Apache Spark

Agenda

- Introduction to Spark
- Spark Architecture
- Explore Spark Components
  - Spark Core
  - Spark SQL
  - Streaming
  - MLLib
  - GraphX
- Case Studies

Learning Objectives

- You should know
  - Core Spark
  - Building blocks of Spark
  - Write programs in Spark
  - How Spark fits the Big Data ecosystem
  - Spark SQL
  - Machine Learning
  - Streaming data analytics
  - Graph Exploration

Confusion Galore

(2004 – 2013)

(2007 – 2015?)

(2014 – ?)

Pregel
Giraph
Tez
Drill
Dremel
Mahout
Storm
S4
Impala
Graph Lab

General Batch Processing

Specialized Systems

(iterative, interactive, ML, streaming, graph, SQL, etc)

General Unified Engine

Difficulty of Programming in MR

Word Count implementations

- Hadoop MR – 61 lines in Java
- Spark – 1 line in interactive shell

```
sc.textFile('...').flatMap(lambda x: x.split())
    .map(lambda x: (x, 1)).reduceByKey(lambda x, y: x+y)
    .saveAsTextFile('...')
```

VS

Apache Spark

Fast and general cluster computing engine interoperable with Apache Hadoop

Improves efficiency through:

- In-memory data sharing
- General computation graphs

→ Up to 100× faster

Improves usability through:

- Rich APIs in Java, Scala, Python
- Interactive shell

→ 2-5× less code

Apache Spark

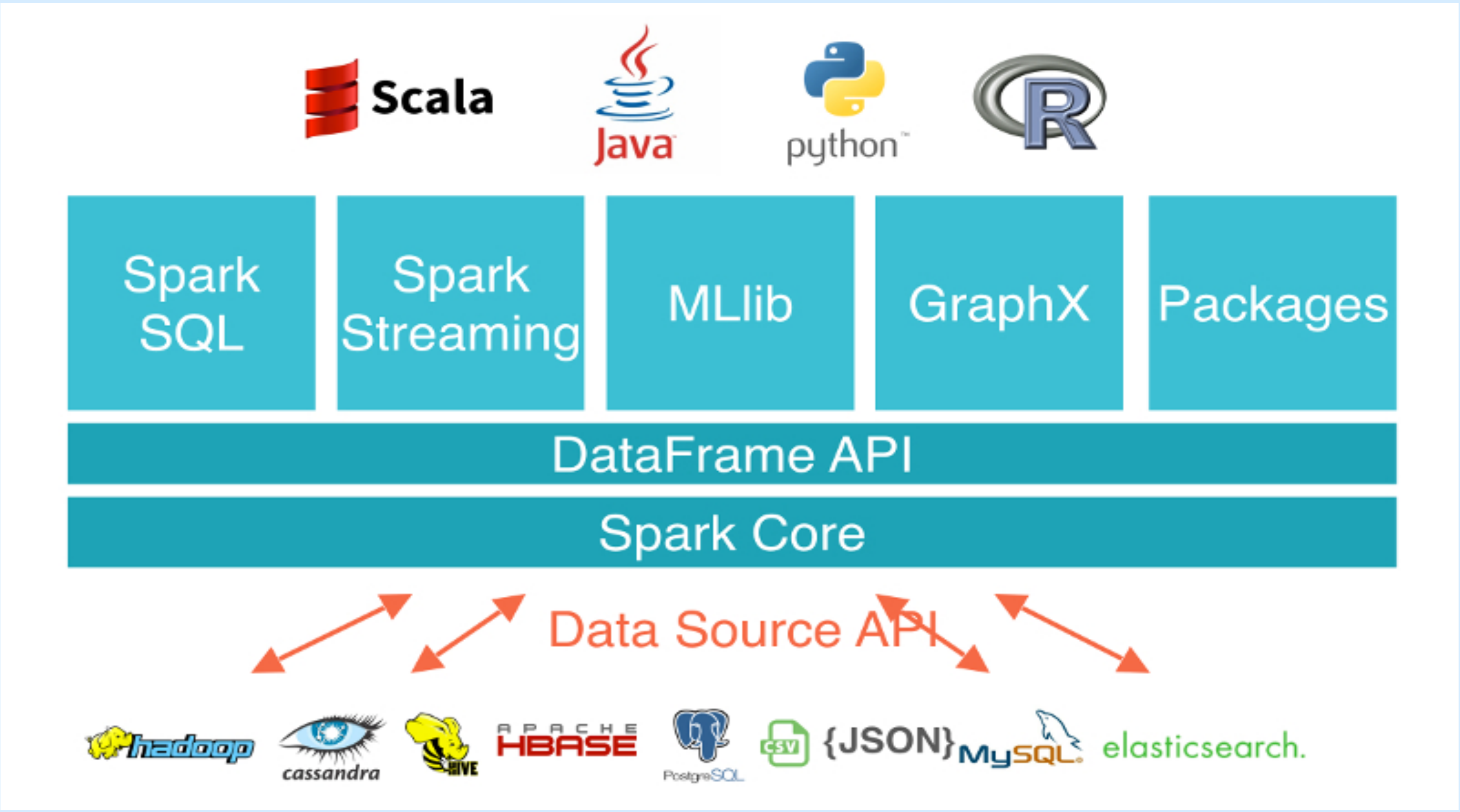
- Started at UC Berkeley (2009)
- General purpose computing engine
- Upto 100 times faster than earlier Big Data processing frameworks
- Useful for iterative machine learning, interactive querying, real-time data processing
- Modular in nature – fits well with legacy systems

Welcome to Spark

- In-memory computing framework
- Distributed Compute Engine
- Attempt at unification
  - Easy to develop
  - Orders of magnitude fast
  - Real time analytics
  - Machine Learning
  - Language Flexibility
  - API to work with different tools

- Spark can work in multiple modes
  - Standalone
  - Hadoop
  - Apache Mesos
- Written in Scala
  - Close to 400,000 lines of code
  - Very active developer community
  - Parallel projects
- ~100 years of effort (COCOMO model)

Spark Components



Spark Users

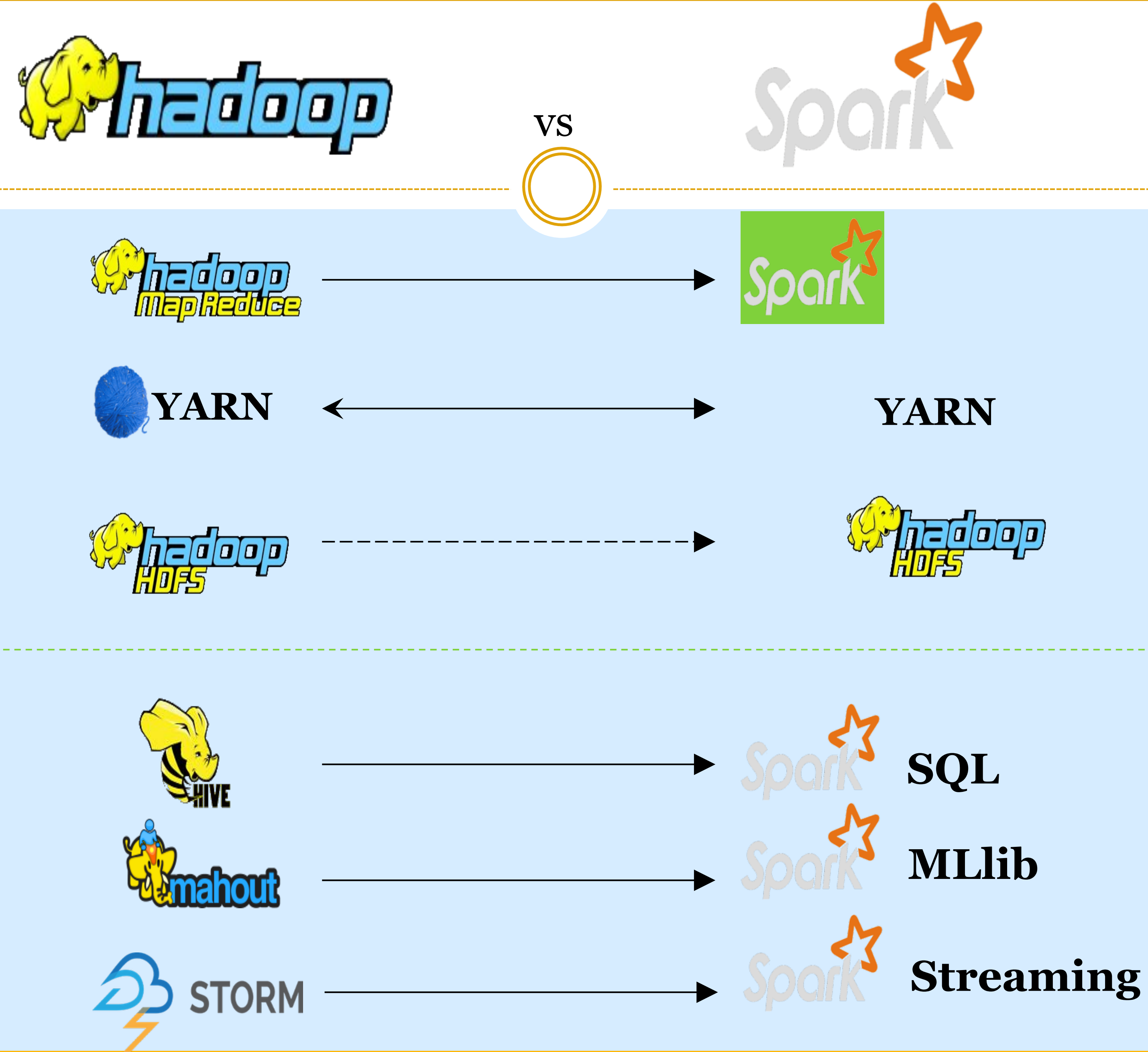
NOVARTIS
UBER
AUTODESK.
YAHOO!
salesforce
Microsoft
airbnb
Capital One
Alibaba.com
NBCUniversal
CISCO
Telefónica
Adobe
TOYOTA
Goldman Sachs
ebay
verizon
Palantir
NETFLIX
THOMSON REUTERS
NTT DATA

Spark Praise

- *"Spark is beautiful. With Hadoop, it would take us six-seven months to develop a machine learning model. Now, we can do about four models a day." - said Rajiv Bhat, senior vice president of data sciences and marketplace at InMobi.*

Spark in Action

- Ooyala
  - Offers analytics services to media organizations
  - >2 billion analytics per day to maximize revenues
  - Deliver real-time insights
- Yahoo
  - Personalization of web pages for visitors
  - Analytics for advertising
- Facebook
  - Uses Spark extensively
- Uber
  - Spark Streaming to analyze data in real-time
  - Drives decisions such as surge pricing


hadoop
VS
Spark
hadoop Map Reduce
Spark
YARN
YARN
hadoop HDFS
hadoop HDFS
HIVE
Spark
SQL
mahout
Spark
MLlib
STORM
Spark
Streaming

Spark MapReduce Competition

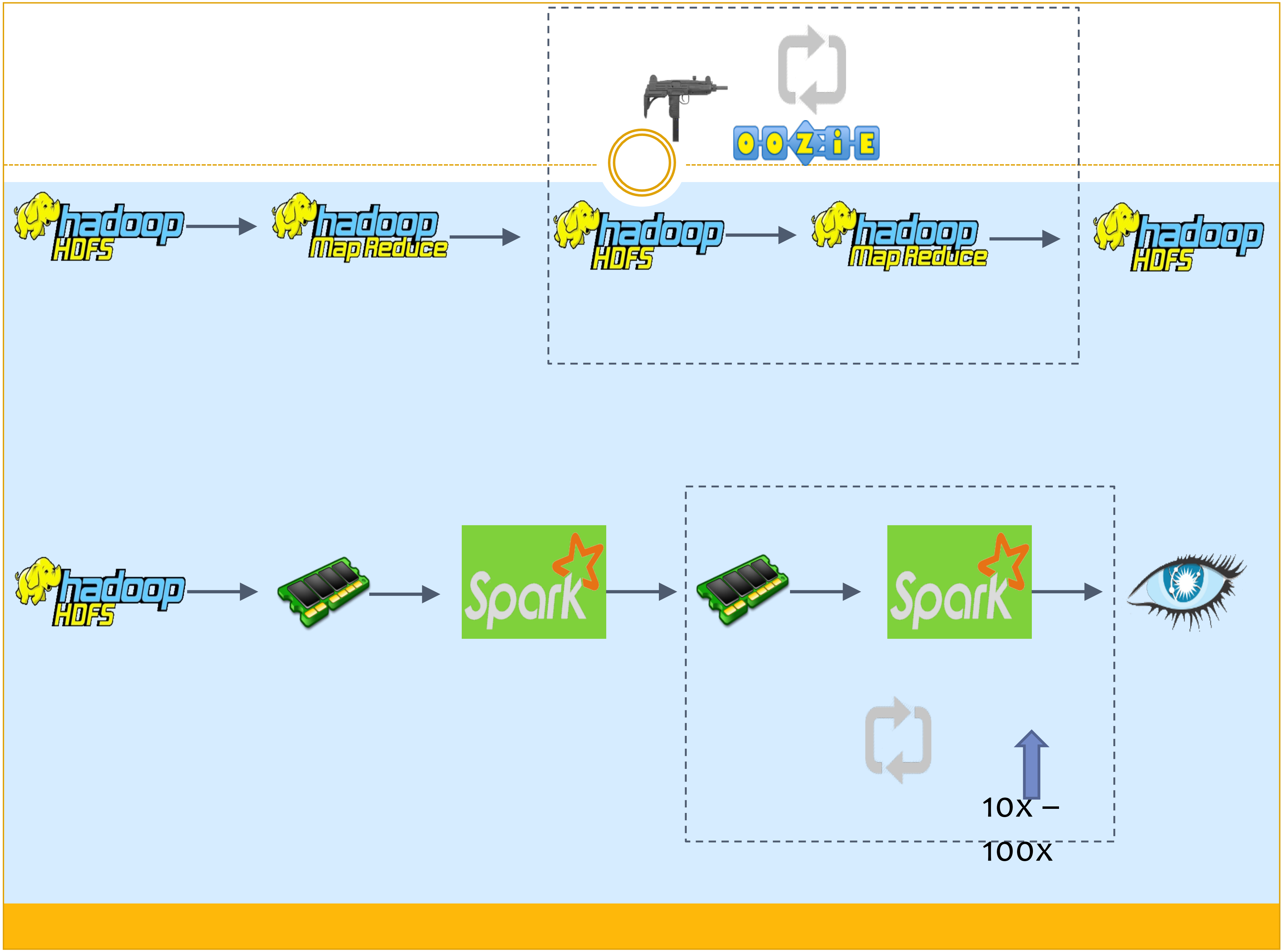
	Hadoop World Record	Spark 100 TB	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

Spark Versions

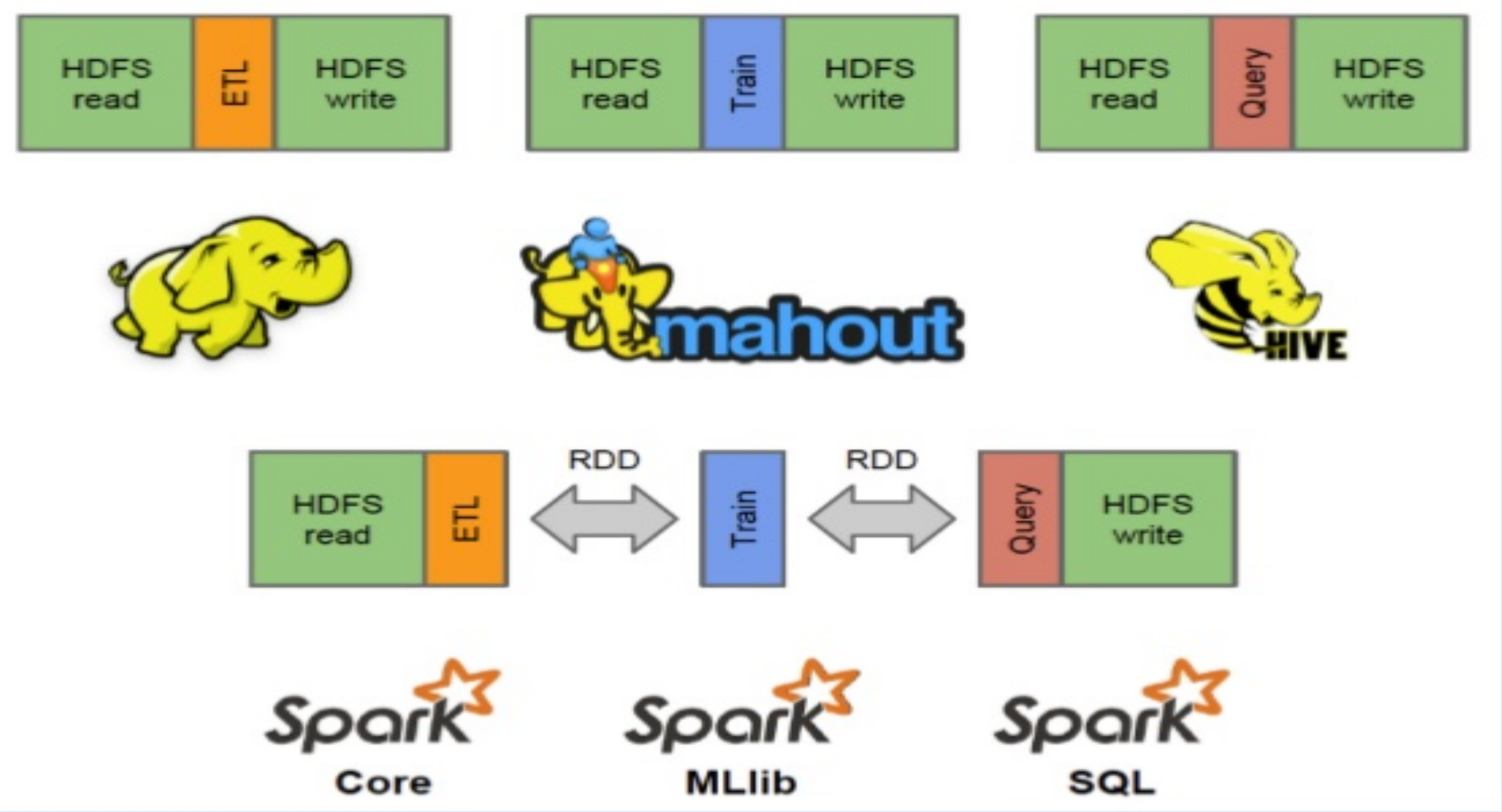
- Two prominent versions:
- Spark 1.6
- Spark 2.x

Spark 1.x vs Spark 2.x

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	13 ns


hadoop HDFS
hadoop Map Reduce
hadoop HDFS
hadoop Map Reduce
hadoop HDFS
OOZIE
hadoop HDFS
Spark
Spark
10X –
100X

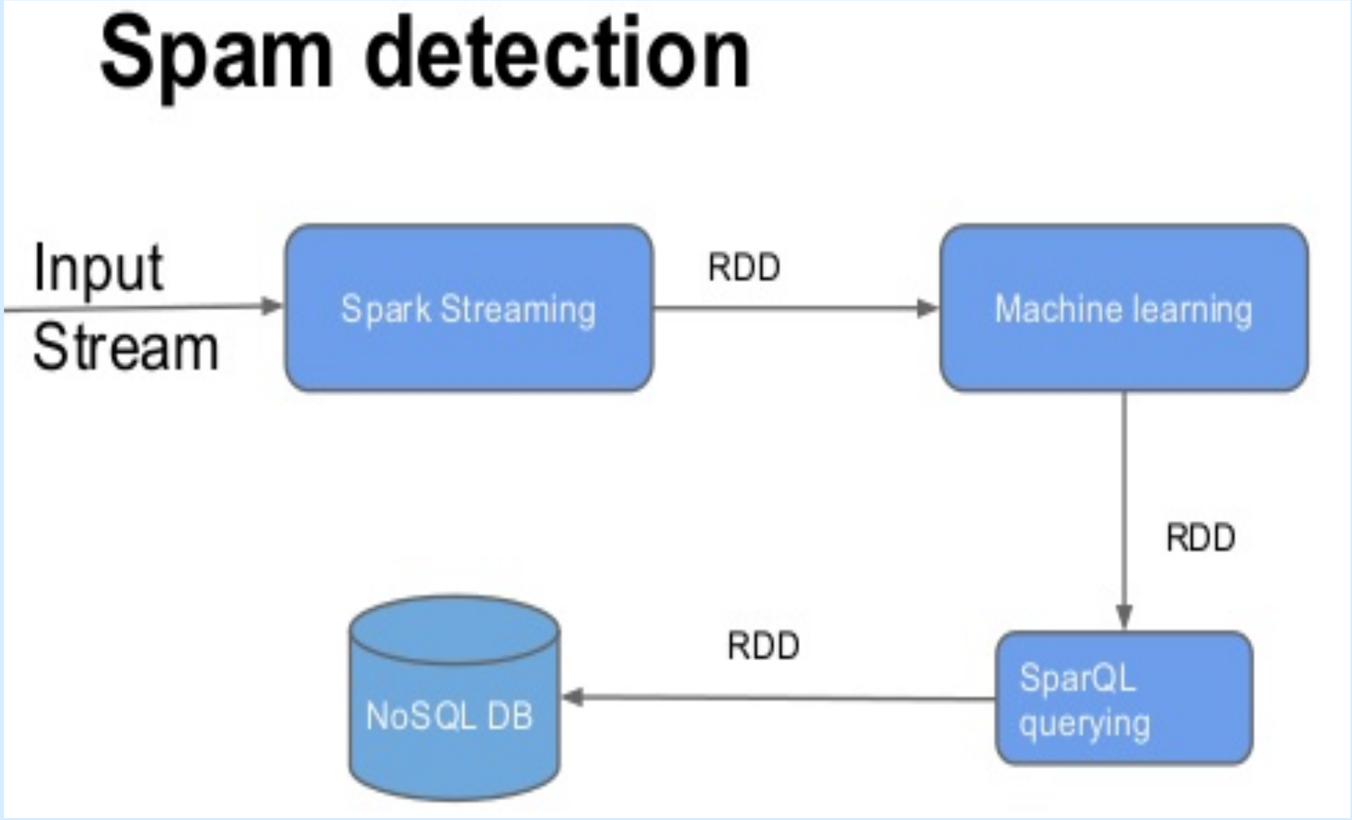
Simple ML Example



Unification – Binding Together

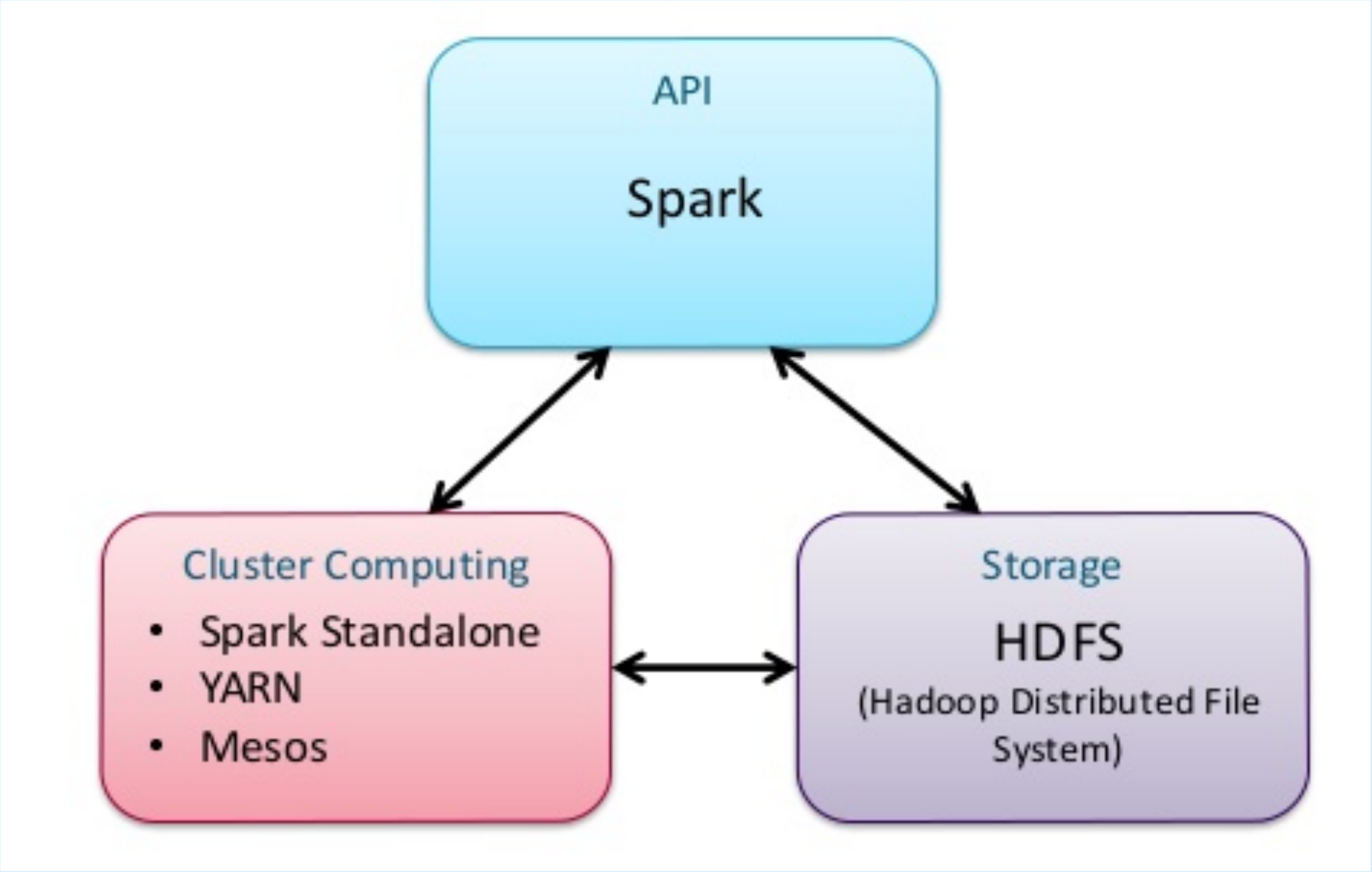
- All different systems in Spark share something known as RDD (more on this later)
    - Think of this as a dataframe for now
- Because of this common RDD, you can now mix and match

Unification



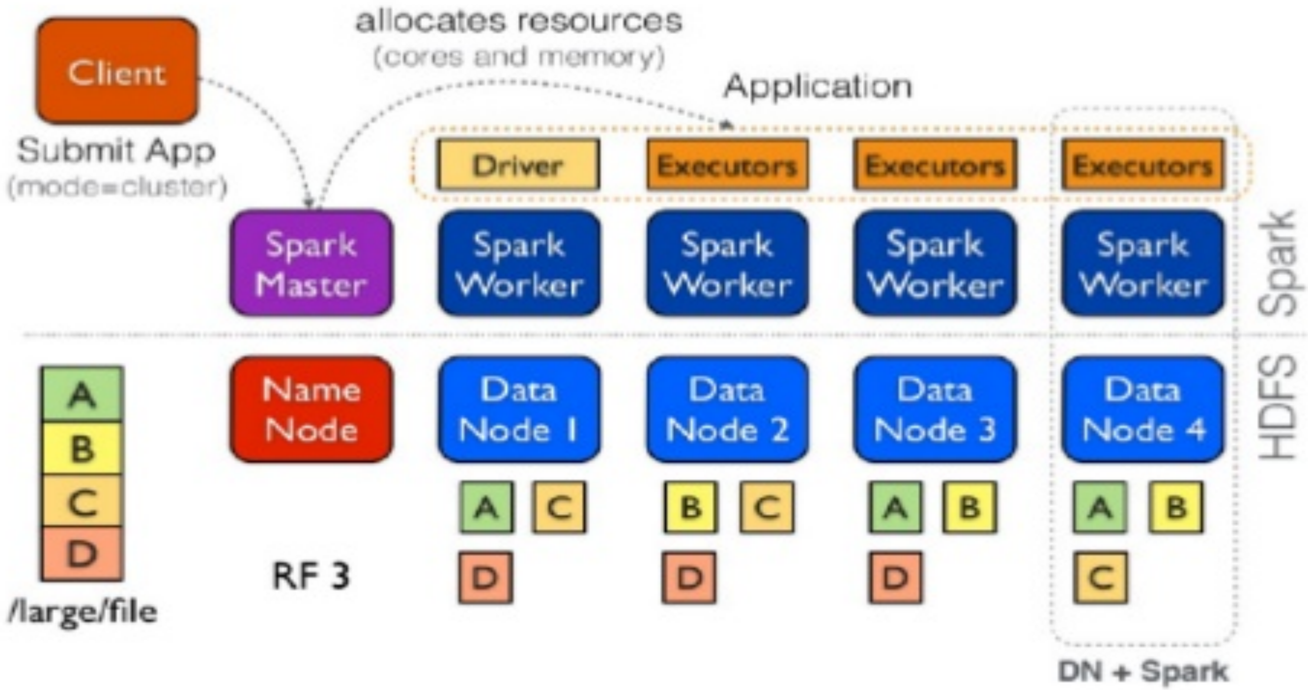
Runs Everywhere

- Run on top of any distributed system
- E.g.
  - Hadoop 1.x
  - Hadoop 2.x
  - Apache Mesos
  - Standalone
  - Own cluster
- Integrates with Hadoop
  - No separate storage layer
  - Works with HDFS


API
Spark
Cluster Computing
• Spark Standalone
• YARN
• Mesos
Storage
HDFS
(Hadoop Distributed File System)

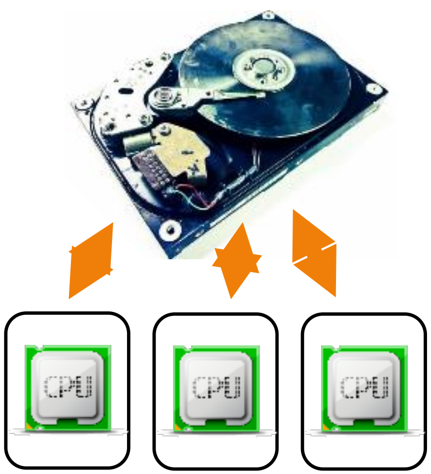
Big Picture

- There are two ways to manipulate data in Spark
  - Use the interactive shell, *i.e.*, the REPL
  - Write standalone applications, *i.e.*, driver programs



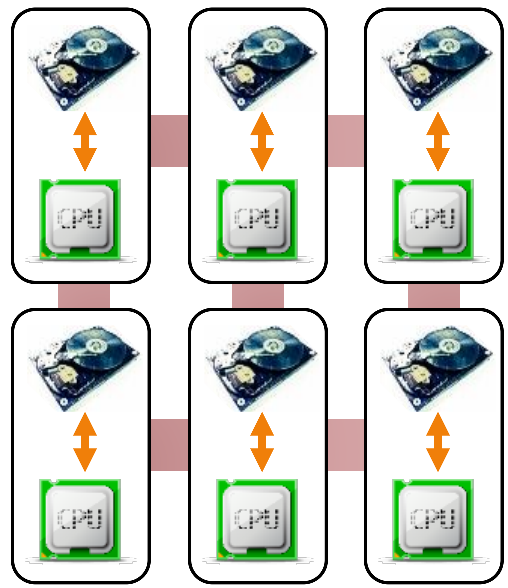
SHARED-NOTHING ARCHITECTURE

So… you want to build a parallel system. What's the best way to do that? There are different models of parallel processing in computers.



The problem is that data must travel over the network within the cluster, and that may be slow. It's useful in some scientific settings where a small data set takes a long time to analyze.

Shared-disk architecture (like many "high-performance computing" systems) distributes computation but centralizes storage



In Hadoop, data is processed where it is stored, so you don't have to send it over the network.

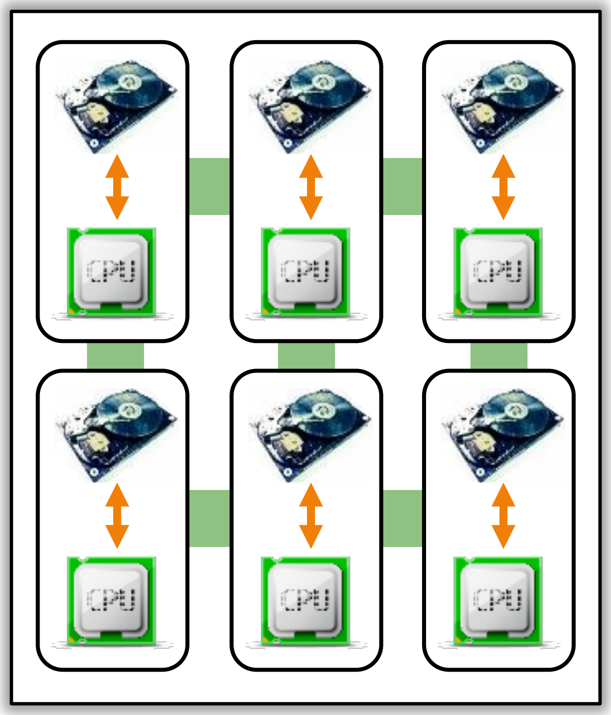
Shared-nothing architecture (like Hadoop) distributes computation *and* storage

DATA LOCALITY

When datasets are very large, as is the case for Google, Yahoo!, Amazon, Facebook, and other web giants, sending data over the network becomes a limiting factor.

Hadoop solves this problem with data locality – it distributes the data, as well as the processing power, so that data can be analyzed (close to) where it is stored.

This "shared nothing" architecture means that relatively little data needs to be sent over the network.



Shared-nothing architecture

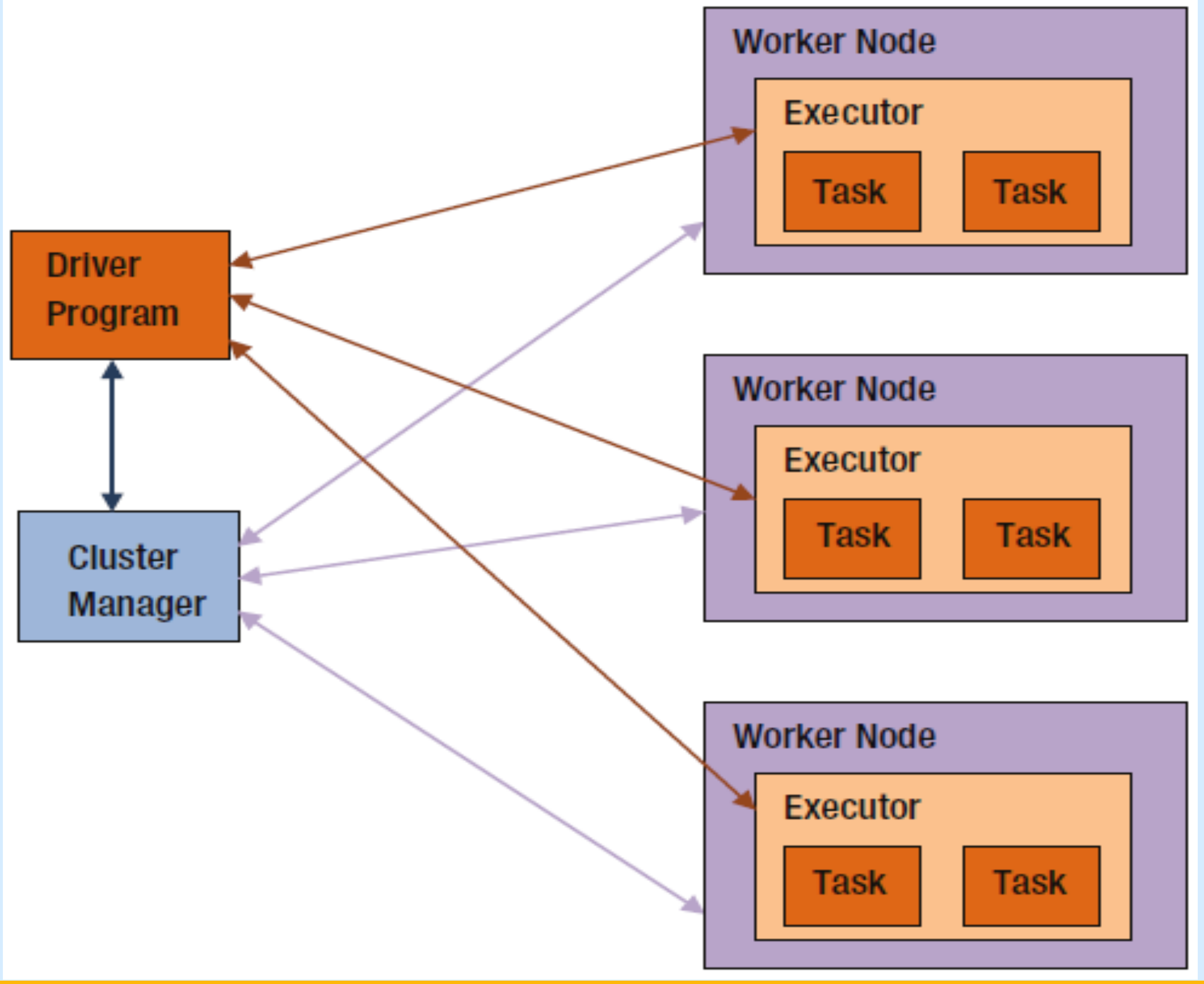
SEND THE PROGRAM TO THE DATA

This saying may help to explain the idea behind Hadoop and its programming model:

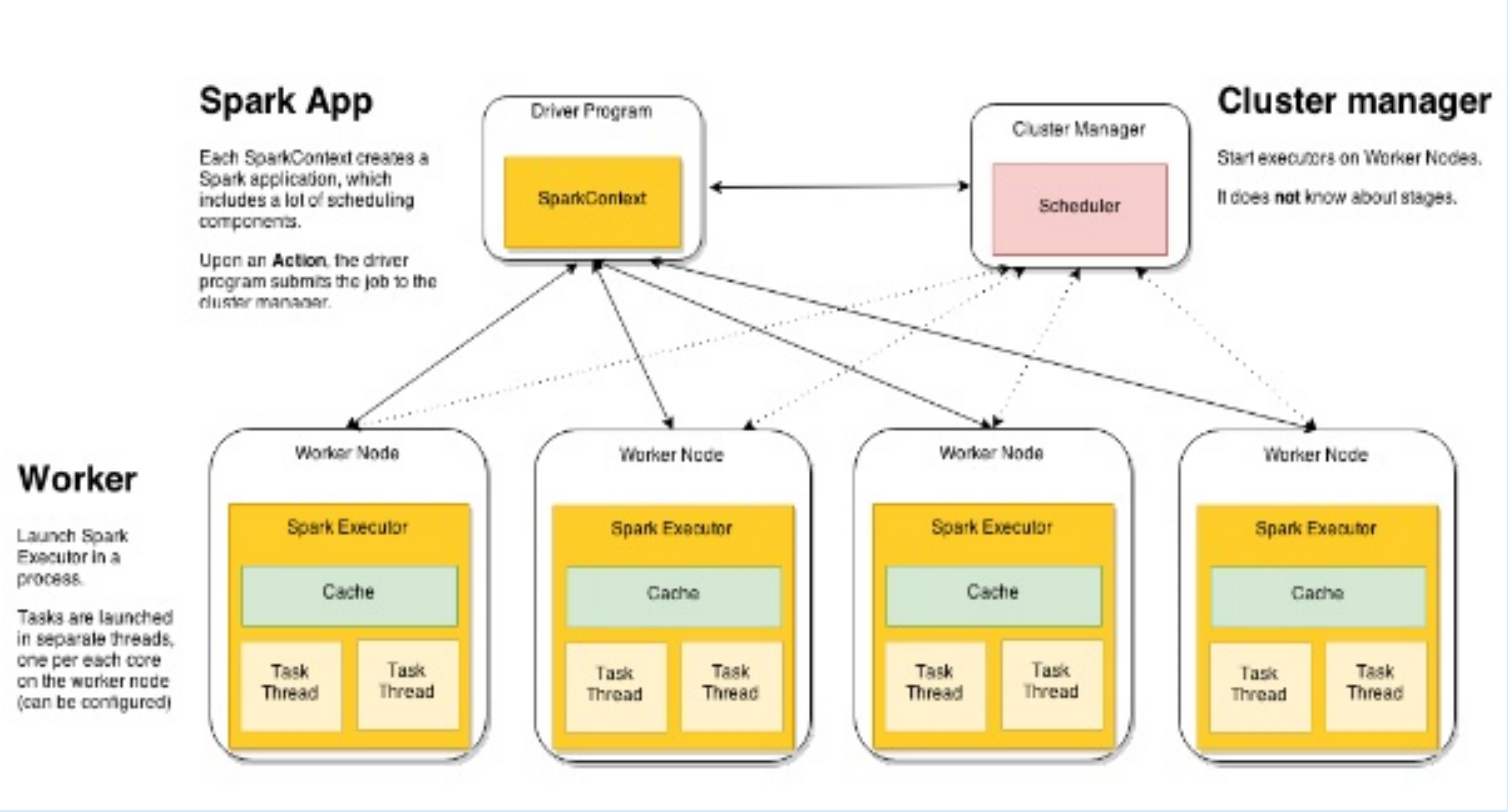
“If you can’t send the data to the program,
send the program to the data!”

At a high level, Hadoop distributes data across the cluster. When you want to process data, Hadoop passes your instructions along to all the computers (“nodes”) in the cluster, and they run the code. Only the answer is sent over the network to you.

High Level Architecture



System Level Architecture



Building Blocks

- Idea of Spark Programming is similar to Excel
  - Think of computation as data flow across multiple stages
  - At every point you have data on which you apply functions
- For this to happen, we need some tools
- Three building blocks
  - Spark Context
  - Resilient Distributed Datasets (RDD)
  - Operations – Transformations & Actions

RDD

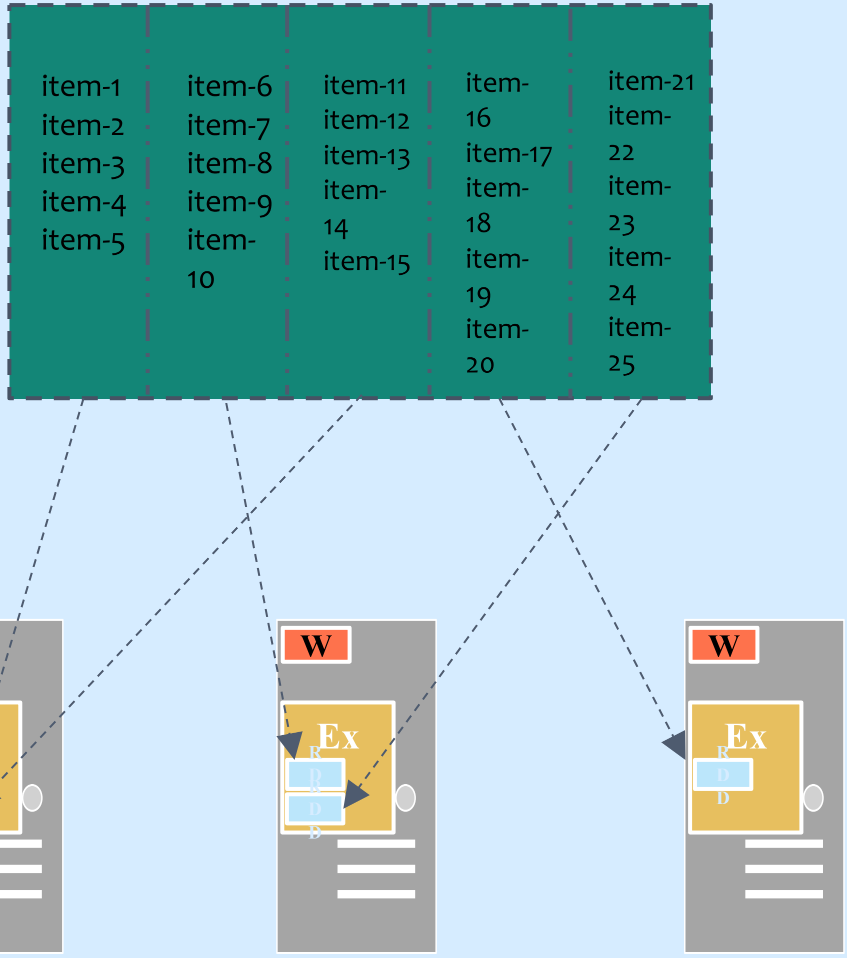
- Fundamental data unit in Spark (think DataFrames in R or Python)
- Resilient Distributed Dataset
  - Resilient – If data in memory is lost, it can be recreated
  - Distributed – stored in memory across the cluster
  - Dataset – data coming from either external files or from internal structures such as lists or other variables
- All functions work on RDD (or its variants)

RDD

- Simple View
  - Collection of data items split into partitions and stored in memory on worker nodes of cluster
- Complex View
  - Interface for data transformation
  - Data stored in either persisted store (HDFS, Cassandra, Hbase etc) or in cache (memory, memory+disks, disk only etc)

more partitions = more parallelism

RDD



More on RDD

- More like a container
- Can have any type of elements
  - Integer, Strings, Boolean
  - List, Dictionaries
  - Complex Objects (More for Java/Scala)

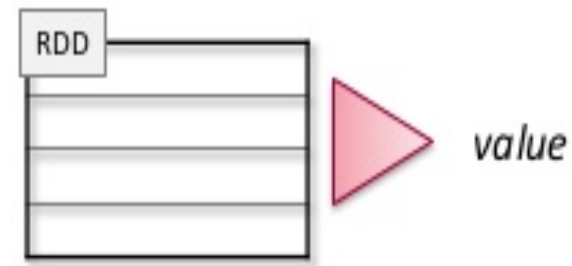
- There are some specific types of RDD as well
  - Key-value – Hold data in key-value pairs
  - Double RDD – Hold only numeric data
- But we won't concern ourselves with them

RDD Operations

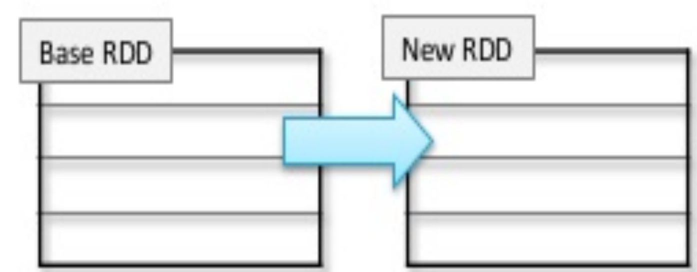
- Two types
  - Transformations
    - Apply code to distributed data in parallel
    - Convert one RDD to another
    - Work with actual data objects
  - Actions
    - Assemble final output from distributed data
    - Trigger entire workflow
    - Present results to the driver

RDD Operations

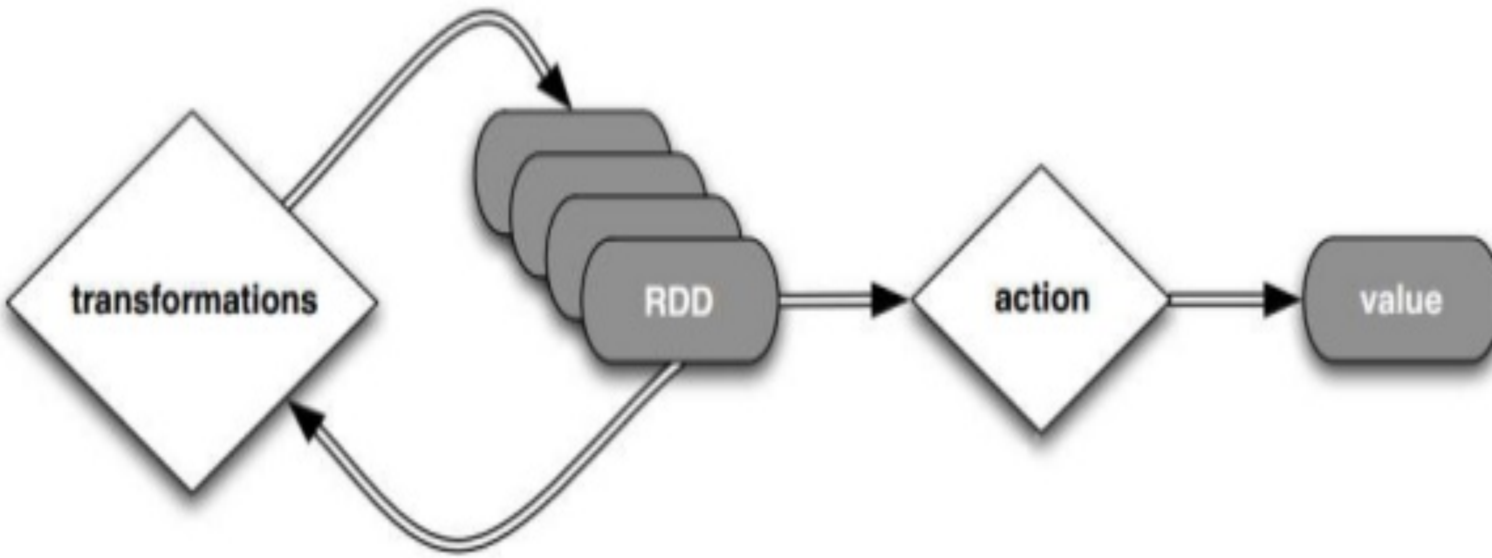
- **Two types of RDD operations**
  - Actions – return values
    - `count`
    - `take(n)`



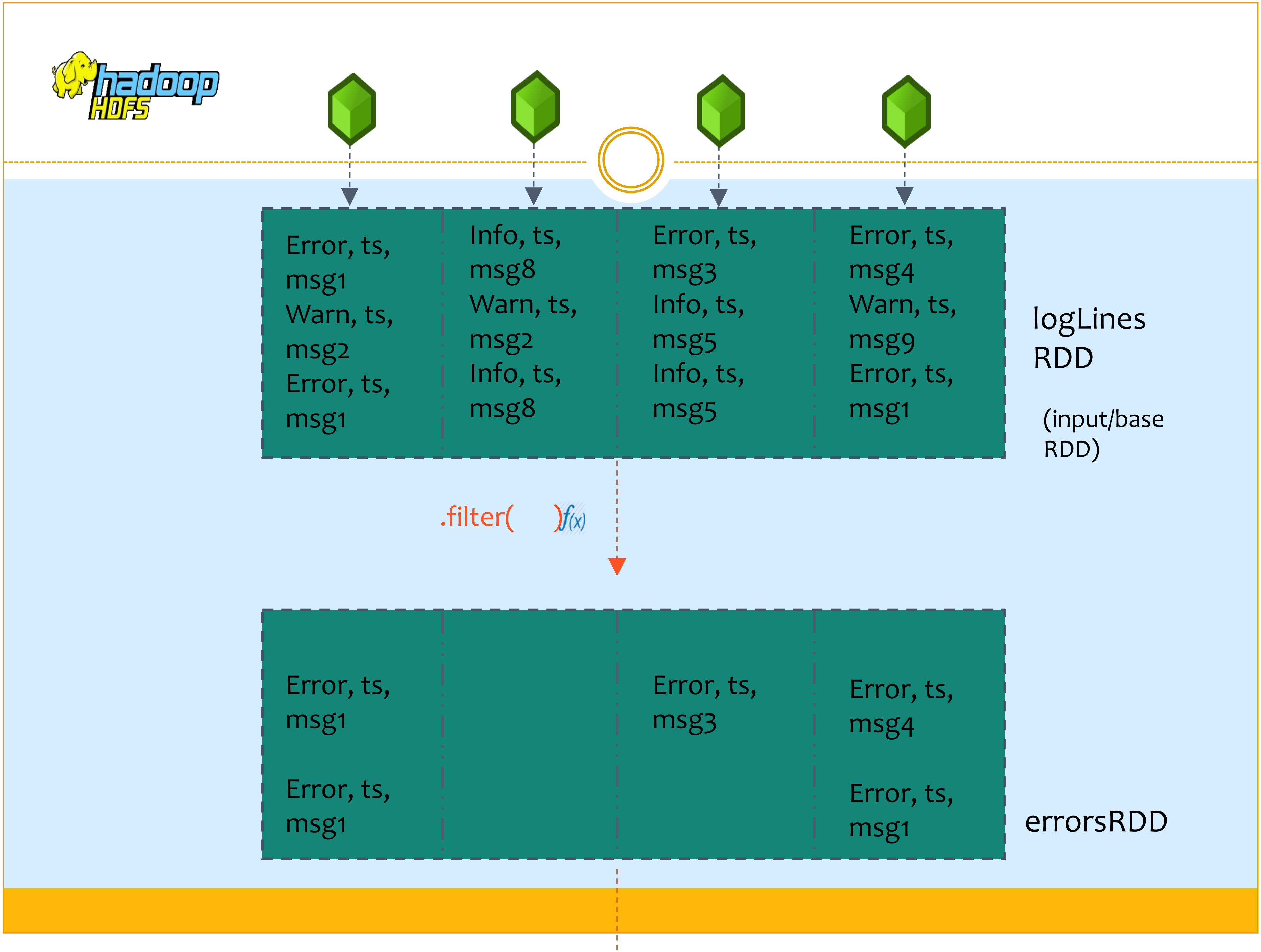
  - Transformations – define new RDDs based on the current one
    - `filter`
    - `map`
    - `reduce`



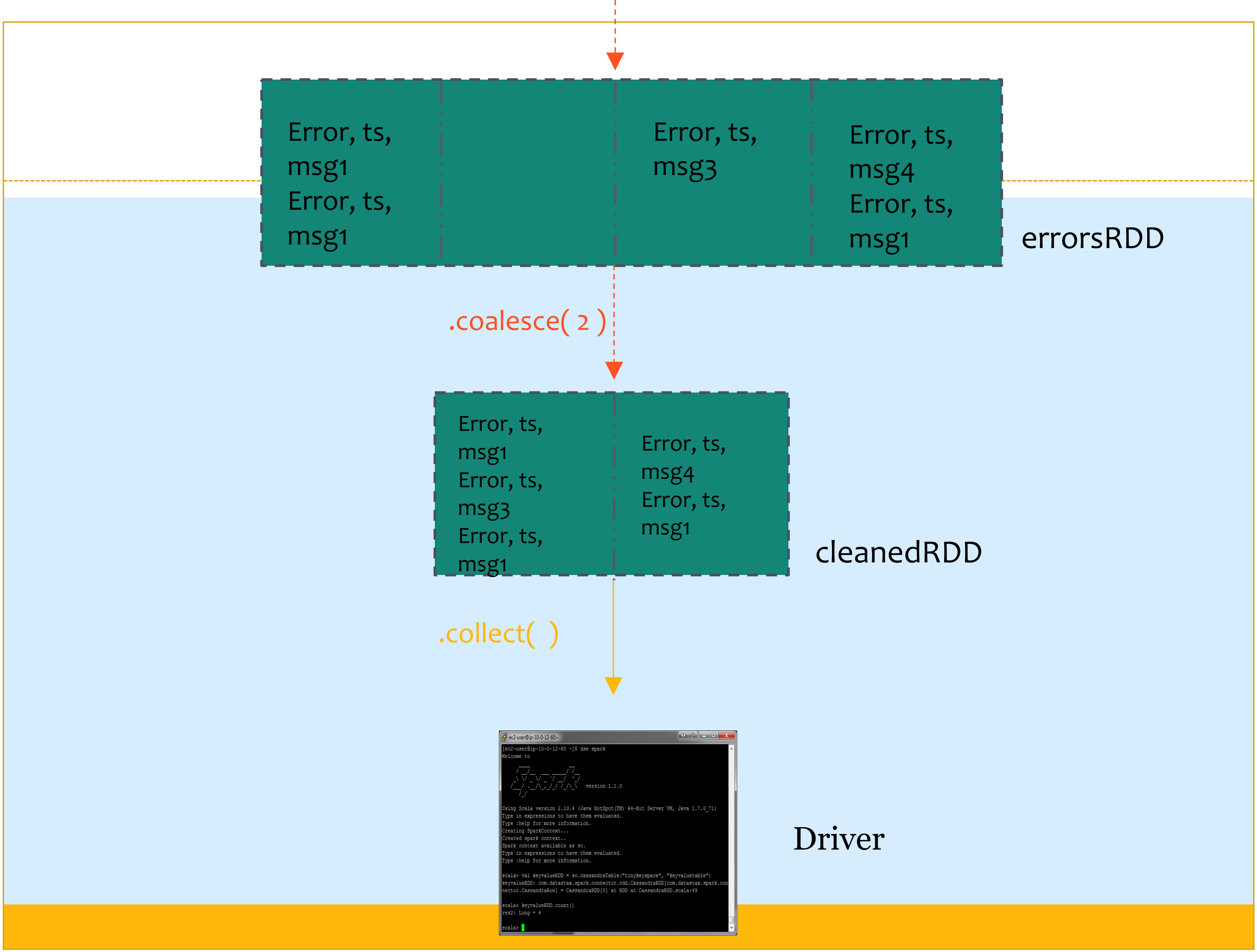
Lazy computations model



Transformation cause only metadata change

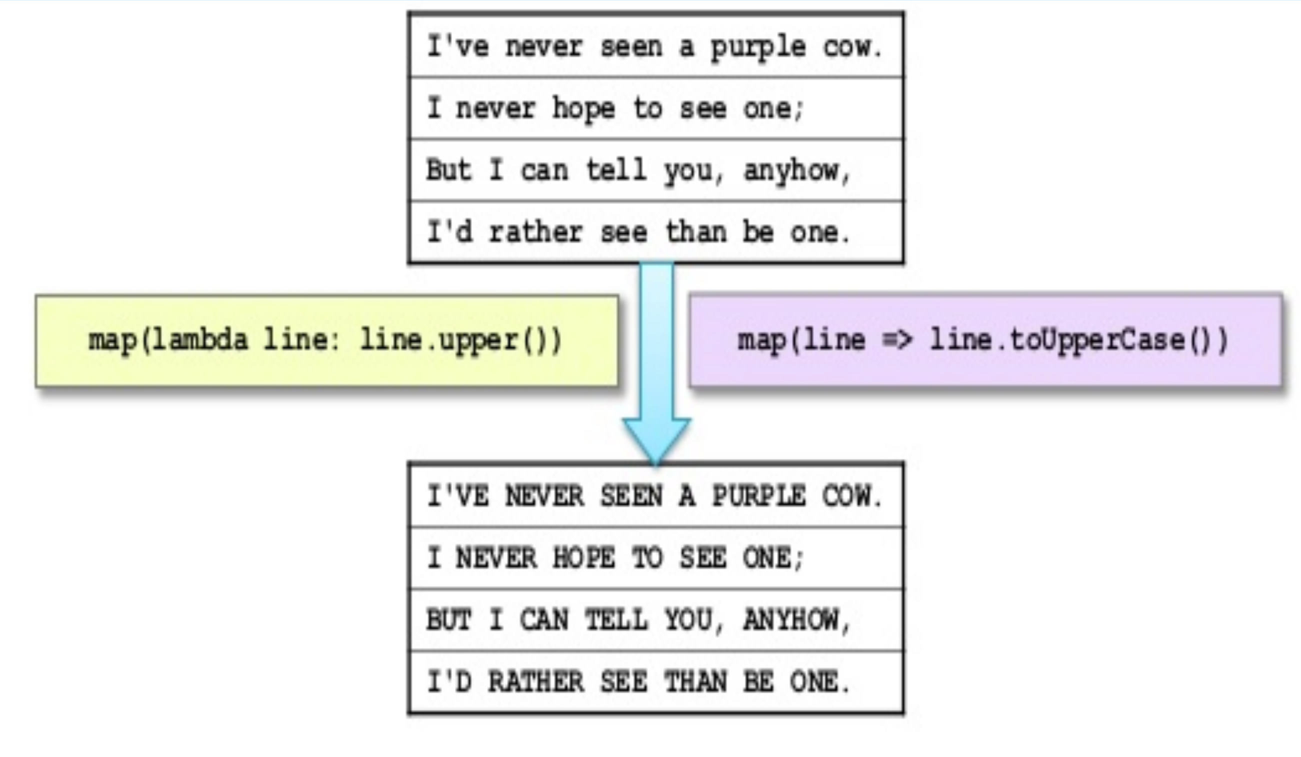


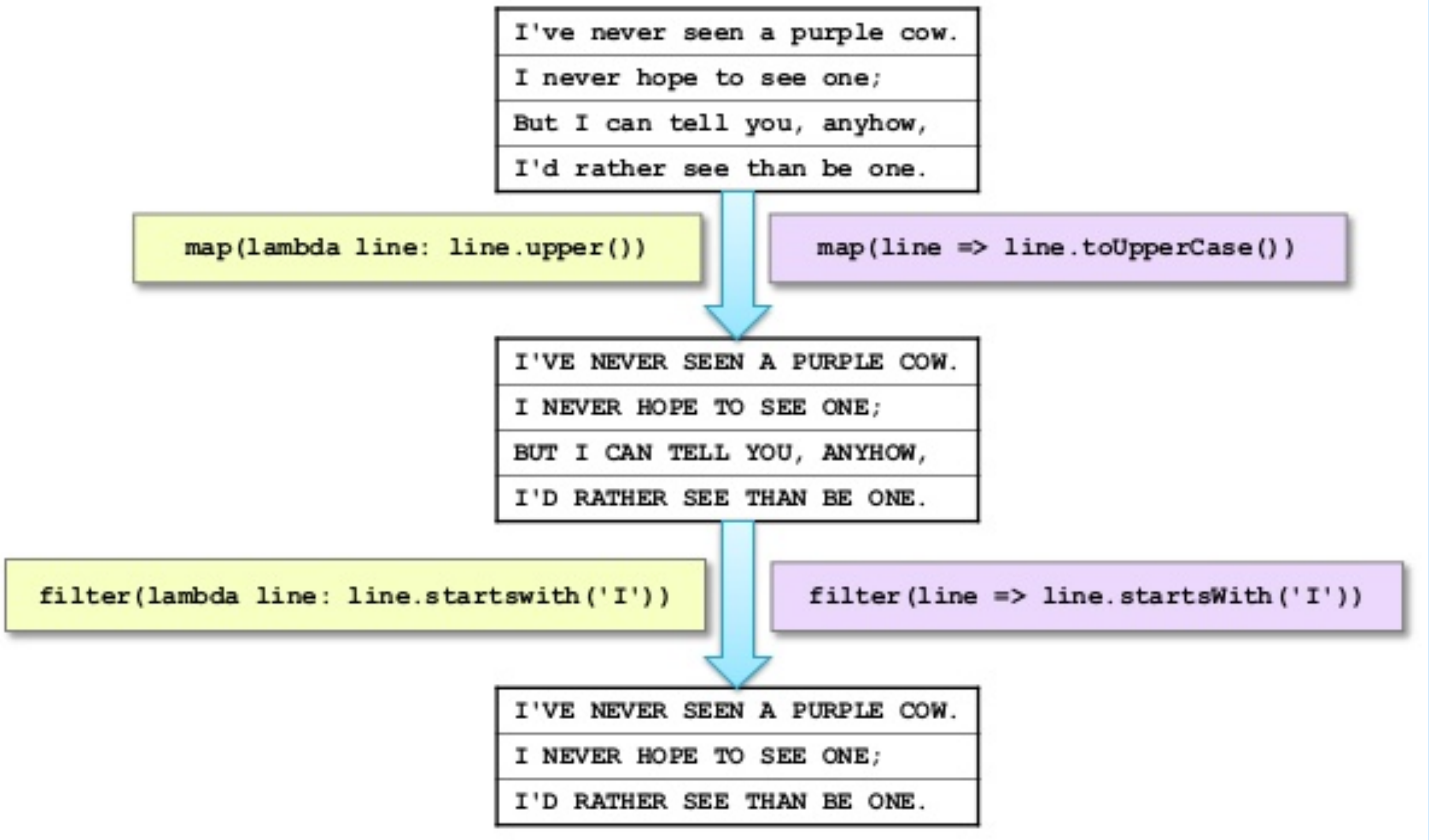
hadoop HDFS
Error, ts, msg1
Warn, ts, msg2
Error, ts, msg1
Info, ts, msg8
Warn, ts, msg2
Info, ts, msg8
Error, ts, msg3
Info, ts, msg5
Info, ts, msg5
Error, ts, msg4
Warn, ts, msg9
Error, ts, msg1
logLines RDD
(input/base RDD)
.filter($f(x)$)
Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD


Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce(2)
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver

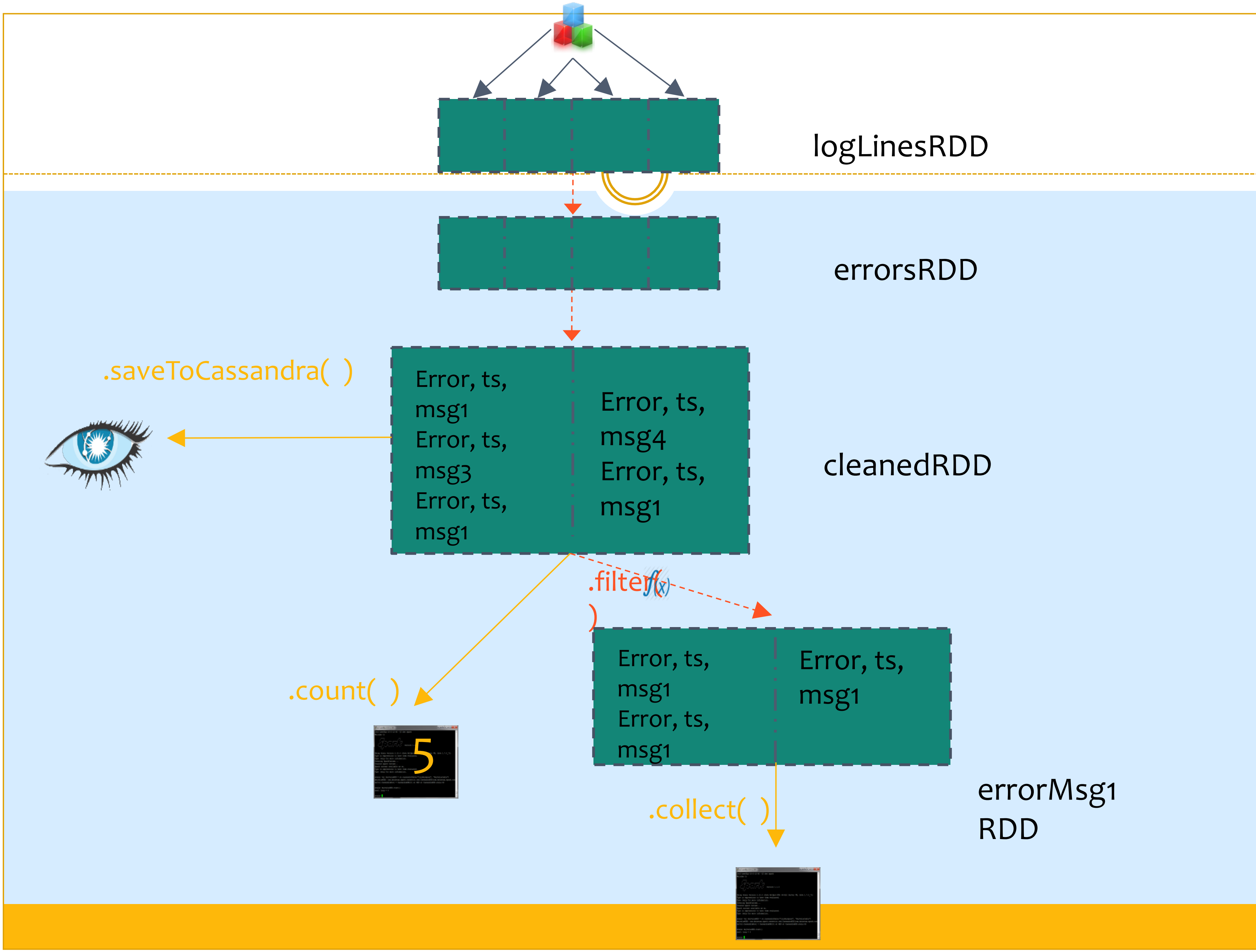
Another Example

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

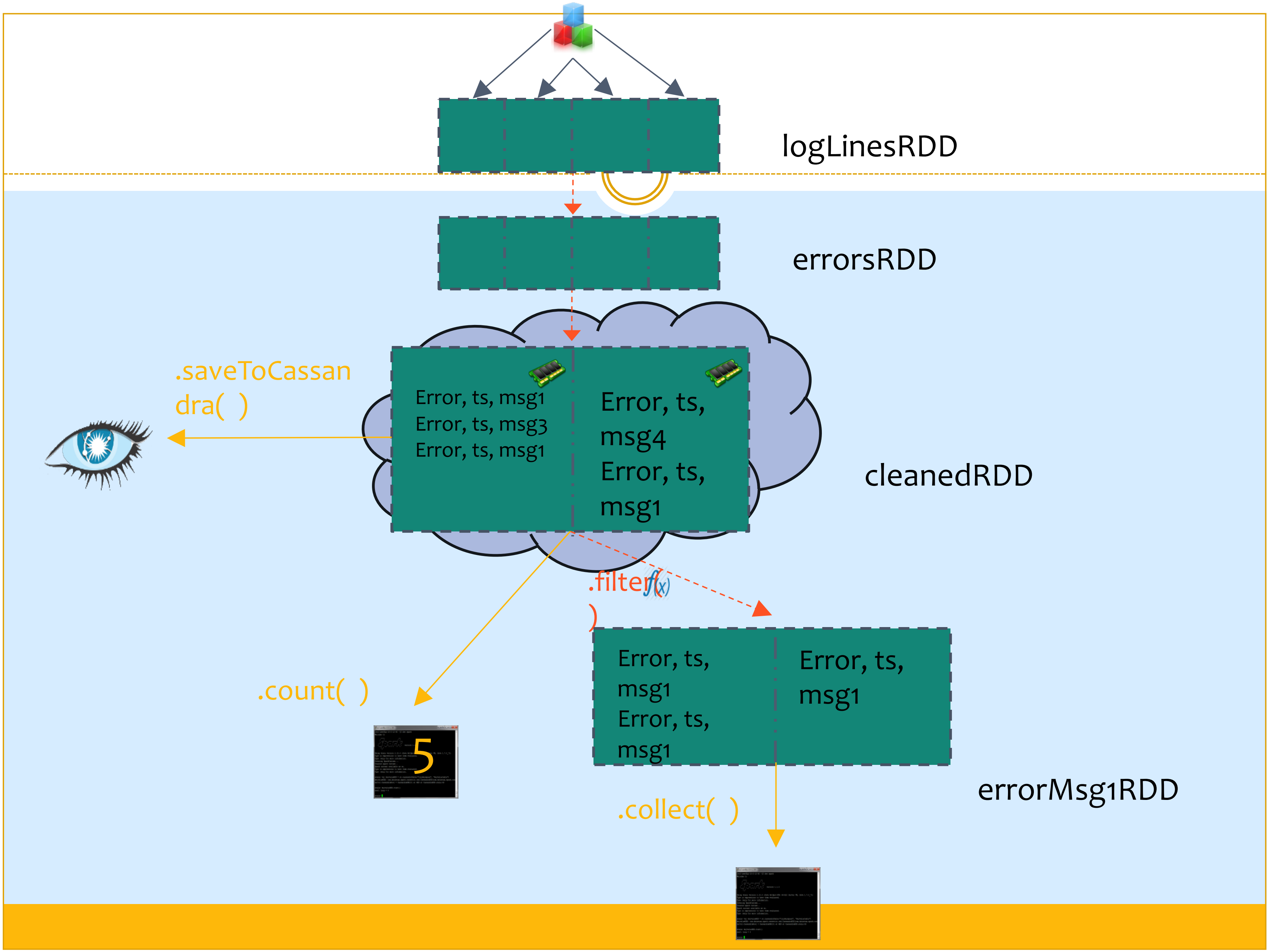

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
map(lambda line: line.upper())
map(line => line.toUpperCase())
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.


I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
map(lambda line: line.upper())
map(line => line.toUpperCase())
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
filter(lambda line: line.startswith('I'))
filter(line => line.startsWith('I'))
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

- What happens once action is called?
  - Data pipelines empty
  - RDD could exist, but no data
- If we want to save intermediate RDD?
  - Repeating process is wasteful
  - Caching is the solution



logLinesRDD
errorsRDD
.saveToCassandra( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter(f(x))
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1 RDD
.collect( )



logLinesRDD
errorsRDD
.saveToCassandra( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter( )
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
.collect( )
errorMsg1RDD

Why Use RDDs?

- Offer control and flexibility
- Low Level API
- Encourage more of “how-to” behavior
- When to use RDD?
  - Low level API and control of dataset
  - Dealing with unstructured data (media, images, text)
  - Manipulate data in complex manner (such as lambda function)
  - Don’t care about schema or structure of data
  - If you are ok if some sacrifices in performance

Problem?

- Encourage “how-to”, not “what-to”
- Not optimized by Spark – hence lower performance
- Slow for non-JVM languages such as Python

Putting it together – Lifecycle of Spark Program

- Create RDD
  - Parallelize
  - Read data from external sources
- Transform operations
- Optional – Cache RDD
- Actions

Code Execution

- Shuffle
  - Redistributes data among cluster of nodes by a criteria
  - Groups data into bucket (partitions)
  - Expensive
- Job
  - Set of computations
  - Application can have multiple jobs
- Stage
  - Collection of tasks
  - Stages could depend on each other
  - Shuffle boundaries

Code Execution

- Run Application
- Connect to cluster manager & get executors on worker nodes
- Spark splits jobs into stages
- Executors run tasks in parallel

Let's get started

- Launch Spark REPL
  - VM Distribution comes pre-installed with Spark
  - Can also install Spark as standalone on Windows
- For now, we will not worry much about installation
  - Focus on being able to program
  - Understand building blocks
- Multiple REPL (Python, Scala)

Ways of programming in Spark

- Similar to Python
  - Command Prompt
  - User Programs
- Spark Shell (Command Prompt version)
  - Good for learning or data exploration
  - Python, Scala
- User Programs (Applications)
  - For large scale data processing
  - Python, R, Java, Scala

Building Blocks

- Driver Program
  - Connects to, and communicates with Spark cluster
  - REPL is a driver program
  - Pushes work to a cluster and brings back

Spark Context

- Every Spark application needs a SparkContext
  - In REPL, it comes pre-started
  - In programs, you have to create one
- Main entry point to Spark library
- Connection to Spark cluster
- Apps create instance of SparkContext
  - One instance per app
  - sc = SparkContext()

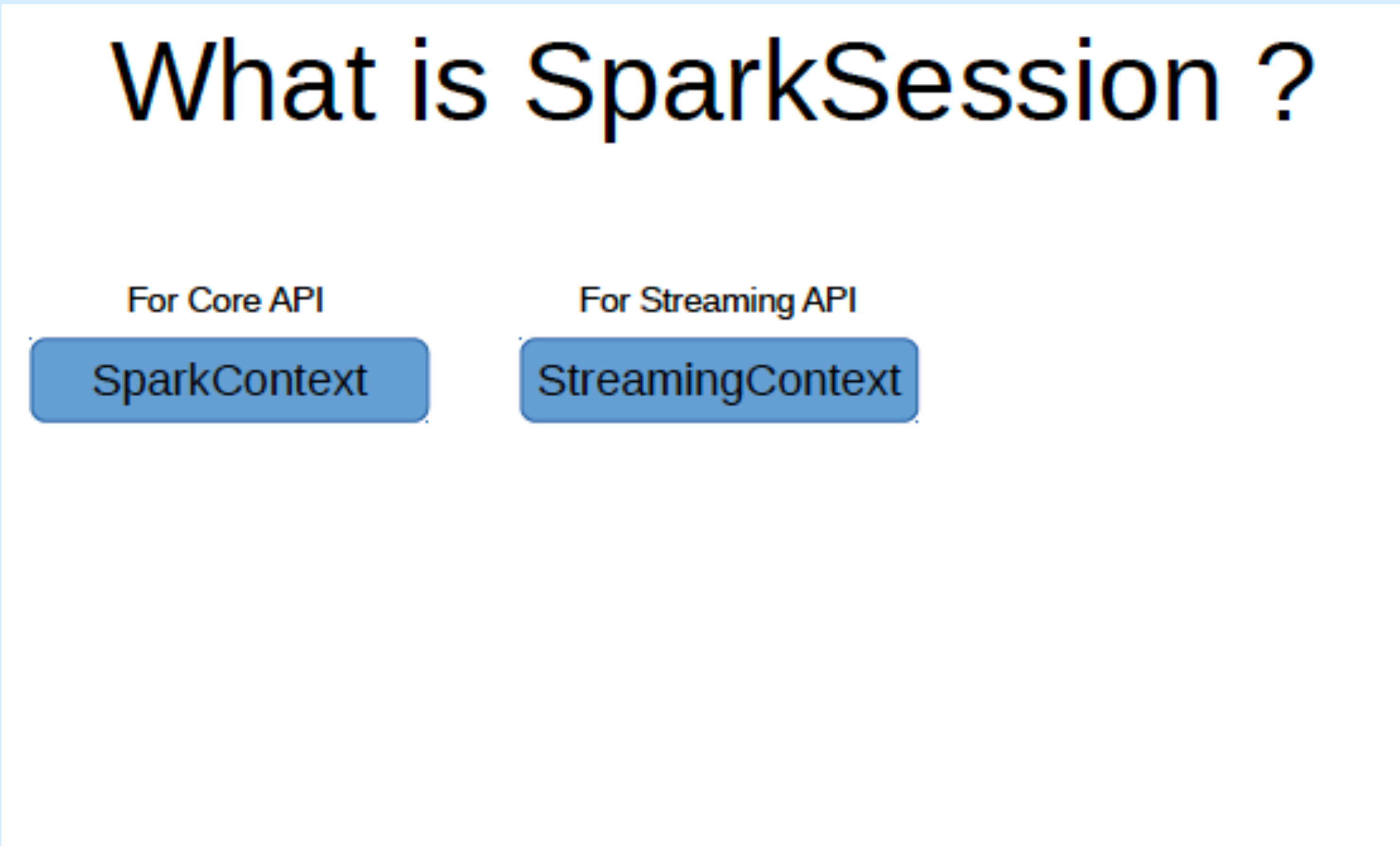
SparkSession – New in Spark 2

What is SparkSession ?

For Core API

SparkContext

SparkSession – New in Spark 2



SparkSession – New in Spark 2

What is SparkSession ?

For Core API

SparkContext

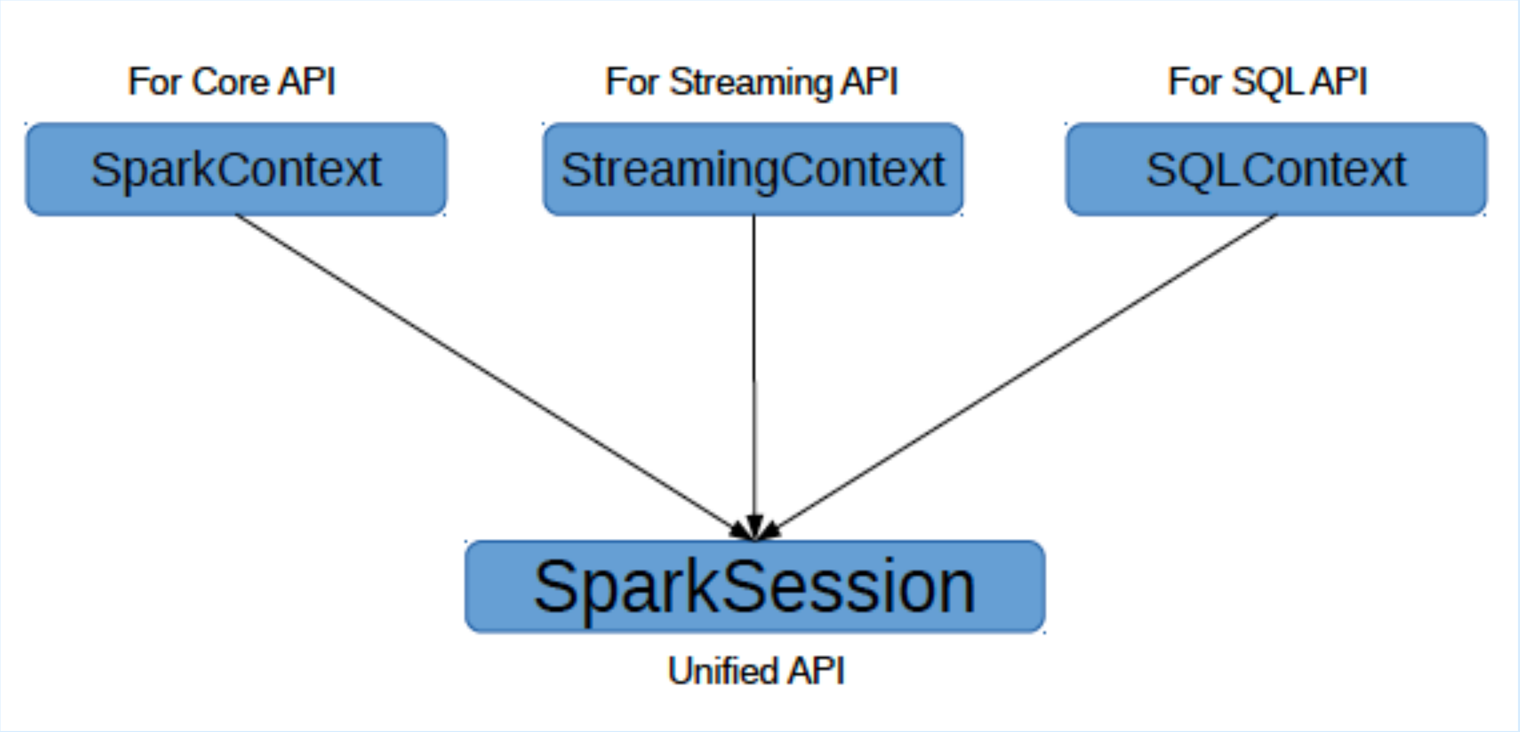
For Streaming API

StreamingContext

For SQL API

SQLContext

SparkSession – New in Spark 2



RDD

- Abstract representation of data
- Breaks data into partitions
- More partition -> More parallelism
- Distributed collection
- Immutable, partitioned, fault tolerant, strongly typed, in-memory
- Can be created in two ways
  - Parallelize a collection – In Memory
  - Read data from external data sources

Example

- Parallelize in Python
  - wordsRDD = sc.parallelize([“fish", “cats“, “dogs”])
- Read a local txt file in Python
  - linesRDD = sc.textFile("/path/to/README.md")

Some common functions

map	reduce	take
filter	count	first
groupBy	fold	partitionBy
union	reduceByKey	pipe
join	groupByKey	distinct
leftOuterJoin	cogroup	save
rightOuterJoin	flatMap	...

Commonly used Transformations

- Work element by element (Most, not all)
- General
  - Map(), filter(), flatMap(), groupBy(), sortBy()
- Math
  - sample(), random(), union(), intersection(), distinct(), subtract(), cartesian()

Commonly used Actions

- General
  - Reduce(), collect(), first(), aggregate()
- Math
  - count(), min(), max(), stdev(), variance()
- IO
  - saveToCassandra(), countByKey(), foreach()

First lines of code

- Let's discuss some commonly used transformations
- Map
  - applies a function to each element of RDD and returns a new RDD
- Python:
  - x = sc.parallelize([“a”,”b”,”c”])
  - Y = x.map(lambda z: (z,z))
  - Print(x.collect())
  - Print(y.collect())

Transformation - Filter

- Filter
  - Keeps an element if condition is true.
- Python
  - x = sc.parallelize([4,5,6,8])
  - Y = x.filter(lambda x:x-5==1)
  - Print(x.collect())
  - Print(y.collect())

Transformation - FlatMap

- Return a new RDD by applying a function to all elements of RDD, and then flattening the results
- Python
  - sentencesRDD = sc.parallelize(['Hello world', 'My name is Peeyush'])
  - wordsRDD = sentencesRDD.flatMap(lambda sentence: sentence.split(" "))
  - print(wordsRDD.collect())
  - print(wordsRDD.count())

Transformations - Intersection

- Takes two RDD as input and returns a new RDD that has common elements
- Python
  - numbersRDD = sc.parallelize([1,2,3])
  - moreNumbersRDD = sc.parallelize([2,3,4])
  - numbersRDD.intersection(moreNumbersRDD).collect()

Transformations - GroupBy

- Similar to groupby in SQL.
- Groups the data, and creates key,value pairs
- Python
  - x = sc.parallelize([‘Ajay’,’Amit’,’Manav’,’Manish’,’Sonam’])
  - Y = x.groupBy(lambda w: w[0])
  - Print [(key, list(value)) for (key,value) in y.collect()]
- Scala
  - val x = sc.parallelize(Array(“Ajay”,”Amit”,”Manav”,”Manish”,”Sonam”])
  - val y = x.groupBy(w => w.charAt(0))
  - println (y.collect().mkString(“,”))

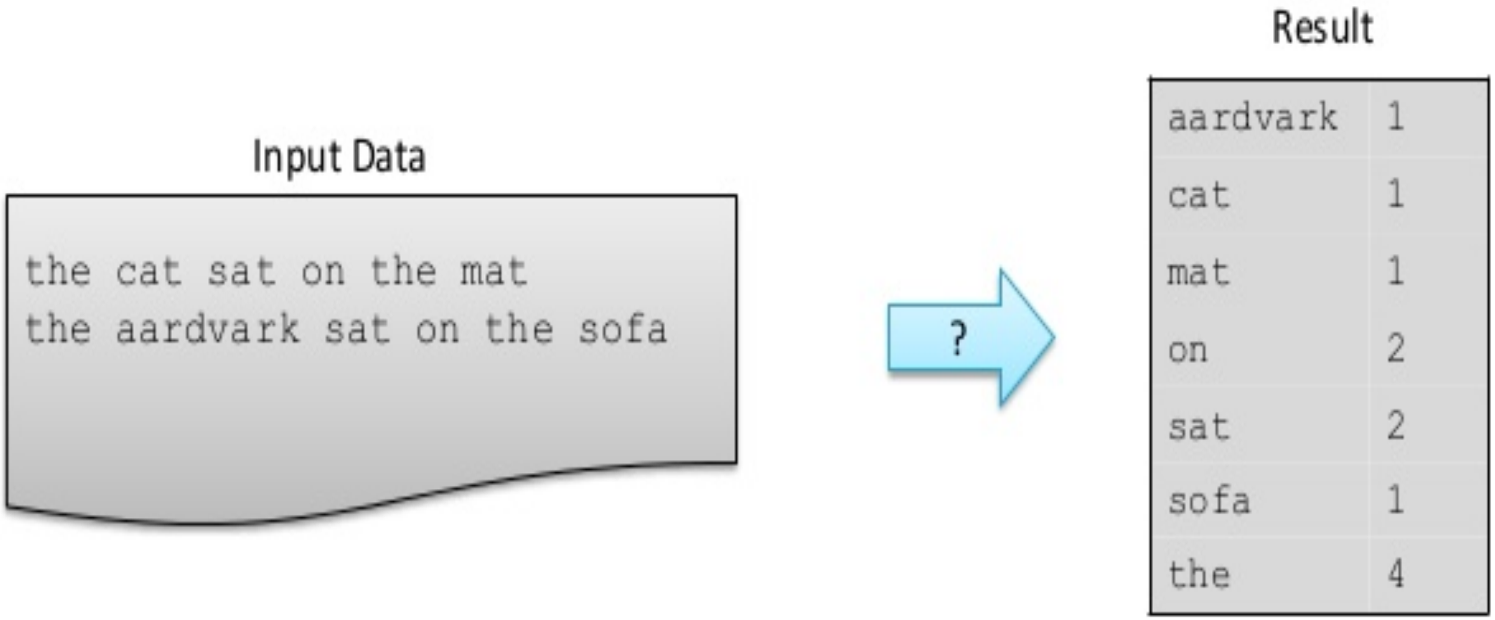
Actions - collect

- Returns elements as array
  - rdd = sc.parallelize((1 to 10000).toList)
  - filteredRdd = rdd filter { x => (x % 1000) == 0 }
  - filterResult = filteredRdd.collect
  - total = rdd.count

Actions - reduce

- Aggregates elements by a rule
  - rdd = sc.parallelize(range(1, 10+1))
  - rdd.reduce(lambda x, y: x * y)

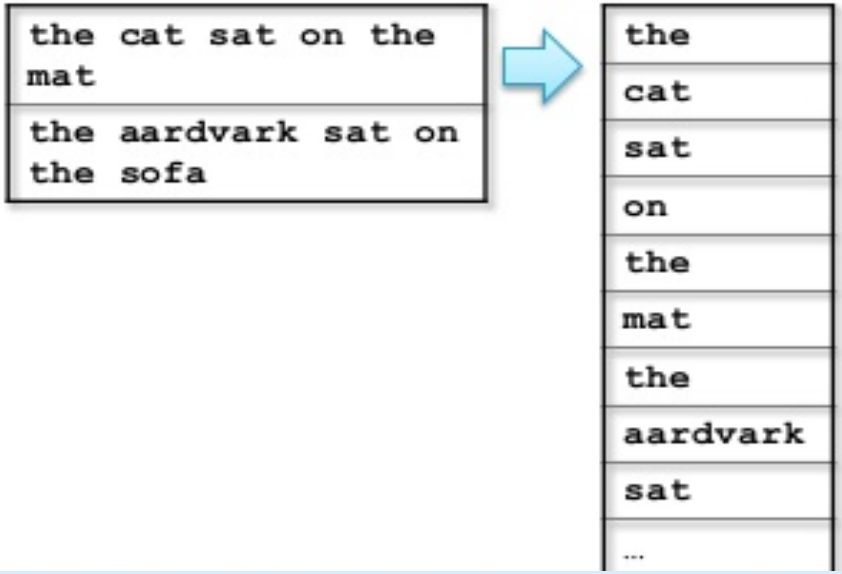
Word Count Example

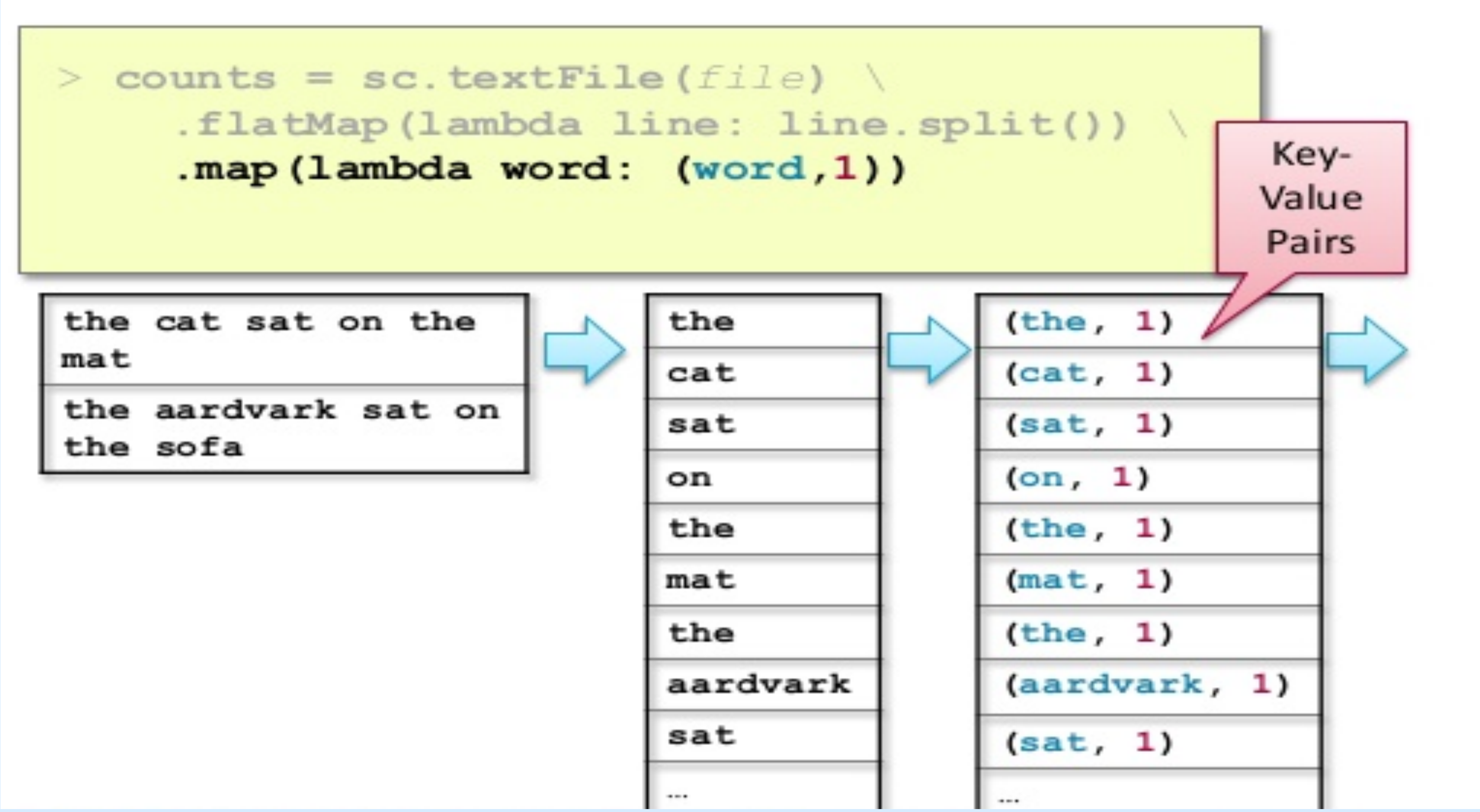


```
> counts = sc.textFile(file)
```

the cat sat on the mat
the aardvark sat on the sofa

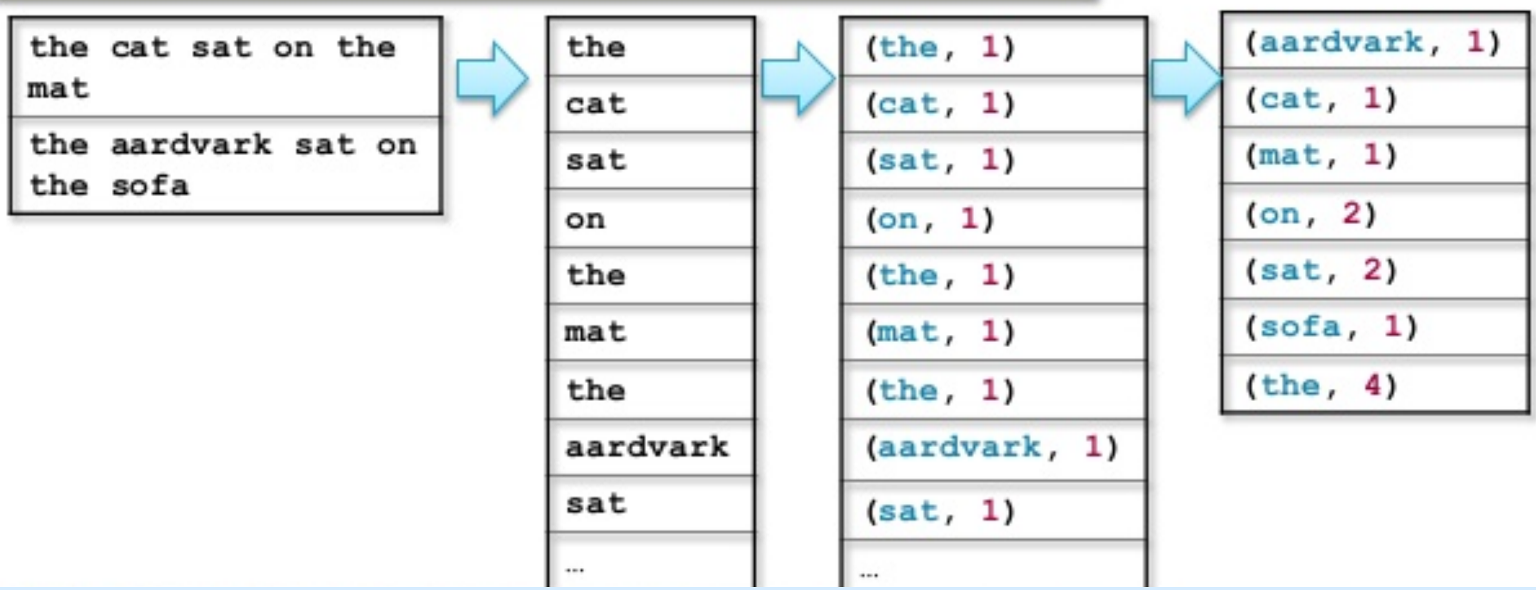
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



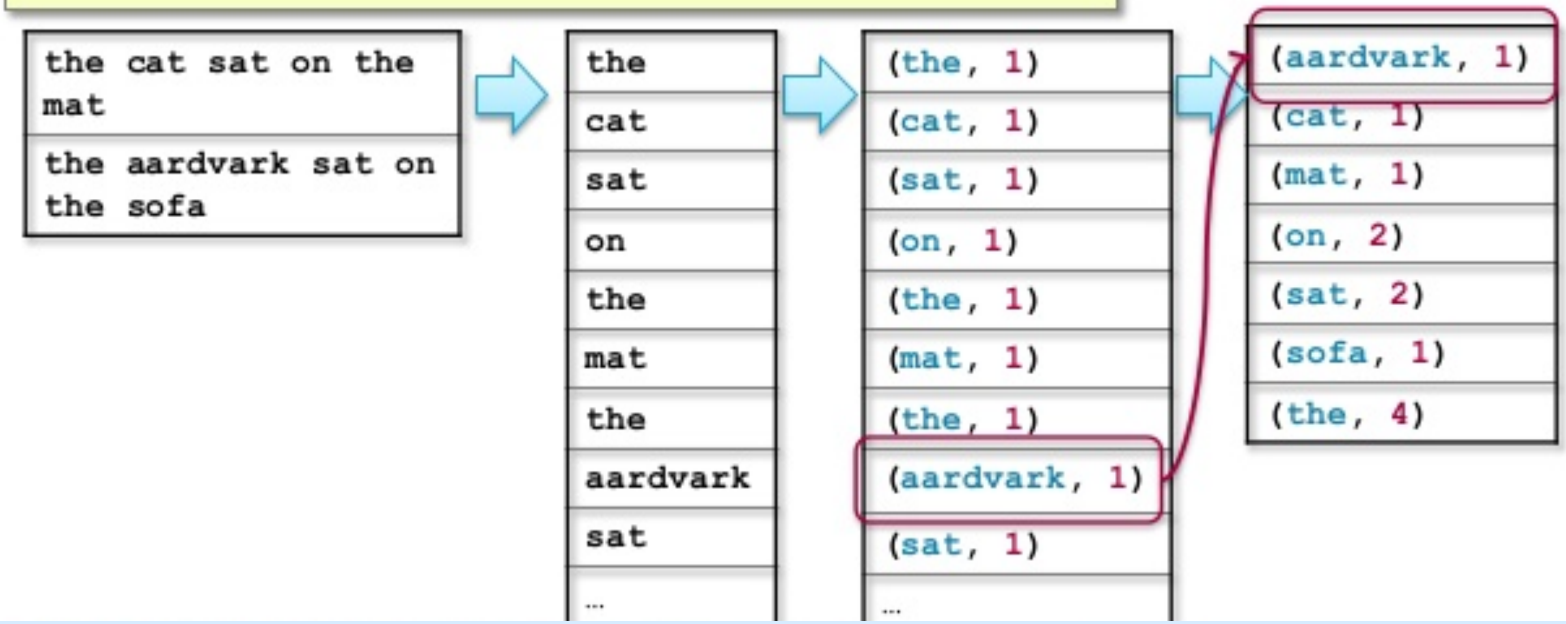


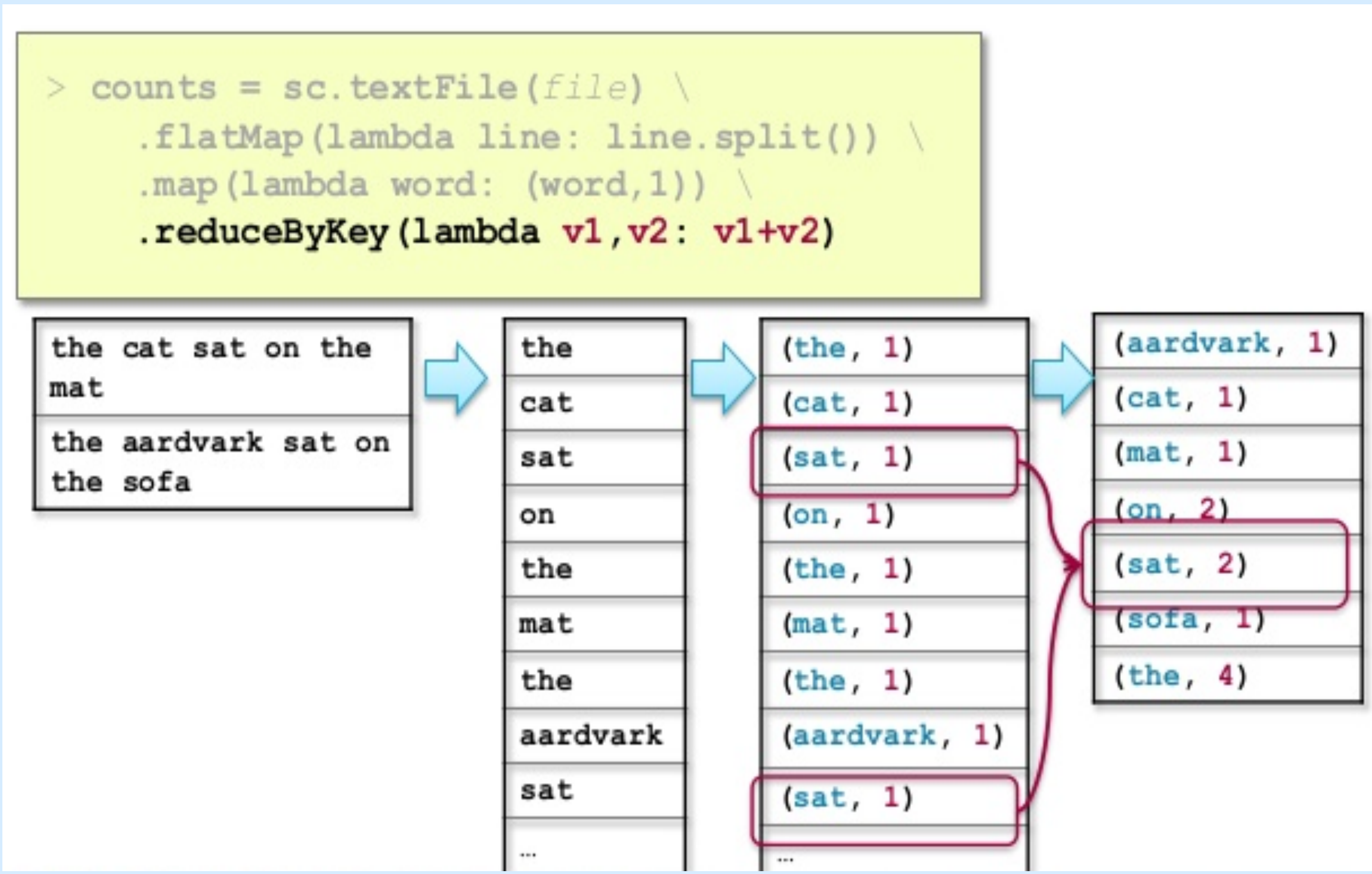
> counts = sc.textFile(file) \
.flatMap(lambda line: line.split()) \
.map(lambda word: (word,1))
Key-Value Pairs
the cat sat on the mat
the aardvark sat on the sofa
the
cat
sat
on
the
mat
the
aardvark
sat
...
(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
(sat, 1)
...

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



```
> counts = sc.textFile(file) \
     .flatMap(lambda line: line.split()) \
     .map(lambda word: (word,1)) \
     .reduceByKey(lambda v1,v2: v1+v2)
```

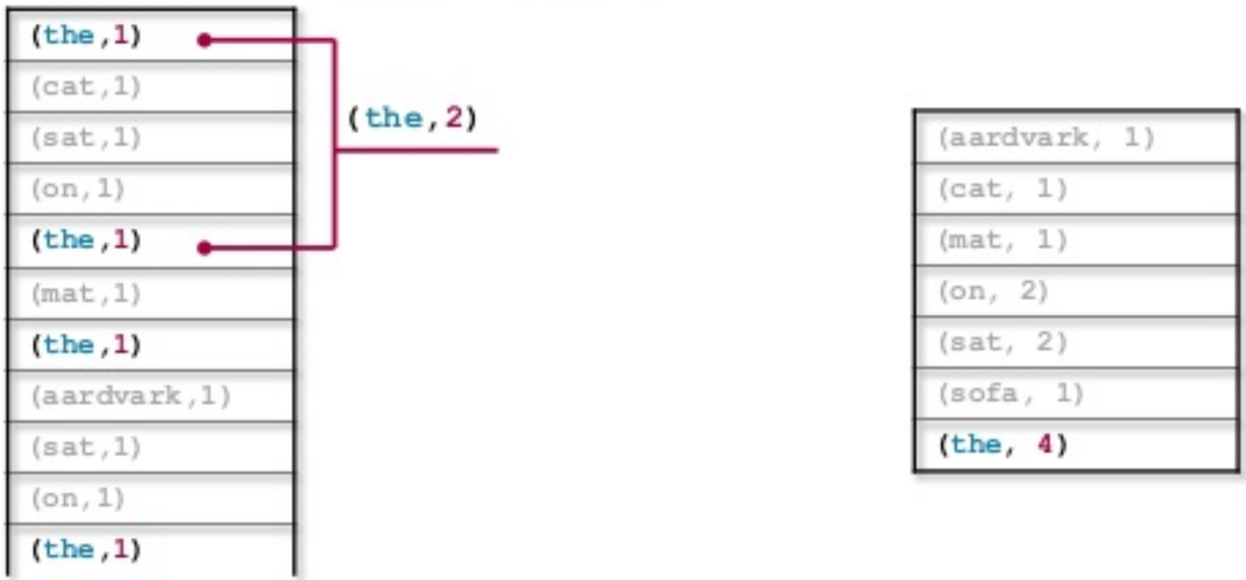



> counts = sc.textFile(file) \
.flatMap(lambda line: line.split()) \
.map(lambda word: (word,1)) \
.reduceByKey(lambda v1,v2: v1+v2)
the cat sat on the mat
the aardvark sat on the sofa
the
cat
sat
on
the
mat
the
aardvark
sat
...
(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
(sat, 1)
...
(aardvark, 1)
(cat, 1)
(mat, 1)
(on, 2)
(sat, 2)
(sofa, 1)
(the, 4)

ReduceByKey

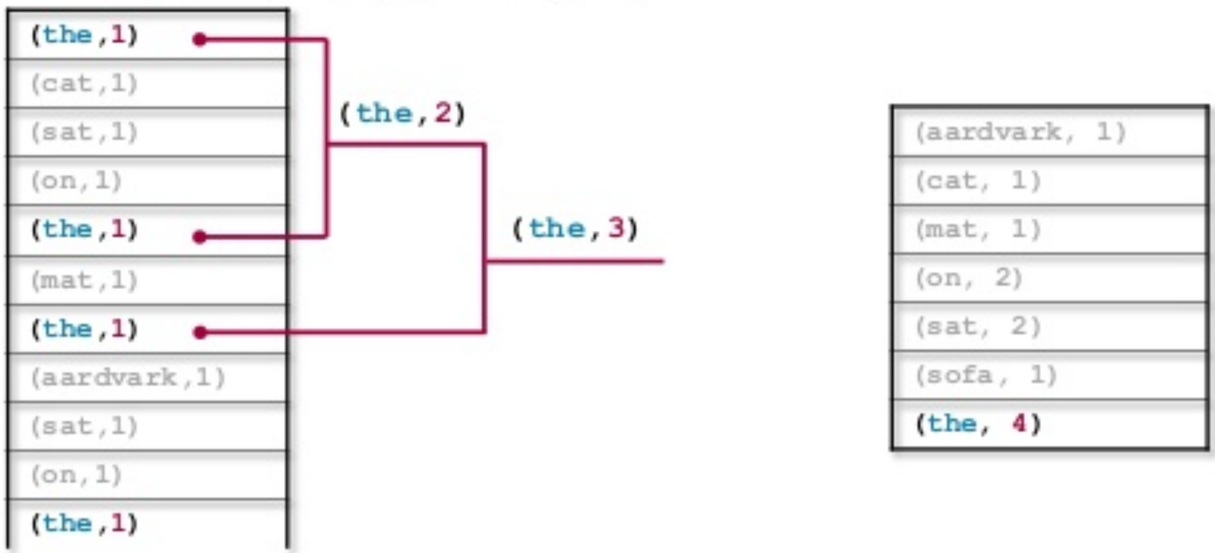
- **ReduceByKey functions must be**
  - Binary – combines values from two keys
  - Commutative – x+y = y+x
  - Associative – (x+y)+z = x+(y+z)

```
> counts = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   .map(lambda word: (word,1)) \
   .reduceByKey(lambda v1,v2: v1+v2)
```



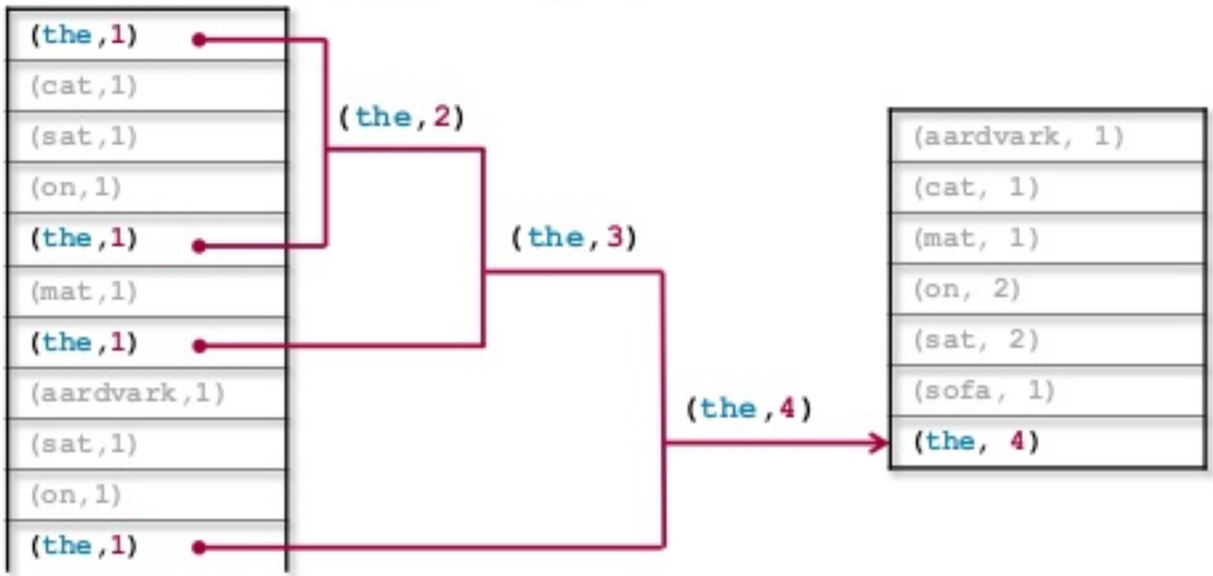
- **ReduceByKey functions must be**
  - Binary – combines values from two keys
  - Commutative – x+y = y+x
  - Associative – (x+y)+z = x+(y+z)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



- **ReduceByKey functions must be**
  - Binary – combines values from two keys
  - Commutative – x+y = y+x
  - Associative – (x+y)+z = x+(y+z)

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



THANK YOU