



Cloud based Data Analytics

Lecture 3



Content

- In this lecture, we will discuss:
- Overview of Spark
- Spark concepts
- Spark operations
- Job execution
- Fundamentals of Scala/Pyspark
- Hadoop vs Spark
- Spark Installation



Introduction to Spark

Introduction

- In this part, we will discuss history of spark, ‘framework of spark’ Resilient Distributed Datasets (RDDs) and Spark execution

History of Spark

The history of Spark is explained below:

Started at UC
Berkeley AMPLab
by Matei Zaharia

2009



BSD

2010

Open sourced
under a BSD
license

The project was given
to the Apache
Software Foundation
and the license was
changed to Apache 2.0

2013



The Apache Software
Foundation



2014

Became an Apache Top-
Level Project. Used by
Databricks to set a world
record in large-scale
sorting in Nov

Exists as a next
generation real-time
and batch processing
framework

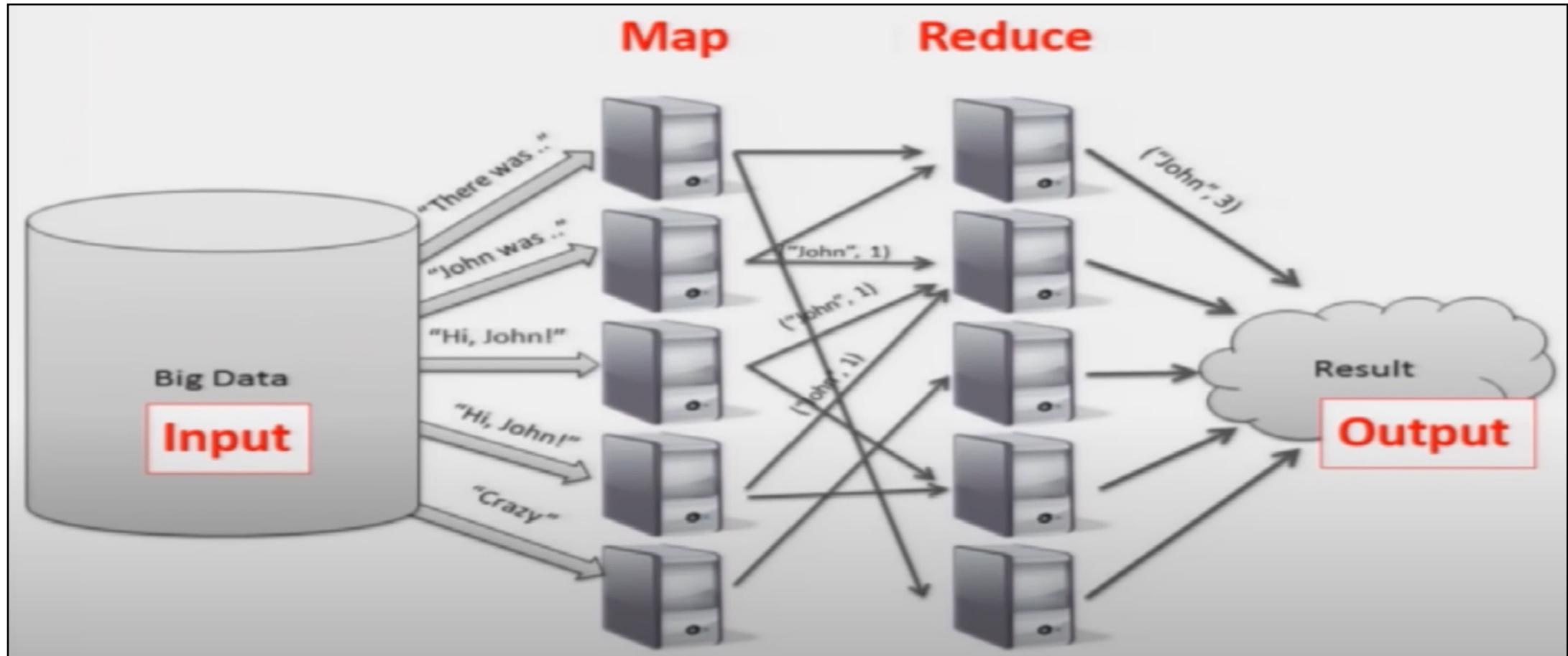
Present



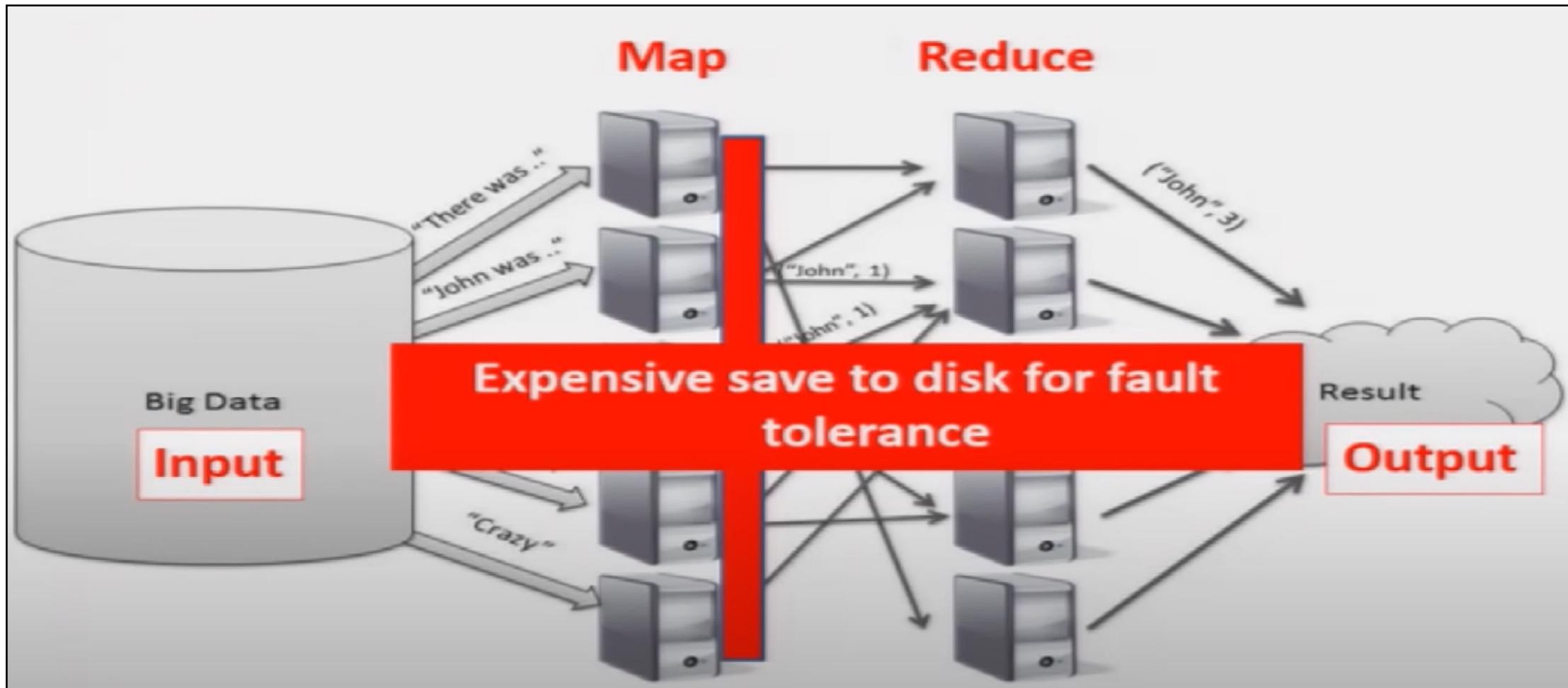
Need of Spark

- Spark is a big data analytics framework that was originally developed at the University of California, Berkeley's AMPLab, in 2009. Since then, it has gained a lot of attraction both in academia and in industry.
- It is another system for big data analytics
- Isn't MapReduce enough?
 - Simplifies batch processing on large commodity clusters

Need of Spark



Need of Spark



Need of Spark

- MapReduce can be expensive for some applications e.g.,
 - Iterative
 - Interactive
- Lacks efficient data sharing
- Specialized frameworks did evolve for different programming models
 - Bulk Synchronous Processing (Pregel)
 - Iterative MapReduce (Hadoop)

Solution: Resilient Distributed Datasets (RDDs)



Resilient Distributed Datasets (RDDs)



Immutable, partitioned collection of records

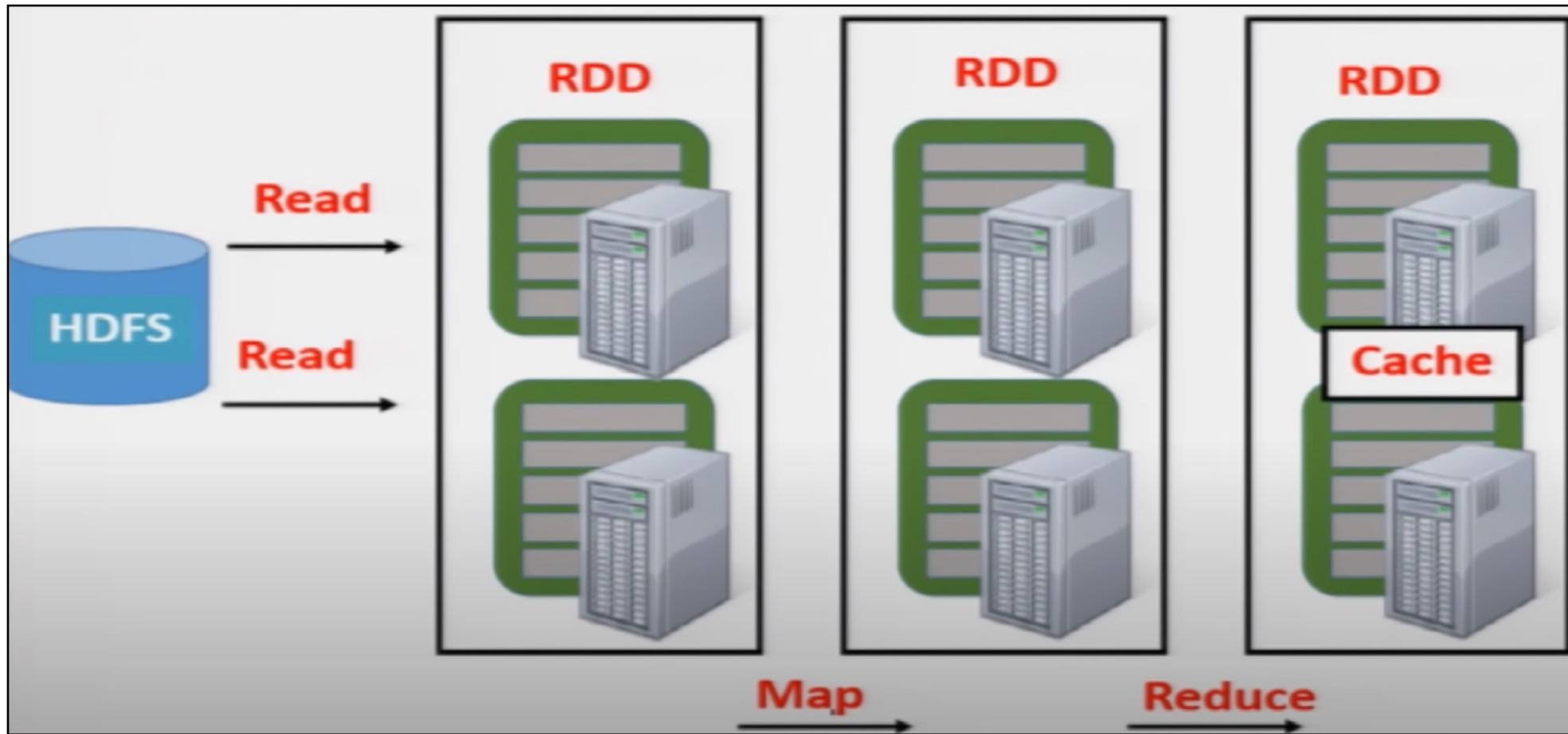


Built through coarse grained transformations (map, join....)



Can be cached for efficient reuse

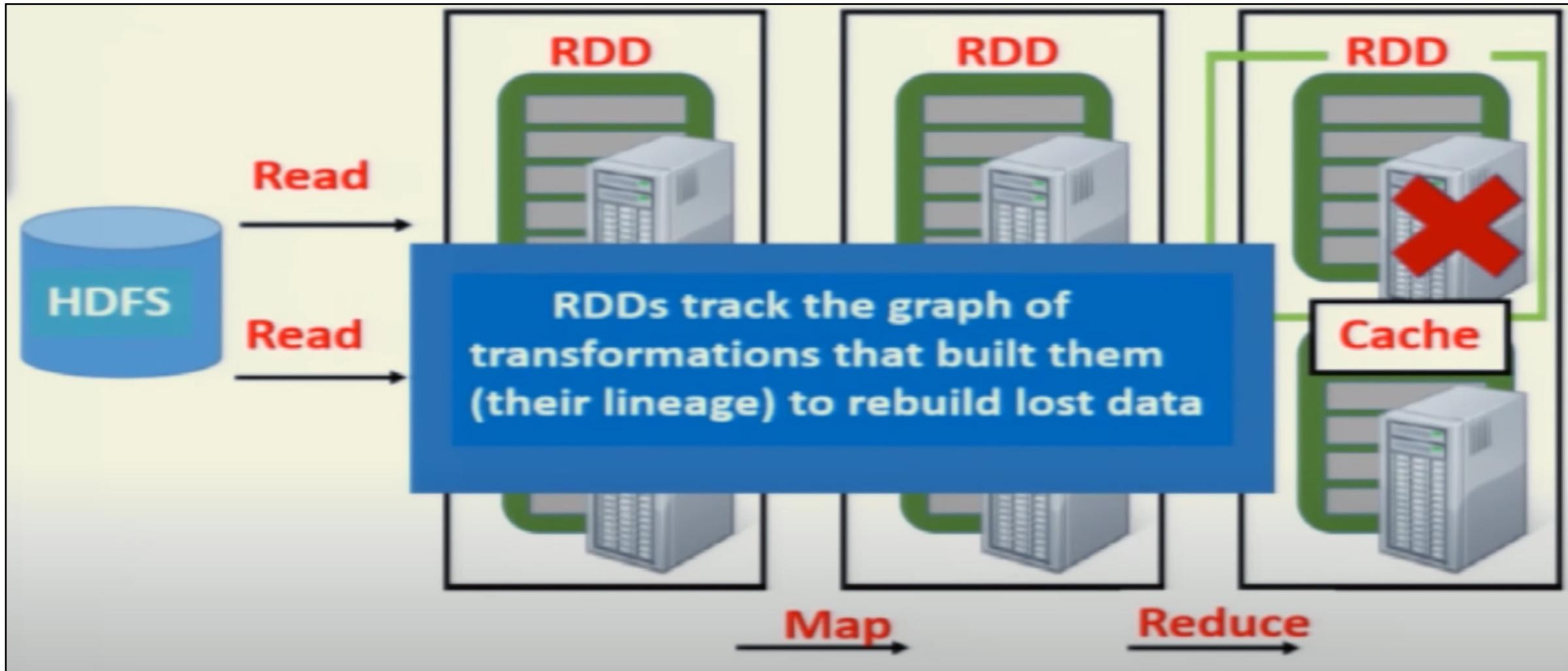
Need of Spark



Solution: Resilient Distributed Datasets (RDDs)

- Resilient Distributed Datasets (RDDs)
 - Immutable, partitioned collection of records
 - Built through coarse grained transformations (map, join, ...)
- Fault Recovery
 - Lineage!
 - Log coarse grained operation applied to a partitioned dataset
 - Simply recompute the lost partition if failure occurs!
 - No cost if no failure





Solution: Resilient Distributed Datasets (RDDs)

- Resilient Distributed Datasets (RDDs)
 - Immutable, partitioned collection of records
 - Built through coarse grained transformations (map, join, ...)
- Fault Recovery
 - Lineage!
 - Log coarse grained operation applied to a partitioned dataset
 - Simply recompute the lost partition if failure occurs!
 - No cost if no failure

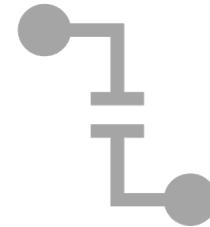
What you can do with Spark?



Resilient Distribute Datasets (RDDs) Operations

Transformations e.g., filter, join,
map, group-by ...

Actions e.g., count, print



Control

Partitioning: Spark also gives you
control over how you can partition
your RDDs.

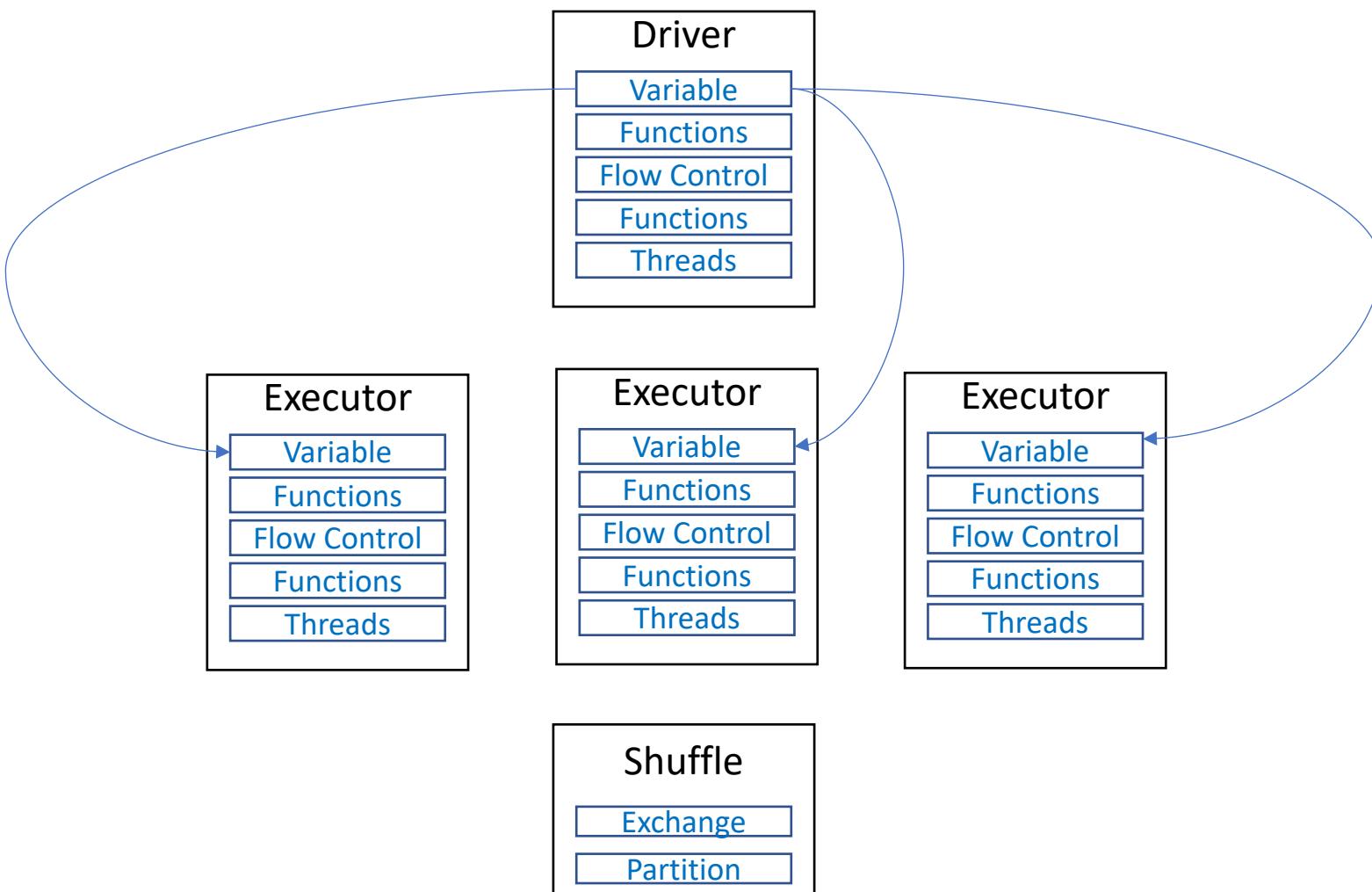
Persistence: Allows you to choose
whether you want to persist RDD
onto disk or not.

Spark Applications

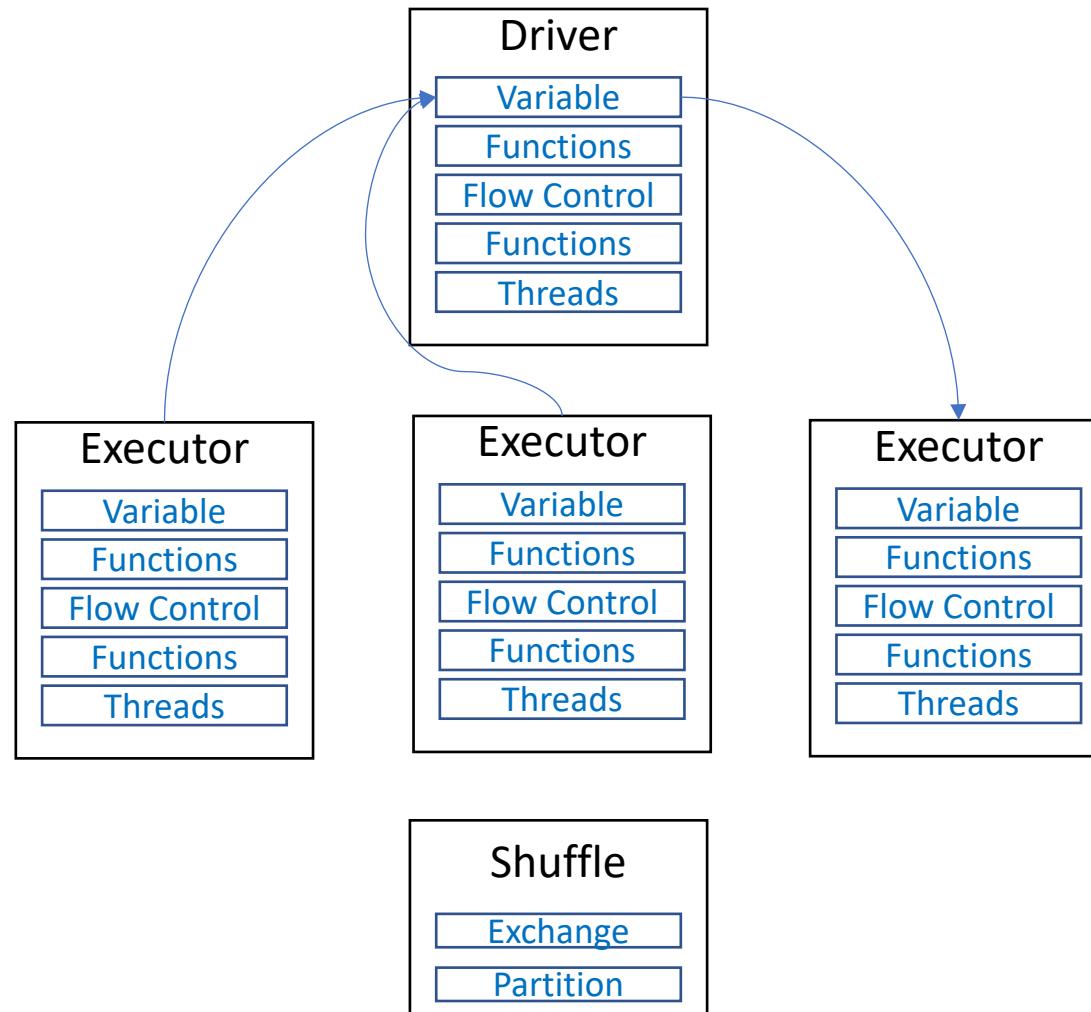
- Twitter Spam Classification
- EM Algorithm for traffic congestion
- K-Means Clustering
- Alternating Least Squares Matrix Factorization
- In-Memory OLAP aggregation on Hive data
- SQL on Spark

Spark Execution

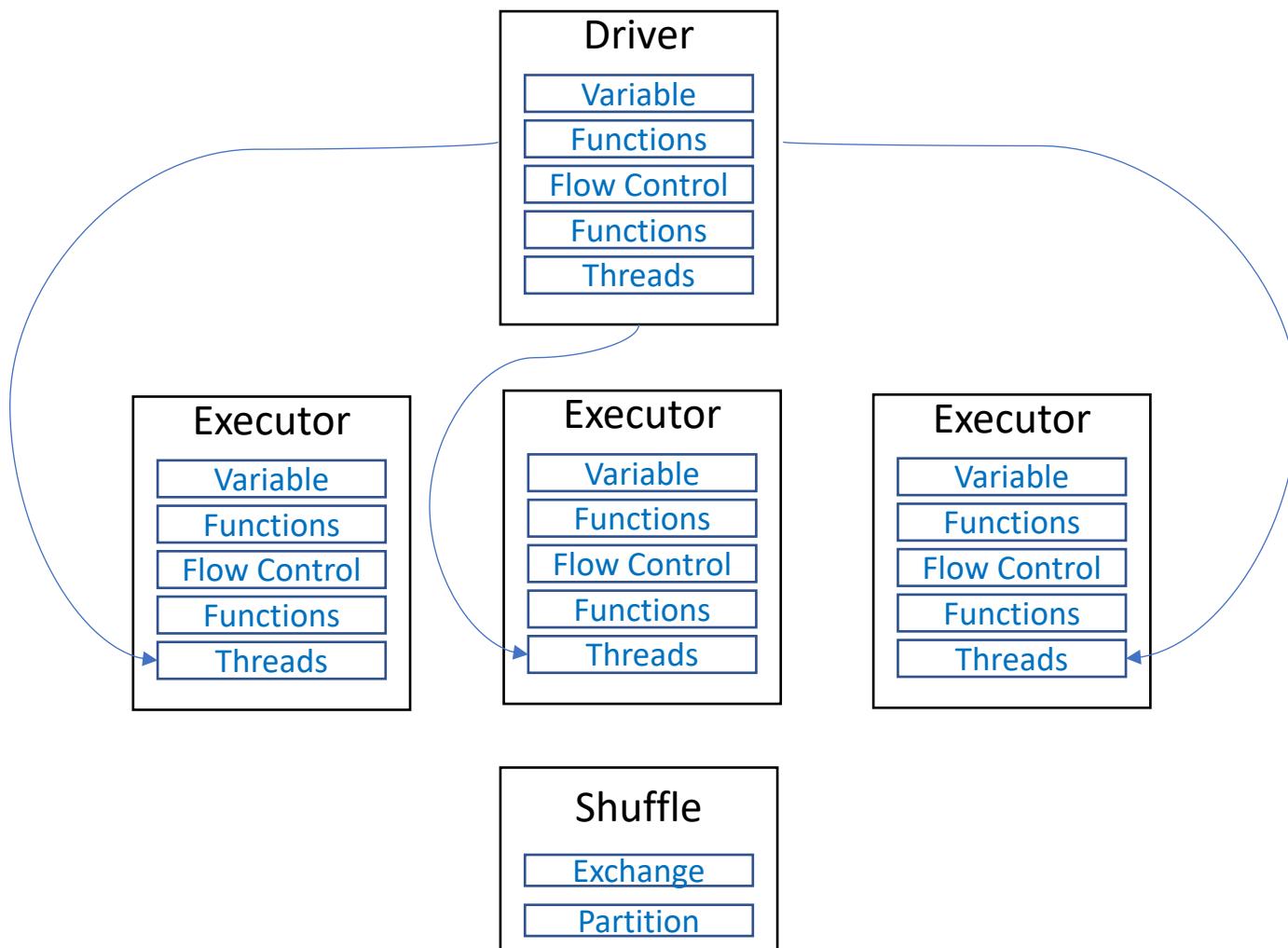
Spark Execution – Distributed Programming (Broadcast)



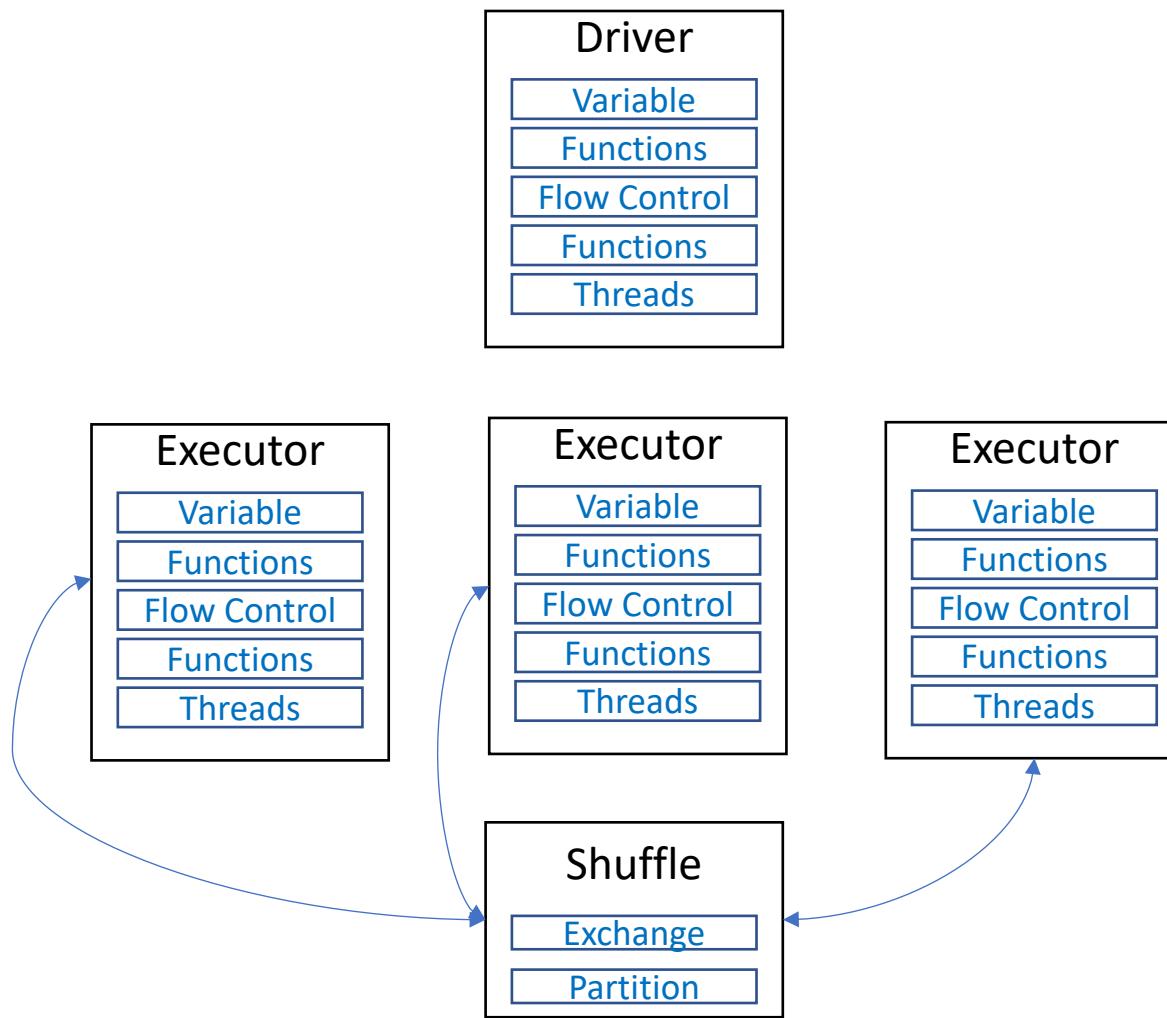
Distributed Programming (Take)



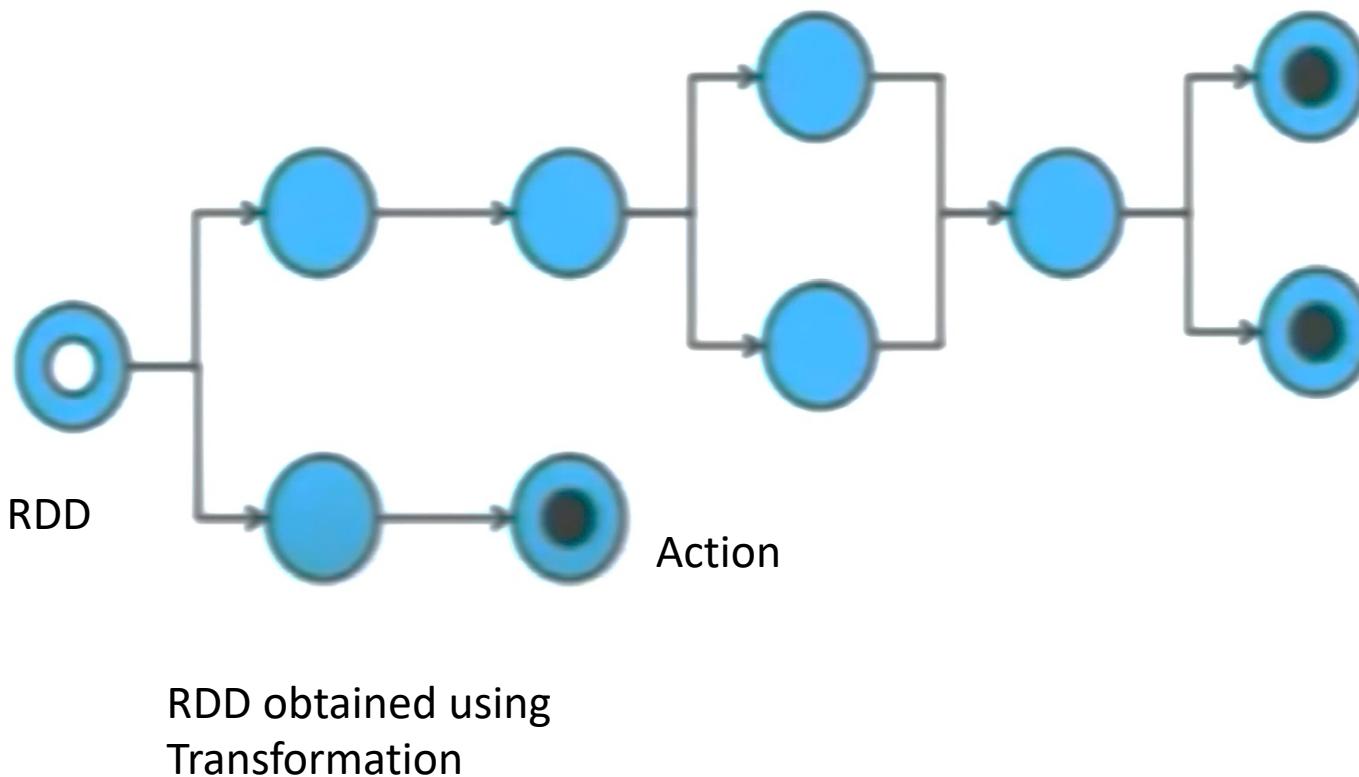
Distributed Programming (DAG Action)



Distributed Programming (Shuffle)



DAG (Directed Acyclic Graph)



DAG (Directed Acyclic Graph)

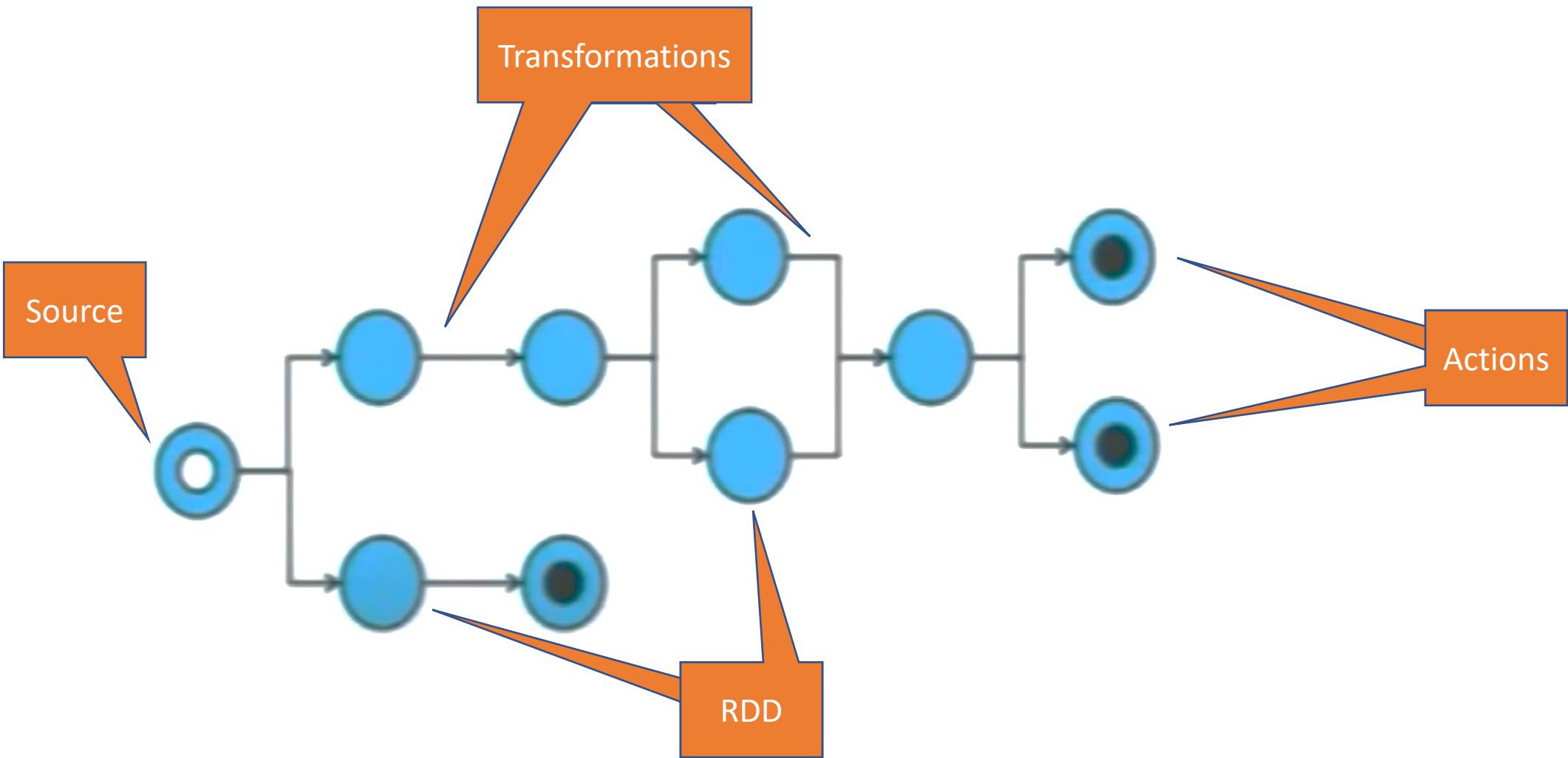
Action

- Count
- Take
- Foreach

Transformation

- Map
- ReduceByKey
- GroupByKey
- JoinByKey

DAG (Directed Acyclic Graph)



Spark Implementation

Spark Ideas

- Not limited to map-reduce model
- Facilitate system memory
 - Avoid saving intermediate results to disk
 - Cache data for repetitive queries (e.g., for machine learning)
- Compatible with Hadoop

RDD Abstraction



Resilient Distributed Datasets



Partitioned collection of records



Spread across the cluster



Read-Only



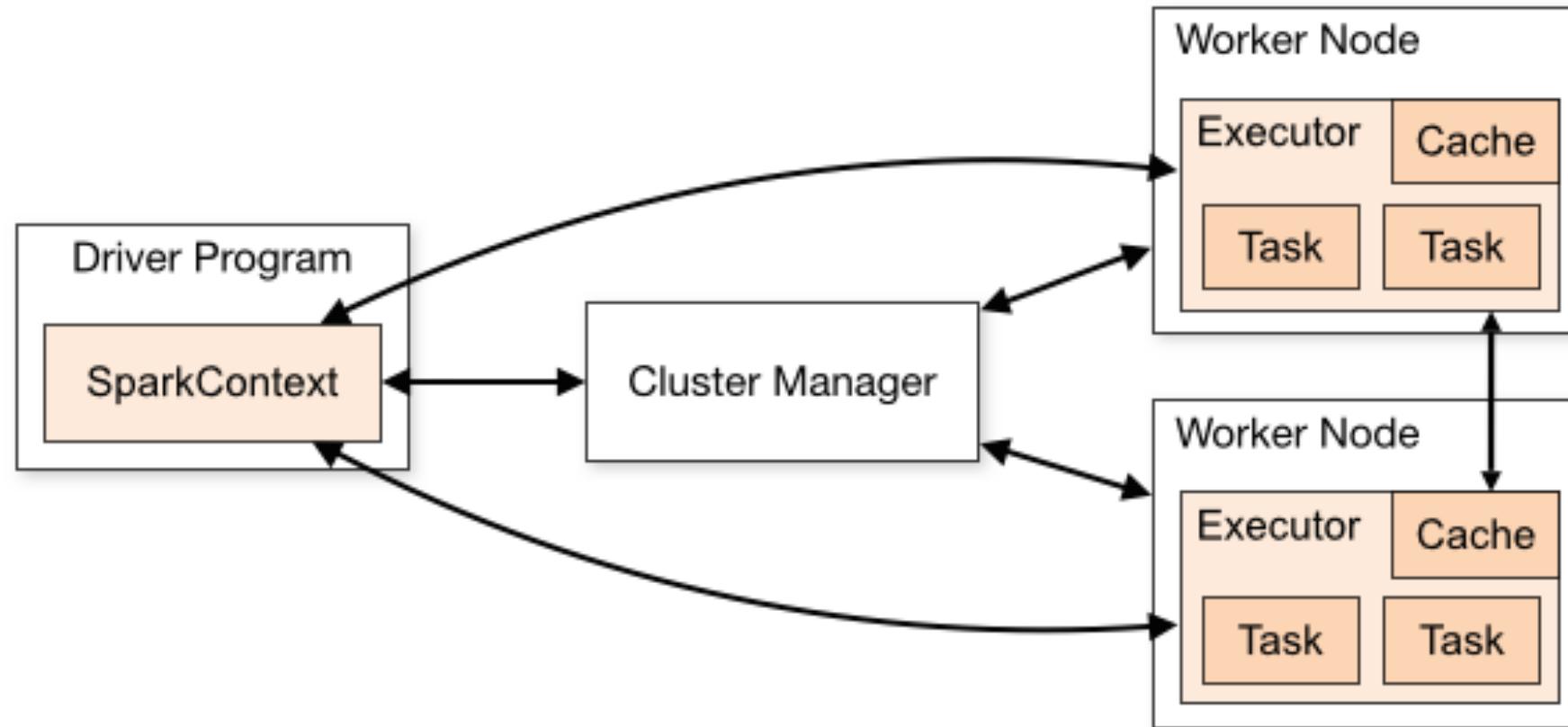
Caching dataset in
memory

Different storage levels
available
Fallback to disk possible

RDD Operations

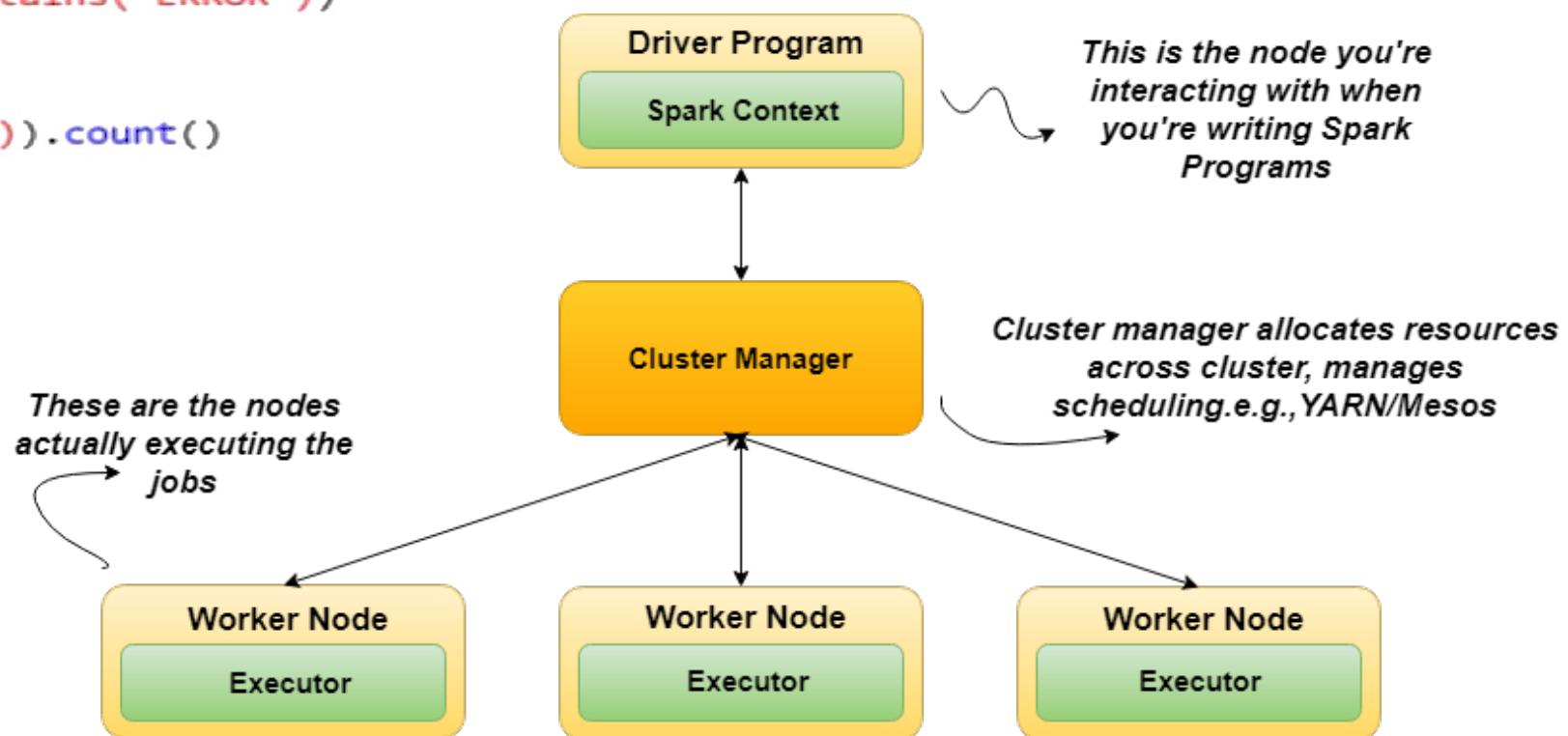
- Transformations to build RDDs through deterministic operations on other RDDs
 - Transformations include map, filter, join
 - Lazy operation
- Actions to return value or export data
 - Actions include count, collect, save
 - Triggers execution

Spark Components

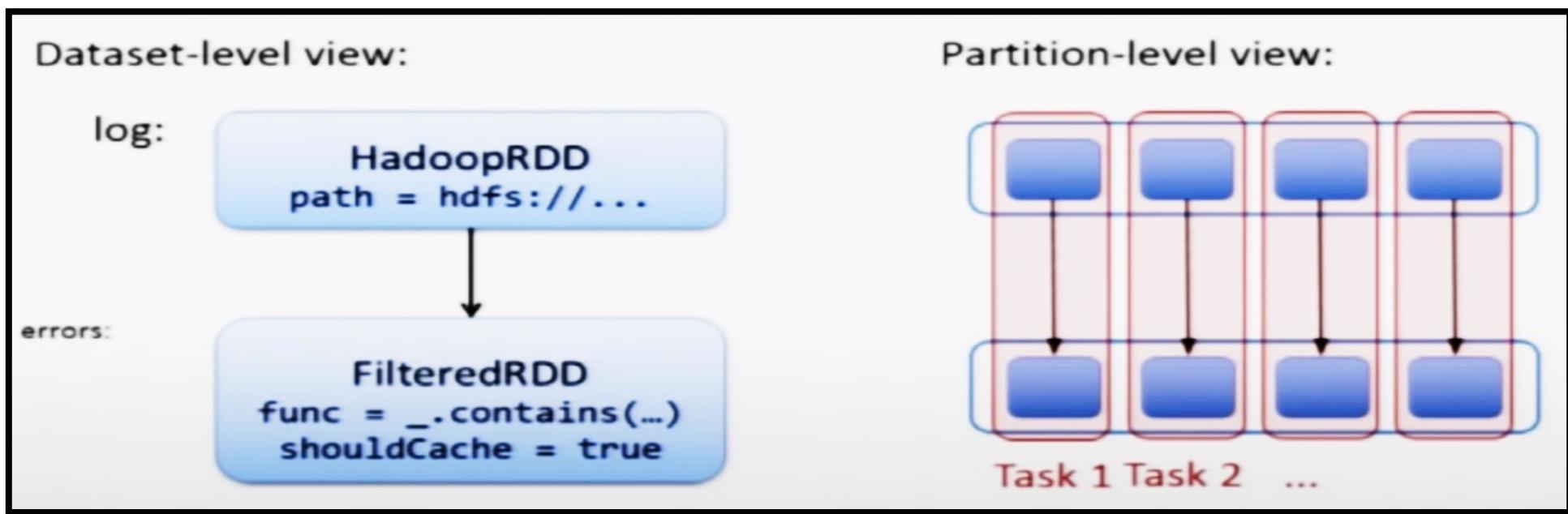


Job Example

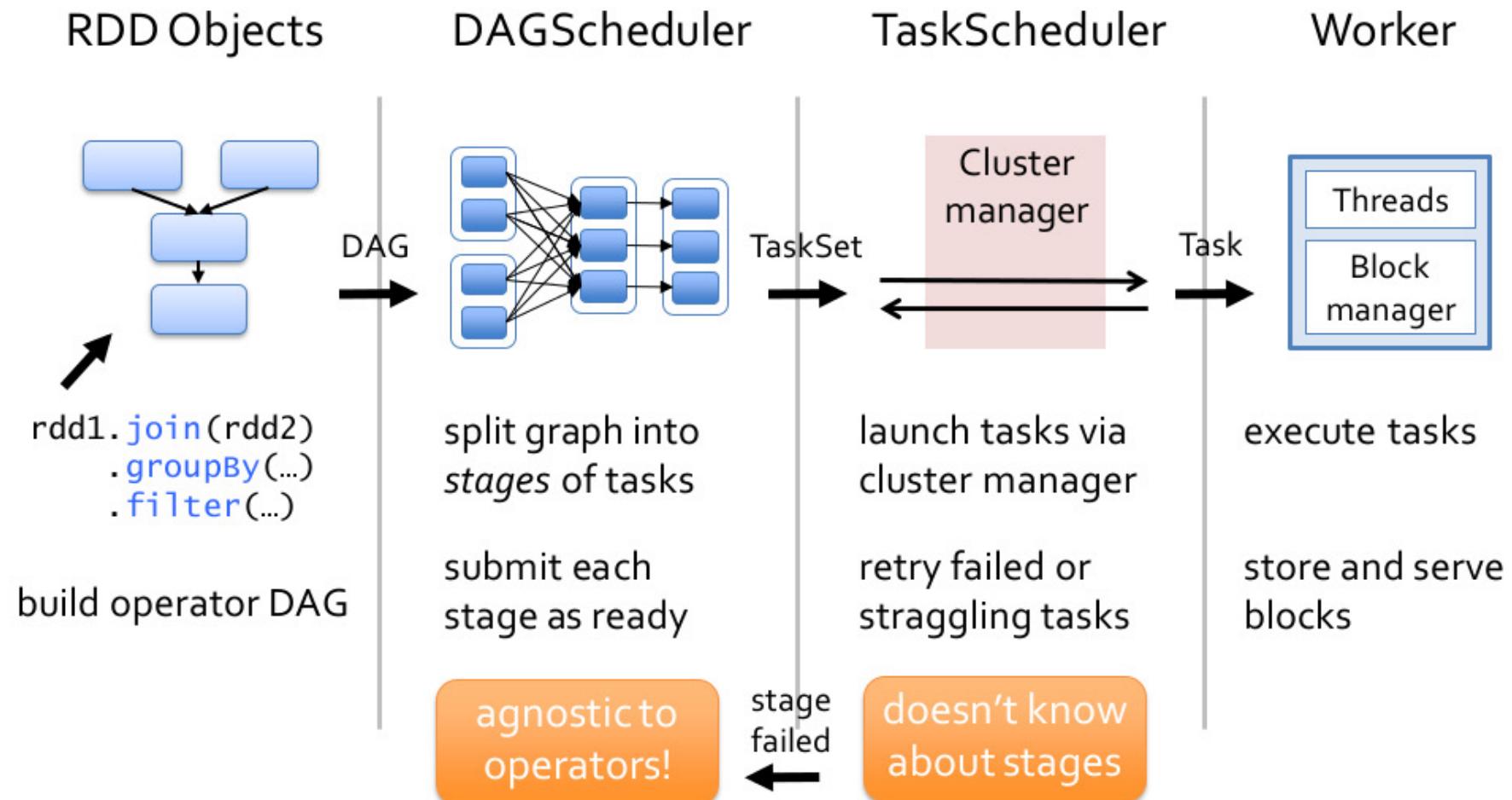
```
val log = sc.textFile("hdfs://...")  
val errors = file.filter(_.contains("ERROR"))  
errors.cache()  
  
errors.filter(_.contains("I/O")).count()
```



RDD partition-level view



Job Scheduling



Summary

- Concept not limited to single pass map-reduce
- Avoid storing intermediate results on disk or HDFS
- Speedup computations when reusing datasets

What is Spark

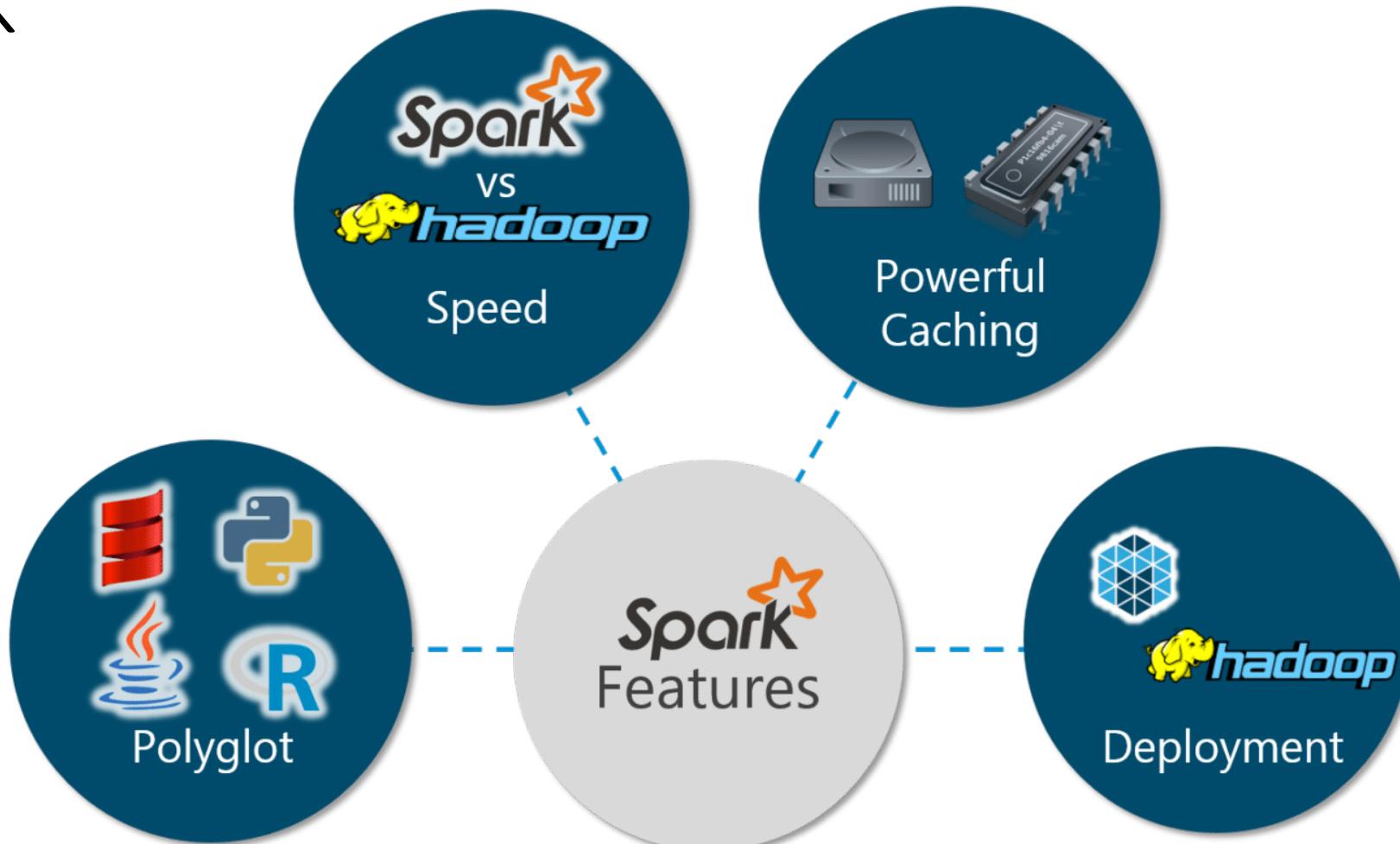
What is Spark?

- Fast, expressive cluster computing system compatible with Apache Hadoop
 - Works with any Hadoop-supported storage system (HDFS, S3, SequenceFile and Avro etc.)
- Improves **efficiency** through:
 - In-memory computing primitives
 - General computation graphs **Up to 100x faster**
- Improves **usability** through:
 - Rich APIs in Java, Python, R and Scala
 - Interactive shell **Often 2-10x less code**

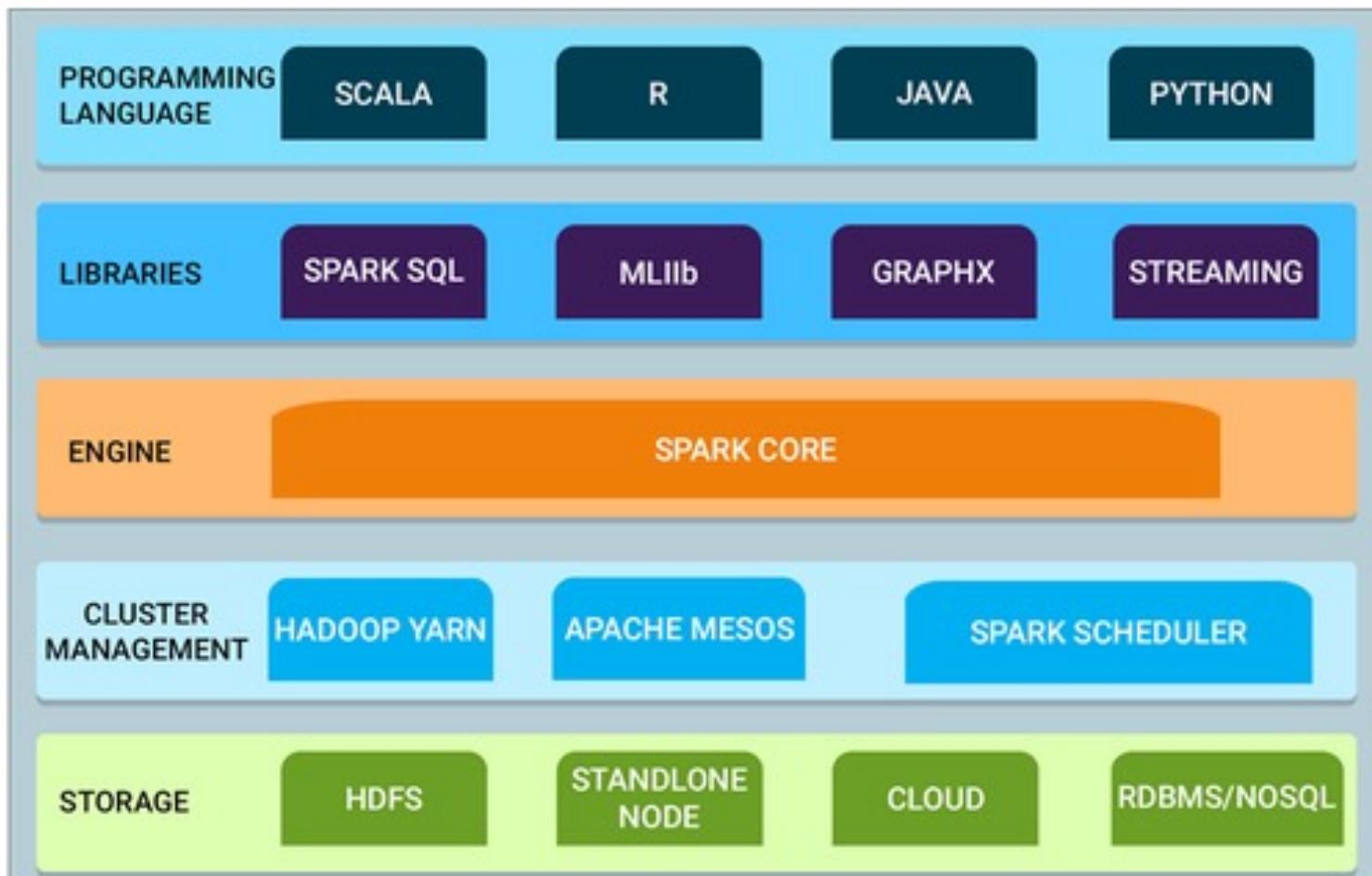
How to Run it

- Local multicore: just a library in your program
- Private cluster: Mesos, YARN, Standalone Mode
- EC2: scripts for launching a Spark cluster
- Databricks: run spark in unified Data Analytics Platform - One cloud platform for massive scale data engineering and collaborative data science
- Azure Synapse Analytics: is a limitless analytics service that brings together enterprise data warehousing and Big Data analytics

Spark

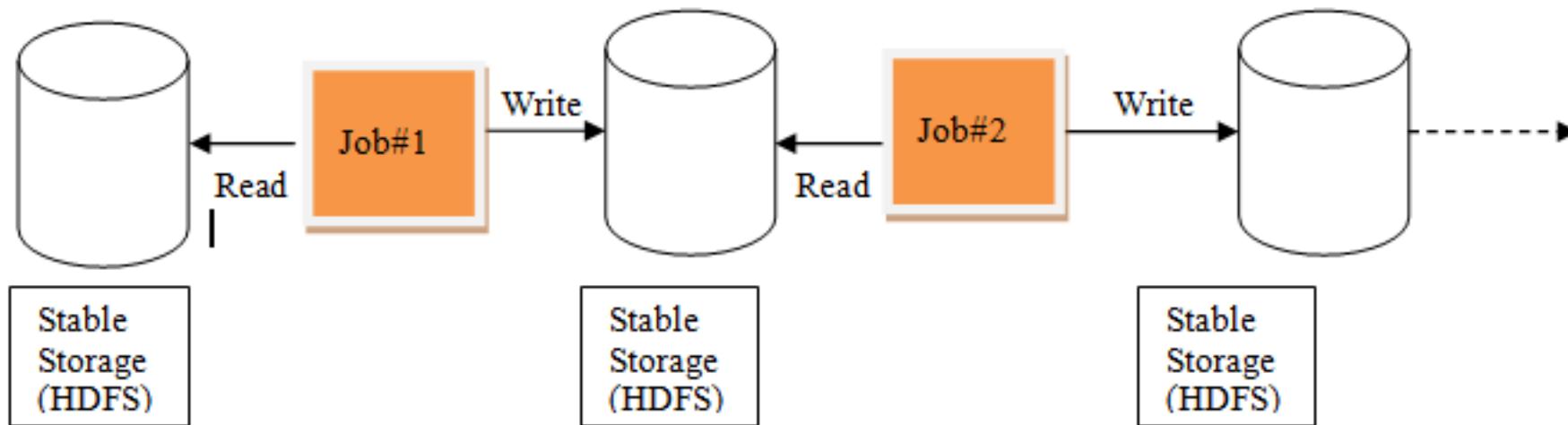


Spark Components



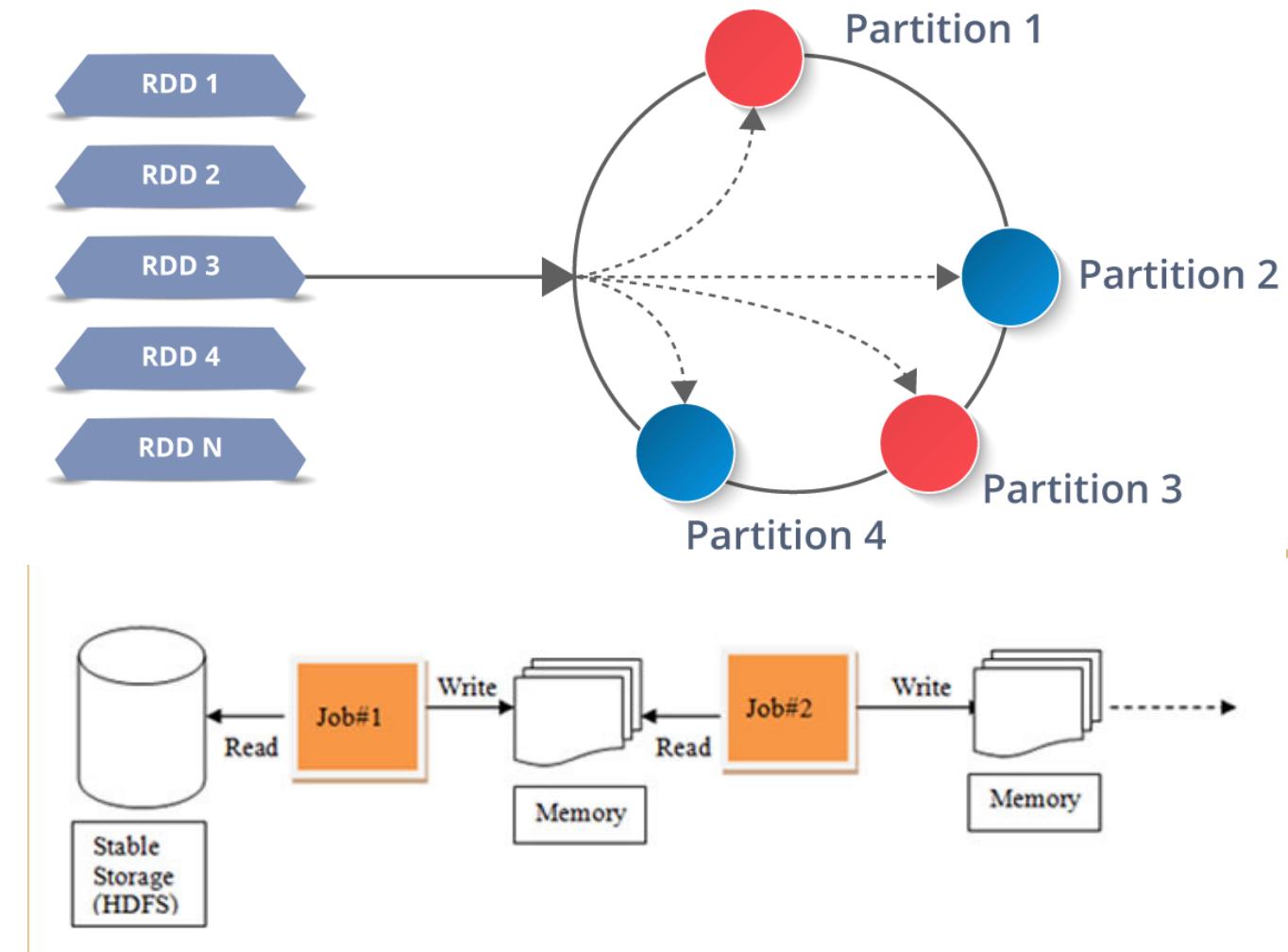
Why RDD?

- Iterative Process
- Reusing Data
- Sharing Data

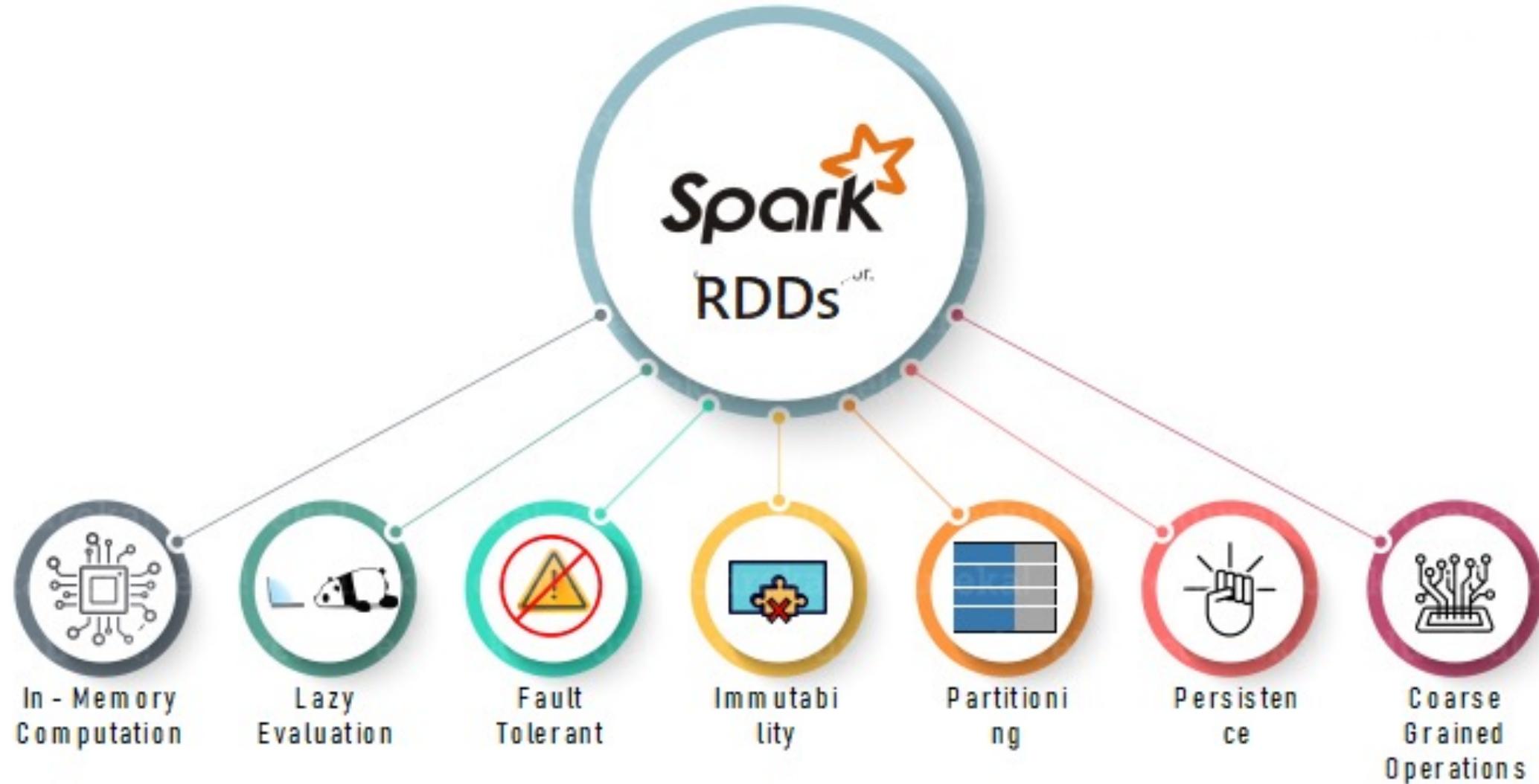


Spark RDD

- Resilient Distributed Datasets (**RDD**) is a fundamental data structure of Spark.
- At the core, an **RDD** is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.
- **RDDs** can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Formally, an **RDD** is a read-only, partitioned collection of records.



Features of the RDD

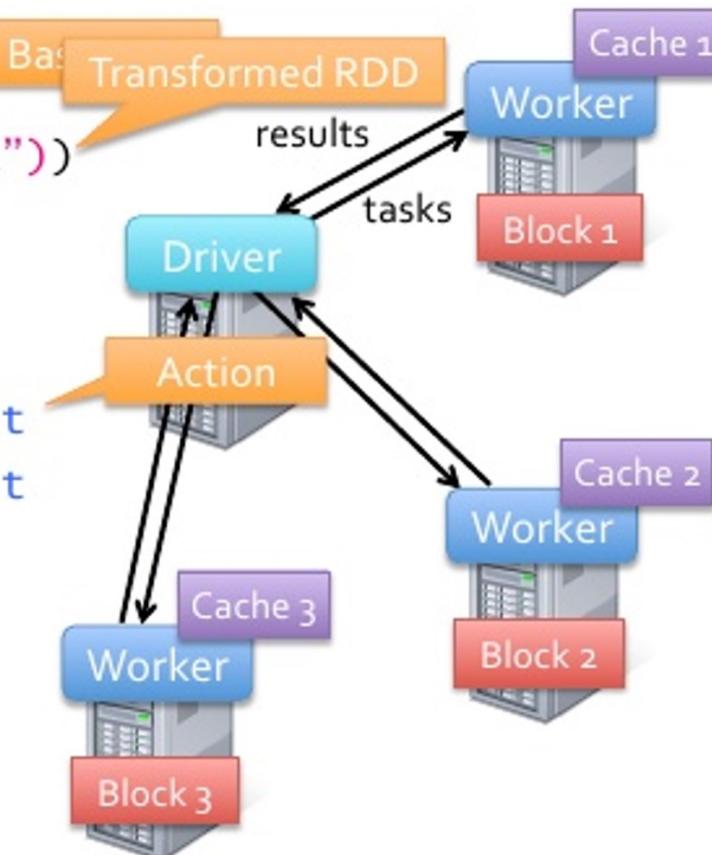


Example: Mining Console Logs

- Load error messages from a log memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(_.startswith("ERROR"))  
messages = errors.map(_.split("\t")(2))  
cachedMsgs = messages.cache()  
  
cachedMsgs.filter(_.contains("foo")).count  
cachedMsgs.filter(_.contains("bar")).count  
...
```

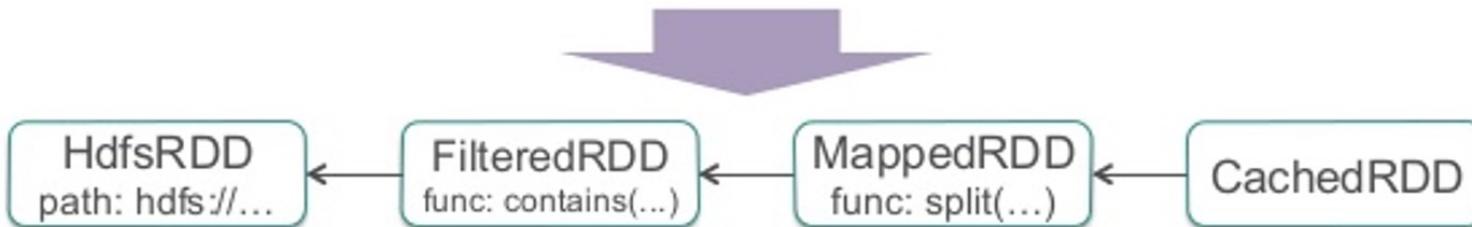
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)



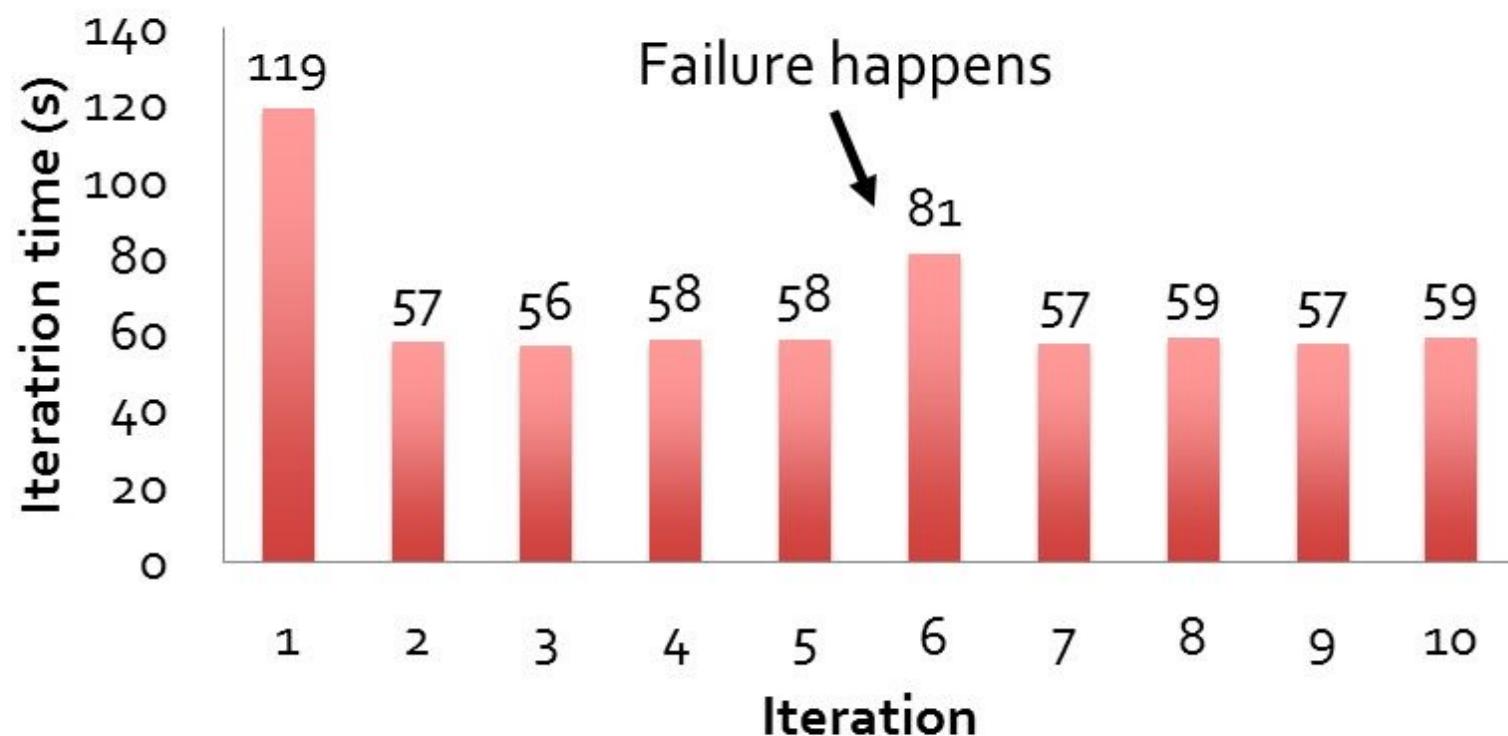
RDD Fault Tolerance

- RDDs track the transformations used to build them (their lineage) to recompute lost data.

```
cachedMsgs = textFile(...).filter(_.contains("error"))
               .map(_.split('\t')(2))
               .cache()
```

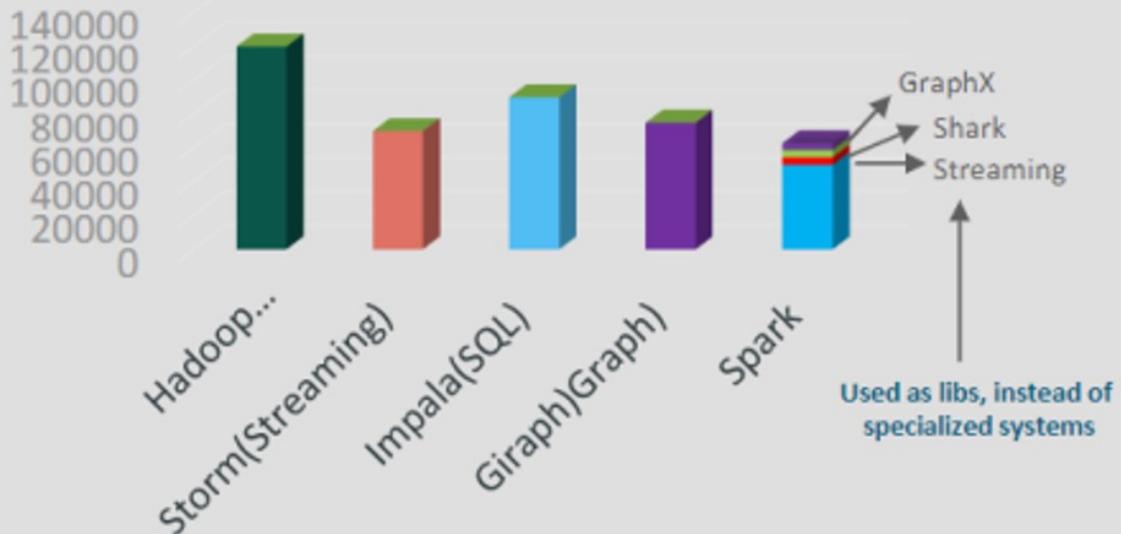


Fault Recovery Results



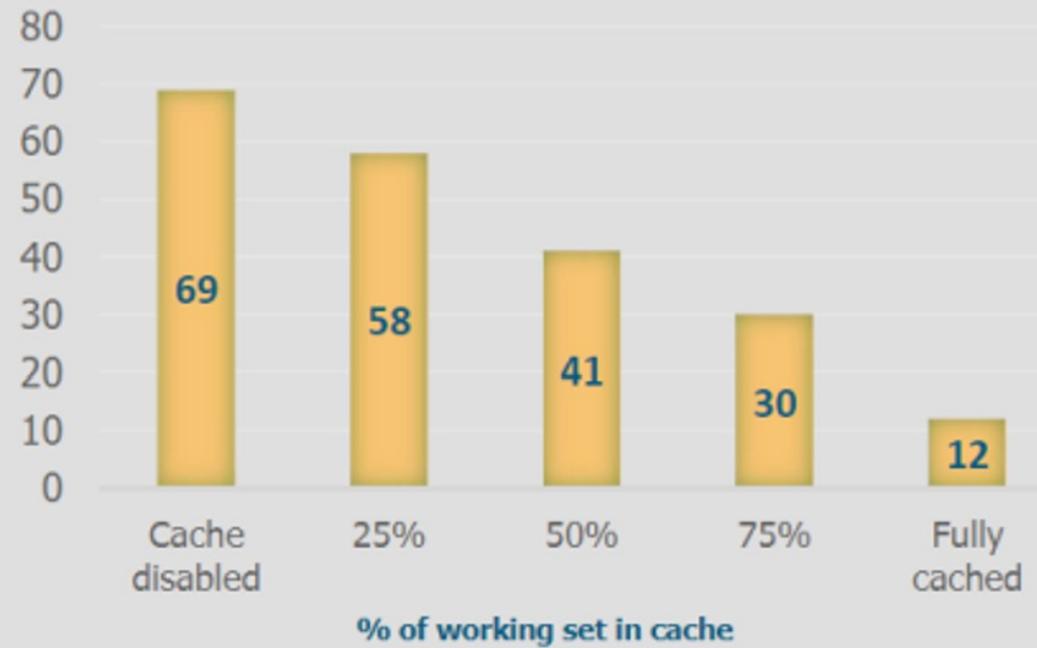
Spark

Code Size



The State of Spark, and where we're going next

Behaviour with Less RAM



Spark uses Cache for In-Memory storage.
When we disable the use of cache there is 69% load on the processor,
but when we use cache fully there is only 12% load on Processor

Summary

- Goal: Work with distributed collections as you would work with local ones
- Concept: Resilient Distributed Datasets (RDDs)
 - Immutable collection of objects spread across a cluster
 - Built through parallel transformations (map, filter etc.)
 - Lazy operations to build RDDs from other RDDs
 - Automatically rebuilt on failure
 - Controllable persistence (e.g., caching in RAM)
 - Actions (e.g., count, collect, save)
 - Return a result or write it to storage

Working with Spark

Available APIs

- Spark originally written in Scala, which allows concise function syntax and interactive use
- You can write in Python, Scala, Java and R
- Interactive interpreter: Scala and Python only
- Standalone applications: any
- Performance: Java and Scala are faster because of static typing

Which language should I use?

- Standalone programs can be written in any, but console is only Python and Scala
- Python Developers: can stay with Python for both
- Java Developers: consider using Scala for console (to learn API)

- Performance: Java/Scala will be faster (statistically typed) but Python code can do well for numerical work with NumPy

Learning Spark

- Easiest way: Spark Interpreter (Pyspark or spark-shell)
 - Special Python and Scala consoles for cluster use
- Runs in local mode on 1 thread by default but can control with Master environment var:

```
MASTER=local      ./spark-shell          # local, 1 thread
MASTER=local[2]   ./spark-shell          # local, 2 threads
MASTER=spark://host:port ./spark-shell  # Spark standalone
cluster
```

Hand on - interpreter

- Run Scala spark interpreter
- \$ spark-shell
- Or Python Interpreter
- \$ pyspark

Spark-Shell and PySpark

```
[snehadhanda@snehas-MacBook-Air ~ % spark-shell
21/01/08 18:22:31 WARN Utils: Your hostname, Snehas-MacBook-Air.local resolves to a lo
opback address: 127.0.0.1; using 192.168.2.12 instead (on interface en0)
21/01/08 18:22:31 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another addres
s
21/01/08 18:22:32 WARN NativeCodeLoader: Unable to load native-hadoop library for your
 platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newL
evel).
21/01/08 18:22:41 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attemptin
g port 4041.
21/01/08 18:22:41 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attemptin
g port 4042.
Spark context Web UI available at http://192.168.2.12:4042
Spark context available as 'sc' (master = local[*], app id = local-1610148162229).
Spark session available as 'spark'.
Welcome to
```

version 3.0.1

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_261)
Type in expressions to have them evaluated.
Type :help for more information.

scala>

Spark- Scala

PySpark

```
[snehadhanda@snehas-MacBook-Air ~ % pyspark
Python 3.9.0 (v3.9.0:9cf6752276, Oct 5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
21/01/23 12:53:17 WARN Utils: Your hostname, Snehas-MacBook-Air.local resolves to
a loopback address: 127.0.0.1; using 192.168.2.12 instead (on interface en0)
21/01/23 12:53:17 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another ad-
dress
21/01/23 12:53:19 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(
newLevel).
Welcome to
```

version 3.0.1

```
Using Python version 3.9.0 (v3.9.0:9cf6752276, Oct 5 2020 11:29:23)
SparkSession available as 'spark'.
>>> 
```

First Stop: SparkContext



Main entry to Spark
functionality



Created for you in Spark shells
as variable sc



In standalone programs, you
would make your own

Ways to create RDD

- RDD is used for efficient work by a developer, it is a read-only partitioned collection of records.
 - Using parallelized collection
 - From existing Apache Spark RDD &
 - From external datasets.

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt,
conf)
```

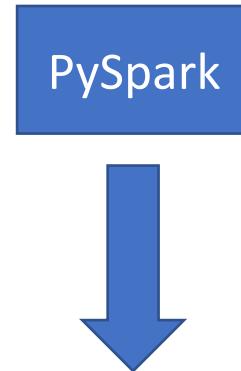
Spark with Scala and Python

```
[scala> sc.parallelize(0 to 100).collect()
res0: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
6, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
6, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
6, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
6, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
6, 97, 98, 99, 100)
```

scala>



Spark- Scala



```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
```

version 3.0.1

Using Python version 3.9.0 (v3.9.0:9cf6752276, Oct 5 2020 11:29:23)
SparkSession available as 'spark'.

```
[>>> sc.parallelize(range(0, 100)).collect()
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 2  
2, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 4  
2, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 6  
2, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 8  
2, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]  
>>>  
```

Spark-Getting Data from HDFS

```
[scala> val text=sc.textFile("hdfs://localhost:9000/Dataset/wc.txt")
text: org.apache.spark.rdd.RDD[String] = hdfs://localhost:9000/Dataset/wc.txt MapPartitionsRDD[1] at textFile at <console>:24

[scala> text.collect;
res0: Array[String] = Array(Deer Bear River, Car Car River, Deer Car Bear)

[scala>

scala> █
```

Spark-Getting Data from local system

```
scala>

scala> val book = sc.textFile("/Users/snehadhanda/work/Book")
book: org.apache.spark.rdd.RDD[String] = /Users/snehadhanda/work/Book MapPartitionsRDD[10] at
textFile at <console>:24
[

scala> book.collect
res8: Array[String] = Array(userId,jobId,Clicks, 1,31,3, 1,1029,66, 1,1061,93, 1,1129,0, 1,117
[2,77, 1,1263,30, 1,1287,38, 1,1293,44, 1,1339,17, 1,1343,76, 1,1371,12, 1,1405,1, 1,1953,11, 1]
,2105,88, 1,2150,20, 1,2193,61, 1,2294,46, 1,2455,18, 1,2968,61, 1,3671,30, 2,10,83, 2,17,19,
2,39,68, 2,47,99, 2,50,40, 2,52,98, 2,62,46, 2,110,13, 2,144,94, 2,150,28, 2,153,34, 2,161,39,
2,165,85, 2,168,33, 2,185,27, 2,186,56, 2,208,30, 2,222,65, 2,223,73, 2,225,28, 2,235,9, 2,24
8,17, 2,253,64, 2,261,34, 2,265,70, 2,266,54, 2,272,51, 2,273,26, 2,292,26, 2,296,77, 2,300,69
[, 2,314,78, 2,317,57, 2,319,36, 2,339,74, 2,349,47, 2,350,36, 2,356,55, 2,357,54, 2,364,67, 2,]
367,74, 2,370,8, 2,371,72, 2,372,20, 2,377,75, 2,382,77, 2,405,58, 2,410,56, 2,454,69, 2,457,4
6, 2,468,71, 2,474,...]

scala> book.partitions.length
res9: Int = 14
[

scala> ]
```

[snehadhanda@Snehas-MacBook-Air Book % ls

BX-Book-Ratings.csv	Users.csv	jobs.txt
BX-Books.csv	books_test.csv	movies.csv
BX-Books.txt	job_clicks.csv	movies.txt
Books.csv	job_clicks.txt	student.csv
Ratings.csv	jobs.csv	

Localhost: 4040 – Spark-shell

Spark 3.0.1 Jobs Stages Storage Environment Executors Spark shell application UI

Spark Jobs (?)

User: snehadhanda
Total Uptime: 2.1 min
Scheduling Mode: FIFO
Completed Jobs: 1

Event Timeline Enable zooming

Executors
Added (Blue)
Removed (Red)

Jobs
Succeeded (Blue)
Failed (Red)
Running (Green)

Timeline markers: 55, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 0, 5, 10, 15, 20, 25, 30.

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:25 collect at <console>:25	2021/01/24 13:45:32	2 s	1/1	5/5

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

DAG Visualization –Spark-shell

APACHE Spark 3.0.1

Jobs Stages Storage Environment Executors

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 0.2 s
Locality Level Summary: Process local: 5
Associated Job Ids: 0

► DAG Visualization

Stage 0

parallelize

ParallelCollectionRDD [0]
parallelize at <console>:25

► Show Additional Metrics
► Event Timeline

Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	19.0 ms	40.0 ms	41.0 ms	41.0 ms	42.0 ms
GC Time	0.0 ms	23.0 ms	23.0 ms	23.0 ms	23.0 ms

Showing 1 to 2 of 2 entries

► Aggregated Metrics by Executor

Localhost:4040 - PySparkShell

APACHE Spark 3.0.1

Jobs Stages Storage Environment Executors SQL

PySparkShell application UI

Spark Jobs [\(?\)](#)

User: snehadhana
Total Uptime: 7.9 min
Scheduling Mode: FIFO
Completed Jobs: 1

Event Timeline

Enable zooming

Executors
Added
Removed

Jobs
Succeeded
Failed
Running

30 35 40 45 50 55 0 5 10

24 January 13:56 24 January 13:57

collect at <stdin>:1 (Job 0)

Completed Jobs (1)

Page: 1 Pages. Jump to . Show items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <stdin>:1 collect at <stdin>:1	2021/01/24 13:57:10	3 s	1/1	4/4

Page: 1 Pages. Jump to . Show items in a page. Go

DAG visualization - PySpark

APACHE SPARK 3.0.1

Jobs Stages Storage Environment Executors SQL

PySparkShell application UI

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 6 s
Locality Level Summary: Process local: 4
Associated Job Ids: 0

▼ DAG Visualization

```
graph TD; A[ParallelCollectionRDD [0]  
readRDDFromFile at PythonRDD.scala:262] --> B[PythonRDD [1]  
collect at <stdin>:1]
```

Stage 0

parallelize

ParallelCollectionRDD [0]
readRDDFromFile at PythonRDD.scala:262

PythonRDD [1]
collect at <stdin>:1

▼ Show Additional Metrics

- Select All
- Scheduler Delay
- Task Deserialization Time
- Result Serialization Time
- Getting Result Time
- Peak Execution Memory

▶ Event Timeline

Summary Metrics for 4 Completed Tasks

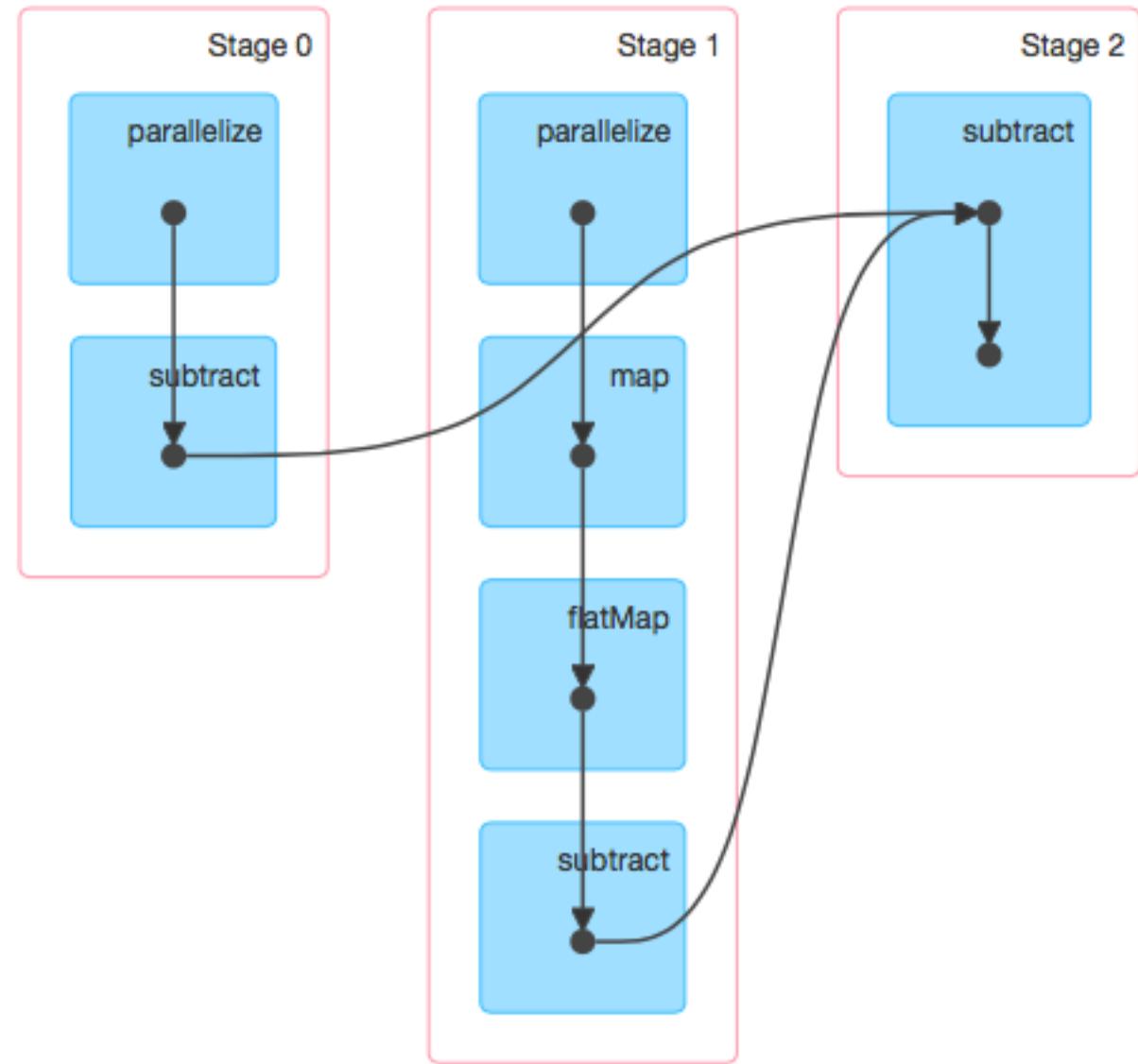
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 s	1 s	1 s	1 s	1 s
GC Time	17.0 ms	17.0 ms	17.0 ms	17.0 ms	17.0 ms

Showing 1 to 2 of 2 entries

▶ Aggregated Metrics by Executor

DAG visualization

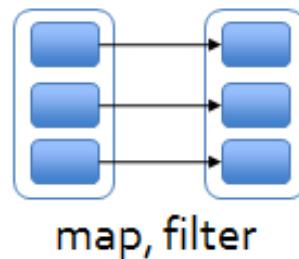
Multiple
stages DAG



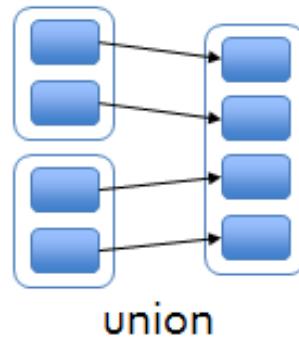
Transformations

- Map Transformation
- Filter Transformation
- flatMap Transformation
- distinct Transformation
- union Transformation
- intersection Transformation
- subtract Transformation

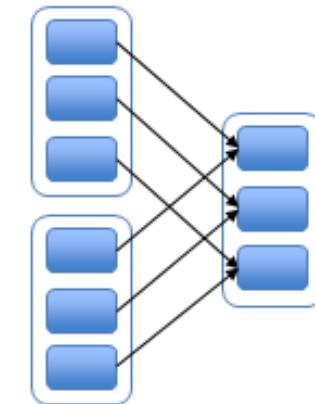
“Narrow” deps:



map, filter

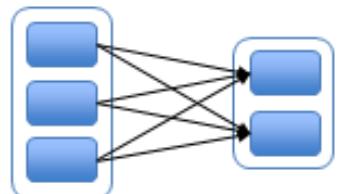


union

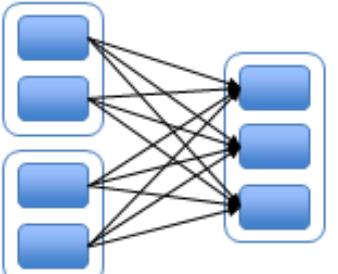


join with
inputs co-
partitioned

“Wide” (shuffle) deps:



groupByKey



join with inputs not
co-partitioned

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)    # => {1,
4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x)) # => {0,
0, 1, 0, 1, 2}
```

Map Transformation

```
[scala> val rdd1=sc.parallelize(List(1,2,3,4))  
      rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:24  
]  
  
[scala> val maprdd1 = rdd1.map(x=>x+5)  
      maprdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at map at <console>:25  
]  
  
[scala> maprdd1.collect  
      res5: Array[Int] = Array(6, 7, 8, 9)  
]  
scala> █
```

Filter Transformation

```
[scala> val rdd1=sc.parallelize(List(1,2,3,4))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:24

[scala> val maprdd1 = rdd1.map(x=>x+5)
maprdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at map at <console>:25

[scala> maprdd1.collect
res5: Array[Int] = Array(6, 7, 8, 9)

[scala> val filterrdd1 = rdd1.filter(x => x!=3)
filterrdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[7] at filter at <console>:25

[scala> filterrdd1.collect
res6: Array[Int] = Array(1, 2, 4)

scala> █
```

flatMap Transformation

```
[scala> val rdd2=sc.parallelize(List(1,2))  
      rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:24  
  
[scala> val maprdd2 = rdd2.map(x=>x.to(3))  
      maprdd2: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.Inclusive] = MapPartitionsRDD[9  
      ] at map at <console>:25  
  
[scala> maprdd2.collect  
      res7: Array[scala.collection.immutable.Range.Inclusive] = Array(Range 1 to 3, Range 2 to 3)  
  
[scala> val flatmaprdd2 = rdd2.flatMap(x=>x.to(3))  
      flatmaprdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[10] at flatMap at <console>:25  
  
[scala> flatmaprdd2.collect  
      res8: Array[Int] = Array(1, 2, 3, 2, 3)  
  
scala> █
```

Distinct Transformation

```
[scala> val rdd3 = sc.parallelize(List(1,1,3,4,5,3,4,5,2,3,2,3,7,3,2))  
      rdd3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelize at <console>:24  
  
[scala> val distinctrdd3 = rdd3.distinct  
      distinctrdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[14] at distinct at <console>:25  
  
[scala> distinctrdd3.collect  
      res9: Array[Int] = Array(4, 1, 5, 2, 3, 7)  
  
scala> ]
```

Union Transformation

```
[scala] val rdd4 = sc.parallelize(List(1,2,3,4))
rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[15] at parallelize at <console>:24

[scala] val rdd5 = sc.parallelize(List(4,5))
rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:24

[scala] val unionrdd = rdd4.union(rdd5)
unionrdd: org.apache.spark.rdd.RDD[Int] = UnionRDD[17] at union at <console>:27

[scala] unionrdd.collect
res10: Array[Int] = Array(1, 2, 3, 4, 4, 5)

scala>
```

Intersection Transformation

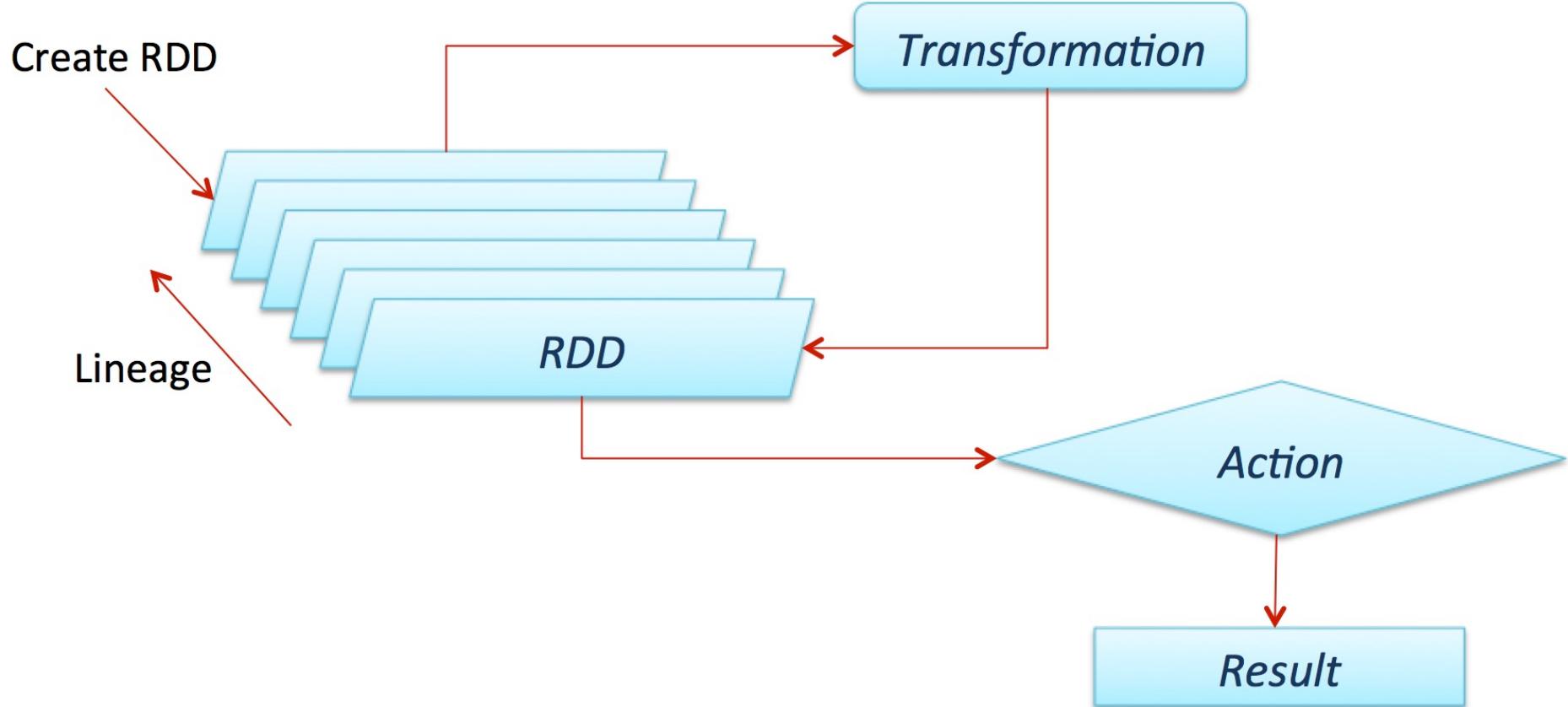
```
[scala] val rdd4 = sc.parallelize(List(1,2,3,4)) ]  
rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[15] at parallelize at <console>:24  
  
[scala] val rdd5 = sc.parallelize(List(4,5)) ]  
rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:24  
  
[scala] val unionrdd = rdd4.union(rdd5) ]  
unionrdd: org.apache.spark.rdd.RDD[Int] = UnionRDD[17] at union at <console>:27  
  
[scala] unionrdd.collect ]  
res10: Array[Int] = Array(1, 2, 3, 4, 4, 5)  
  
[scala] val intersectionrdd = rdd4.intersection(rdd5) ]  
intersectionrdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[23] at intersection at <console>:  
:27  
  
[scala] intersectionrdd.collect ]  
res11: Array[Int] = Array(4)  
  
scala> █
```

Subtract Transformation

```
[scala> val rdd4 = sc.parallelize(List(1,2,3,4))  
rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[15] at parallelize at <console>:24  
]  
  
[scala> val rdd5 = sc.parallelize(List(4,5))  
rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:24  
]  
  
[scala> val unionrdd = rdd4.union(rdd5)  
unionrdd: org.apache.spark.rdd.RDD[Int] = UnionRDD[17] at union at <console>:27  
]  
  
[scala> unionrdd.collect  
res10: Array[Int] = Array(1, 2, 3, 4, 4, 5)  
]  
  
[scala> val intersectionrdd = rdd4.intersection(rdd5)  
intersectionrdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[23] at intersection at <console>  
:27  
]  
  
[scala> intersectionrdd.collect  
res11: Array[Int] = Array(4)  
]  
  
[scala> val subtractrdd = rdd4.subtract(rdd5)  
subtractrdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[27] at subtract at <console>:27  
]  
  
[scala> subtractrdd.collect  
res12: Array[Int] = Array(1, 2, 3)  
]  
scala> █
```

Actions

- collect
- reduce
- take
- top



Basic Actions

```
nums = sc.parallelize([1, 2, 3])
# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]
# Return first K elements
nums.take(2) # => [1, 2]
# Count number of elements
nums.count() # => 3
# Merge elements with an associative function
nums.reduce(lambda x, y: x + y) # => 6
# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

Reduce Action

```
[scala] val rdd6 = sc.parallelize(List(1,2,3,4,5))
rdd6: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[28] at parallelize at <console>:24

[scala] rdd6.reduce((x,y) => x+y)
res13: Int = 15

scala> █
```

Take Action

```
[scala> val rdd6 = sc.parallelize(List(1,2,3,4,5))
rdd6: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[28] at parallelize at <console>:24

[scala> rdd6.reduce((x,y) => x+y)
res13: Int = 15

[scala> rdd6.take(2)
res14: Array[Int] = Array(1, 2)

scala> █
```

Top Action

```
[scala> val rdd6 = sc.parallelize(List(1,2,3,4,5))
rdd6: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[28] at parallelize at <console>:24

[scala> rdd6.reduce((x,y) => x+y)
res13: Int = 15

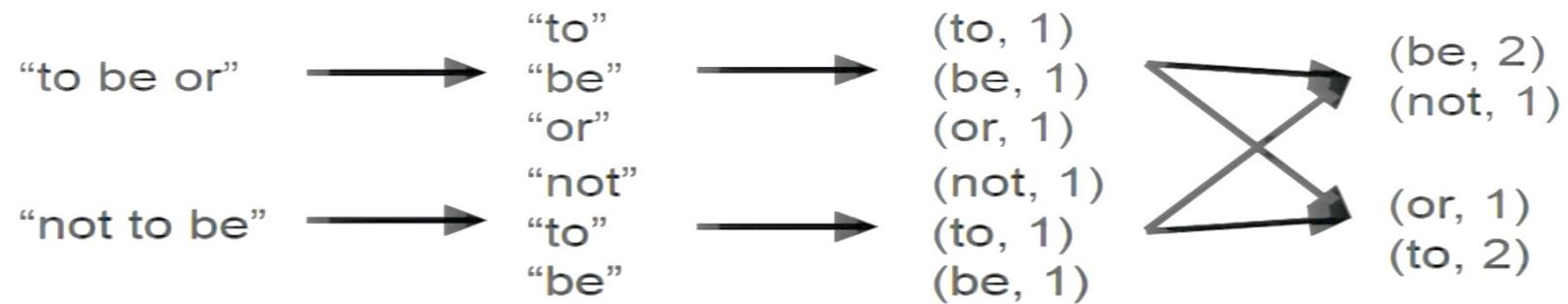
[scala> rdd6.take(2)
res14: Array[Int] = Array(1, 2)

[scala> rdd6.top(3)
res15: Array[Int] = Array(5, 4, 3)

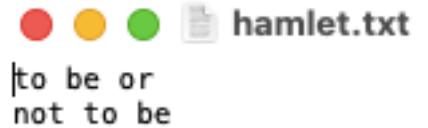
scala> █
```

Example: Word count

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda x, y: x + y)
```



MapReduce - Scala



```
[scala]> [redacted]  
[scala]> val text=sc.textFile("hdfs://localhost:9000/Demo/hamlet.txt")  
text: org.apache.spark.rdd.RDD[String] = hdfs://localhost:9000/Demo/hamlet.txt MapPartitionsRDD[7]  
[ at textFile at <console>:24 ]  
[redacted]  
[scala]> val counts = text.flatMap(line => line.split(" ")){  
counts: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[8] at flatMap at <console>:25 ]  
[scala]> counts.collect  
res7: Array[String] = Array(to, be, or, not, to, be)  
[redacted]  
scala> val map = counts.map(word=>(word, 1))  
map: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[9] at map at <console>:25  
[redacted]  
scala> map.collect  
res8: Array[(String, Int)] = Array((to,1), (be,1), (or,1), (not,1), (to,1), (be,1))  
[redacted]  
scala> val reduce = map.reduceByKey(_+_){  
reduce: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[10] at reduceByKey at <console>:25  
[redacted]  
scala> reduce.collect  
res9: Array[(String, Int)] = Array((not,1), (or,1), (be,2), (to,2))  
[redacted]  
scala> [redacted]
```

WordCount - Spark

```
val textFile = sc.textFile("hdfs://...")  
val counts = textFile.flatMap(line => line.split(" "))  
    .map(word => (word, 1))  
    .reduceByKey(_ + _)  
counts.saveAsTextFile("hdfs://...")
```

Scala

```
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://...")
```

Python

<http://spark.apache.org/examples.html>

MapReduce in Python

```
pyspark-map.py ×  
1  from pyspark import SparkContext  
2  
3  sc = SparkContext.getOrCreate()  
4  
5  text=sc.textFile("hdfs://localhost:9000/Demo/hamlet.txt")  
6  
7  counts = text.flatMap(lambda line: line.split(" ")) \  
8      .map(lambda word: (word, 1)) \  
9      .reduceByKey(lambda a, b: a + b)  
10  
11 for element in counts.collect():  
12     print(element)  
13  
14 counts.saveAsTextFile("/Users/snehadhanda/work/result.txt")
```

```
Run: pyspark-map ×  
▶ Up ↑ Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties  
▶ Down ↓ Setting default log level to "WARN".  
▶ Left ⌂ To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).  
21/01/09 11:49:24 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.  
21/01/09 11:49:24 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.  
▶ Right ⌂ ('to', 2)  
▶ Bottom ⌂ ('be', 2)  
▶ Bottom ⌂ ('or', 1)  
▶ Bottom ⌂ ('not', 1)  
  
Process finished with exit code 0
```

Working with Key-Value Pairs

- Spark's 'distributed reduce' transformations act on RDDs of key value pairs

- Python:

```
pair = (a, b)
        pair[0] # => a
        pair[1] # => b
```
- Scala:

```
val pair = (a, b)
        pair._1 // => a
        pair._2 // => b
```
- Java:

```
Tuple2 pair = new Tuple2(a, b);
// class scala.Tuple2
        pair._1 // => a
        pair._2 // => b
```

Some Key-Value Operations

```
pets = sc.parallelize([('cat', 1), ('dog', 1),  
                      ('cat', 2)])  
  
pets.reduceByKey(lambda x, y: x + y)  
# => {('cat', 3), ('dog', 1)}  
  
pets.groupByKey()  
# => {('cat', Seq(1, 2)), ('dog', Seq(1))}  
  
pets.sortByKey()  
# => {('cat', 1), ('cat', 2), ('dog', 1)}
```

- `reduceByKey` also automatically implements combiners on the map side

Other Key-Value Operations

```
val visits = sc.parallelize(List(  
    ("index.html", "1.2.3.4"),  
    ("about.html", "3.4.5.6"),  
    ("index.html", "1.3.3.1")))  
val pageNames = sc.parallelize(List(  
    ("index.html", "Home"), ("about.html", "About")))  
visits.join(pageNames)  
// ("index.html", ("1.2.3.4", "Home"))  
// ("index.html", ("1.3.3.1", "Home"))  
// ("about.html", ("3.4.5.6", "About"))  
visits.cogroup(pageNames)  
// ("index.html", (Seq("1.2.3.4", "1.3.3.1"),  
Seq("Home")))  
// ("about.html", (Seq("3.4.5.6"), Seq("About")))
```

Controlling the level of Parallelism

- All the pair RDD operations take an optional second parameter for number of tasks

```
words.reduceByKey(lambda x, y: x + y, 5)
```

```
words.groupByKey(5)
```

```
visits.join(pageViews, 5)
```

Can also set `spark.default.parallelism` property

Using Local Variables

- External Variables you use in a closure will automatically be shipped to the cluster:

```
query = raw_input("Enter a query:")
pages.filter(lambda x:
x.startswith(query)).count()
```

- Some caveats:
 - Each task gets a new copy (updates aren't sent back)
 - Variable must be serializable (Java/Scala) or Pickle-able (Python)
 - Don't use fields of an outer object (ships all of it!)

Other RDDs Operations

- `sample()`: deterministically sample a subset
- `cartesian()`: cross product
- `pipe()`: pass through external program
- See programming guide for more
- <http://spark.apache.org/docs/latest/>

Hadoop Compatibility

- Spark can read/write from any storage / format that has plugin for Hadoop!
- Examples: HDFS, Hbase, Cassandra, Avro, SequenceFile
- Reuses Hadoop's InputFormat and Output APIs
- APIs like `SparkContext.textFile` support filesystems, while `SparkContext.hadoopRDD` allows passing any Hadoop JobConf to configure an input source

Create a SparkContext

Scala

```
import spark.SparkContext  
import spark.SparkContext._  
  
val sc = new SparkContext("masterUrl", "name", "sparkHome", Seq("app.jar"))
```

Cluster URL, or local /
local[N]

App
name

Spark install path
on cluster

Java

```
import spark.api.java.JavaSparkContext;  
  
JavaSparkContext sc = new JavaSparkContext(  
    "masterUrl", "name", "sparkHome", new String[] {"app.jar"});
```

Python

```
from pyspark import SparkContext  
  
sc = SparkContext("masterUrl", "name", "sparkHome", ["library.py"]))
```

Complete App: Scala

```
import spark.SparkContext
import spark.SparkContext._

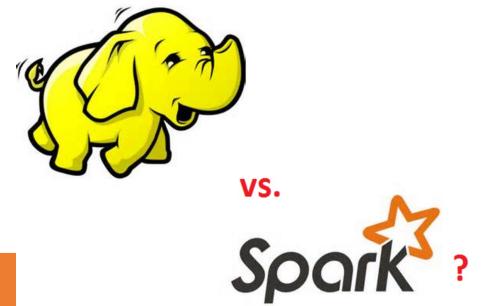
object WordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "WordCount", args(0), Seq(args(1)))
    val lines = sc.textFile(args(2))
    lines.flatMap(_.split(" "))
      .map(word => (word, 1))
      .reduceByKey(_ + _)
      .saveAsTextFile(args(3))
  }
}
```

Complete App: Python

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
    sc = SparkContext("local", "WordCount", sys.argv[0], None)
    lines = sc.textFile(sys.argv[1])

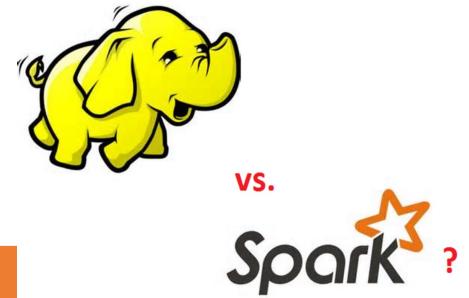
    lines.flatMap(lambda s: s.split(" ")) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda x, y: x + y) \
        .saveAsTextFile(sys.argv[2])
```



Hadoop vs Spark

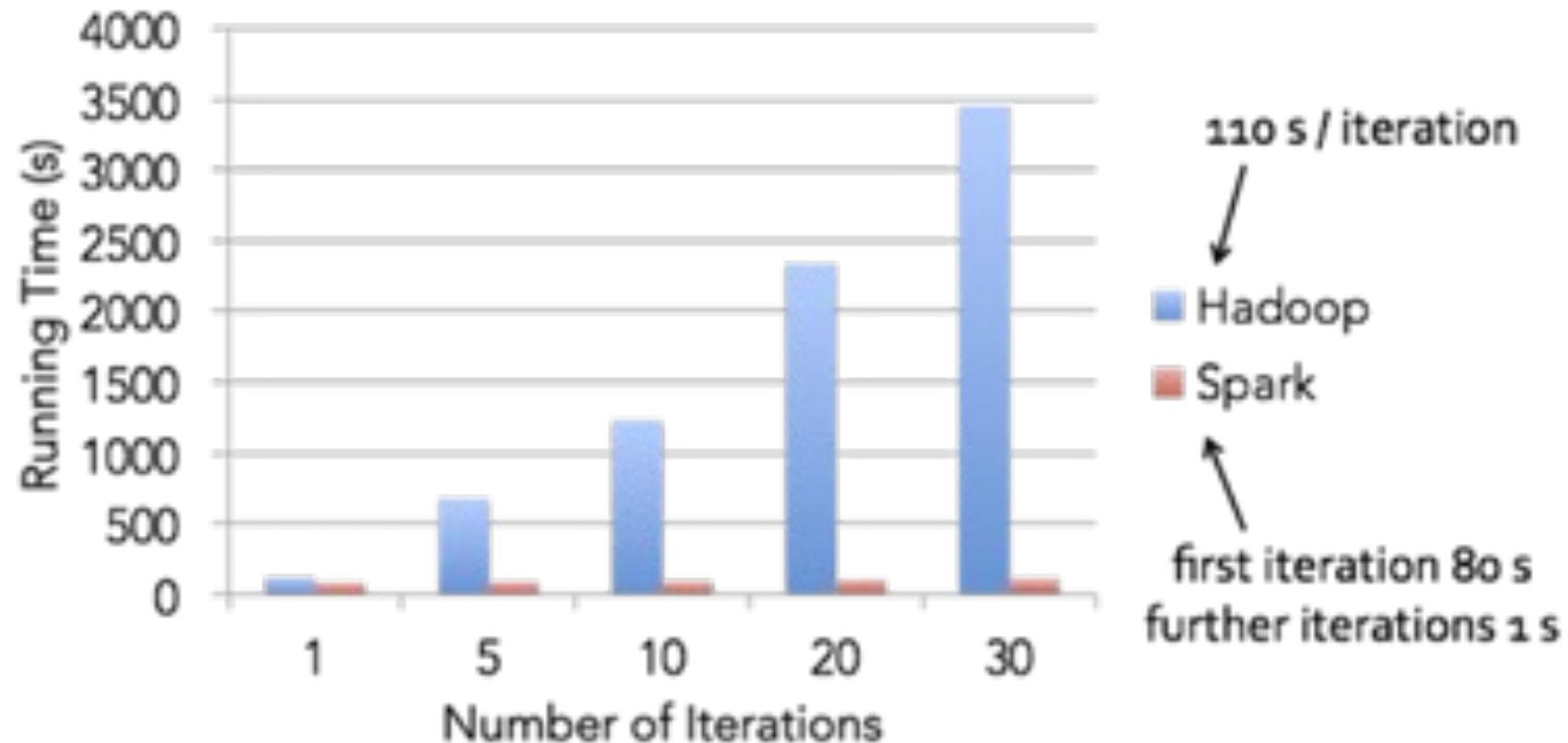
Parameter	Hadoop	Spark
Data Processing	Batch Processing	Batch/ Real-time/ Iterative/ Interactive/ Graph Processing
Performance	Faster than traditional system	100x times than MapReduce
Compatibility	Compatible	Spark and MapReduce are compatible with each other
Ease of use	Hadoop provides add-ons to support users but doesn't have interactive mode	User-friendly APIs for Scala, Java Python and R etc. Provides interactive mode for developers.
Price	Less costly as MapReduce model provides cheaper strategy	Costlier than Hadoop since it uses in-memory processing
Fault Tolerance	Storing data in HDFS	RDDs for fault tolerance
Latency	High latency computing framework	Low latency computing framework
Scalability	Yahoo has around 40,000 node Hadoop cluster	Spark cluster has 7,000 nodes
Difficulty	Need to hand code every operation	Tons of high-level operators

Hadoop vs Spark

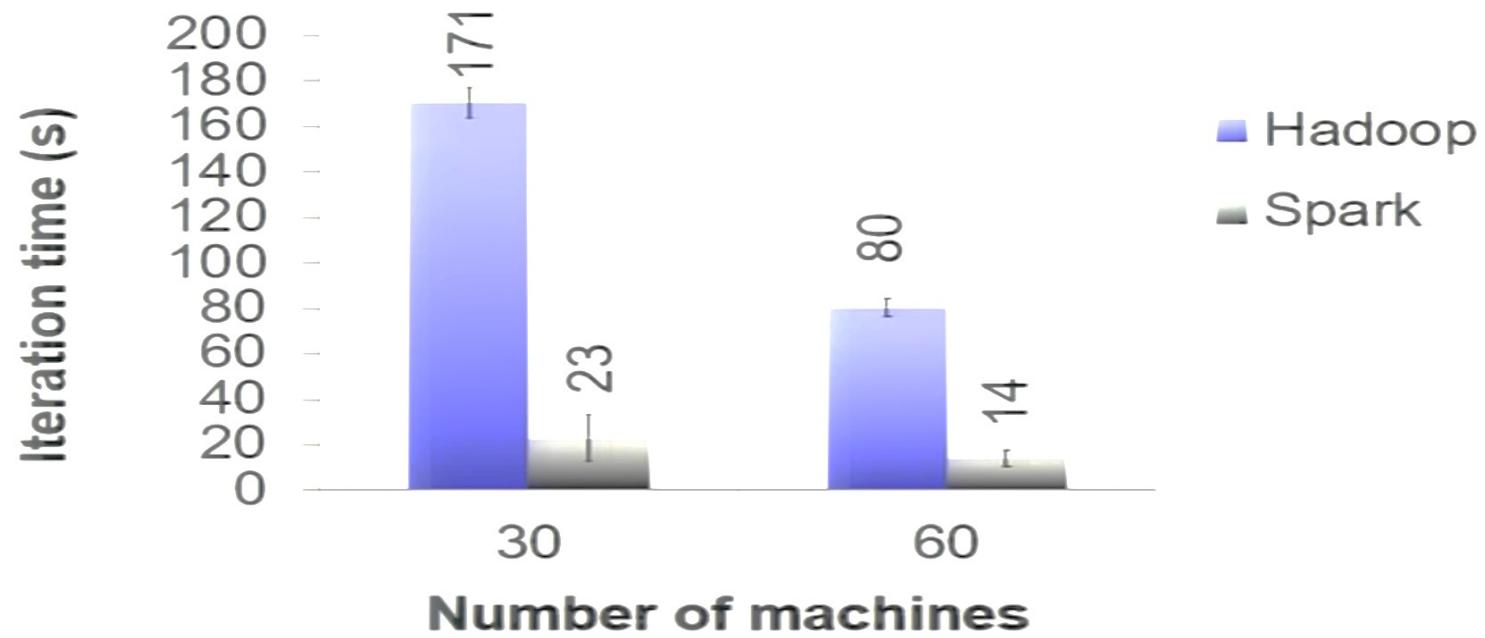


Parameter	Hadoop	Spark
Easy to Manage	Only provides the batch engine	Batch/interactive/streaming in same cluster
Real-time analysis	Perform batch processing	Process real-time data
Interactive mode	No	Yes
Recovery	Naturally resilient to system faults	RDDs allows recovery
Scheduler	Needs an external job scheduler	Own flow scheduler
Programming Language support	Primarily Java others C, C++, Ruby, Groovy, Perl, Python	Scala, Java, Python, R, SQL
SQL support	Hive	Spark SQL
Language Developed	Java	Scala
Machine Learning	Apache Mahout	Mlib
Caching	No	Yes
Hardware Requirements	Commodity hardware	Needs mid to high-level hardware

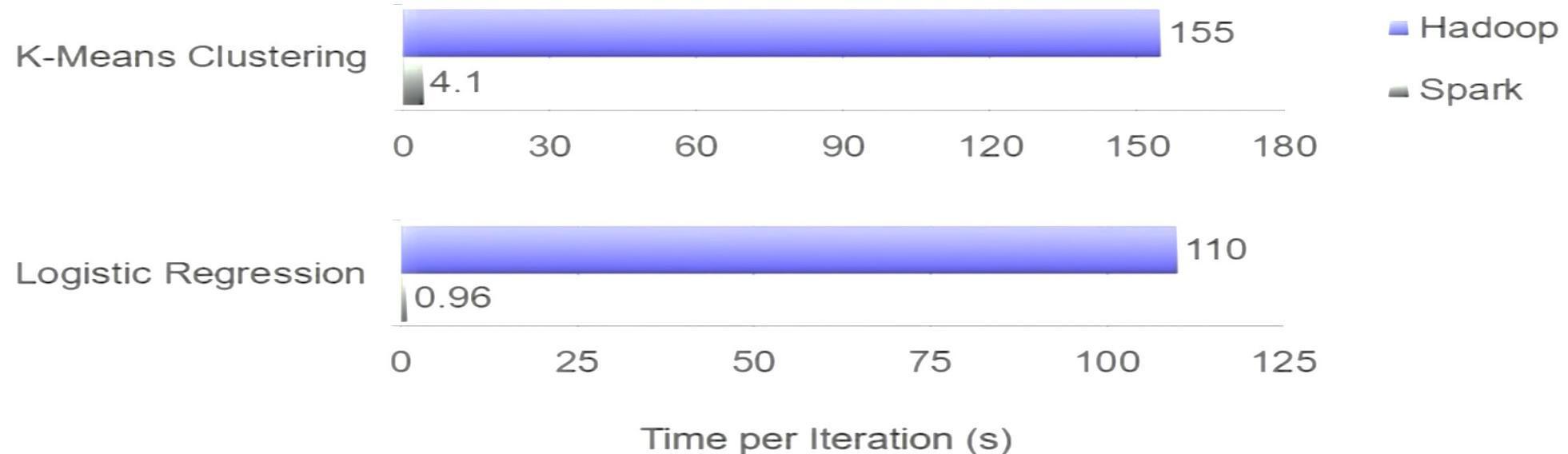
Performance Comparison



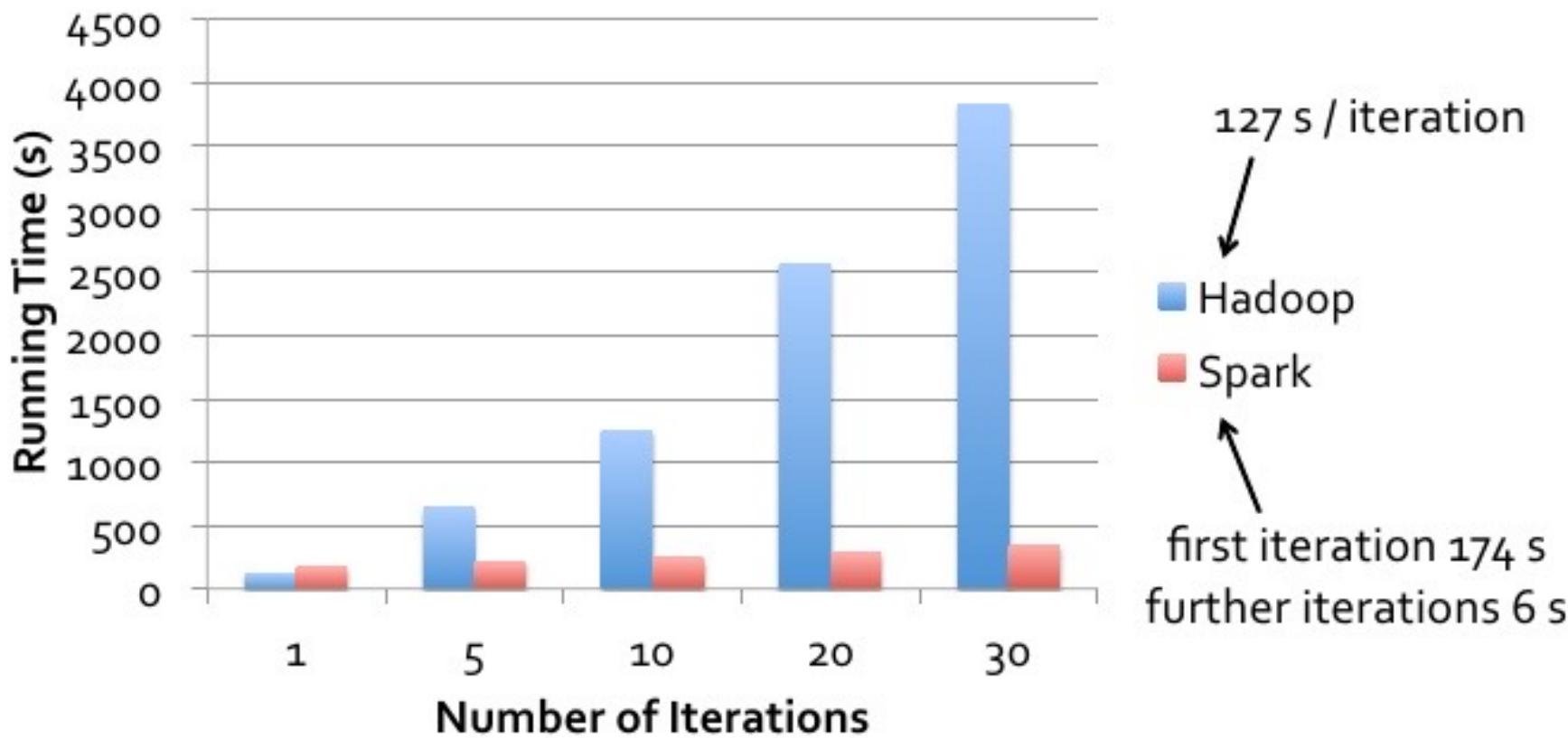
PageRank Performance Comparison



Performance Comparison

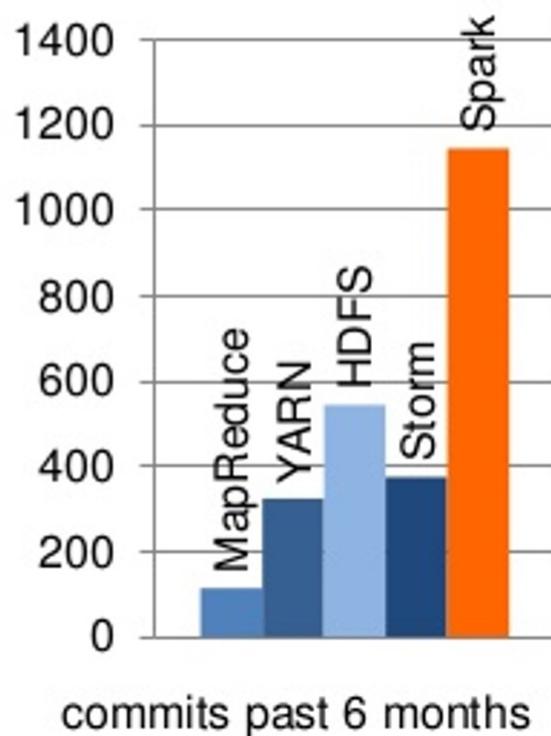


Logistic Regression Performance



Spark Community

250+ developers, 50+ companies contributing
Most active open source project in big data



Spark Installation

- [Apache Spark](#)



Lightning-fast unified analytics engine

[Download](#) [Libraries](#) ▾ [Documentation](#) ▾ [Examples](#) [Community](#) ▾ [Developers](#) ▾

Apache Software Foundation ▾

Download Apache Spark™

1. Choose a Spark release: [3.0.1 \(Sep 02 2020\) ▾](#)
2. Choose a package type: [Pre-built for Apache Hadoop 2.7 ▾](#)
3. Download Spark: [spark-3.0.1-bin-hadoop2.7.tgz](#)
4. Verify this release using the 3.0.1 [signatures](#), [checksums](#) and [project release KEYS](#).

Note that, Spark 2.x is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. Spark 3.0+ is pre-built with Scala 2.12.

Latest Preview Release

Preview releases, as the name suggests, are releases for previewing upcoming features. Unlike nightly packages, preview releases have been audited by the project's management committee to satisfy the legal requirements of Apache Software Foundation's release policy. Preview releases are not meant to be functional, i.e. they can and highly likely will contain critical bugs or documentation errors. The latest preview release is Spark 3.0.0-preview2, published on Dec 23, 2019.

Link with Spark

Spark artifacts are [hosted in Maven Central](#). You can add a Maven dependency with the following coordinates:

```
groupId: org.apache.spark  
artifactId: spark-core_2.12  
version: 3.0.1
```

Installing with PyPi

[PySpark](#) is now available in pypi. To install just run `pip install pyspark`.

Release Notes for Stable Releases

- [Spark 3.0.1 \(Sep 02 2020\)](#)
- [Spark 2.4.7 \(Sep 12 2020\)](#)

Latest News

- Next official release: Spark 3.1.1 (Jan 07, 2021)
Spark 2.4.7 released (Sep 12, 2020)
Spark 3.0.1 released (Sep 08, 2020)
Spark 3.0.0 released (Jun 18, 2020)

[Archive](#)



[Download Spark](#)

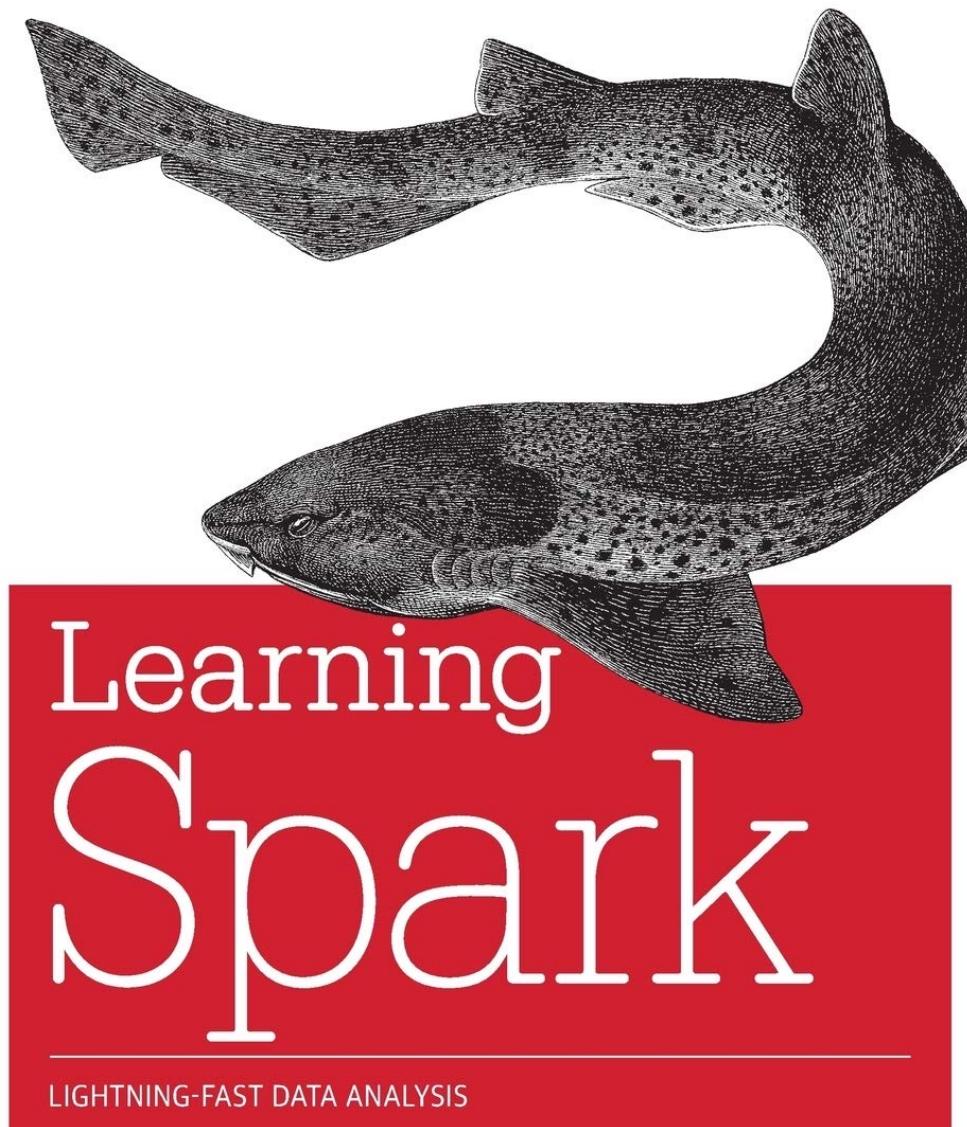
Built-in Libraries:

[SQL and DataFrames](#)
[Spark Streaming](#)
[MLlib \(machine learning\)](#)
[GraphX \(graph\)](#)

Third-Party Projects

Summary

- Spark offers a rich APIs to make data analytics fast; both fast to write and fast to run
- Achieves 100x speedups in real applications
- Growing community with 50 companies contributing
- Details: <http://spark.apache.org/>



Holden Karau, Andy Konwinski,
Patrick Wendell & Matei Zaharia

References

- Pyspark and SQL Basics:
 - <https://github.com/runawayhorse001/CheatSheet>
 - <https://datacamp-community-prod.s3.amazonaws.com/65076e3c-9df1-40d5-a0c2-36294d9a3ca9>
 - <https://spark.apache.org/>
- Reference books:
 - [Learning Spark: Lightning-Fast Big Data Analysis](#)
 - [Spark: The Definitive Guide: Big Data Processing Made Simple](#)