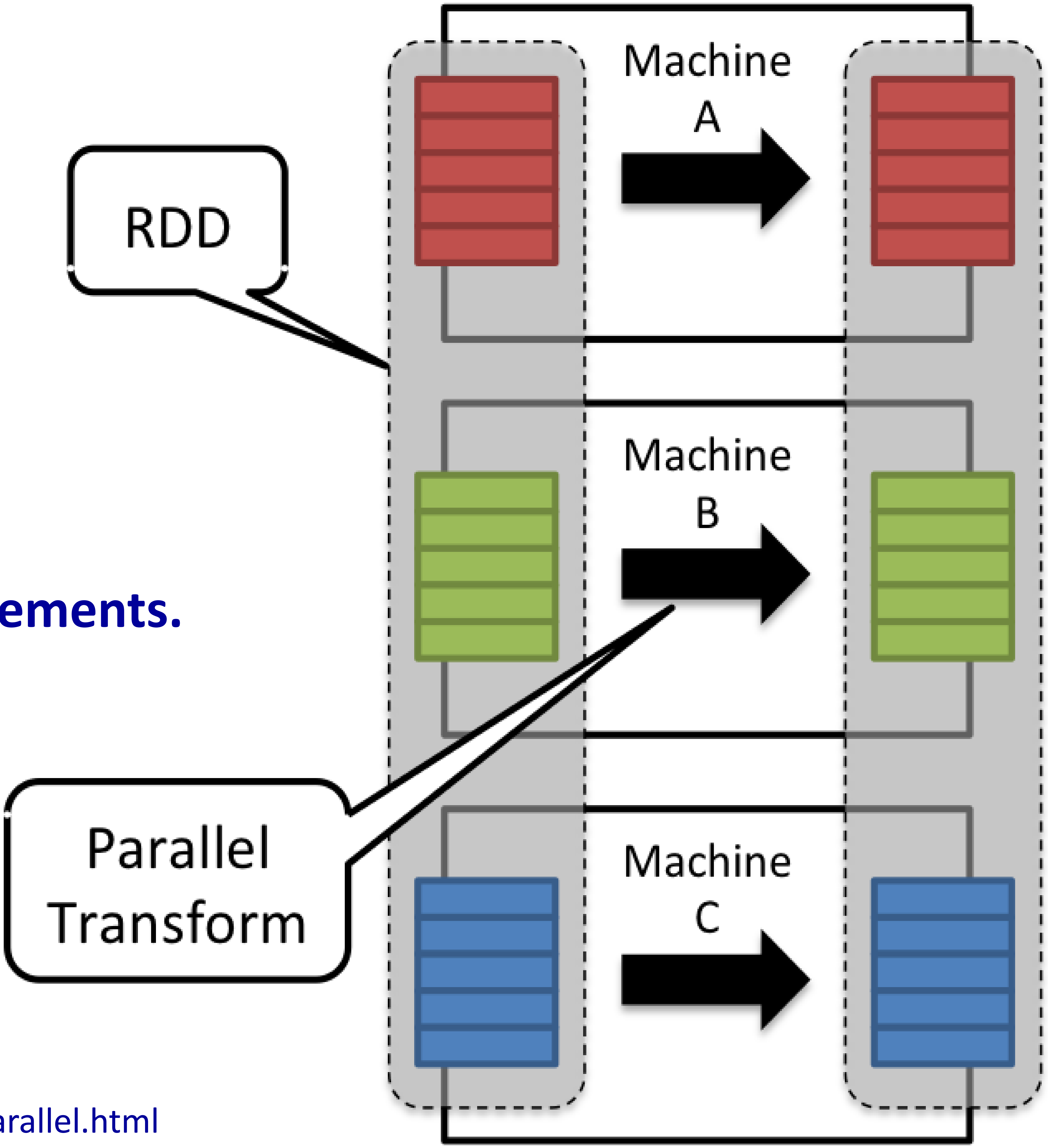


Image Credits: http://horicky.blogspot.in/2013/12/spark-low-latency-massively-parallel.html

What is RDD?

Properties

- **Resilient -** i.e. fault-tolerant with the help of RDD lineage graph and so we are able to re-compute missing or damaged partitions due to node failures.
- **Distributed -** with data residing on multiple nodes in a cluster, we can enable operations on collection of elements in parallel.
- **Dataset -** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects.

Note: RDD are Immutable - **Cannot be changed** once it is created

How to construct RDDs?

RDDs can be constructed

1. By **parallelizing** existing Python collections **(Parallelized collections)**
2. By **transforming** an existing RDDs **(Pipelined RDD)**
3. From **external data** (dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop InputFormat.)

RDDs from Parallelized Collections

- Parallelized collections are created by calling SparkContext’s **“parallelize” method** on an existing iterable or collection in your driver program.

- The elements in the collection will be **converted as a distributed dataset** that can be operated on in **parallel.**

```
data = [1, 2, 3, 4, 5, 6, 7]

RDD1 = sc.parallelize(data, 4)
```

- Important parameter for parallel collections is the number of partitions to cut the dataset into.

- Spark will run **one task for each partition of the cluster**.

- **More Partitions → More parallelism**

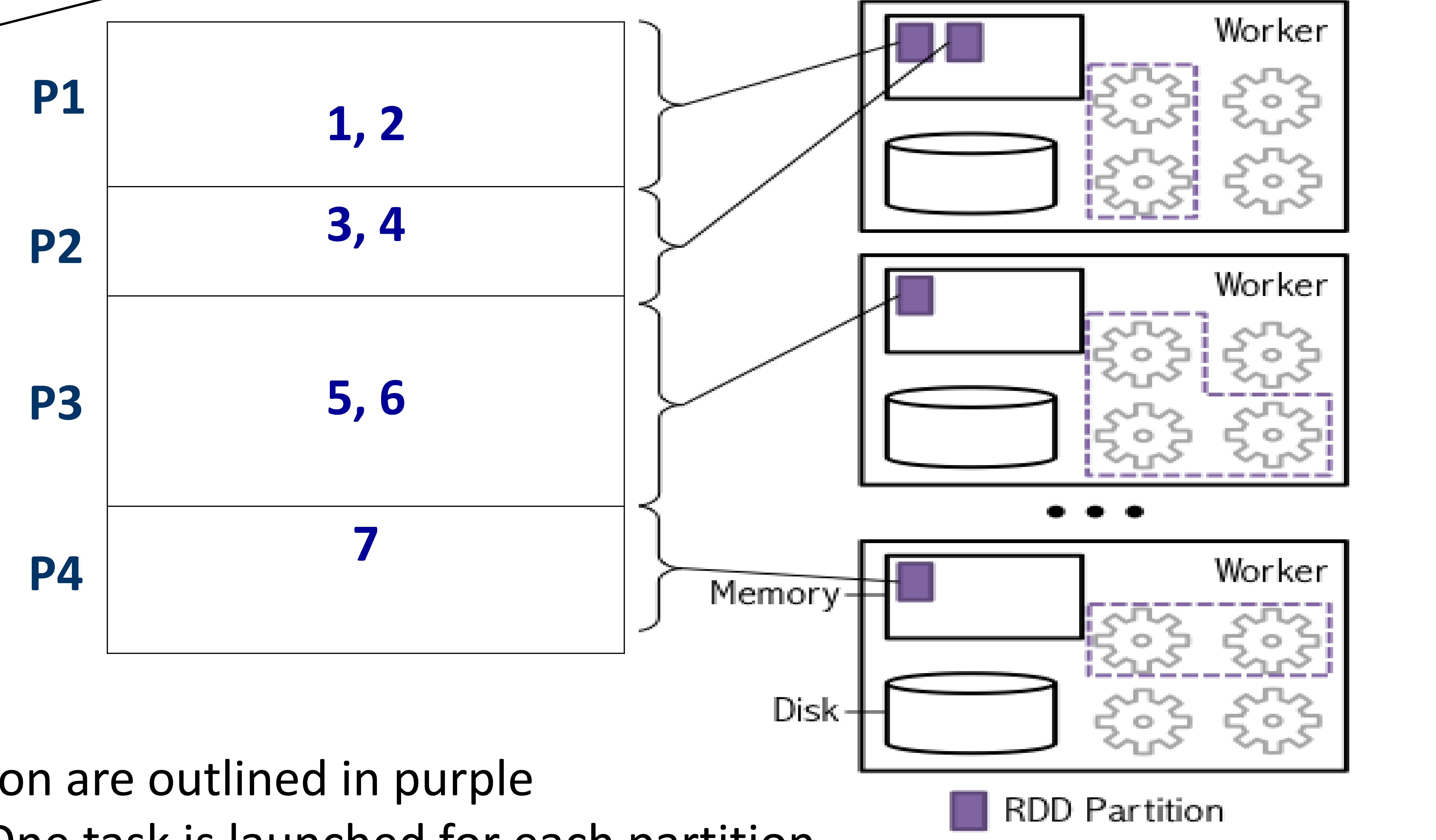
- If you want to make any change to an existing RDD, we have to create a New RDD.

How data is partitioned?

```
data = [1, 2, 3, 4, 5, 6, 7]
RDD1 = sc.parallelize(data, 4)
```

Number of Partitions

Partitions are each stored in a worker's memory

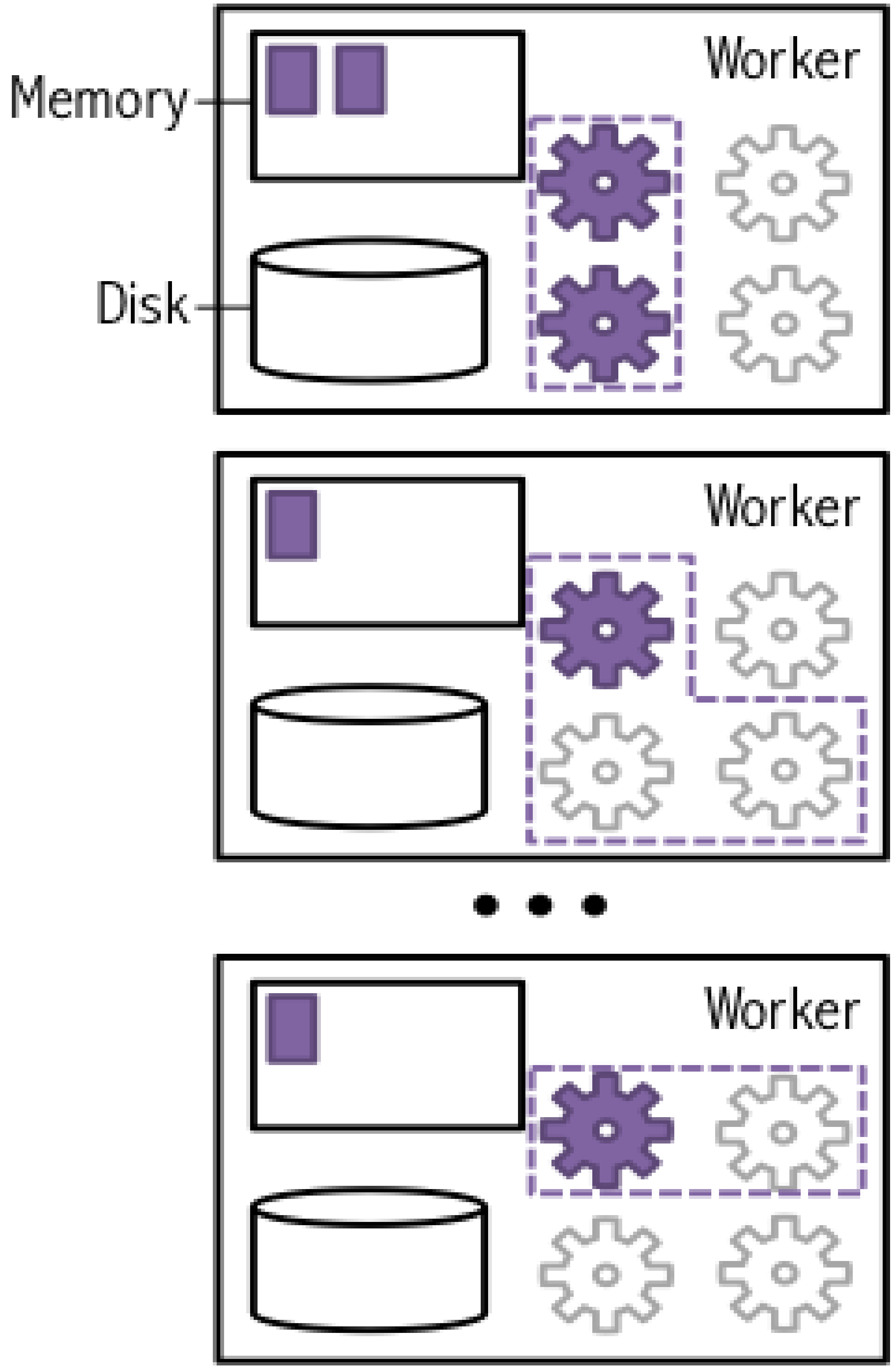


Cores allocated for an application are outlined in purple

7 cores are allocated (2+3+2). One task is launched for each partition.

How data set is partitioned?

```
data = [1, 2, 3, 4, 5, 6, 7]
RDD1 = sc.parallelize(data, 4)
```



One task is launched for each partition

⚙ Running task

⚙ Unused core

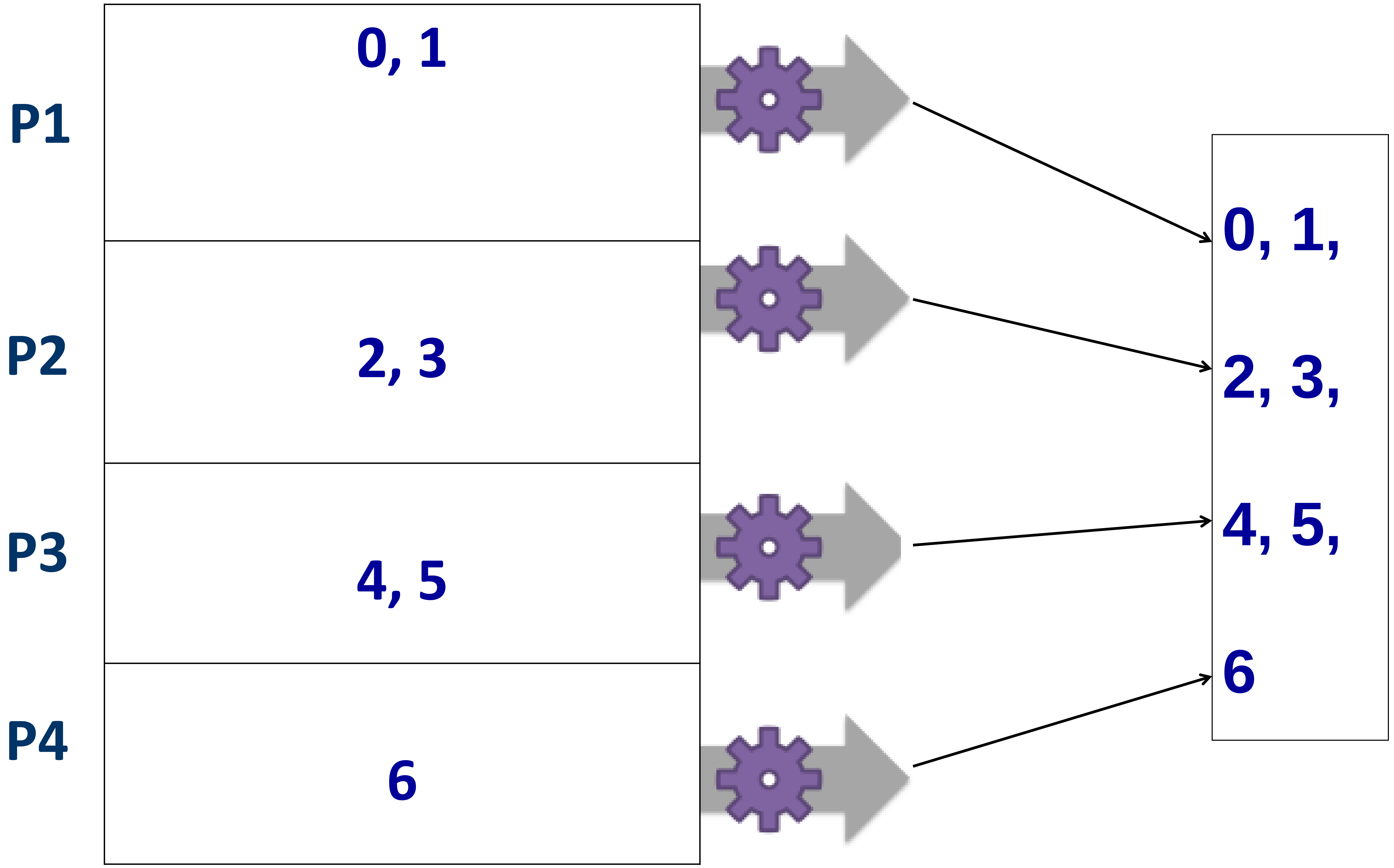
4 partitions → 4 Tasks → 4 Cores out of 7

How data set is transformed and collected?

```
# Definition of sub function
def sub(value):
  return (value – 1)

# Use map for subtraction
subRDD1 = RDD1.map(sub)

# Apply action collect on
  subRDD to collect numbers
print subRDD1.collect()
```



RDDs from External Datasets

PySpark can create distributed datasets from any storage source supported by Hadoop:

- Local File System
- HDFS
- Cassandra
- HBase
- Amazon S3, etc.
- Spark supports textFiles, SequenceFiles, and any other Hadoop InputFormat.

Text file RDDs can be created using SparkContext’s textFile method.

```
distFile = sc.textFile(“/home/bda/data.txt“, minPartitions = 4,
                     use_unicode=True)
```

External Datasets…

NOTE

- If using a path on the local file system, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system.
- All of Spark's file-based input methods, including textFile, support running on directories, compressed files and wildcards as well.
- Number of Partitions of a file:
  - Mention the number of partitions as 2nd argument of the method "textFile".
  - By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS),
  - But, you can also ask for a higher number of partitions by passing a larger value.
  - Note that you cannot have fewer partitions than blocks.

Reading and Saving RDDs

Spark Python API supports additional formats:

Reading Directories

- **SparkContext.wholeTextFiles** lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file.

Saving RDDs

- **RDD.saveAsTextFile(“PATH”)**
- **RDD.saveAsPickleFile** and SparkContext.pickleFile support saving an RDD in a simple format consisting of pickled Python objects. Batching is used on pickle serialization, with default batch size 10.

Others

- SequenceFile and Hadoop Input/Output Formats

PySpark’s writable support of RDDs

Spark Python API supports additional formats:

Writable Support:

- PySpark **SequenceFile** support loads an **RDD of key-value pairs within Java**
- Converts Writables to base Java types and **pickles** the resulting Java objects using **Pyrolite**.
- When saving an RDD of key-value pairs to SequenceFile, PySpark does the reverse.
- It **unpickles** Python objects into Java objects and then converts them to Writables.
- The following Writables are automatically converted:

Examples: Writable type “text” is converted into python type “unicode str”

Writable type “intwritable” is converted into python type “int”

RDD Life Cycle

1. **Create an input RDD** from parallelizing a collection OR from an external data source.
2. Perform the required **transformations** on the base RDD to create new RDDs.
3. Instruct Spark to **store/persist** them in disk/cache. Persist using cache if it can be reused later.
4. Perform one or more **actions** on them by parallel computations using Spark.

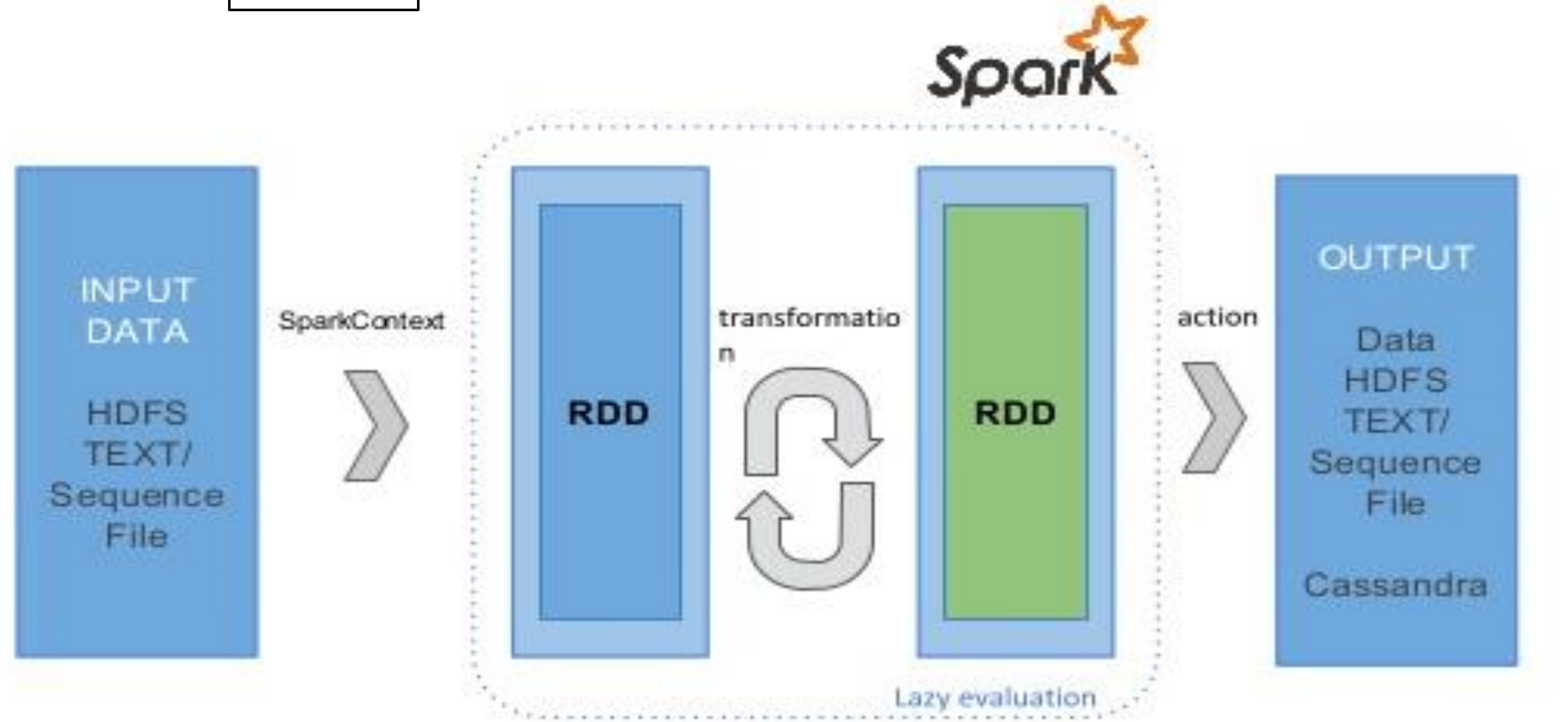


Image Credits: Spark devoxx2014, Andre Petrella

Transformations

Transformations

- Creates new datasets (RDDs) from the existing ones
- In other words, transformations are functions that take a RDD as the input and produce one or many RDDs as the output.
- Input RDD will not be changed (since RDDs are immutable)
- **Lazy Evaluation:**
  - They do not compute their results right away. Instead, they just remember the transformations applied **(Lineage)** to some base dataset (e.g. a file).
  - Spark optimizes the required calculations
  - Computes when there is an "action"
- Transformations can be treated as recipes for creating a result of an action.
- **Lineage** helps to Recovers from failures and slow workers.

Why Transformations?

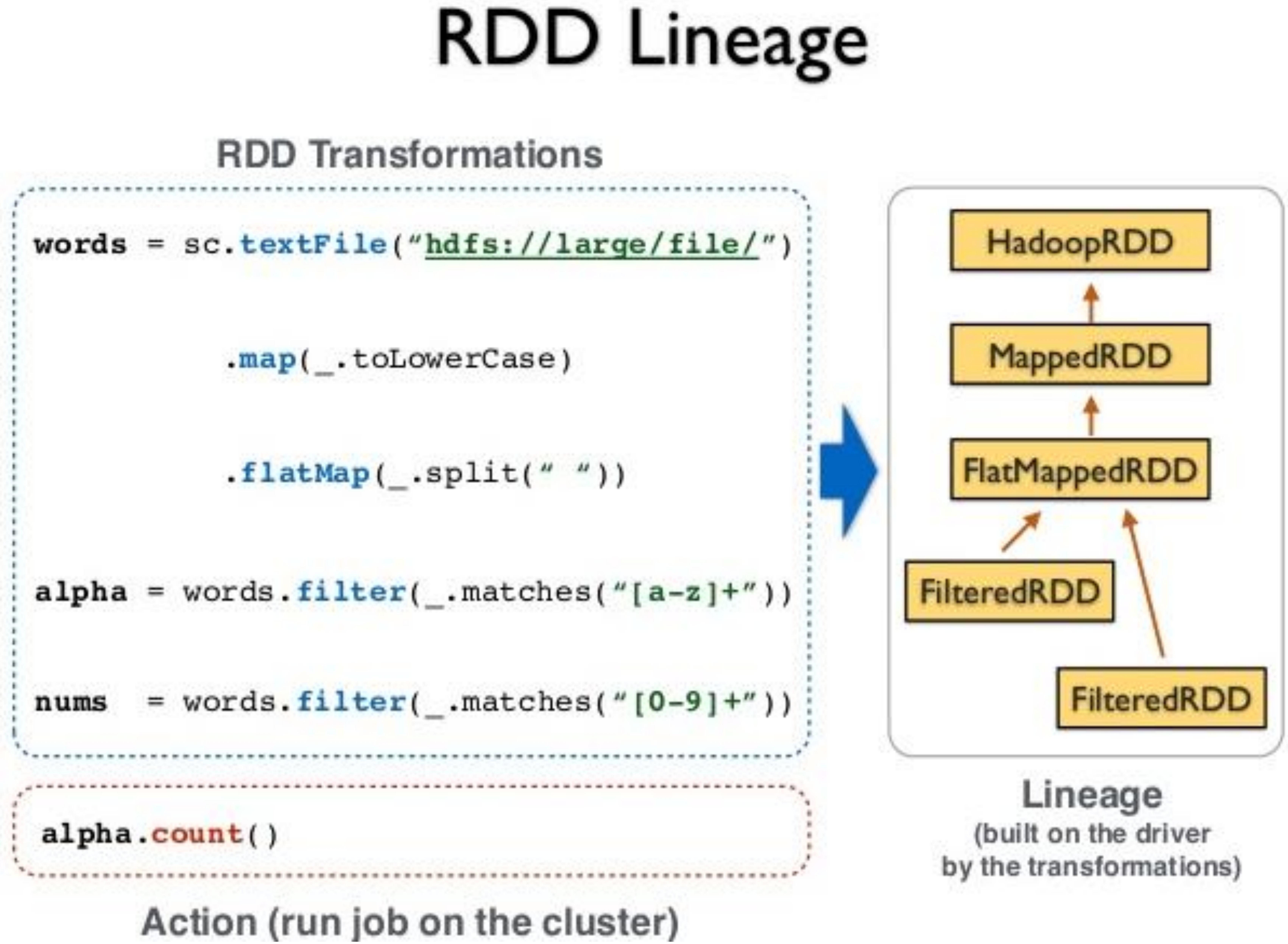
1. To understand the data (List out the number of columns in data and their type).
2. Preprocess the data (Remove null values, Remove outliers).
3. Filter the data.
4. Fill the null values or missing values in data (Filling the null values in data by constant, mean, median, etc)
5. Deriving new features from data.

And so on…

Transformations

Stores the lineage of a base RDD

By applying transformations, you incrementally build a **RDD lineage** with all the parent RDDs of the final RDD(s).



RDD Lineage

RDD Transformations

```
words = sc.textFile(“hdfs://large/file/”)

                 .map(_.toLowerCase)

                 .flatMap(_.split(“ “))

alpha = words.filter(_.matches(“[a-z]+”))

nums  = words.filter(_.matches(“[0-9]+”))
```

```
alpha.count()
```

Action (run job on the cluster)

Lineage
(built on the driver
by the transformations)

Transformations

Manipulations (On single RDD)	Across RDDs (On Multiple RDDs)	Reorganization (On single RDD) (Key-Value)	Tuning (On single RDD)
map	union	groupByKey	coalesce
flatMap	subtract	reduceByKey	repartition
filter	intersection	sortByKey	
distinct	join		
	Cartesian		

Examples – Transformations - map

```
# Create an RDD
data = [1, 2, 3, 4, 5, 6, 7]
RDD1 = sc.parallelize(data, 4)
```

```
# Create sub function to subtract 1# Args: One integer value,
Returns: One def sub(value):
          return (value - 1)
```

```
# Transform xrangeRDD thro’ map transformation using sub function
subRDD = RDD1.map(sub)
```

```
# Print the lineage
print(subRDD.toDebugString())
```

More Examples – Transformations – Filter & Distinct

```
# Create an RDD
data = [1, 2, 2, 3, 4, 4, 5, 6, 7]
RDD1 = sc.parallelize(data, 4)
```

```
# Create an RDD of only even numbers (Filter odd numbers)
RDD2 = RDD1.filter( lambda x: x%2 == 0 )
```

```
# Get the distinct even numbers
 RDD3 = RDD2.distinct()
```

More Examples – Transformations – Map & flatMap

```
# Create an RDD
data = [1, 2, 3]
RDD1 = sc.parallelize(data, 3)
```

```
# Create an RDD of x and x+5 pair using map
RDD2 = RDD1.map( lambda x: [ x, x+5 ]
Output: [1, 2, 3] → [ [1,6], [2,7], [3,8] ]
```

```
# Create an RDD of x and x+5 pair using flatMap
 RDD3 = RDD2.flatMap()
Output: [1, 2, 3] → [ 1, 6, 2, 7, 3, 8 ]
```

Pair RDDs

- A pair RDD is an RDD where **each element is a pair tuple** (key, value).
- For example, sc.parallelize([('a', 1), ('a', 2), ('b', 1)]) would create a pair RDD where the keys are 'a', 'a', 'b' and the values are 1, 2, 1.
- Transformations:
    - groupByKey()
    - reduceByKey()

```
pairRDD = sc.parallelize([('a', 1), ('a', 2), ('b', 1)])
# mapValues only used to improve format for printing
print pairRDD.groupByKey().mapValues(lambda x:
list(x)).collect()
Output: [('a', [1, 2]), ('b', [1])]
```

Actions

Actions

- Return a value to the driver program after running a computation on the dataset
- Mechanism to get the results to the driver from workers
- Cause spark to execute the recipe to transform source

Data Fetching	Aggregation	Output
collect take(n) first takeSample	reduce count countByKey	foreach foreachPartition saveAsTextFile saveAsSequenceFile saveAsObjectFile saveToCassandra

Actions – Examples – count, collect

```
# Create an RDD
data = [1, 2, 2, 3, 4, 4, 5, 6, 7]
RDD1 = sc.parallelize(data, 4)
```

```
# Count the number of elements in RDD1 and print.
print(RDD1.count( ))
```

```
# Collect the data of RDD1 and print
 print(RDD1.collect())
```

Additional Actions – Examples

```
# Get the first element
print(filteredRDD.first())


# The first 4
print(filteredRDD.take(4))
```

```
# Create new base RDD to show countByValue
repetitiveRDD = sc.parallelize([1, 2, 3, 1, 2, 3, 1, 2, 1, 2, 3, 3,
3, 4, 5, 4, 6])
print(repetitiveRDD.countByValue())
Output:defaultdict(<type 'int'>, {1: 4, 2: 4, 3: 5, 4: 2, 5: 1, 6:
1})
```

Types of RDDs

- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD
- HadoopRDD

- DoubleRDD
- JdbcRDD
- JsonRDD
- SchemaRDD
- VertexRDD
- EdgeRDD

- CassandraRDD (DataStax)
- GeoRDD (ESRI)

Shared Variables

- When a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all the variables used in the function.
- These variables are copied to each machine and no updates to the variables on the remote machine are propagated back to the driver program.
- Supporting general, read-write shared variables across tasks would be inefficient.
- However, **Spark does provide two limited types of shared variables** for two common usage patterns:
  - **Broadcast Variables**
  - **Accumulators**

Broadcast Variables

- **Keep a read-only variable cached on each machine**
- For example, give every node a copy of a large input dataset
- Reduce communication cost
- Spark actions are executed through a set of stages, separated by distributed “shuffle” operations.
- Spark automatically broadcasts the common data needed by tasks within each stage.

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>
>>> broadcastVar.value
[1, 2, 3]
```

Broadcast Variables

- After the broadcast variable is created, it should be used instead of the value v in any functions run on the cluster so that v is not shipped to the nodes more than once.

- **Broadcast object v should not be modified after it is broadcast** in order to ensure that all nodes get the same value of the broadcast variable (Ex: if the variable is shipped to a new node later).

```
>>> bvar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>
>>> bvar.value
[1, 2, 3]
```

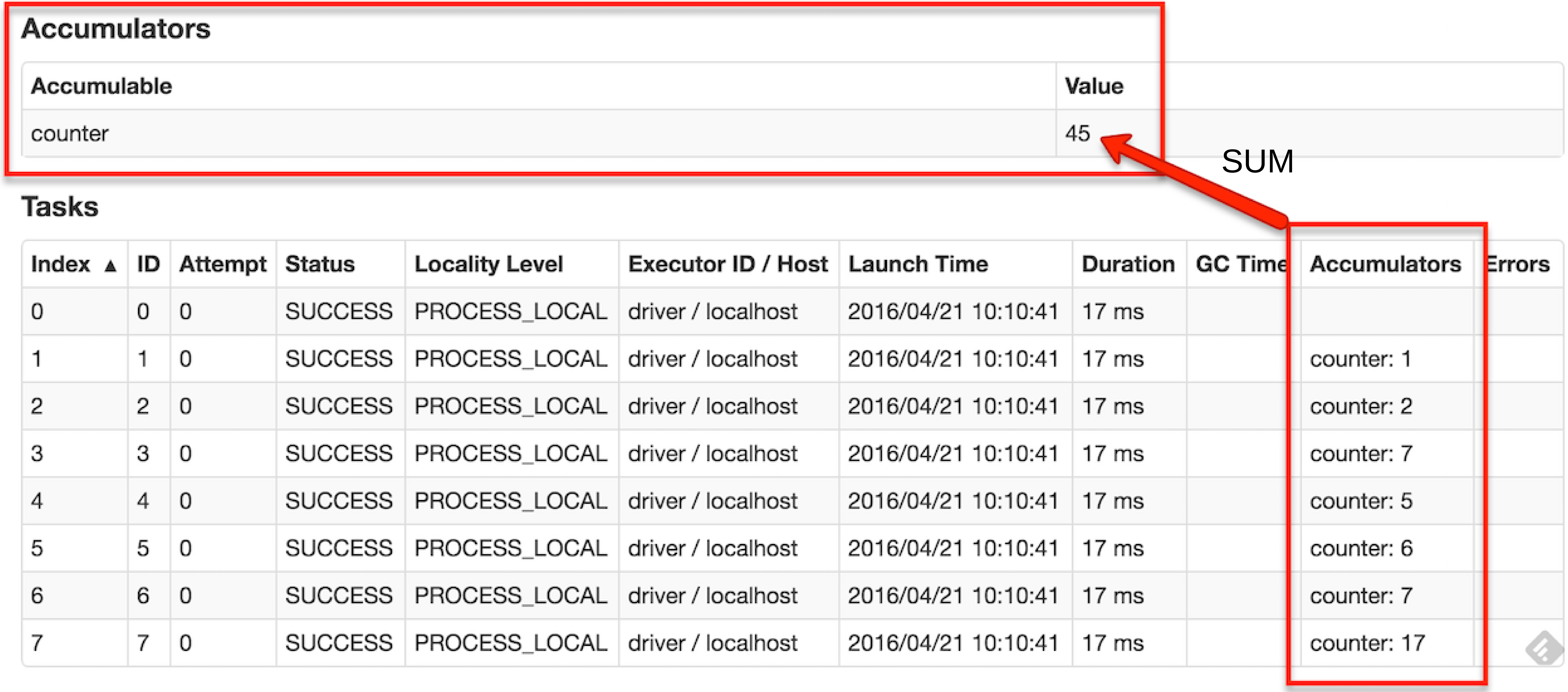
Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation.
- They can be used to implement counters or sums.
- Spark natively supports accumulators of numeric types and programmers can add support for new types.
- User can create **named** or **unnamed** accumulators.

Accumulators

- As seen in the image below, a named accumulator (in this instance counter) will display in the web UI for the stage that modifies that accumulator. Spark displays the value for each accumulator modified by a task in the "Tasks" table.

NOTE: this is not yet supported in Python.



Accumulators

Accumulable	Value
counter	45

SUM

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Thank You