UE Large Scale Processing @ ESIGELEC
2019/2020

10 – Spark

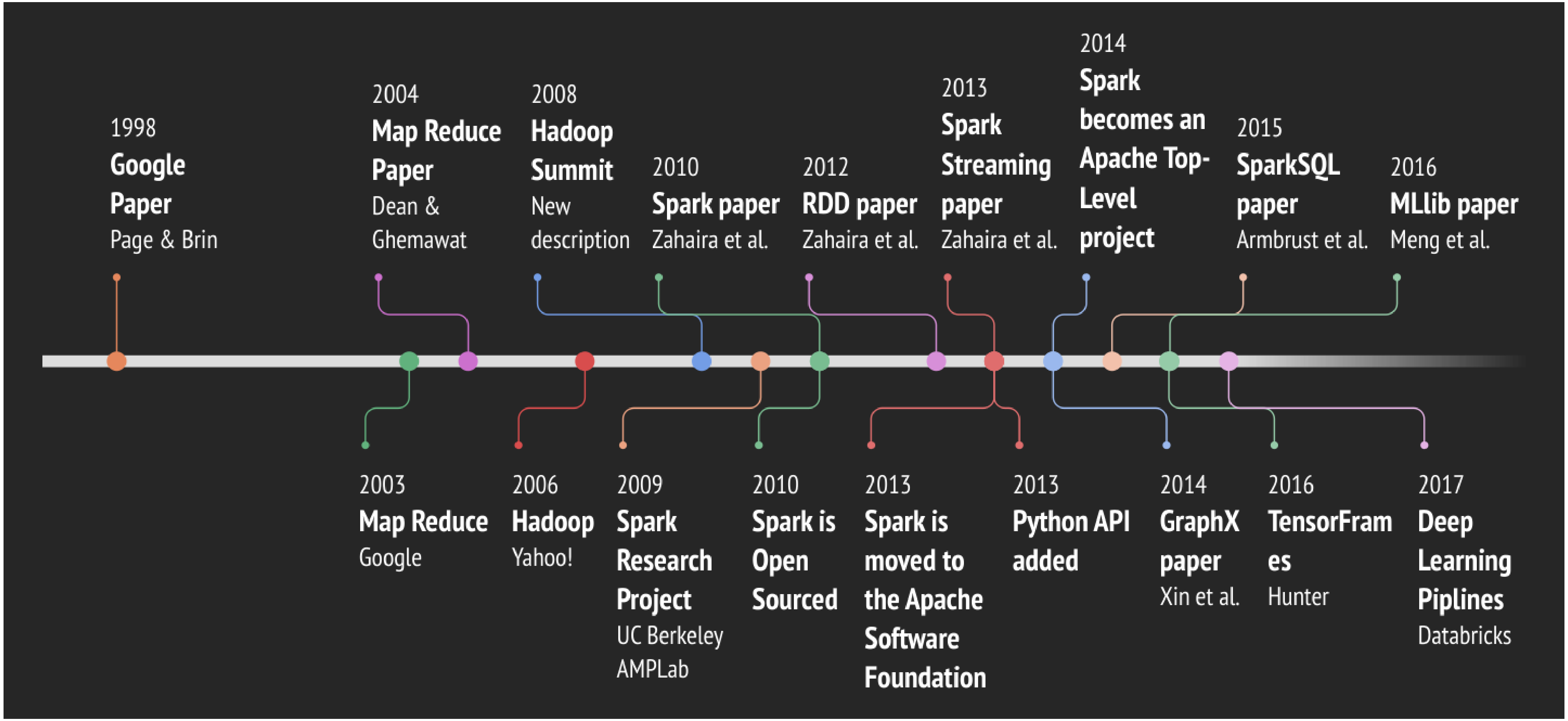
Abdel Dadouche
DJZ Consulting

adadouche@hotmail.com
@adadouche

Some History

- Started by Matei Zaharia in 2009 at the AMPLab (Algorithms, Machines and People Lab) from UCB (home of Apache Mesos too!)
- Open sourced in 2010
- Donated to the Apache Software Foundation in 2013
- Backed by Databricks (founded by Matei Zaharia)
- Most active projects @ ASF (more than 1k contributors)

History Timeline



Source: https://www.kdnuggets.com/2018/05/deep-learning-apache-spark-part-2.html

Spark Goals?

- Build a common toolset/platform for:
  - Data Scientist
  - Data Engineer
  - Data Analysts
- Provide better performance than MapReduce (use memory, not disks)
- Improves usability with diverse but more concise API
- Be more open (Hadoop, but not only)
- Scale ! Scale ! Scale !

What is the Spark DNA?

- Use memory instead of disks
- Uses a DAG (directed acyclic graph) execution engine with support for:
  - in-memory storage
  - data locality
  - (micro) batch & streaming support
- Uses RDD (resilient distributed dataset) - but not only anymore!
  - An immutable collection of fault tolerant elements that can be operated on “in-parallel”
- No transformation is applied until an actions requires it!

Remember, memory is still faster than disk!
Network is killing it! SSD is expensive!

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

4 X

100 X

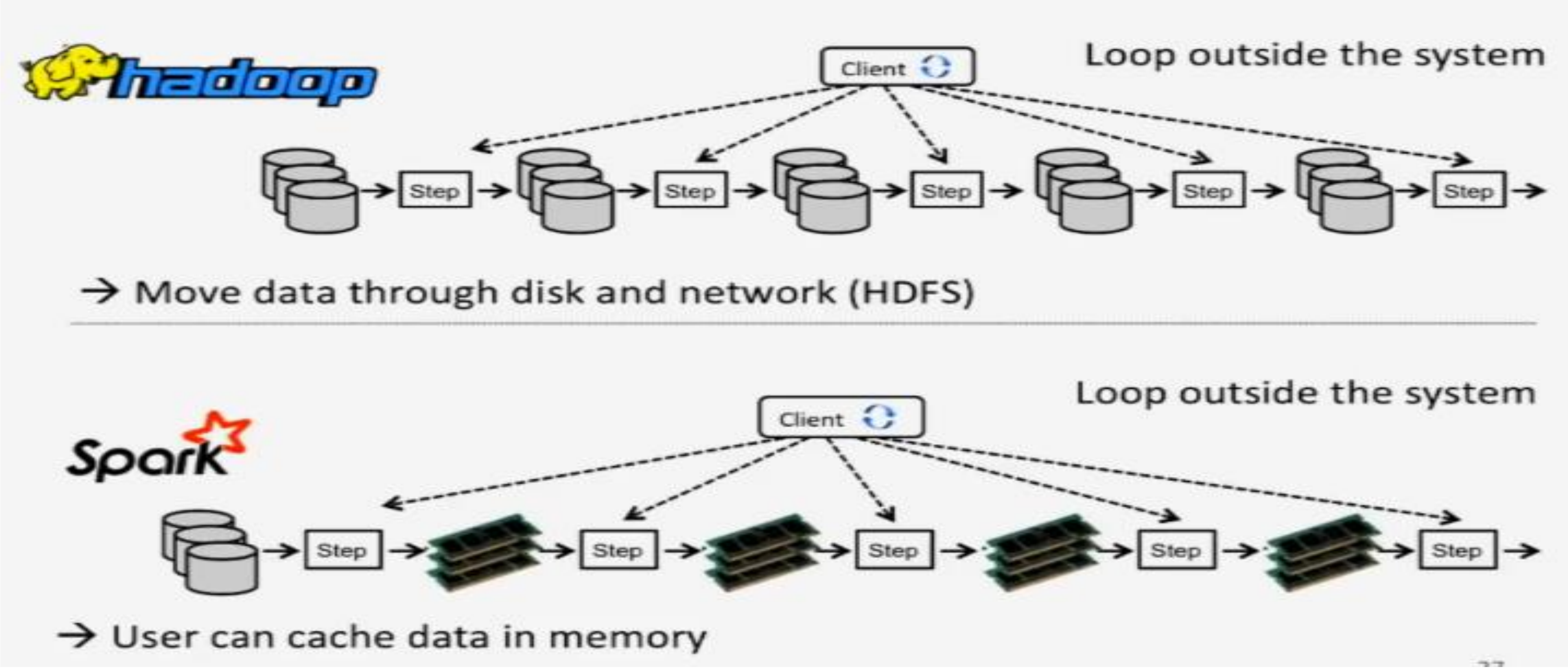
Réf : Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Spark Benefits?

- Hadoop compatible:
  - Native integration with Hive, HDFS & any Hadoop File System implementation
- Different execution mode:
  - Standalone, on YARN, on Mesos…
- Faster development
  - Concise & various API: Scala (~3x lesser code than Java), Python and R
- Faster execution:
  - In-memory caching for iterative jobs
- Promotes code reuse:
  - APIs and data types are similar for batch and streaming

Spark vs Hadoop MapReduce

- More consistent and concise API in Spark
- Spark Shell (no need to compile to test!)
- As distributed as Hadoop MapReduce/YARN
- Can use the same commodity hardware as Hadoop MapReduce
- More open and flexible than Hadoop MapReduce



hadoop
Client
Loop outside the system
Step
Step
Step
Step
Step
→ Move data through disk and network (HDFS)
Spark
Client
Loop outside the system
Step
Step
Step
Step
Step
→ User can cache data in memory

Achievement: **3X faster using 10X fewer machines**

Startup Crunches 100 Terabytes of Data in a Record 23 Minutes

BY KLINT FINLEY 10.13.14 | 2:36 PM | PERMALINK

Share 1.1k | Tweet 800 | 8+1 75 | in Share 573 | Pin it

Paul Price/Getty

There's a new record holder in the world of "big data."

On Friday, Databricks—a startup spun out of the University California, Berkeley—announced that it has sorted 100 terabytes of data in a record 23 minutes using a number-crunching tool called Spark, eclipsing the previous record held by Yahoo and the popular big-data tool Hadoop.

Apache Spark Beats the World Record for Fastest Processing of Big Data

Open-Source Big Data Processing Engine Spark Beats Previous World Record of Sorting 100 Terabyte On-Disk; and Follows Up With 1 Petabyte Sort

MARKET WIRED

Databricks
October 10, 2014 9:00 AM

BERKELEY, CA--(Marketwired - Oct 10, 2014) - Databricks, the company founded by the creators of popular open-source Big Data processing engine Apache Spark, announced today that it has broken the world record for the GraySort, a third-party, industry benchmarking competition for sorting large on-disk datasets. Databricks completed the Daytona GraySort, which is a distributed sort of 100 terabyte (TB) of on-disk data, in 23 minutes with 206 machines with 6,592 cores during this year's Sort Benchmark competition. That feat beat the previous record, held by Yahoo, of 70 minutes using a large, open-source Hadoop cluster of 2100 machines for data processing. This means that Spark sorted the same data three times faster using ten times fewer machines.

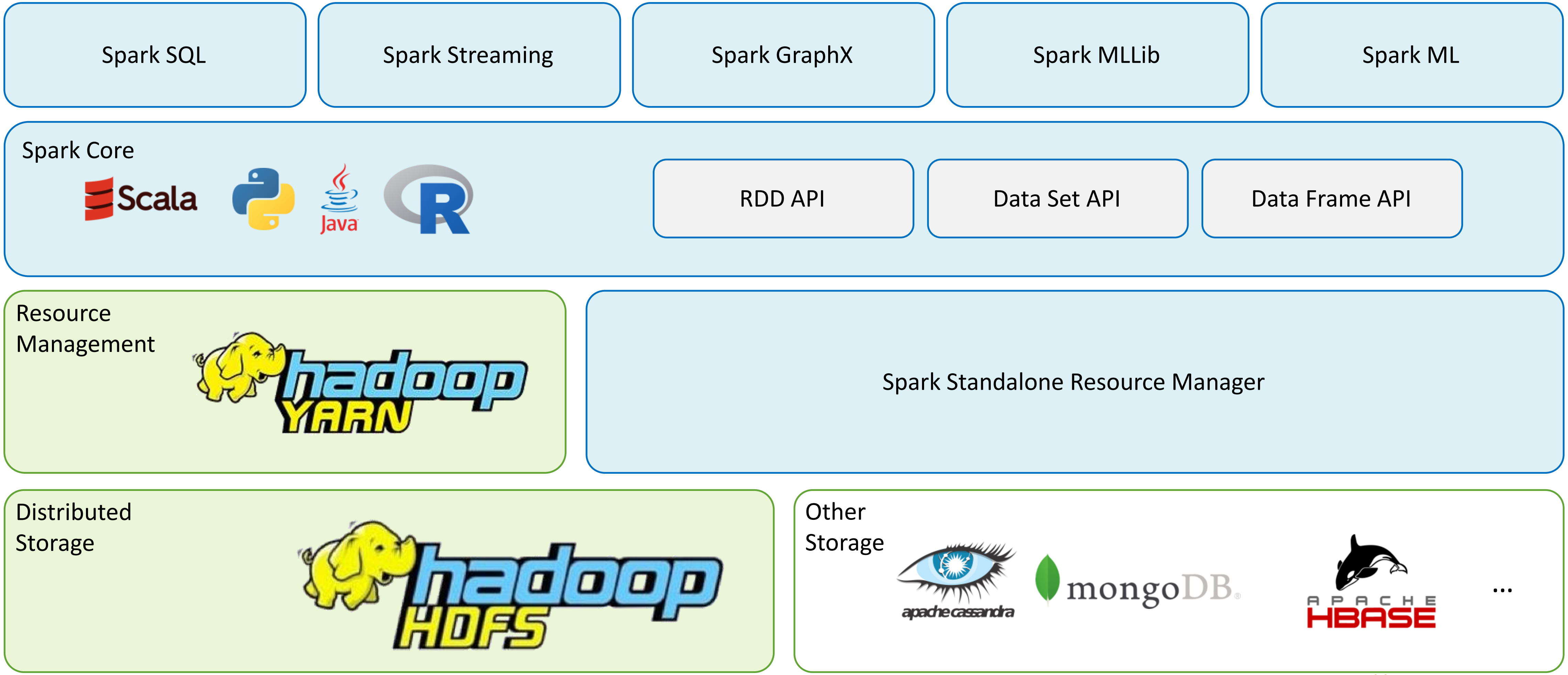
Additionally, while no official petabyte sort competition exists, Databricks pushed Spark further to also sort one petabyte (PB) of data on 190 machines in under four hours (234 minutes). This PB time beats previously reported results based on Hadoop MapReduce.

"Spark is well known for its in-memory performance, but Databricks and the open source community have also invested a great deal in optimizing on-disk performance, scalability, and stability," said Ion Stoica, CEO of Databricks. "Beating this data processing record previously set on a large Hadoop MapReduce clusters not only validates the work we've done, but also demonstrates that Spark is fulfilling its promise to serve as a faster and more scalable engine for all data processing needs. This

Source: http://www.wired.com/2014/10/startup-crunches-100-terabytes-data-record-23-minutes
http://finance.yahoo.com/news/apache-spark-beats-world-record-130000796.html

Spark Ecosystem, Architecture & Components

A Simple View …

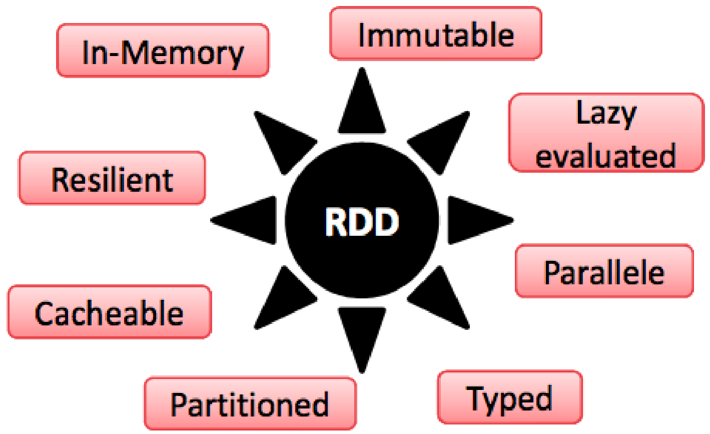


Logos: https://www.apache.org/logos

Spark Core / RDD

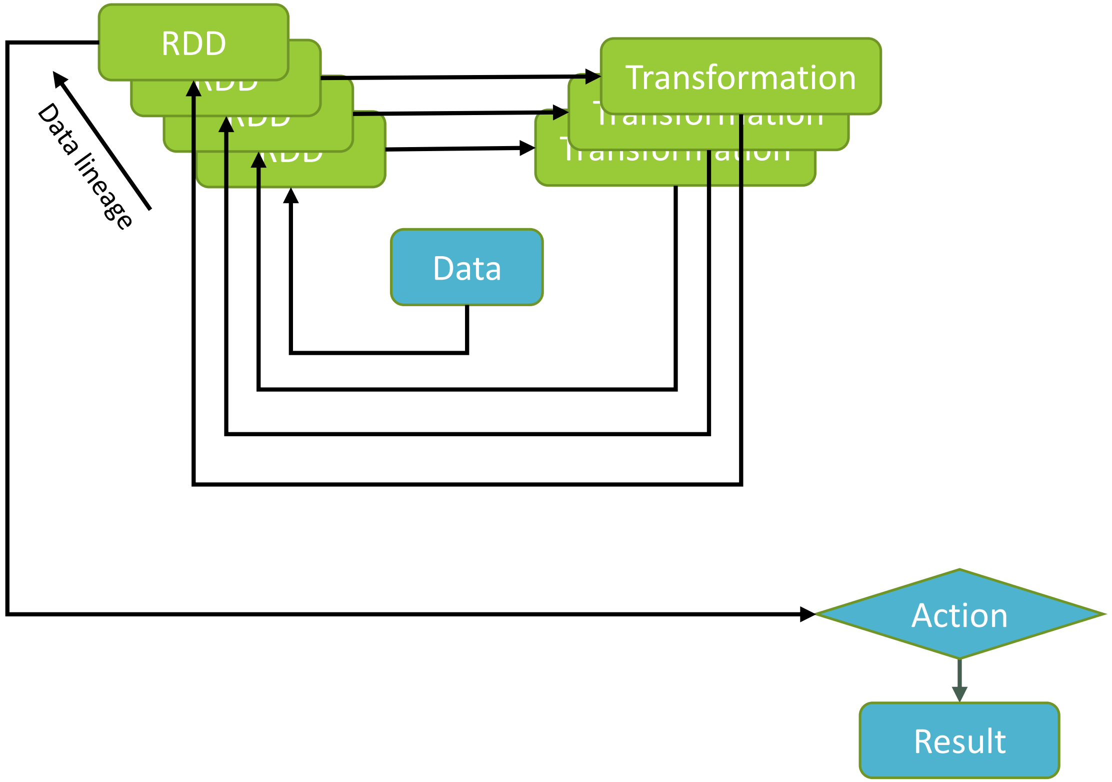
Spark Core : RDD (resilient distributed dataset)

- Immutable collection of fault tolerant elements that can be operated on "in parallel" that represents:
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - An optional partitioner
  - An optional list of preferred block locations for an HDFS file



Spark Core : RDD in a nutshell

- RDD
  - immutable, iter-able, partion-able & lazy loaded data structure
  - Provide data lineage capabilities (can be reconstructed)
  - Represent each and every step of the execution
- Transformation
  - Create a new RDD from an existing one applying an operation (map, filter, Sample, Union…)
- Action:
  - Evaluate the chain of transformation on the RDD object and return a result

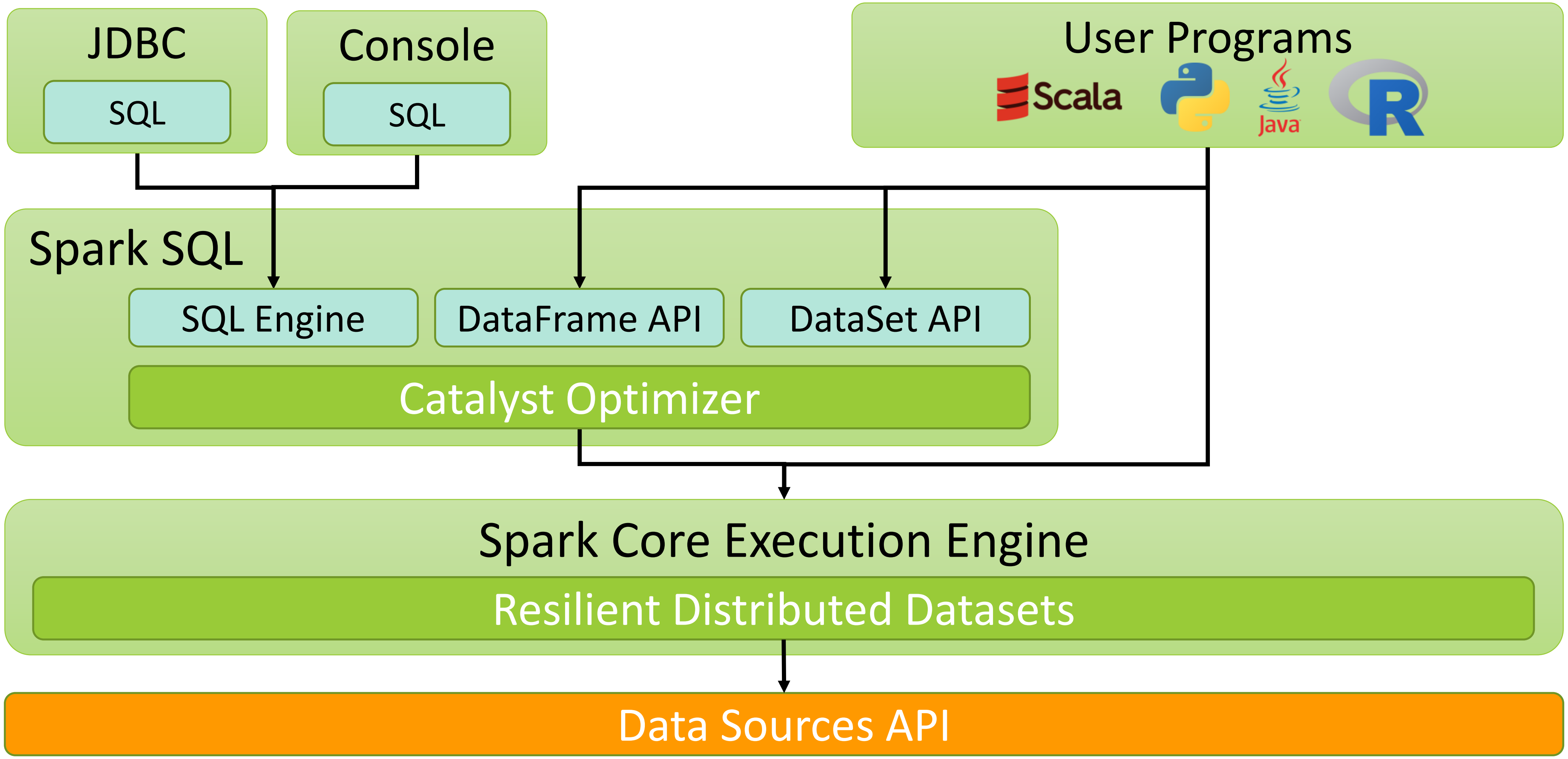


Spark SQL

Spark SQL

- Integrates relational processing with Spark's functional programming API
- Offers integration between relational and procedural processing through DataFrame API
- Includes a highly extensible optimizer, *Catalyst*
- Data Model:
  - Uses a nested data model based on Hive for tables and DataFrames
  - Supports major SQL data types, complex data types and user-defined types

Spark SQL



JDBC
SQL
Console
SQL
User Programs
Scala
Java
R
Spark SQL
SQL Engine
DataFrame API
DataSet API
Catalyst Optimizer
Spark Core Execution Engine
Resilient Distributed Datasets
Data Sources API

Spark MLlib / ML

Spark MLlib / ML

- Mllib / ML are a Spark implementation of some common machine learning algorithms and utilities, including:
  - classification
  - regression
  - clustering
  - collaborative filtering
  - dimensionality reduction…

Full List here: https://spark.apache.org/docs/latest/ml-guide.html

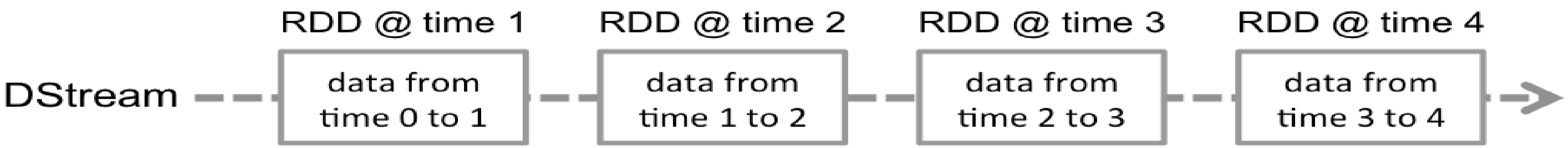
Spark MLLib vs ML

- Spark MLlib is the official name (there is no Spark ML)
- As of Spark 2.0, the RDD-based APIs in the ***spark.mllib*** package have entered maintenance mode
- The primary Machine Learning API for Spark is now the DataFrame-based API in the ***spark.ml*** package.

Spark Streaming

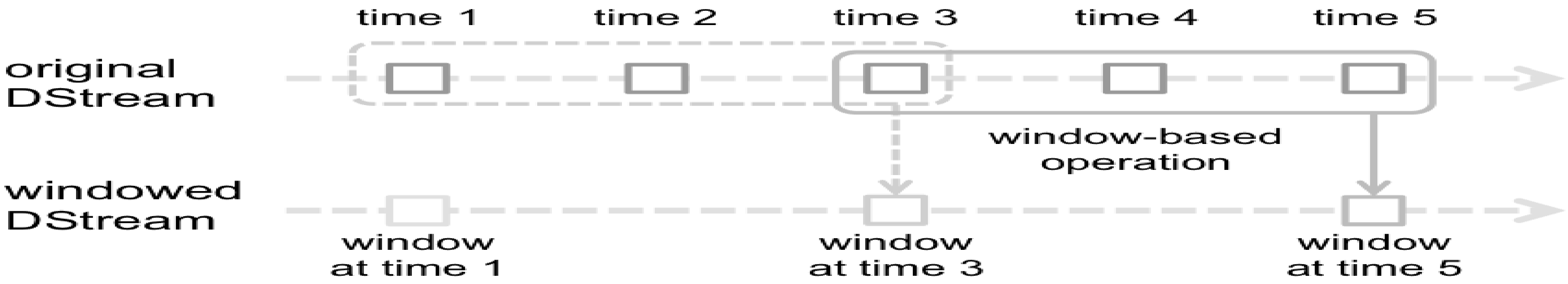
Spark Streaming

- Provides an API to handle streaming data in your applications
- Uses Discretized Stream (DStream), a sequence of RDD to handle each state change (new data streamed in/out)



Spark Streaming – Windowed Operations

- Useful when a certain operation must be executed over a certain period
  - Sum the last n
  - Moving average over the last

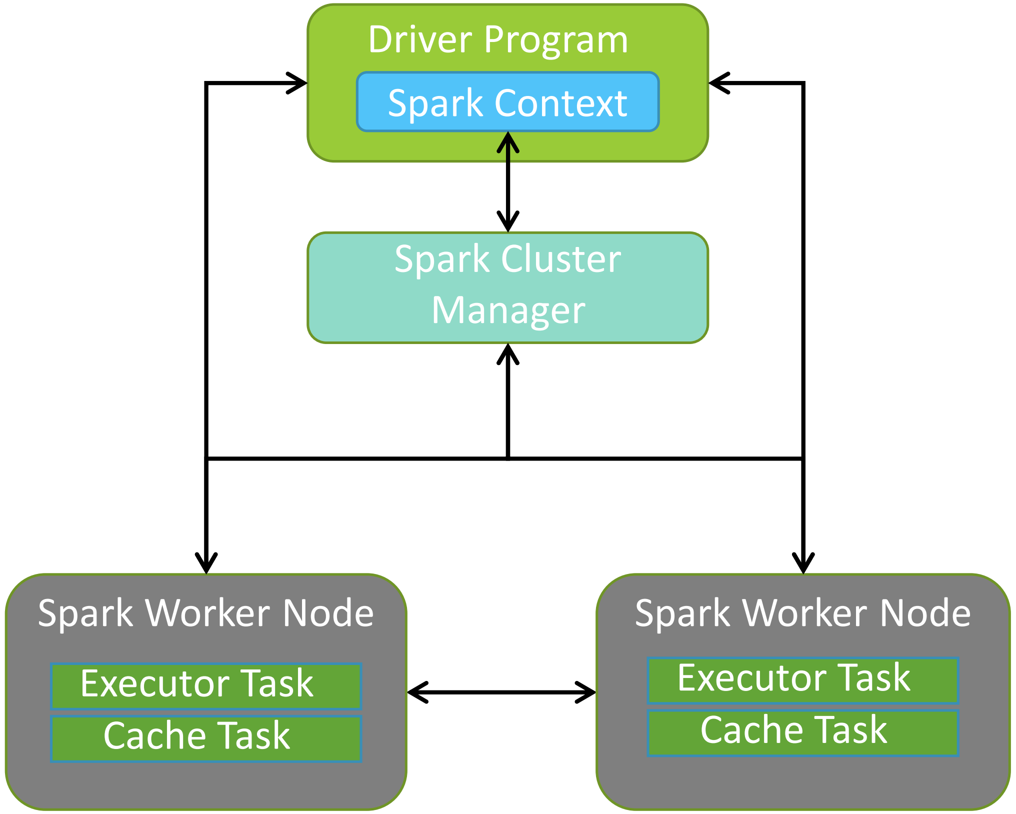


Building a Spark Application

Components & Deployments

Spark Application Components

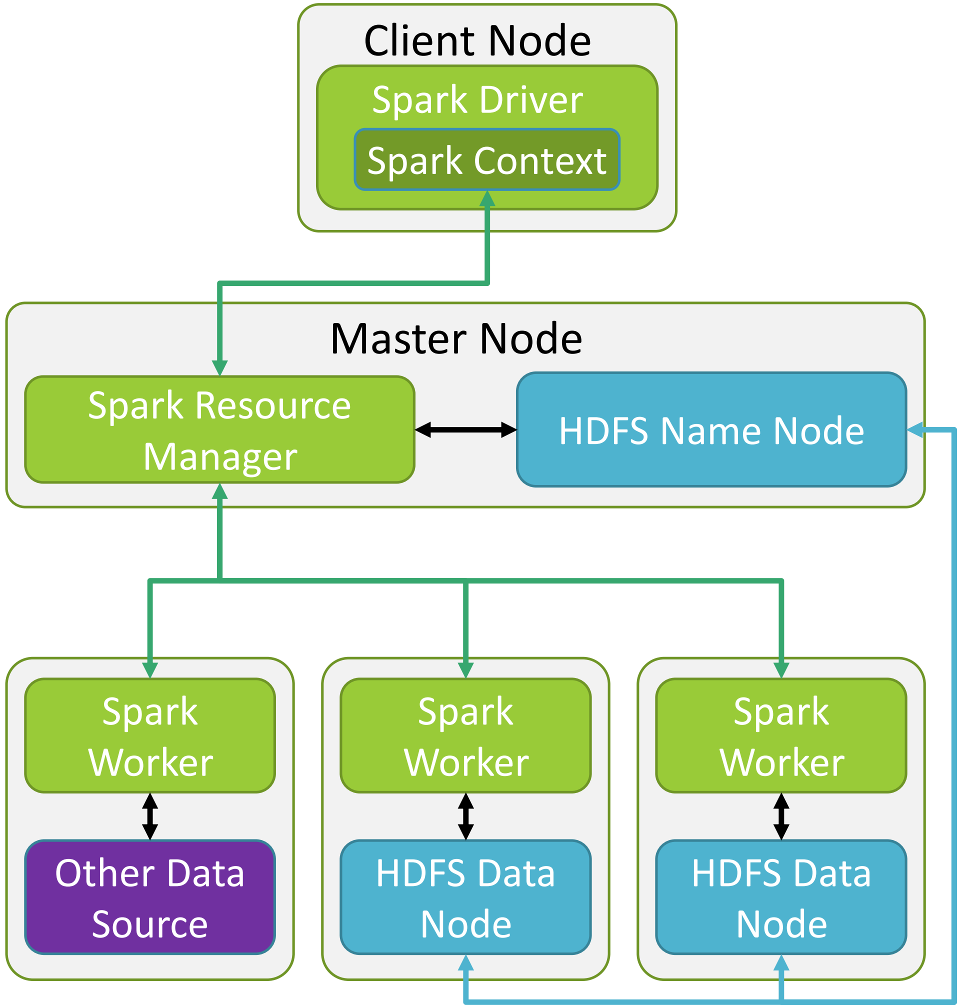
- Spark Driver Program
  - Contains main() function
  - Defines RDDs
  - Performs operations on RDDs
  - Access Spark through SparkContext
- Spark Context
  - Represents a connection to a cluster
  - Builds RDDs
  - Contacts Cluster Manager to Launch Executors
- Cluster Manager
  - Allocate resources across applications
- Executors – Spark worker “bees”
  - Run computations
  - Store data



Note: Spark shell or pyspark REPL is a Spark driver program & automatically created as context in REPL environments

Spark Application Deployments – Standalone Cluster

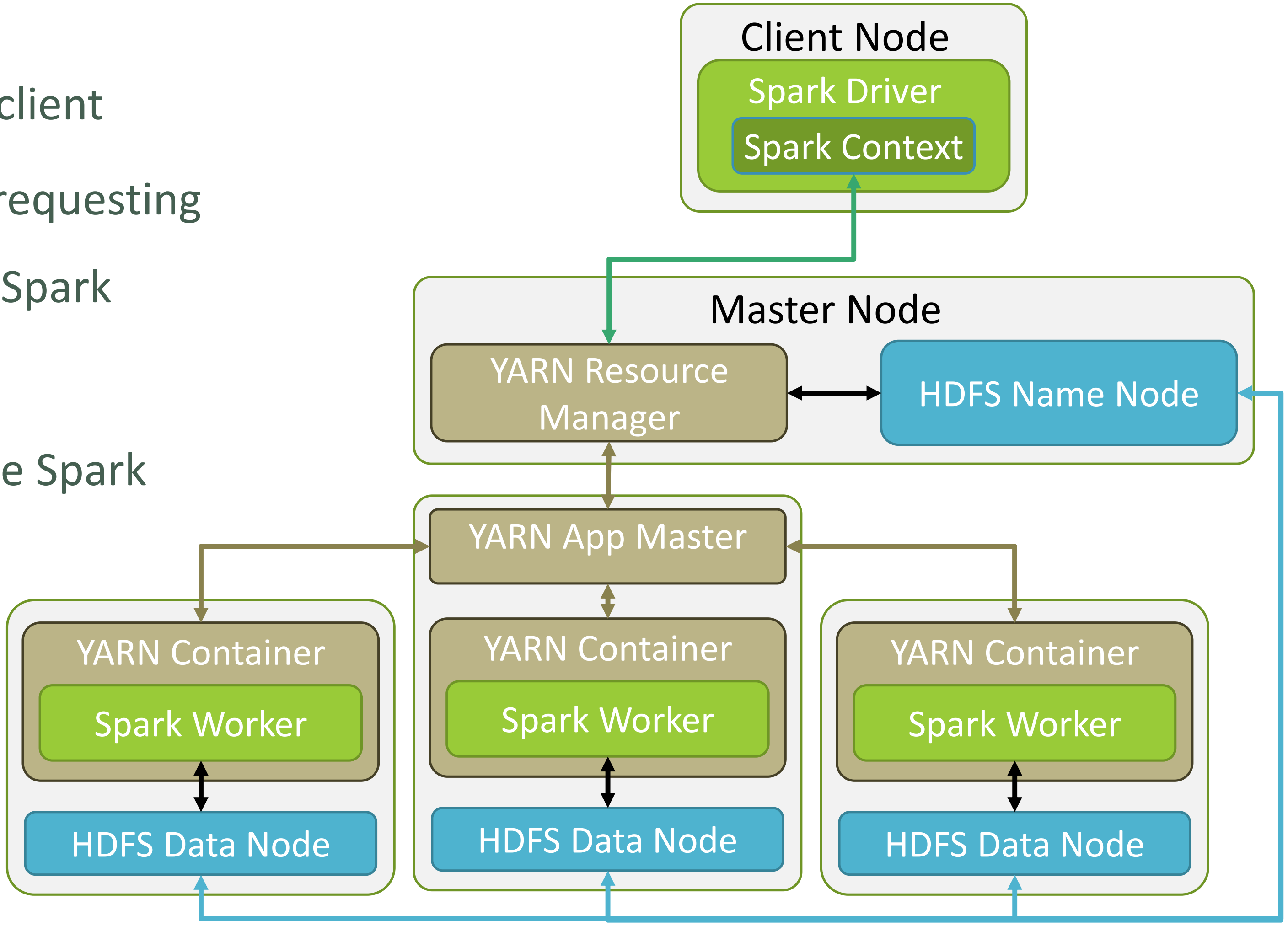
- Manage your own Cluster:
  - Network topology
  - Node resources
- You can access any type of data:
  - HDFS, S3…
- Colocation between the “worker” and the data is key to better performance but not mandatory



More details: https://spark.apache.org/docs/latest/spark-standalone.html

Spark Application Deployments – YARN Client Mode

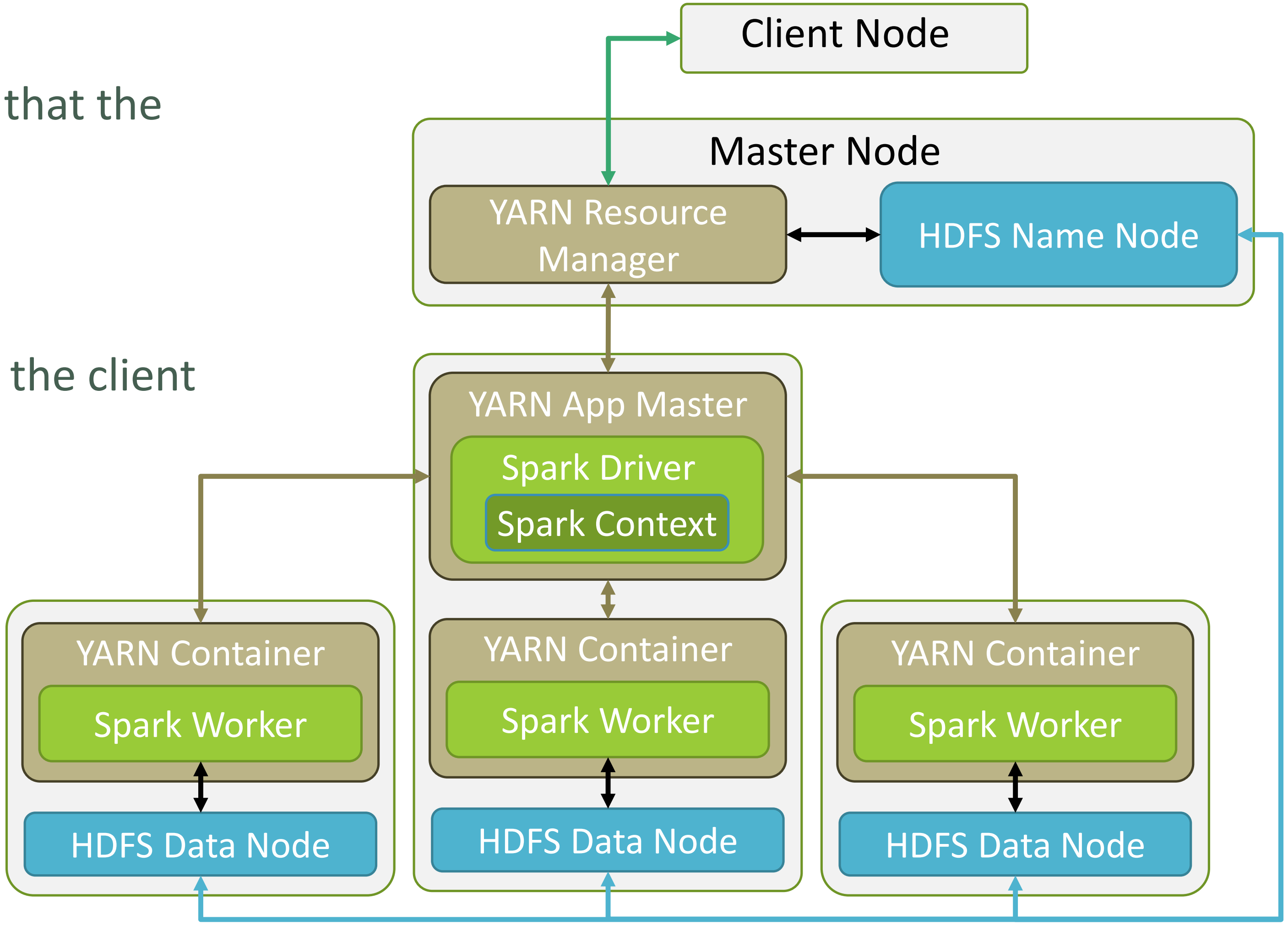
- The Spark Driver program runs on the client
- The YARN App master is only used for requesting resources for the Spark Worker by the Spark Driver
- The YARN Container are used to run the Spark Worker Executor and Cache Tasks



More details: https://spark.apache.org/docs/latest/running-on-yarn.html

Spark Application Deployments – YARN Cluster Mode

- Very similar to the Client mode except that the Spark Driver is deployed into the YARN Application Master
- You can submit you program and close the client machine, the job will continue

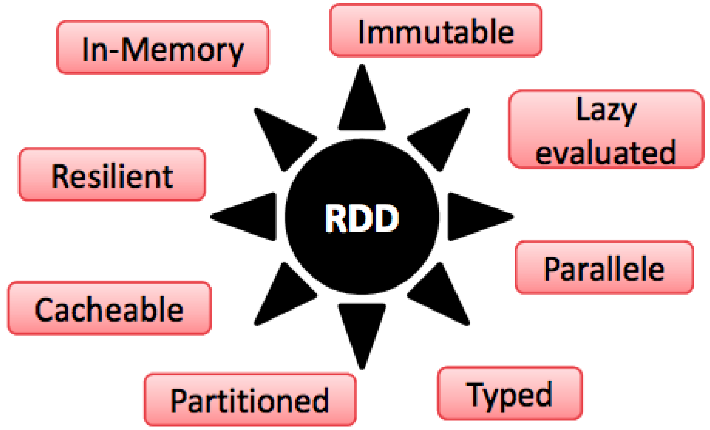


More details: https://spark.apache.org/docs/latest/running-on-yarn.html

A few more words about Spark RDD

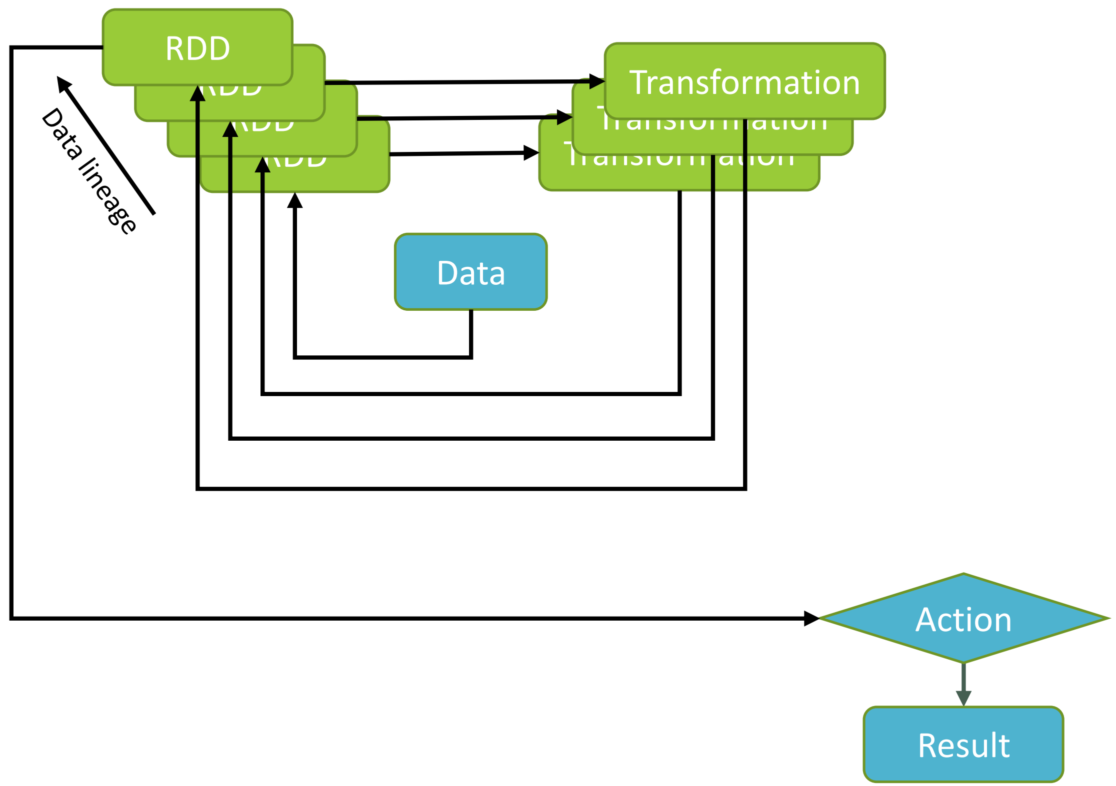
Spark RDD (resilient distributed dataset)

- Immutable collection of fault tolerant elements that can be operated on "in parallel" that represents:
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - An optional partitioner
  - An optional list of preferred block locations for an HDFS file



Spark RDD approach

- RDD
  - immutable, iter-able, partion-able & lazy loaded data structure
  - Provide data lineage capabilities (can be reconstructed)
  - Represent each and every step of the execution
- Transformation
  - Create a new RDD from an existing one applying an operation (map, filter, Sample, Union…)
- Action:
  - Evaluate the chain of transformation on the RDD object and return a result



Transformations

- **map**(func)
- **filter**(func)
- **flatMap**(func)
- **mapPartitions**(func)
- **mapPartitionsWithIndex**(func)
- **union**(otherDataset)
- **intersection**(otherDataset)
- **distinct**([numTasks]))
- **groupByKey**([numTasks])
- **sortByKey**([ascending], [numTasks])
- **reduceByKey**(func, [numTasks])
- **aggregateByKey**(zeroValue)(seqOp, combOp, [numTasks])
- **join**(otherDataset, [numTasks])
- **cogroup**(otherDataset, [numTasks])
- **cartesian**(otherDataset)
- **pipe**(command, [envVars])
- **coalesce**(numPartitions)
- **sample**(withReplacement,fraction, seed)
- **repartition**(numPartitions)

Full List here: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

Actions

- **reduce**(func)
- **collect**()
- **count**()
- **first**()
- **countByKey**()
- **foreach**(func)
- **take**(n)
- **takeSample**(withReplacement,num, [seed])
- **takeOrdered**(n, [ordering])
- **saveAsTextFile**(path)
- **saveAsSequenceFile**(path) (Only Java and Scala)
- **saveAsObjectFile**(path) (Only Java and Scala)

Full List here: https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Cache

- Use persist() or cache() to “save” any intermediate RDDs that will need to be reused
- RDDs’ partitions will be stored in memory buffers when calling persist() or cache()
- Limit the amount of memory using:
  - spark.storage.memory: if limit is exceeded, older partitions will be dropped from memory
  - spark.storage.memoryFraction: fraction of Java heap to use for Spark's memory cache

```
// cache data for reuse by reduceByKey()
val data = salaryData
        .map(line => line.split(',’))
        .map(line => (
                line(0), line(2).toInt)
        )
data.cache()
val totalSalary = data.reduceByKey {_ + _}
```

Cache - Storage Level

- Storage levels are set by passing a StorageLevel object to persist()
- The cache() method uses the default storage level:
  - StorageLevel.MEMORY_ONLY (store deserialized objects in memory)

Storage Level	Description
MEMORY_ONLY (default)	Deserialized. Partitions will not be cached. Recomputed them on demand
MEMORY_AND_DISK	Deserialized. Store the partitions that don’t fit on disk. Read them on demand
MEMORY_ONLY_SER	Serialized. More space-efficient than deserialized objects. CPU-intensive to read.
MEMORY_AND_DISK_SER	Spill partitions that don't fit in memory to disk instead of recomputing on demand.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Reduces GC overhead, allows executors to be smaller / share a pool of memory

Partitioning

- Each RDD is split into multiple partitions
- Each partitions can be computed on different nodes of the cluster
- More parallelism can be obtained with more partitions
- Tasks are executed on a partition
- Each HDFS block will require one partition
- Repartitioning can improve performance

Partitioning – Impact on Operations

- Operations that affect partitioning
  - cogroup
  - groupWith
  - join
  - leftOuterJoin
  - rightOuterJoin
  - groupByKey
  - reduceByKey
  - combineByKey
  - partitionBy
  - sort
  - mapValues
  - flatMapValues
- Operations that benefit from partitioning
  - cogroup
  - groupWith
  - join
  - leftOuterJoin
  - rightOuterJoin
  - groupByKey
  - reduceByKey
  - combineByKey
  - lookup

Spark RDD by Example: WordCount

Spark RDD by Example: WordCount

```
// Step 1. Create RDD from Hadoop text files
val docs = sc.textFile(“hdfs://docs/“)

// Step 2. Convert lines to lower case
val lower = docs.map(
        line => line.ToLowerCase
)

// Step 3. Split lines into words
val words = lower.flatMap(
        line => line.split(“\\s+“)
)
```

```
// Step 4. Convert into tuples
val counts = words.map(word => (word,1))

// Step 5. Count all words
val freq = counts.reduceByKey(_ + _)

// Step 6. Swap tuples (Partial code)
freq.map(_.swap)

// Step 7. Swap tuples (Complete code)
val top = freq.map(_.swap).top(N)
```

Create a Spark RDD

Use the parallelize method to convert an existing data collection into an RDD

```
from pyspark import SparkContext
```

```
myarray = range(1,20)
myarray
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
dist_array = sc.parallelize(myarray)
dist_array
```

```
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:223
```

Reference HDFS files (or any Hadoop storage)

```
data = sc.textFile("/user/root/whitehouse_visits.txt")
data
```

```
MappedRDD[19] at textFile at NativeMethodAccessorImpl.java:-2
```

Step 2. Convert lines to lower case

RDD[String]

Hello World

This is Spark

Spark

…

The end

```
.map(line => line.ToLowerCase)
```

=

```
.map(_.ToLowerCase)
```

RDD[String]

hello world

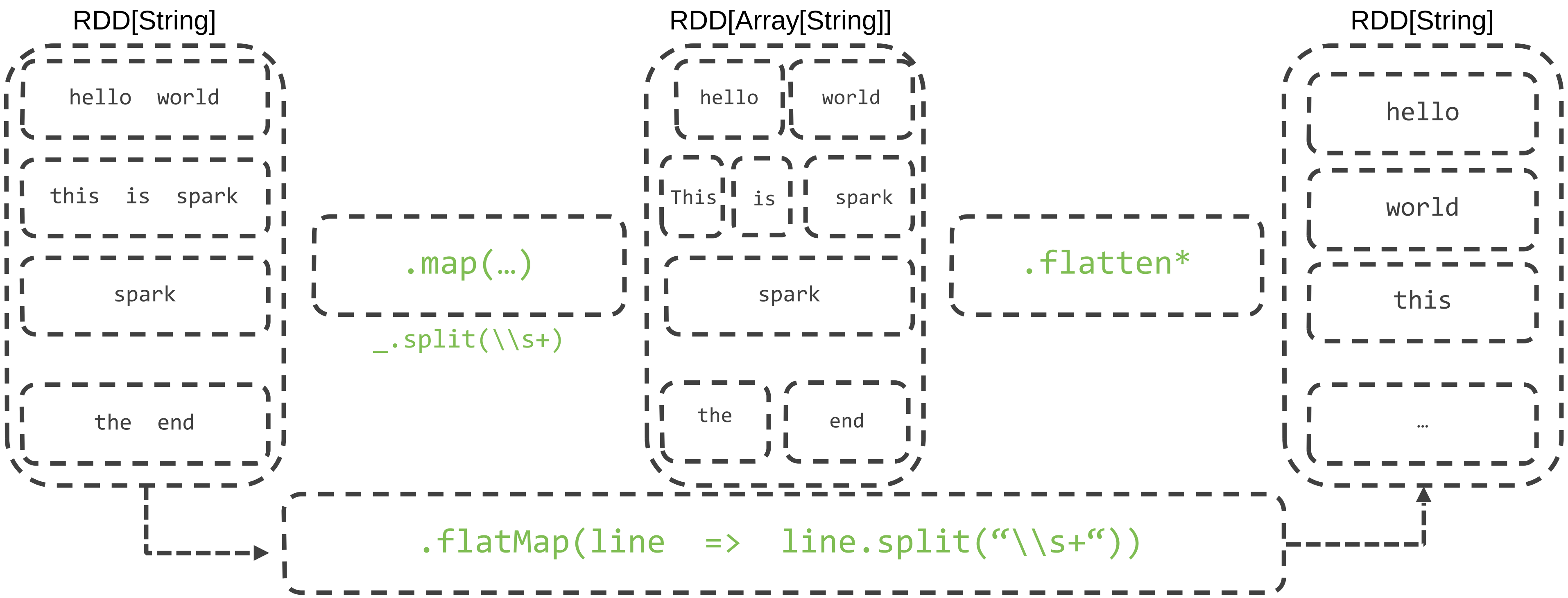
this is spark

spark

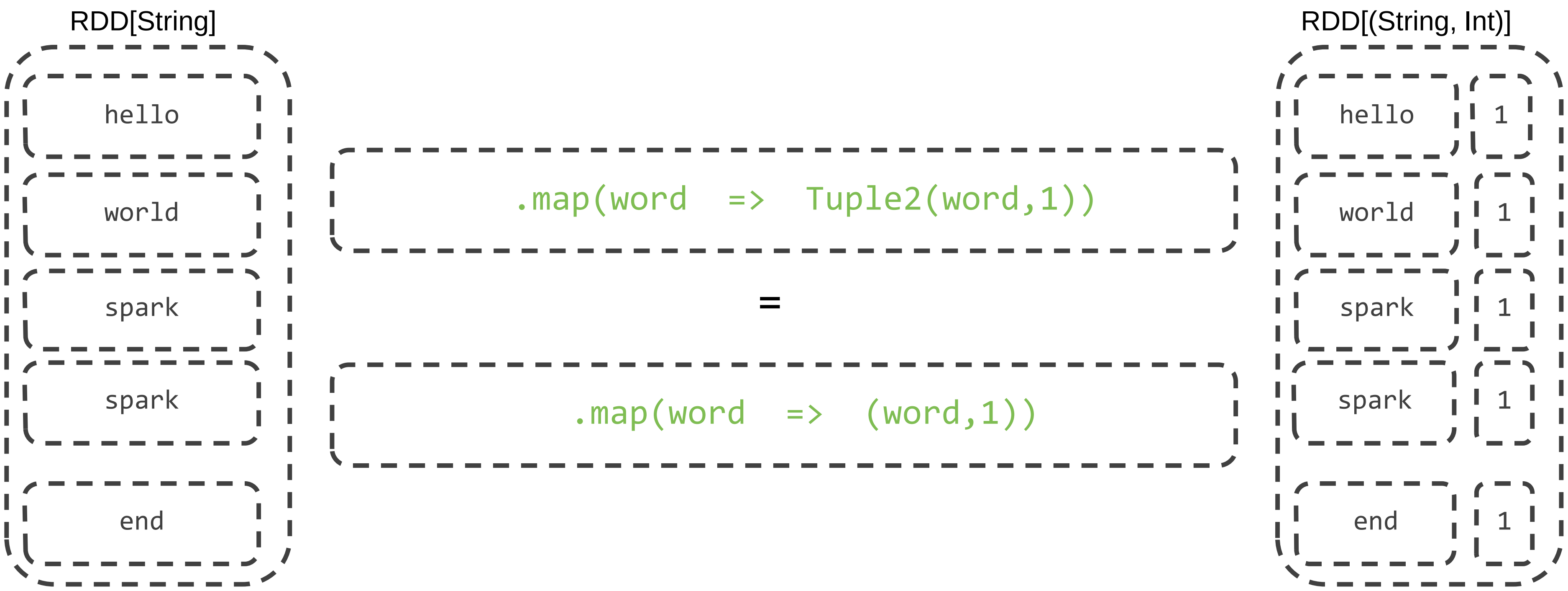
…

the end

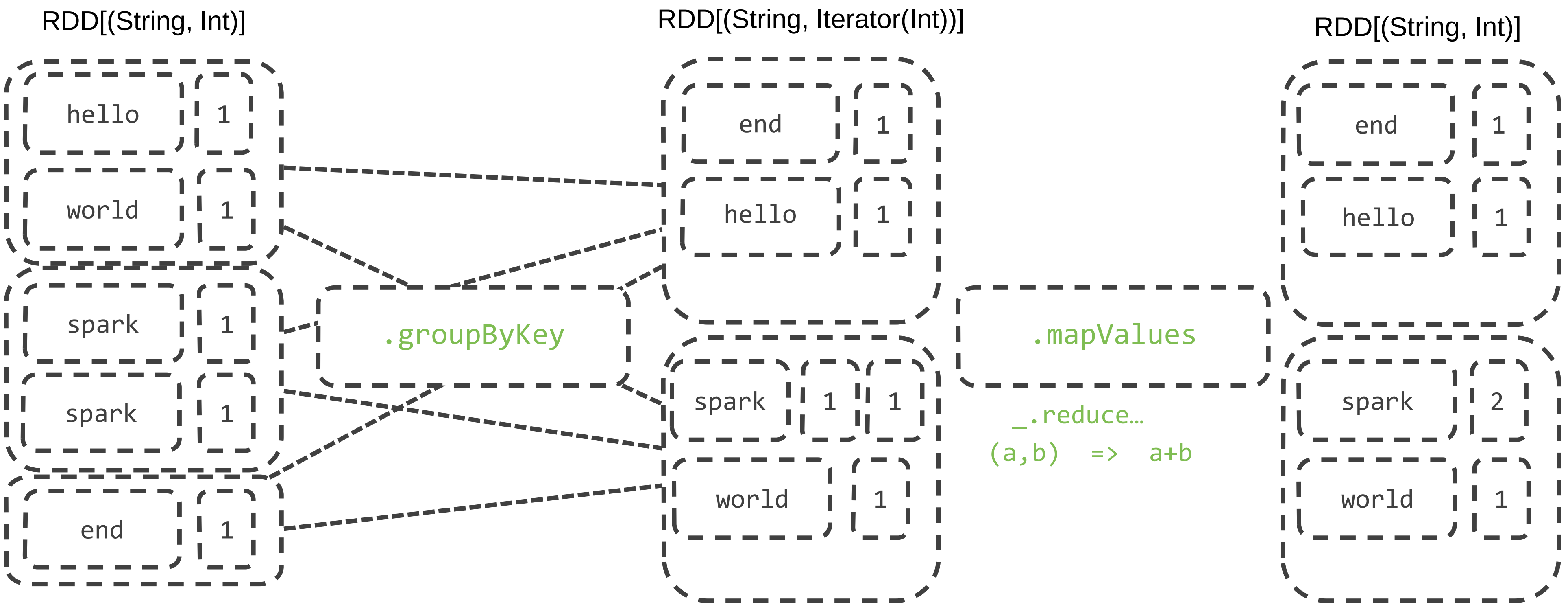
Step 3. Split lines into words



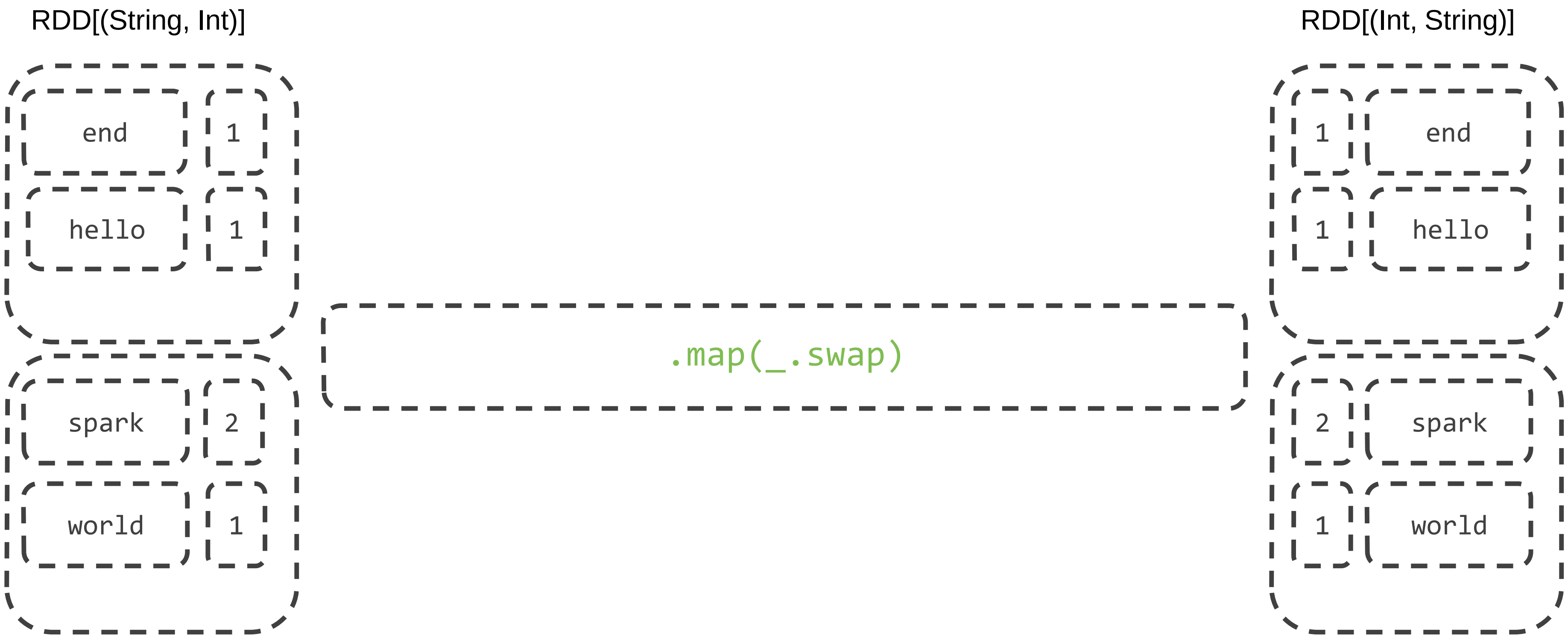
Step 4. Convert into tuples



Step 5. Count all words



Step 6. Swap tupels (Partial code)



Step 7. Swap tuples (Complete code)