COPERNICUS MASTER IN DIGITAL EARTH

HPC FOR BIG DATA

6. Spark

Frédéric RAIMBAULT, Nicolas COURTY

University of South Brittany, France

IRISA laboratory, OBELIX team

Outlines

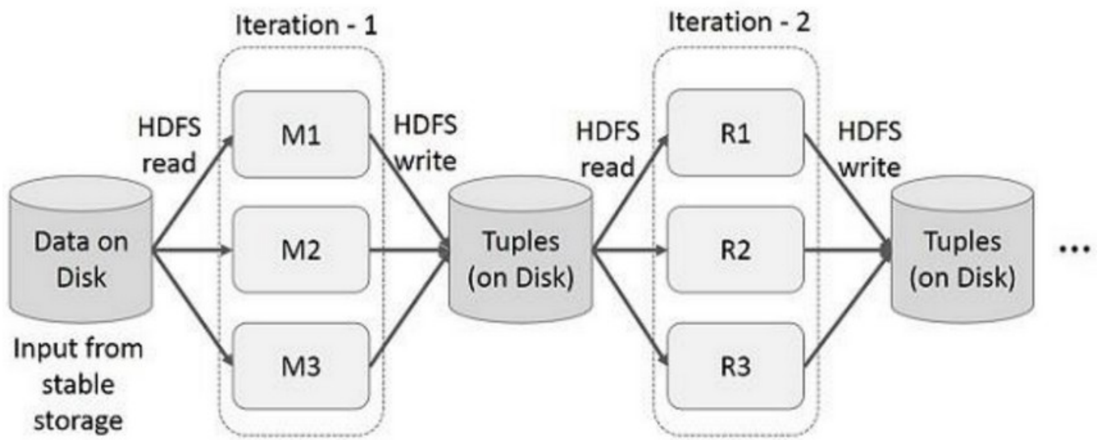
1. **Motivations**
2. **Programming model**
3. **Runtime Environment**

MapReduce Limitations

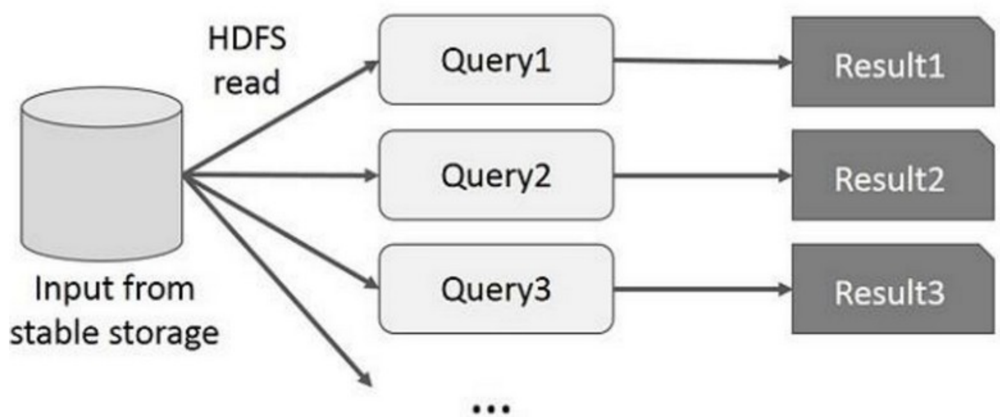
- **Programming model**
  - Requires advanced programming skills and deep understanding of the system architecture
  - Common data analysis tasks are not trivial
  - Batch processing only: no real-time data processing, no stream processing, no iterative processing
  - Computation steps are fixed
- **Performances**
  - Issue with small files (leading to small tasks with great overheads)
    - HDFS block is the unit of input data of the tasks map
  - Slow processing speed
    - intensive disk I/O during shuffle step
    - No caching: cascading MR jobs require the intermediate data to be materialized on disks
  - High latency
    - task initialization, scheduling, coordination and monitoring

Data Sharing Is Slow In MapReduce

- **Iterative operations on MapReduce**



- **Interactive operations on MapReduce**



Apache Spark Project

- **Started as a research project at UC Berkeley in 2009**
  - Open source in 2010
  - Donated to Apache Software Foundation in 2013
  - The founder of Spark created the company Databricks
- **One of the most active open source big data projects**
  - The MapReduce community has moved to Spark.
- **Faster and more general purpose data processing engine.**
- **Covers a wide range of workflow for example batch, interactive, iterative and streaming.**

Apache Spark vs. Hadoop MapReduce

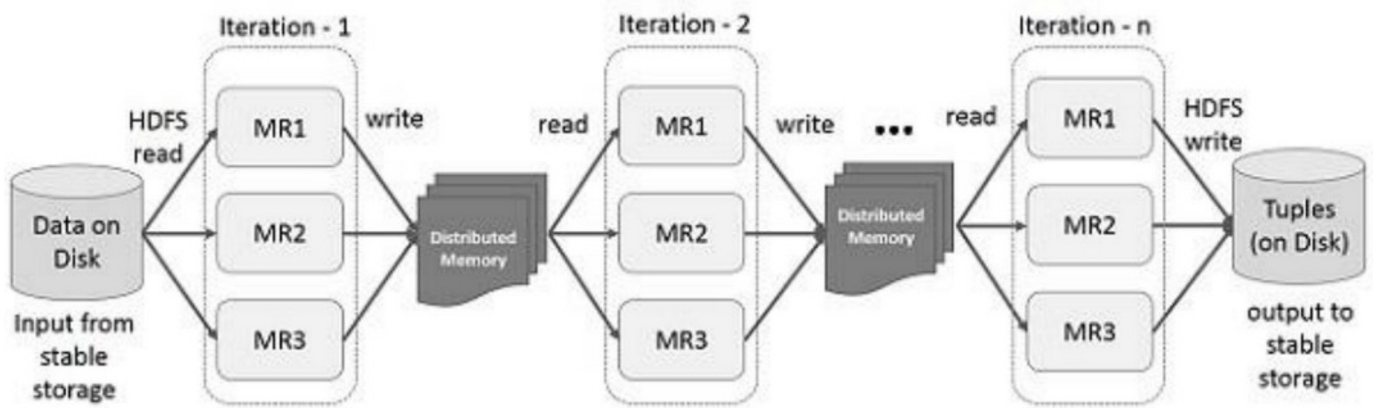
DataFlair

Spark vs Hadoop MapReduce

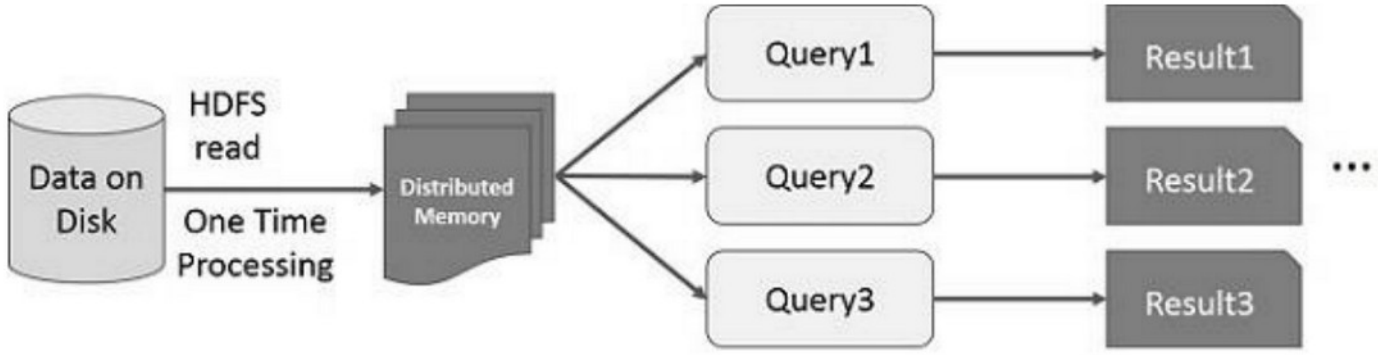
Factors	Spark	Hadoop MapReduce
Speed	100x times than MapReduce	Faster than traditional system
Written In	Scala	Java
Data Processing	Batch / real-time / iterative / interactive /graph	Batch processing
Ease of Use	Compact & easier than Hadoop	Complex & lengthy
Caching	Caches the data in-memory & enhances the system performance	Doesn't support caching of data

Data Sharing using Spark RDD

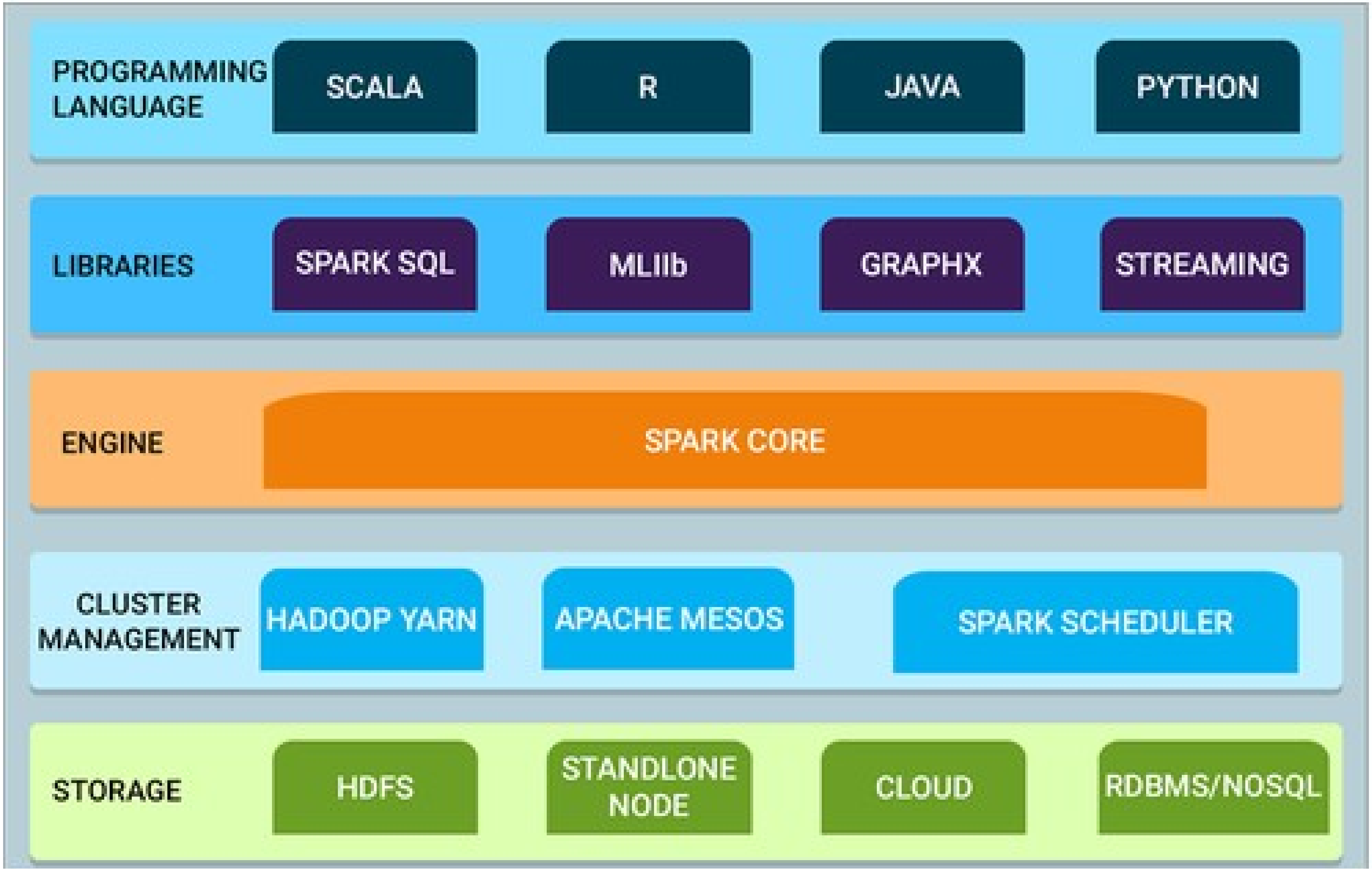
- **Iterative operations on Spark RDD**



- **Interactive operations on Spark RDD**



Spark Ecosystem



Spark Core

- **Contains the basic functionality for**
  - task scheduling,
  - memory management,
  - fault recovery,
  - interacting with storage systems, ...
- **Defines the Resilient Distributed Datasets (RDDs)**
  - main Spark programming abstraction.

Spark SQL

- **For working with structured data.**
- **View datasets as relational tables**
- **Define a schema of columns for a dataset**
- **Perform SQL queries**
- **Supports many sources of data**
  - Hive tables, Parquet and JSON
- **Works on DataFrame and Datasets abstractions**
- **Other interesting abstraction over DataFrame for EO:**
  - RasterFrame,
  - Apache Sedona (aka GeoSpark)
    - Specialized/optimized RDD

Spark Streaming

- **Data analysis of streaming data**

  e.g. log files generated by production web servers

- **Aimed at hight-throughput and fault-tolerant stream processing**

- **Near real-time processing**

Spark MLlib

- **MLlib is a library that contains common Machine Learning (ML) functionality:**
  - Basic statistics
  - Classification (Naïve Bayes, decision tress, LR)
  - Clustering (k-means, Gaussian mixture, …)
  - …
- **All the methods are designed to scale out across a cluster.**

Spark GraphX

- **Graph Processing Library**
- **Defines a graph abstraction**
  - Directed multi-graph
  - Properties attached to each edge and vertex
  - RDDs for edges and vertices
- **Provides various operators for manipulating graphs (e.g. subgraph and join vertices)**

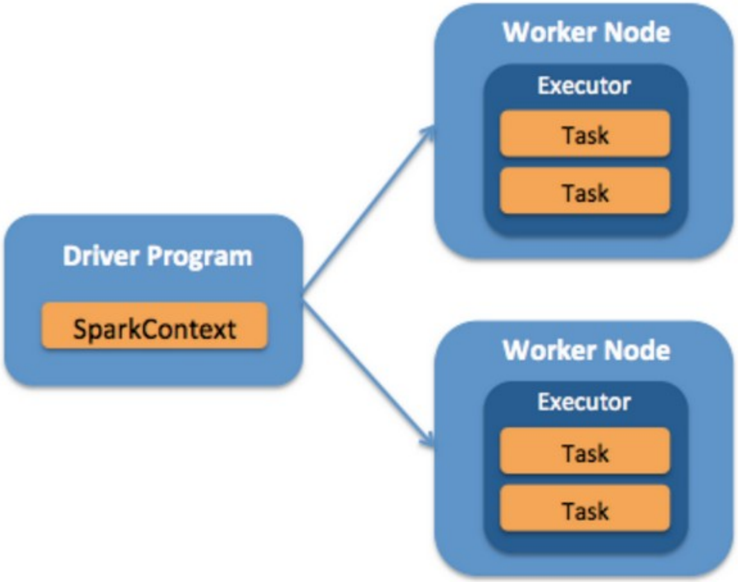
Outlines

1. Motivations
2. **Programming Model**
3. Runtime Environment

SparkContext

- **Spark applications consist of a driver program that controls the execution of parallel operations across a cluster.**
  - A driver program manages a number of workers nodes called executors (master-slaves model)
- **The driver program accesses the Spark environment through a `SparkContext` object**

1. **Sets the deployment environment**
   - local vs cluster
   - number of cores/nodes, memory size, ...
2. **Creates a `SparkContext`**
3. **Manages running tasks on the cluster**

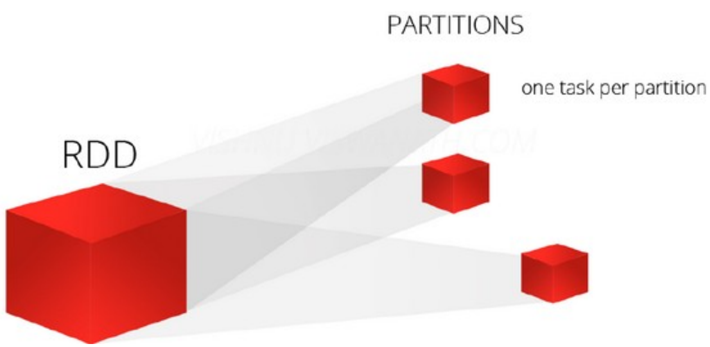


RDD: Resilient Distributed Dataset

Immutable distributed collection of elements that can be operated on memory in parallel

– Resilient, i.e. fault tolerant: lineage of data is preserved, so data can be re-created on a new node at any time

– Distributed: RDD are split into multiples partitions which can be computed on different nodes of the cluster



– Dataset: represents records of the data coming from external data sets

Features of Spark RDD

- In-memory computation
- Lazy evaluations of transformations until action
- Fault tolerance thanks to lineage of data
- Immutability
- Partitioning
- Persistence (user defined)
- Coarse-grained operations
- Location stickiness, close to data

RDD Creation

- **Parallelizing an existing collection in the driver program, e.g.**

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

  - Not generally used outside of prototyping since it requires entire dataset (`data`) in memory in one machine
- **Referencing an existing dataset in a external storage system, e.g.**

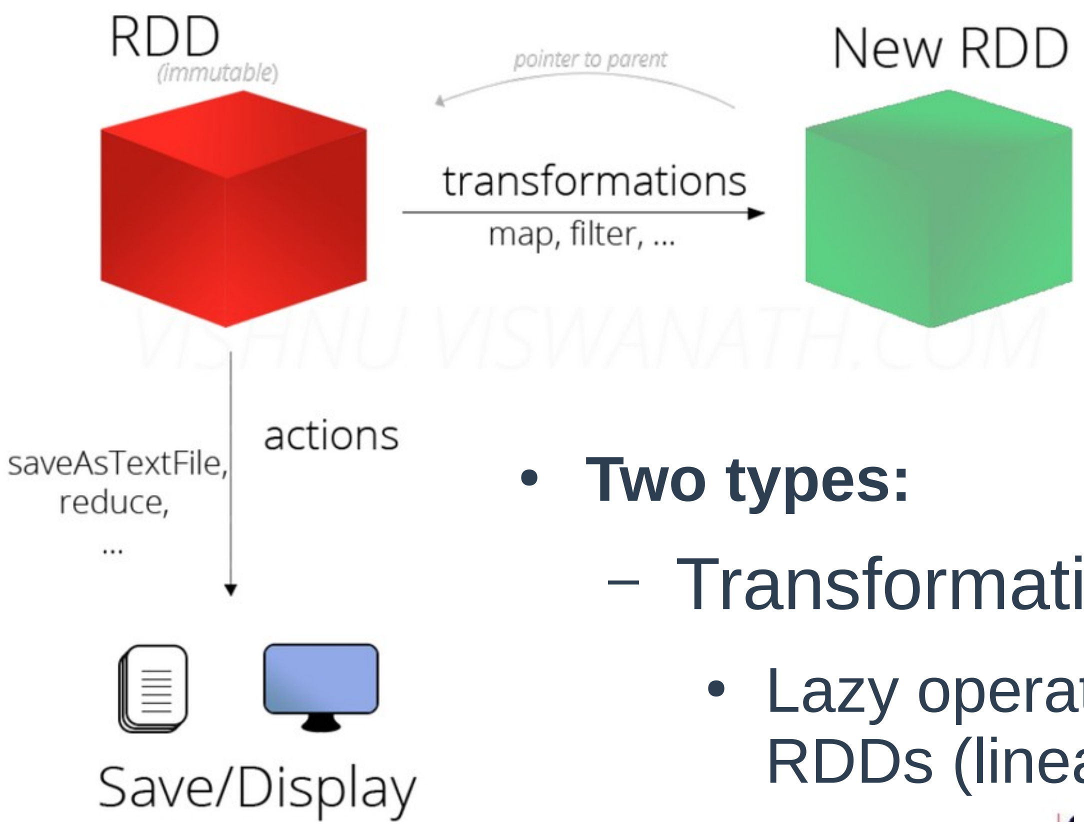
```
file= sc.textFile("data.txt")
```

  - Others methods exist to read data from HDFS,…
- **Applying transformations on existing RDDs, e.g.**

```
words= file.flatMap(lambda line: line.split(' '))
```

RDD Operations

Transformations	Actions
map (func)	reduce(func)
flatMap(func)	collect()
filter(func)	count()
groupByKey()	first()
reduceByKey(func)	take(n)
mapValues(func)	saveAsTextFile(path)
sample(...)	countByKey()
union(other)	foreach(func)
distinct()	...
sortByKey()	
...	



- **Two types:**
  - Transformations
    - Lazy operation to build RDDs from other RDDs (lineage saved)
  - Actions
    - Return a result or write it to storage

Programming with RDDs

- **General workflow for working with RDDs:**
  - creating a RDD from a data source
  - Apply transformations to RDDs
  - Apply action RDDs to compute a result.
- **Example in Python:**

```
lines= sc.textFile('data.txt')
line_len= lines.map(lambda x: len(x))
# line_len.persist()
doc_len= line_len.reduce(lambda x,y:x+y)
print doc_len
```

Passing Functions to Spark

- **Transformations and actions require function objects to be passed from the driver to the executors.**
  - Defined and passed through Python lambda functions
- **Anonymous functions created at runtime, e.g.:**

```
lambda x,y: x+y
lambda x: len(x)
```

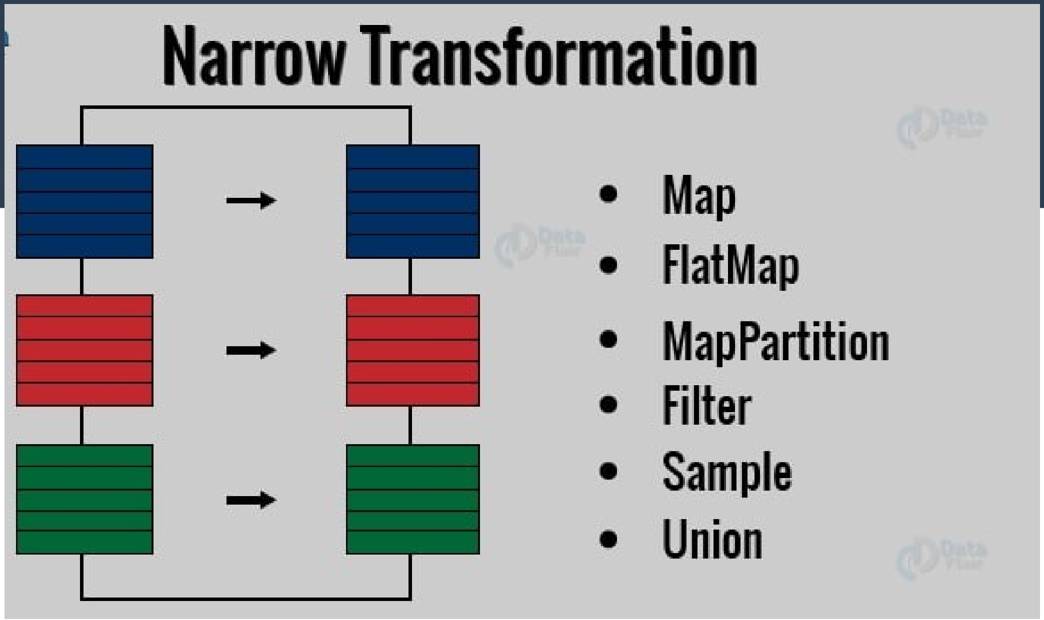
- **Externals variables in a closure will be shipped to the cluster, e.g:**

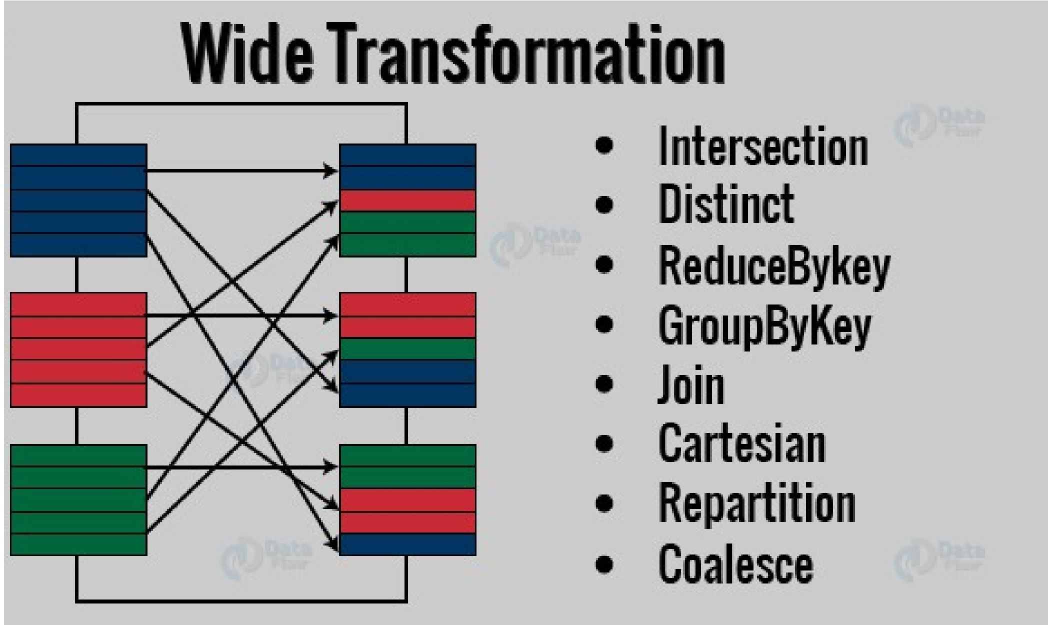
```
query= raw_input(“Enter a query:”)
pages.filter(lambda x: x.startswidth(query)).count()
```

- **Don’t use fields outside an outer object (ships all of it)**

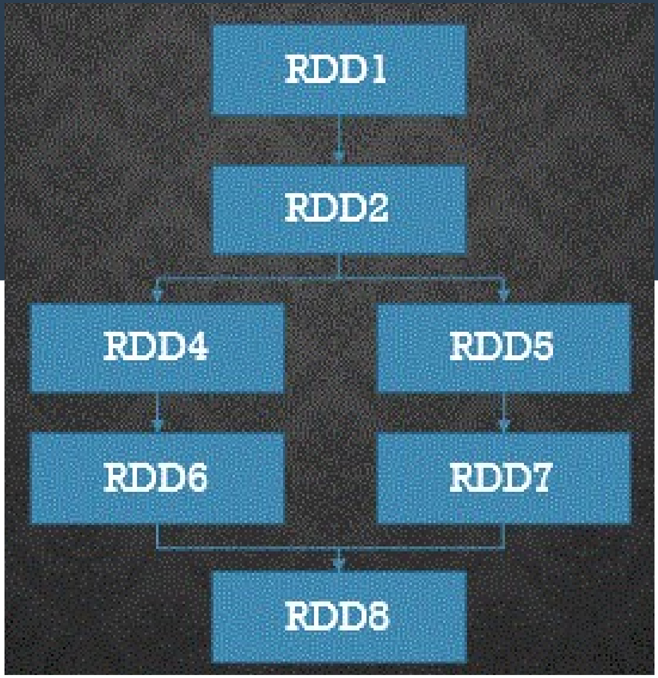
Transformations

- **Create new datasets from existing ones**
- **Two types:**
  - Narrow: an input partition is converted to only one output partition
  - Wide: input partitions contribute to many output partitions





RDD Lineage



- **Applying transformation built an RDD lineage**
  - this RDD dependency graph (a Directed Acyclic Graph) is named the logical plan
- **Lazy evaluation of transformations enables Spark to optimize the calculations**
  - The execution does not start until an action is triggered
  - Data is not loaded until it is necessary
  - Operations are grouped

Transformations: `map`

- **map(func)**
  - applies a function to each element in the source RDD to create a new RDD.
  - The input function must take a single input parameter and returns a value.

```
data = [1, 2, 3, 4, 5, 6]
rdd = sc.parallelize(data)
map_result = rdd.map(lambda x: x * 2)
map_result.collect()
```

→ [2, 4, 6, 8, 10, 12]

Transformations: `flatMap`

- **flatMap(func)**
  - As map() it applies a function to each element in the source RDD to create a new RDD.
  - The input function takes a single input parameter but returns a list of values.
  - The new RDD is formed by flattening the lists of value

```
data = [1, 2, 3, 4]
rdd = sc.parallelize(data)
map = rdd.map(lambda x:
              [x,pow(x,2)])
map.collect()
```

→ [[1, 1], [2, 4], [3, 9], [4, 16]]

```
rdd = sc.parallelize()
flat_map =
rdd.flatMap(lambda x:
              [x, pow(x,2)])
flat_map.collect()
```

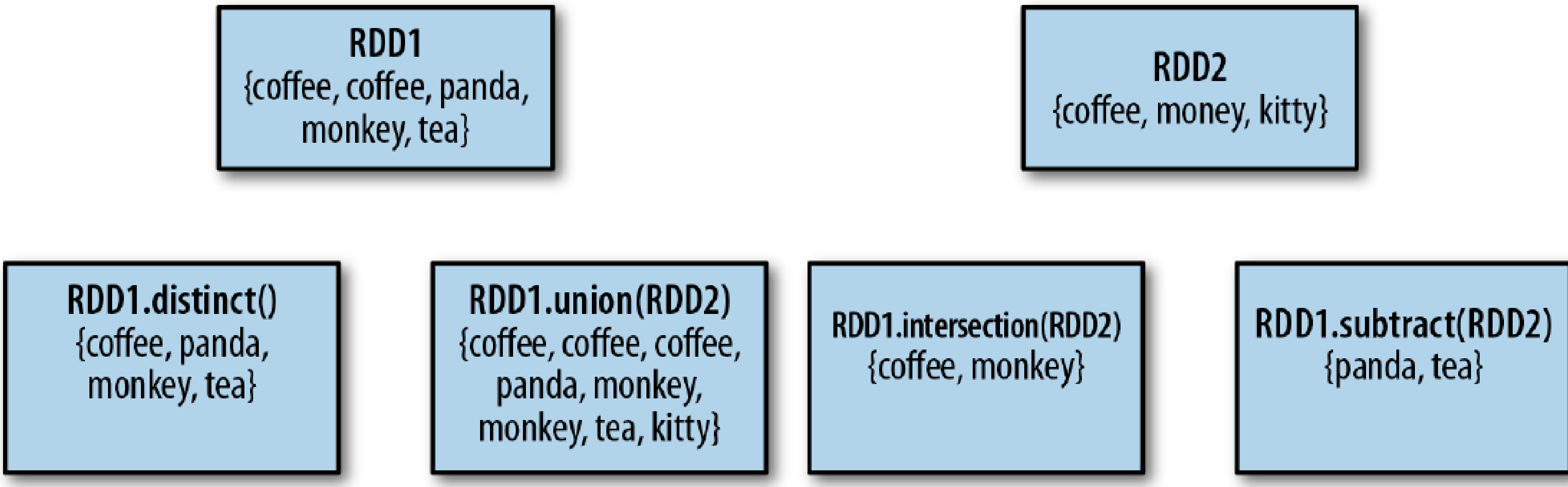
→ [1, 1, 2, 4, 3, 9, 4, 16]

Transformations: `filter`

- **filter(func)**
  - Returns a new RDD containing only the elements of the source that the supplied function returns as true.

```
lines= sc.textFile("README.md")
pythonlines= lines.filter(lambda line:
                                  "Python" in line)
```

Pseudo Set Transformations



+ cartesian product

- **Note that RDD themselves are not properly sets**

Actions

- **Actions are operations that give non-RDD value**
- **Actions force the evaluation of the transformations**
  - To avoid recompute RDD users can cache (persist) intermediate results
- **The result is returned to the driver program**
  - Or written to an external storage system
- **Some actions are available on certain types of RDDs, e.g.**
  - `mean()` and `variance()` on numeric RDD
  - `join()` on key-value pairs RDD

Common Actions

- **Reduce action**

```
rdd= sc.parallelize([1,2,3,3])
sum = rdd.reduce(lambda x, y: x + y)
```

→ 9

- **Fold action**

```
sum = rdd.fold(0,lambda x, y: x + y)
```

→ 9

Others Actions
Examples on {1,2,3,3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`

Caching

- **RDD persistence (or caching) is an optimization technique in which saves the result of RDD evaluation.**

```
logs= sc.textFile("path/to/log-files")

logs.cache()

errorLogs= logs.filter(lambda l:
                       l.contains("ERROR"))

warningLogs= logs.filter(lambda l:
                         l.contains("WARN"))

errorCount = errorLogs.count

warningCount = warningLogs.count
```

Paired RDD

- **A RDD containing a key-value pair**
  – Used to perform aggregations
  – Exposes new operations
- **Created by:**
  – Running a map function that returns key-value pairs

```
hashtagsNum = hashtags.map(lambda word:
                           (word, 1))
```

```
→ [('#sea',1),('#sea', 1),('#sky', 1)]
```

  – Loading from Hadoop SequenceFiles
  – Parallelizing an existing collection of pairs

Transformations on a Pair RDD (1)

- **reduceByKey(func, [numTasks]):**
  - 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.
  - optional numTasks argument sets the number of tasks.
  - WordCount example:

```
rdd = sc.textFile("hdfs:/data/Gutenberg")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).
               reduceByKey(lambda x, y: x + y)
```

Transformations on a Pair RDD (2)

- **groupByKey([numTasks]) : called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.**
- **sortByKey([ascending], [numTasks]): When called on a dataset of (K, V) pairs, 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 also supported through leftOuterJoin and rightOuterJoin.**
- **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.**

Actions on Pair RDD
Examples on {(1,2),(3,4),(3,6)}

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

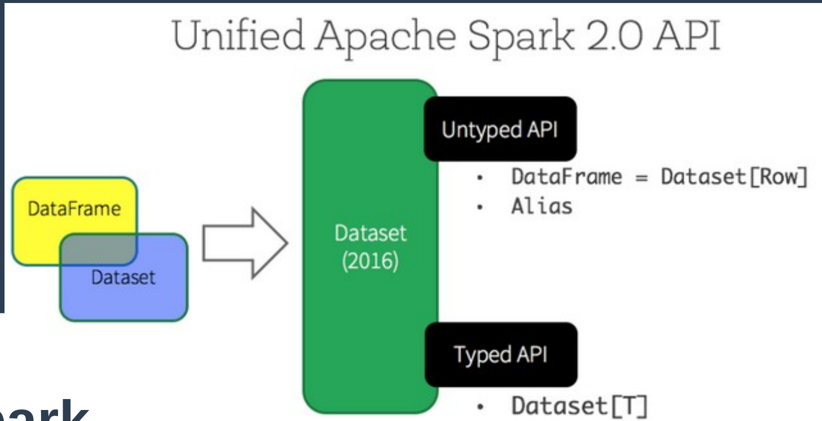
Loading and Saving Data

- **Spark can create datasets from any file system supported by Hadoop:**
  - Local file system (`file://`)
  - HDFS (`hdfs://`)
  - Amazon S3 (`s3n://`)
- **Formats**
  - Text files (with compression)
  - JSON
  - CSV
  - Hadoop formats (Sequence files...)

Life cycle of Spark Program

- Create some input RDDs from external data (or parallelize a collection) in your driver program
- Lazily transform the RDDs to define new RDDs using transformations like `filter`() or `map`()
- Ask Spark to `cache`() any intermediate RDDs that will need to be reused
- Launch actions as `count`() and `collect`() to kick off a parallel computation which is then optimized and executed by Spark

Others Spark API



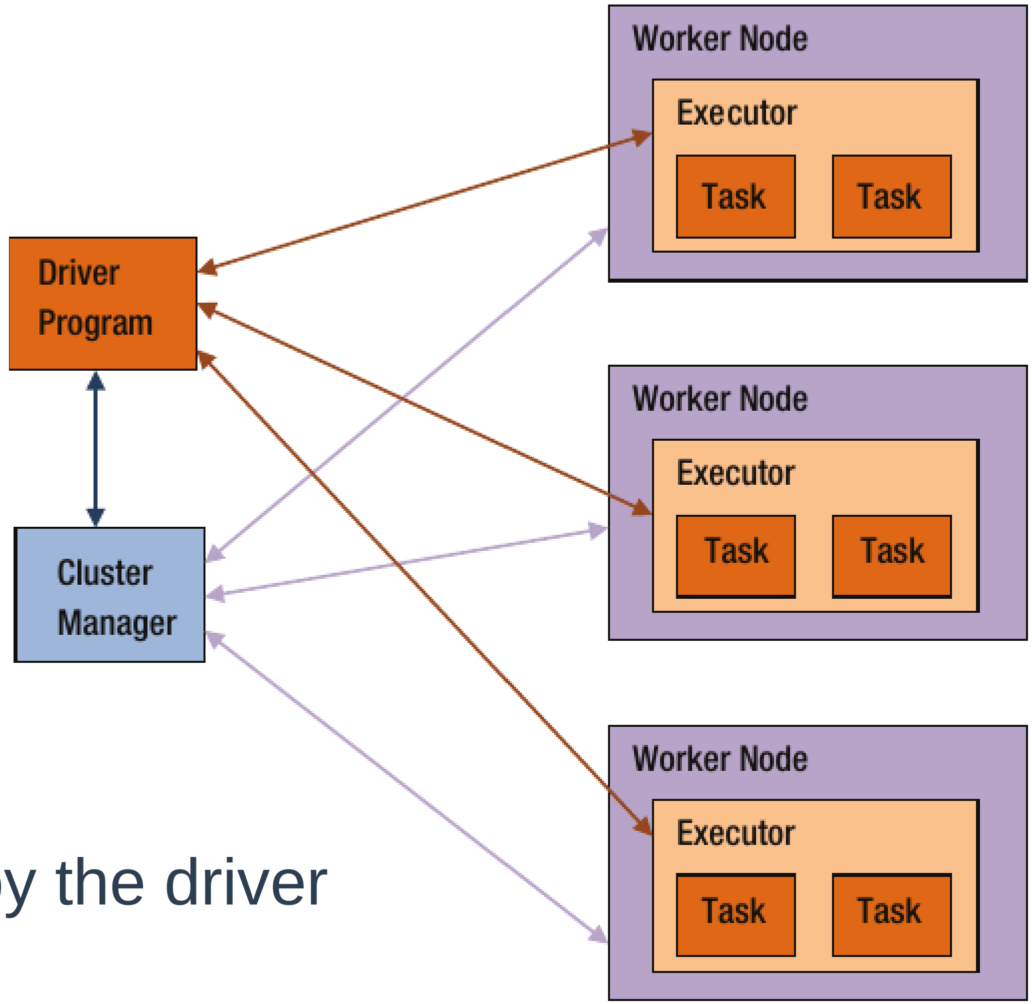
- **RDD was the primary user-facing API in Spark**
  - low-level control
  - unstructured data (media streams or streams of text)
  - functional programming constructs rather than domain specific expressions
- **DataFrames and Datasets**
  - Distributed collection of data ordered into named columns, like table in RDBMS
    - Impose a structure onto a distributed collection of data
    - Higher-level abstraction, domain specific language API
    - Optimization and performance benefits
  - Built on top of RDDs

Outlines

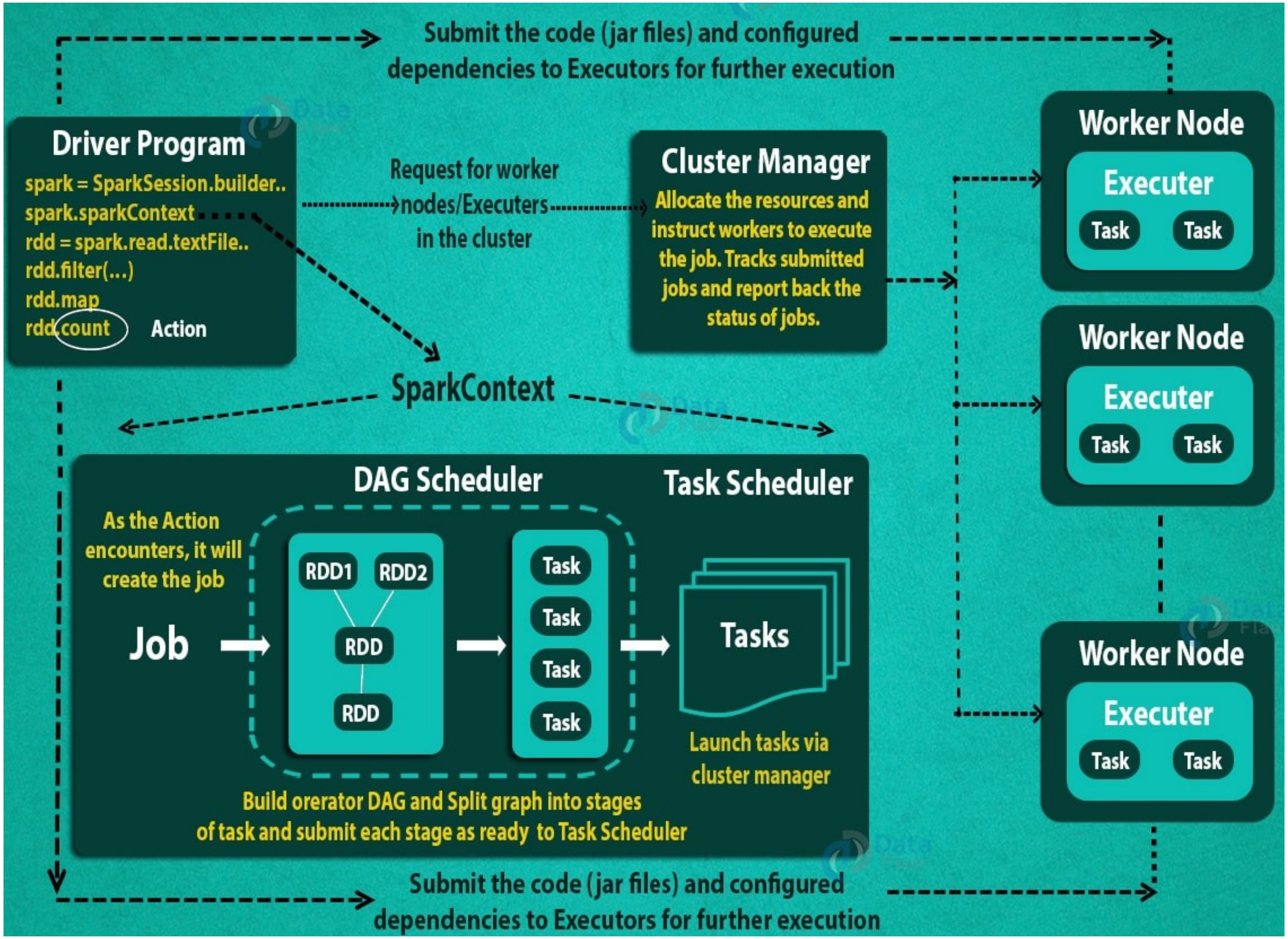
1. Motivations
2. Programming Model
3. **Runtime Environment**

Spark Application Architecture

- **Driver program**
  - Application using Spark
- **Cluster manager**
  - Manages the computing resources
- **Worker nodes**
  - Runs the distributed processes of the application
- **Executors**
  - Executes the tasks submitted by the driver
- **Tasks**
  - Units of work per partition of data



Application Execution on a Cluster



Python Spark

- **`pyspark`: Python ‘s interface to Spark**
  - Interactive shell
    - run by the `pyspark` command
  - A `SparkContext` is created when shell launches
    - Held in variable `sc`
- **Self-contained applications**
  - Executed with `spark-submit`, e.g.:
    - `spark-submit --master local[2] wordcount.py`
      runs locally with 2 threads
    - `spark-submit --master yarn --num-executor=10 --deploy-mode client wordcount.py`
      runs on a YARN cluster with 10 worker nodes, leaving the driver on the frontal (`cluster` mode: driver on a worker node)

Summary

- 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.
- Ease of Use: Write applications quickly in Java, Scala, Python, R.
- Generality: Combine SQL, streaming, and complex analytics.
- Runs Everywhere: Spark runs on Hadoop YARN, Mesos, standalone, or in the cloud.

http://spark.apache.org/