# CS5412 / Lecture 25
# Apache Spark and RDDs

Kishore Pusukuri,
Spring 2019

## Recap

**MapReduce**

- For easily writing applications to process vast amounts of data in-parallel on large clusters in a reliable, fault-tolerant manner
- Takes care of scheduling tasks, monitoring them and re-executes the failed tasks

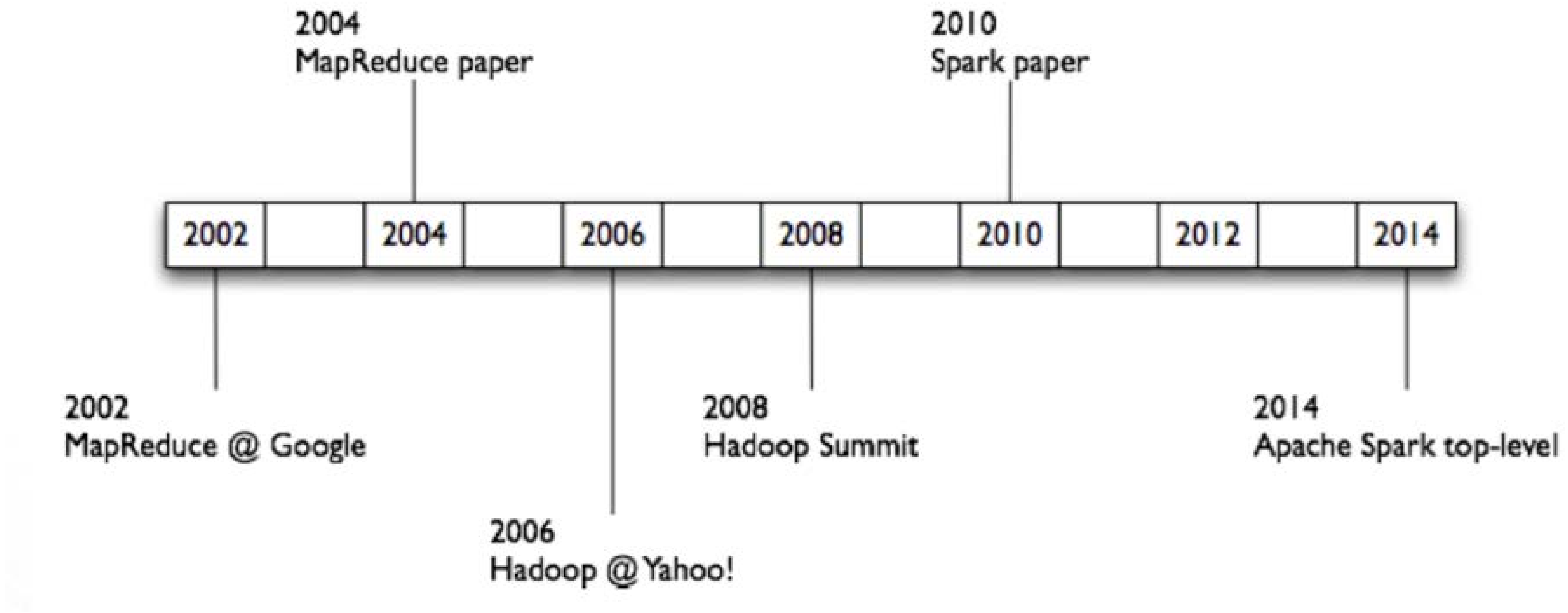
**HDFS & MapReduce**: Running on the same set of nodes → compute nodes and storage nodes same (keeping data close to the computation) → very high throughput

**YARN & MapReduce**: A single master resource manager, one slave node manager per node, and AppMaster per application

## Today’s Topics

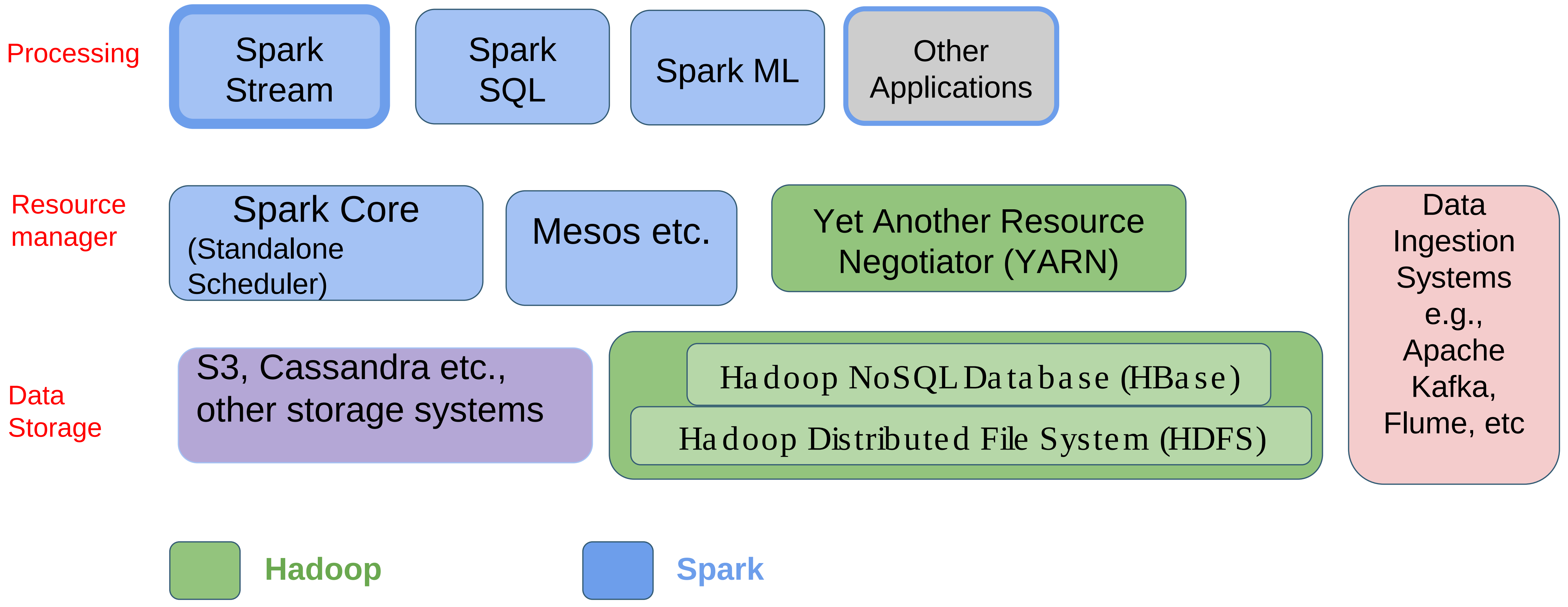
- Motivation
- Spark Basics
- Spark Programming

## History of Hadoop and Spark

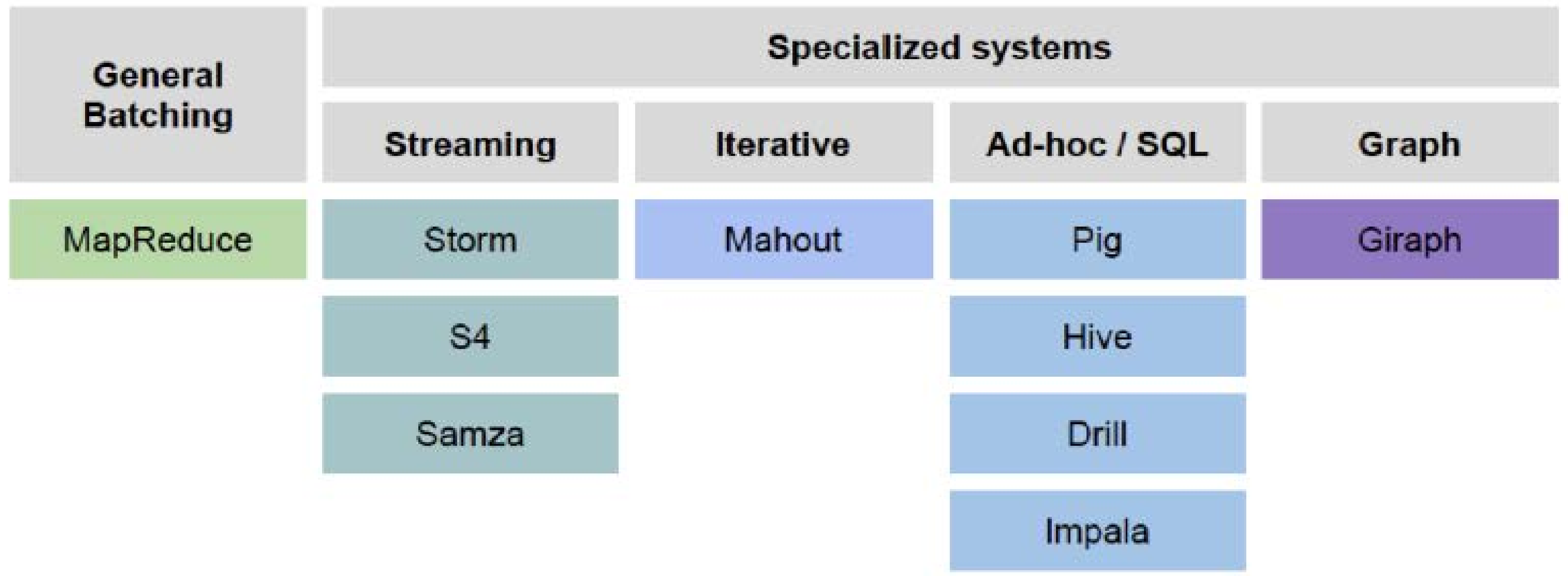


# Apache Spark

** Spark can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos or YARN)


Processing
Spark Stream
Spark SQL
Spark ML
Other Applications
Resource manager
Spark Core (Standalone Scheduler)
Mesos etc.
Yet Another Resource Negotiator (YARN)
Data Ingestion Systems e.g., Apache Kafka, Flume, etc
Data Storage
S3, Cassandra etc., other storage systems
Hadoop NoSQL Database (HBase)
Hadoop Distributed File System (HDFS)
Hadoop
Spark

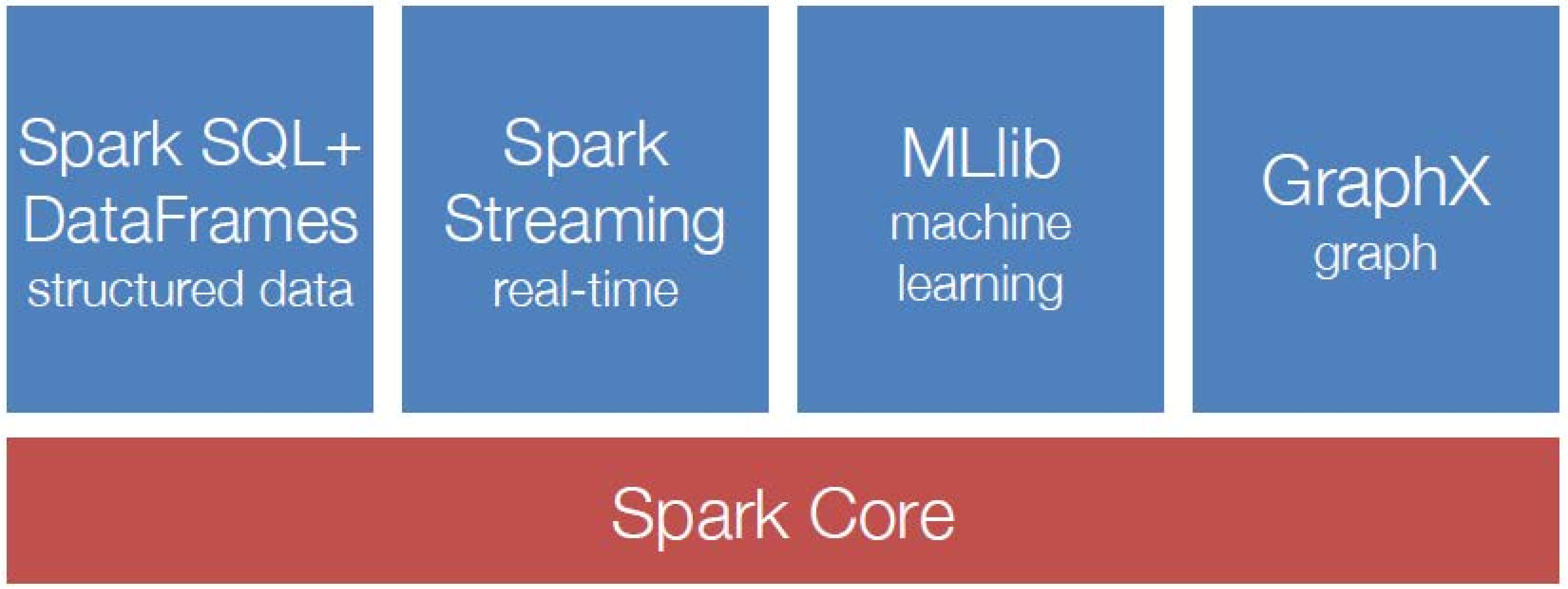
# Apache Hadoop Lack a Unified Vision



| General Batching | Specialized systems | | | |
|---|---|---|---|---|
| | Streaming | Iterative | Ad-hoc / SQL | Graph |
| MapReduce | Storm | Mahout | Pig | Giraph |
| | S4 | | Hive | |
| | Samza | | Drill | |
| | | | Impala | |

- Sparse Modules
- Diversity of APIs
- Higher Operational Costs

## Spark Ecosystem: A Unified Pipeline


Spark SQL+ DataFrames structured data
Spark Streaming real-time
MLlib machine learning
GraphX graph
Spark Core

Note: Spark is <u>not</u> designed for IoT real-time. The streaming layer is used for continuous input streams like financial data from stock markets, where events occur steadily and must be processed as they occur. But there is no sense of direct I/O from sensors/actuators. For IoT use cases, Spark would not be suitable.

## Key ideas

In Hadoop, each developer tends to invent his or her own style of work

With Spark, serious effort to standardize around the idea that people are writing parallel code that often runs for many “cycles” or “iterations” in which a lot of reuse of information occurs.

Spark centers on Resilient Distributed Dataset, RDDs, that capture the information being reused.

## How this works

You express your application as a graph of RDDs.

The graph is only evaluated as needed, and they only compute the RDDs actually needed for the output you have requested.

Then Spark can be told to cache the reuseable information either in memory, in SSD storage or even on disk, based on *when* it will be needed again, *how big it is*, and *how costly it would be to recreate.*

You write the RDD logic and control all of this via hints

## Motivation (1)

**MapReduce**: The original scalable, general, processing engine of the Hadoop ecosystem

- Disk-based data processing framework (HDFS files)
- Persists intermediate results to disk
- Data is reloaded from disk with every query → Costly I/O
- Best for ETL like workloads (batch processing)
- Costly I/O → Not appropriate for iterative or stream processing workloads

# Motivation (2)

**Spark**: General purpose computational framework that substantially improves performance of MapReduce, but retains the basic model

- Memory based data processing framework → avoids costly I/O by keeping intermediate results in memory
- Leverages distributed memory
- Remembers operations applied to dataset
- Data locality based computation → High Performance
- Best for both iterative (or stream processing) and batch workloads

# Motivation - Summary

Software engineering point of view

- Hadoop code base is huge
- Contributions/Extensions to Hadoop are cumbersome
- Java-only hinders wide adoption, but Java support is fundamental

System/Framework point of view

- Unified pipeline
- Simplified data flow
- Faster processing speed

Data abstraction point of view

- New fundamental abstraction RDD
- Easy to extend with new operators
- More descriptive computing model

## Today’s Topics

- Motivation
- Spark Basics
- Spark Programming

## Spark Basics(1)

Spark: Flexible, in-memory data processing framework written in Scala

Goals:

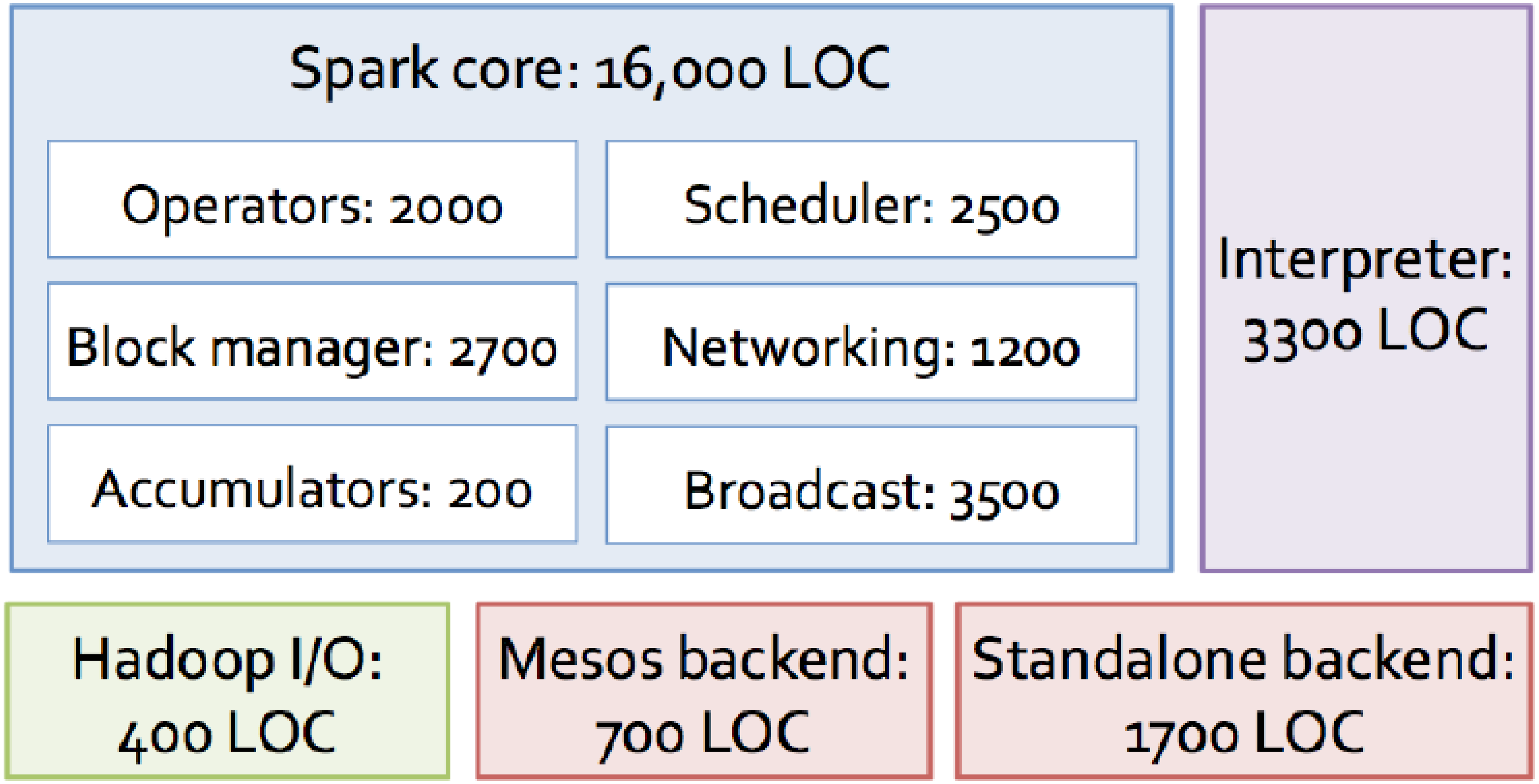
- Simplicity (Easier to use):
  - Rich APIs for Scala, Java, and Python
- Generality: APIs for different types of workloads
  - Batch, Streaming, Machine Learning, Graph
- Low Latency (Performance): In-memory processing and caching
- Fault-tolerance: Faults shouldn’t be special case

## Spark Basics(2)

There are two ways to manipulate data in Spark

- Spark Shell:
  - Interactive – for learning or data exploration
  - Python or Scala
- Spark Applications
  - For large scale data processing
  - Python, Scala, or Java

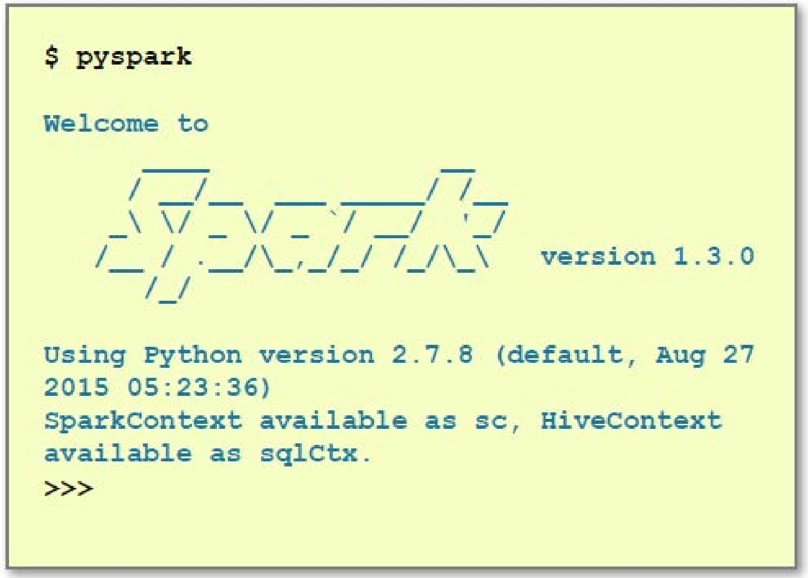
# Spark Core: Code Base (2012)



# Spark Shell

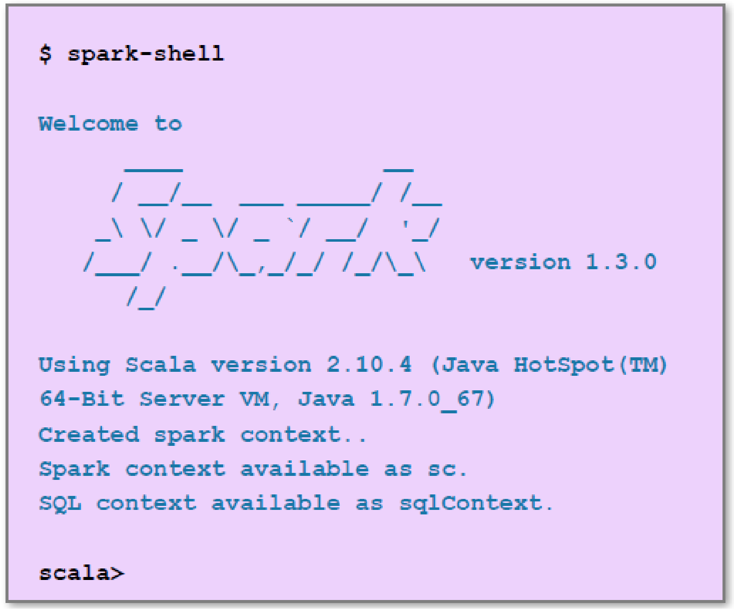
The Spark Shell provides interactive data exploration (REPL)

Python Shell: **pyspark**



```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Python version 2.7.8 (default, Aug 27
2015 05:23:36)
SparkContext available as sc, HiveContext
available as sqlCtx.
>>>
```

Scala Shell: **spark-shell**



```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_67)
Created spark context..
Spark context available as sc.
SQL context available as sqlContext.

scala>
```

REPL: Repeat/Evaluate/Print Loop

# Spark Fundamentals

Example of an application:

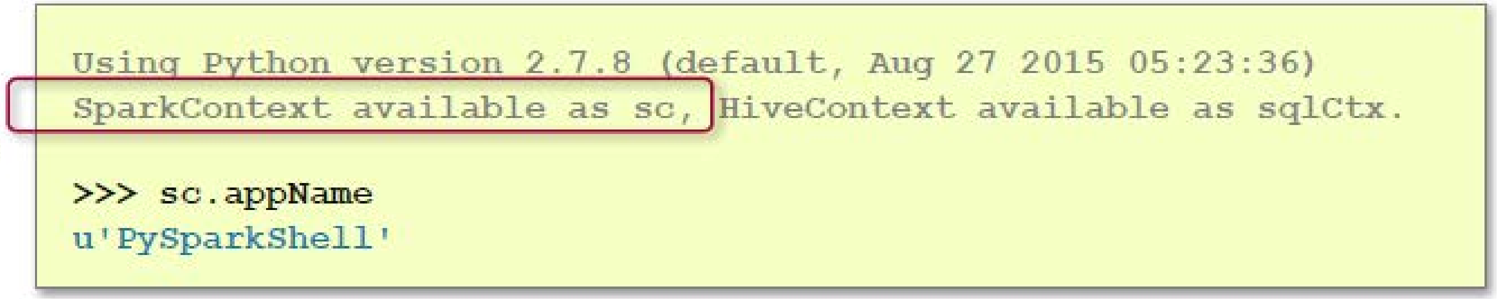
```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

- **Spark Context**
- **Resilient Distributed Data**
- **Transformations**
- **Actions**

# Spark Context (1)

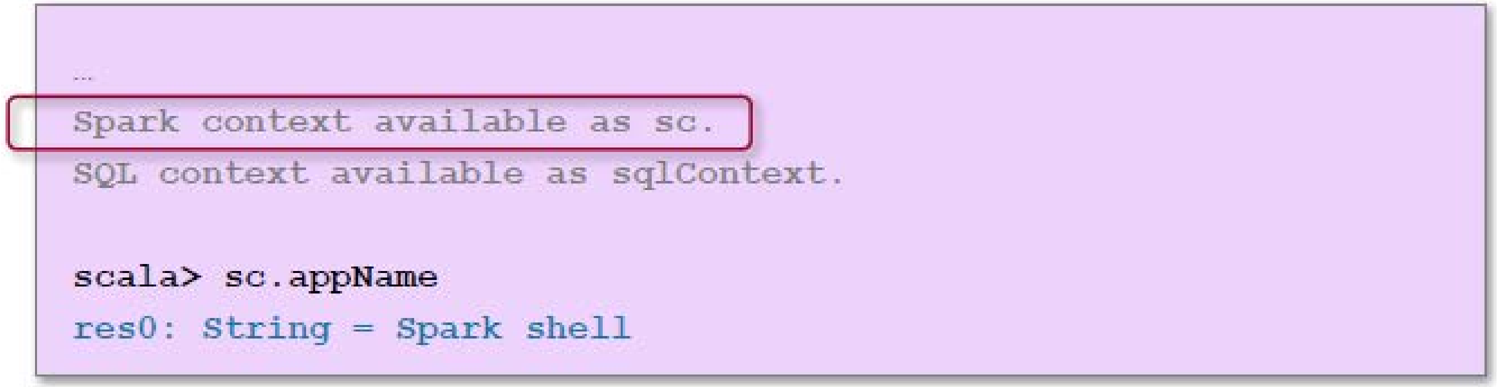
- Every Spark application requires a *spark context*: the main entry point to the Spark API
- Spark Shell provides a preconfigured Spark Context called “sc”

Python



```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

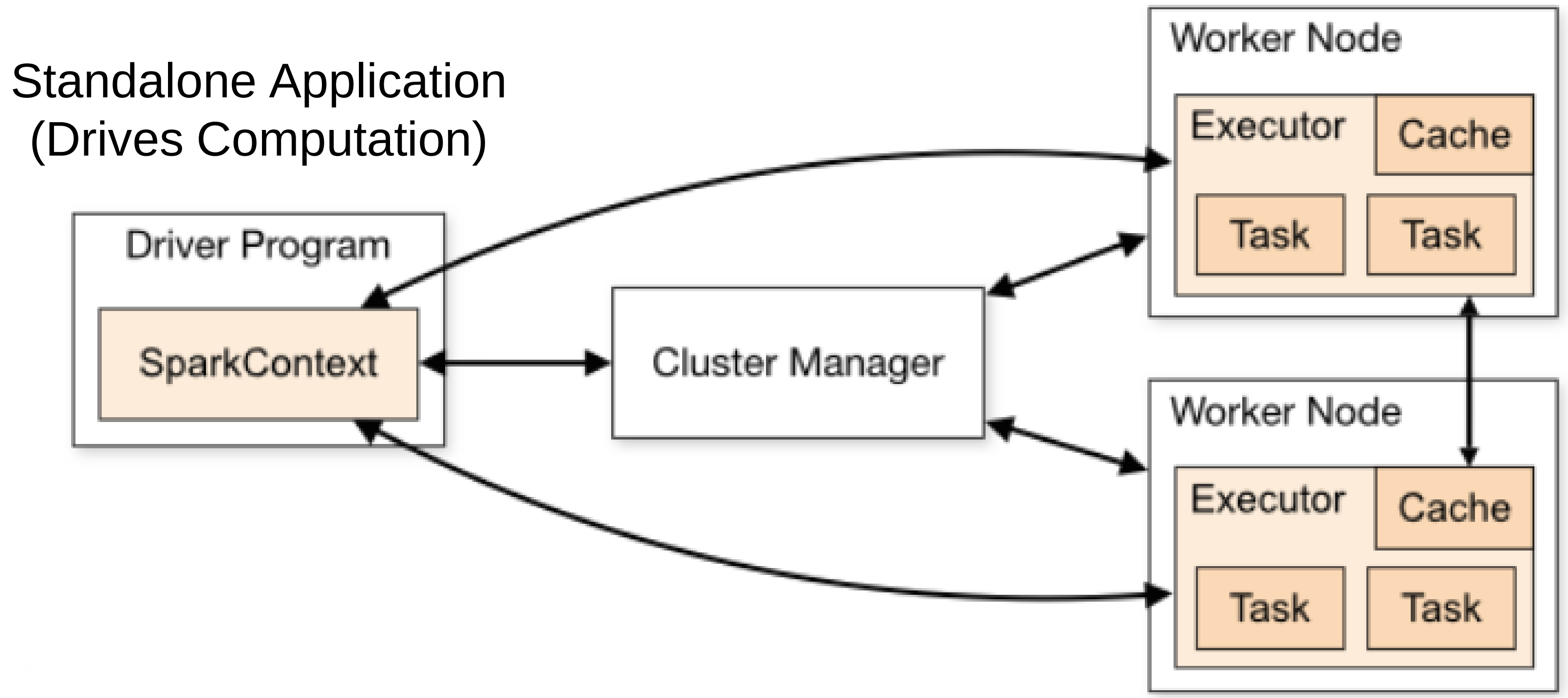
Scala



```
...
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

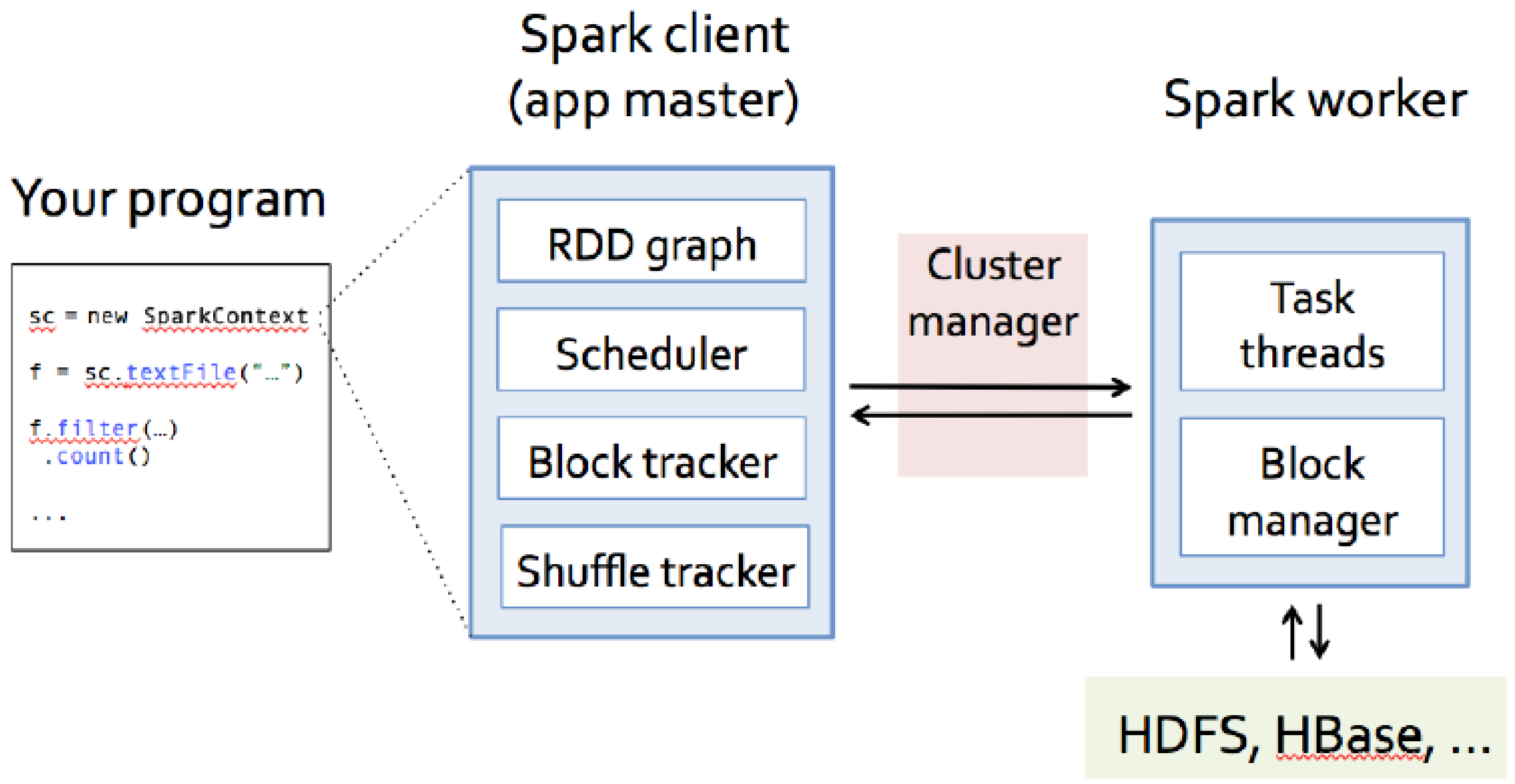
## Spark Context (2)

- Standalone applications → Driver code → Spark Context
- Spark Context holds configuration information and represents connection to a Spark cluster



## Spark Context (3)

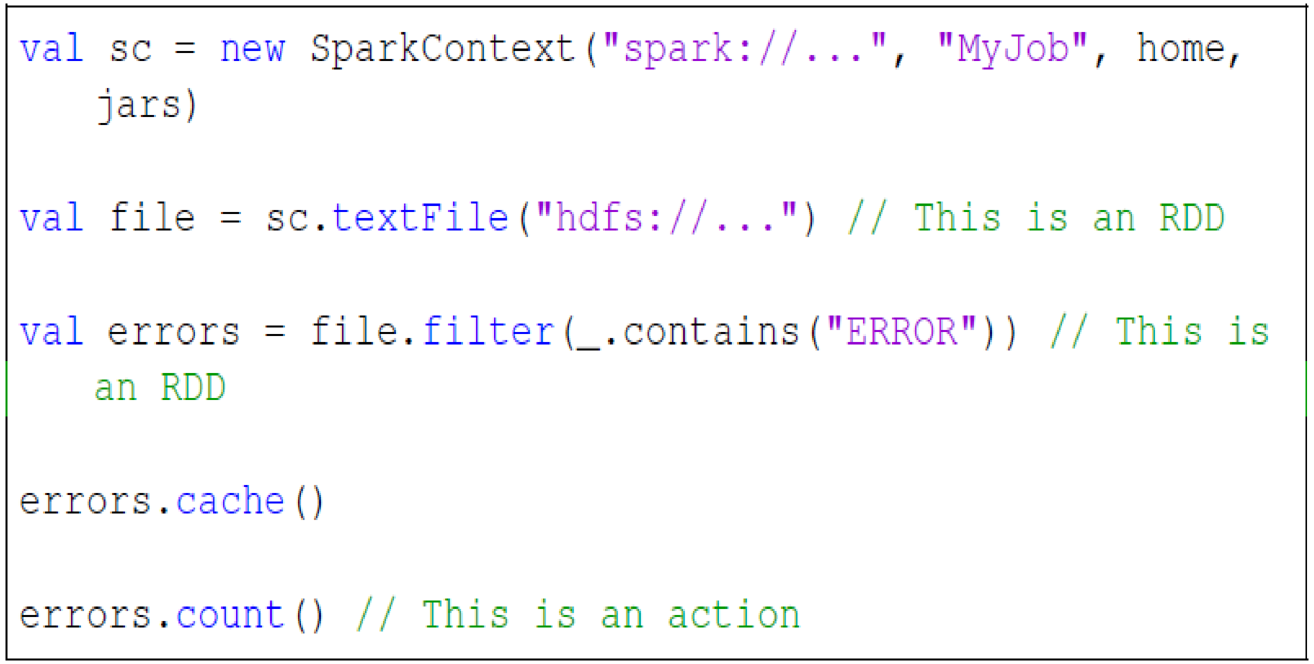
Spark context works as a client and represents connection to a Spark cluster



```
sc = new SparkContext
f = sc.textFile("…")
f.filter(…)
 .count()
...
```

# Spark Fundamentals

## Example of an application:



```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

- Spark Context
- **Resilient Distributed Data**
- Transformations
- Actions

# Resilient Distributed Dataset

**RDD** (Resilient Distributed Dataset) is the fundamental unit of data in Spark: An *Immutable* collection of objects (or records, or elements) that can be operated on "in parallel" (spread across a cluster)

**Resilient** -- if data in memory is lost, it can be recreated

- Recover from node failures
- An RDD keeps its lineage information → it can be recreated from parent RDDs

**Distributed** -- processed across the cluster

- Each RDD is composed of one or more partitions → (more partitions – more parallelism)

**Dataset** -- initial data can come from a file or be created

## RDDs

**Key Idea**: Write applications in terms of transformations on distributed datasets. One RDD per transformation.

- Organize the RDDs into a DAG showing how data flows.
- RDD can be saved and reused or recomputed. Spark can save it to disk if the dataset does not fit in memory
- Built through parallel transformations (map, filter, group-by, join, etc). Automatically rebuilt on failure
- Controllable persistence (e.g. caching in RAM)

## RDDs are designed to be “immutable”

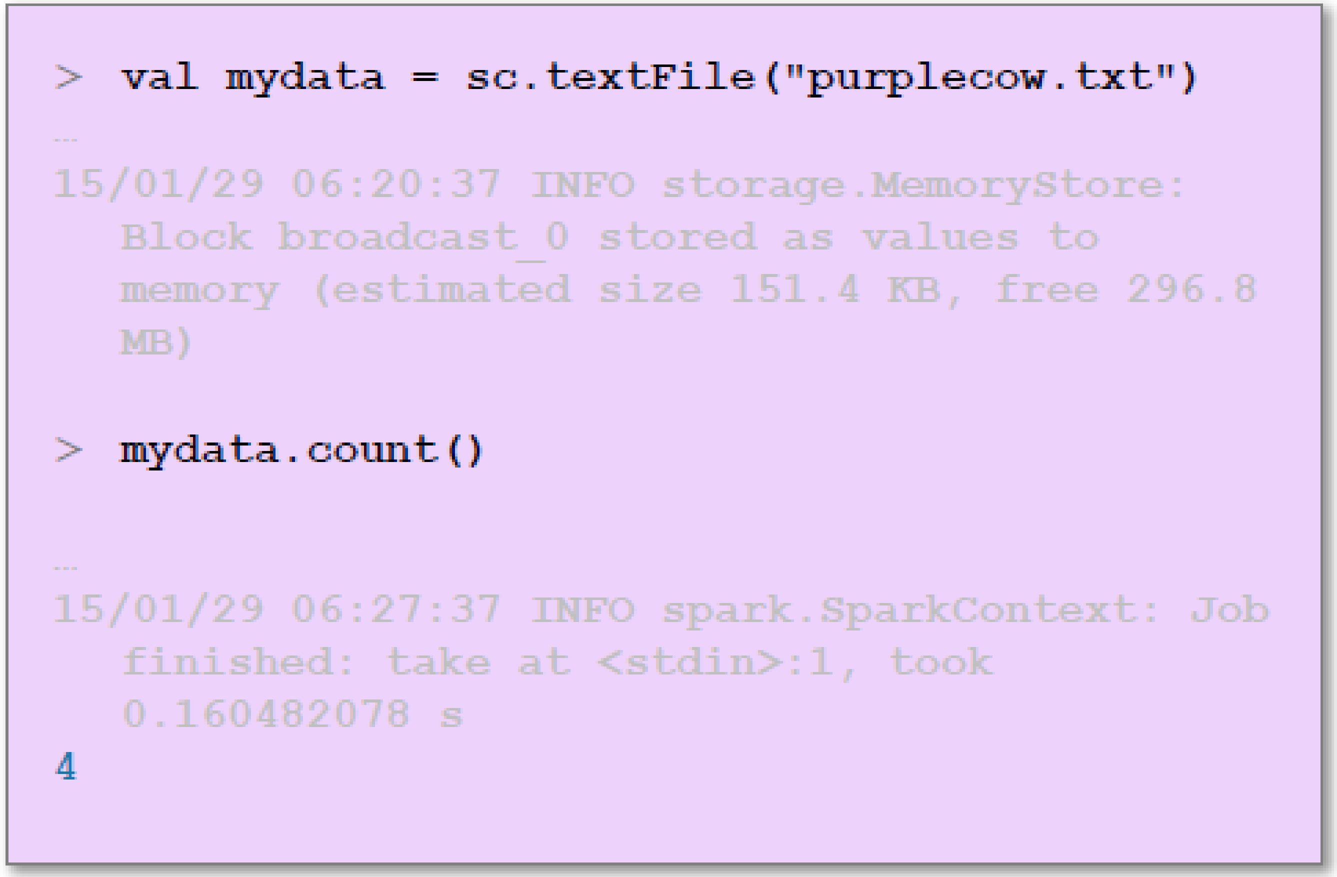
- Create once, then reuse without changes. Spark knows lineage → can be recreated at any time → Fault-tolerance
- Avoids data inconsistency problems (no simultaneous updates) → Correctness
- Easily live in memory as on disk → Caching → Safe to share across processes/tasks → Improves performance
- Tradeoff: (Fault-tolerance & Correctness) vs (Disk Memory & CPU)

## Creating a RDD

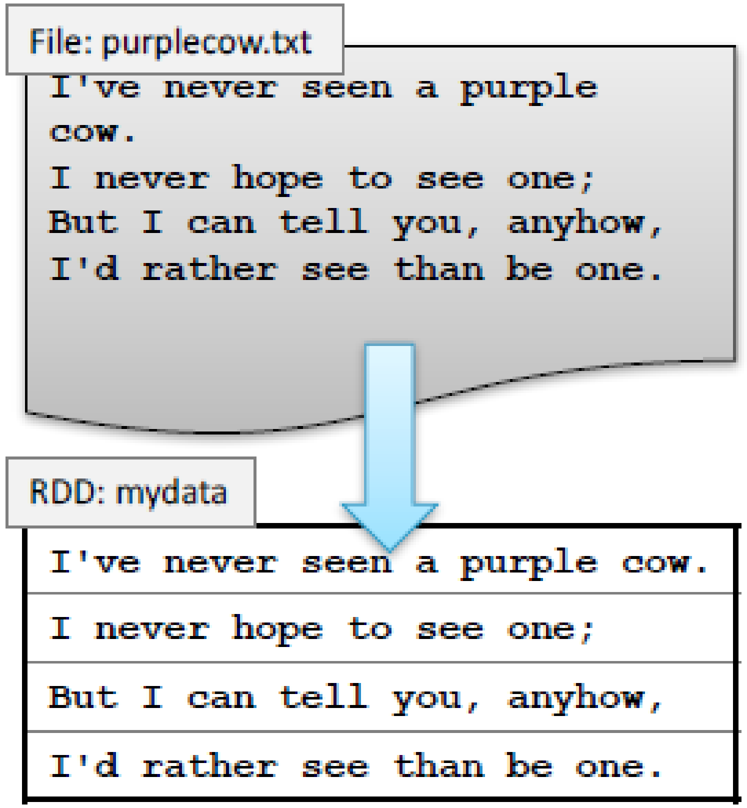
Three ways to create a RDD

- From a file or set of files
- From data in memory
- From another RDD

# Example: A File-based RDD



```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```



File: purplecow.txt
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.
RDD: mydata
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.

## Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

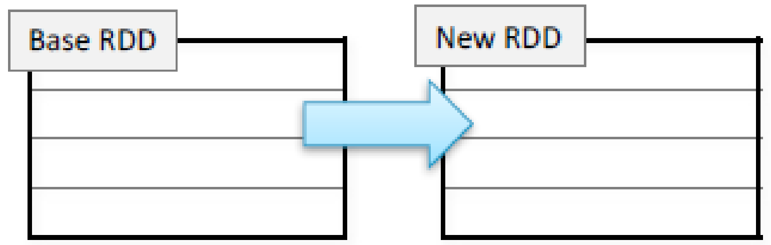
- Spark Context
- **Resilient Distributed Data**
- **Transformations**
- **Actions**

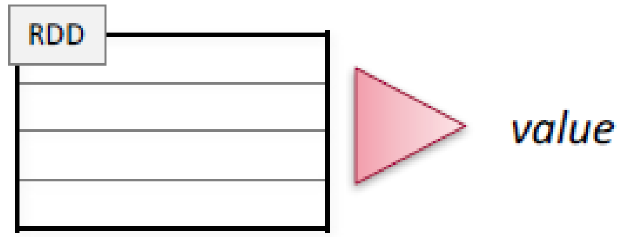
# RDD Operations

Two types of operations

**Transformations**: Define a new RDD based on current RDD(s)

**Actions**: return values



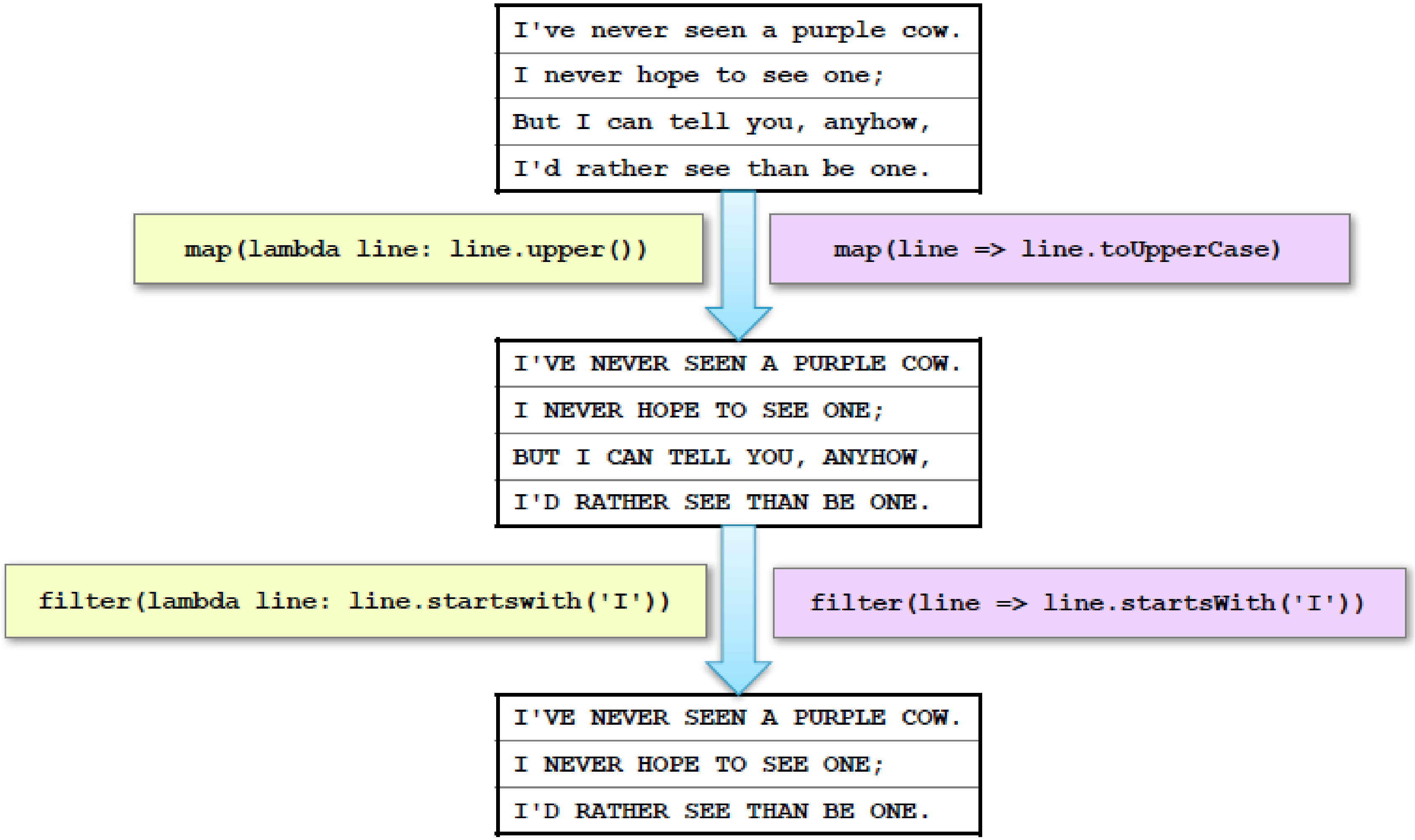


```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

## RDD Transformations

- Set of operations on a RDD that define how they should be transformed
- As in relational algebra, the application of a transformation to an RDD yields a new RDD (because RDD are immutable)
- Transformations are lazily evaluated, which allow for optimizations to take place before execution
- Examples: map(), filter(), groupByKey(), sortByKey(), etc.

# Example: map and filter Transformations



```
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'))
```

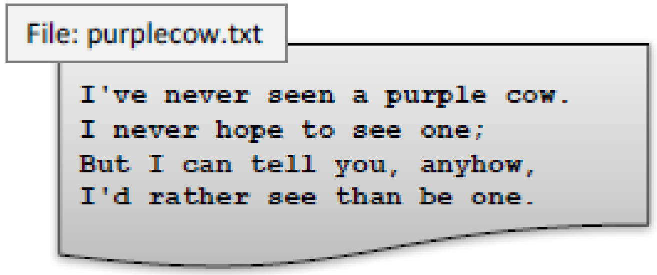
```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

# RDD Actions

- Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
- Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
- Some common actions
  - count() – return the number of elements
  - take(*n*) – return an array of the first *n* elements
  - collect()– return an array of all elements
  - saveAsTextFile(*file*) – save to text file(s)

# Lazy Execution of RDDs (1)

Data in RDDs is not processed until an action is performed



File: purplecow.txt

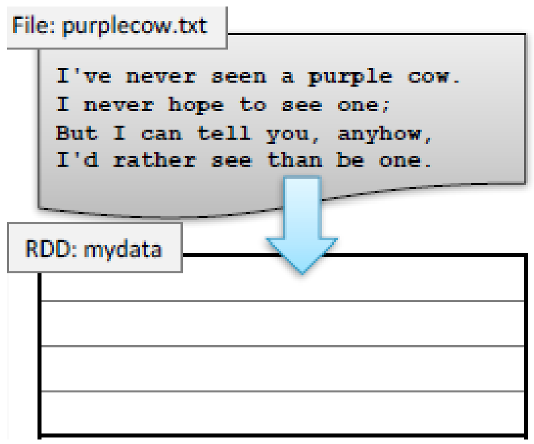
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

## Lazy Execution of RDDs (2)

Data in RDDs is not processed until an action is performed

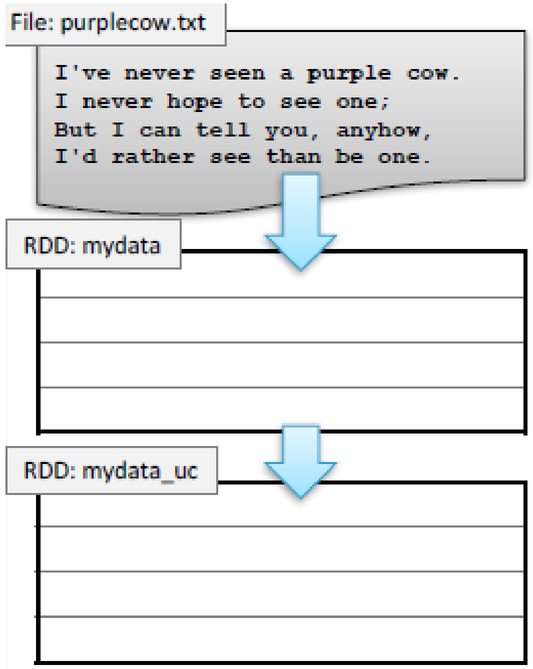
```
> val mydata = sc.textFile("purplecow.txt")
```


File: purplecow.txt
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.
RDD: mydata

# Lazy Execution of RDDs (3)

Data in RDDs is not processed until an action is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```

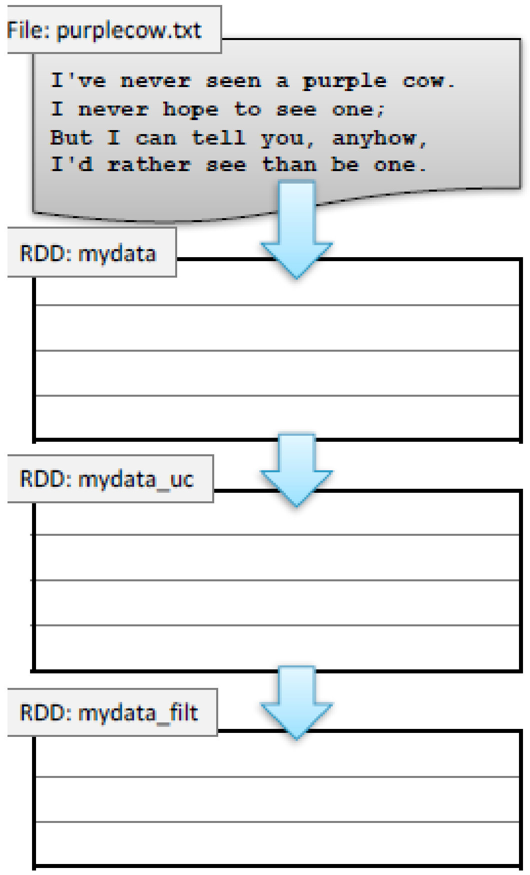


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD: mydata
RDD: mydata_uc

## Lazy Execution of RDDs (4)

Data in RDDs is not processed until an action is performed

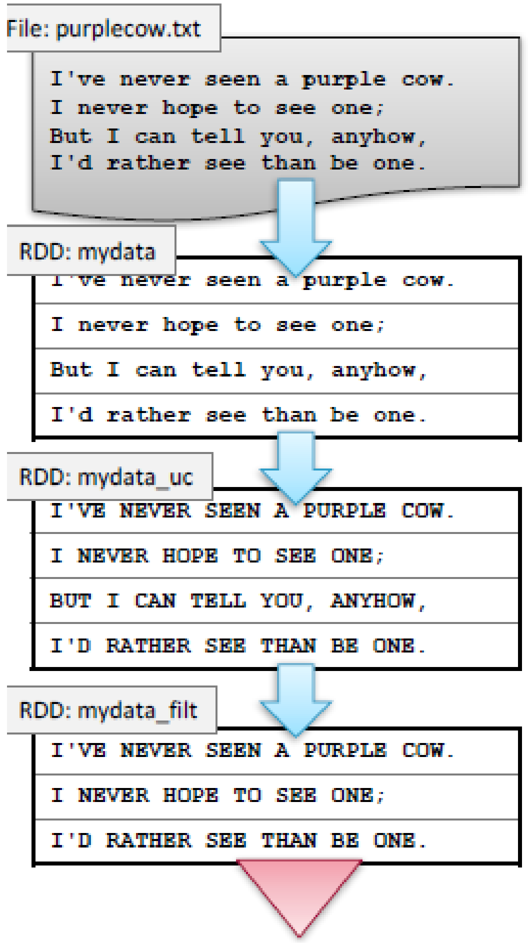
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



## Lazy Execution of RDDs (5)

Data in RDDs is not processed until an action is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```


File: purplecow.txt
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.
RDD: mydata
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.
RDD: mydata_uc
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.
RDD: mydata_filt
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

***Output Action "triggers" computation, pull model***

## Example: Mine error logs

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

```
lines = spark.textFile(“hdfs://...”)   HadoopRDD
errors = lines.filter(lambda s: s.startswith(“ERROR”)) FilteredRDD
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()
messages.filter(lambda s: “foo” in s).count()
```

Result: full-text search of Wikipedia in 0.5 sec (vs 20 sec for on-disk data)

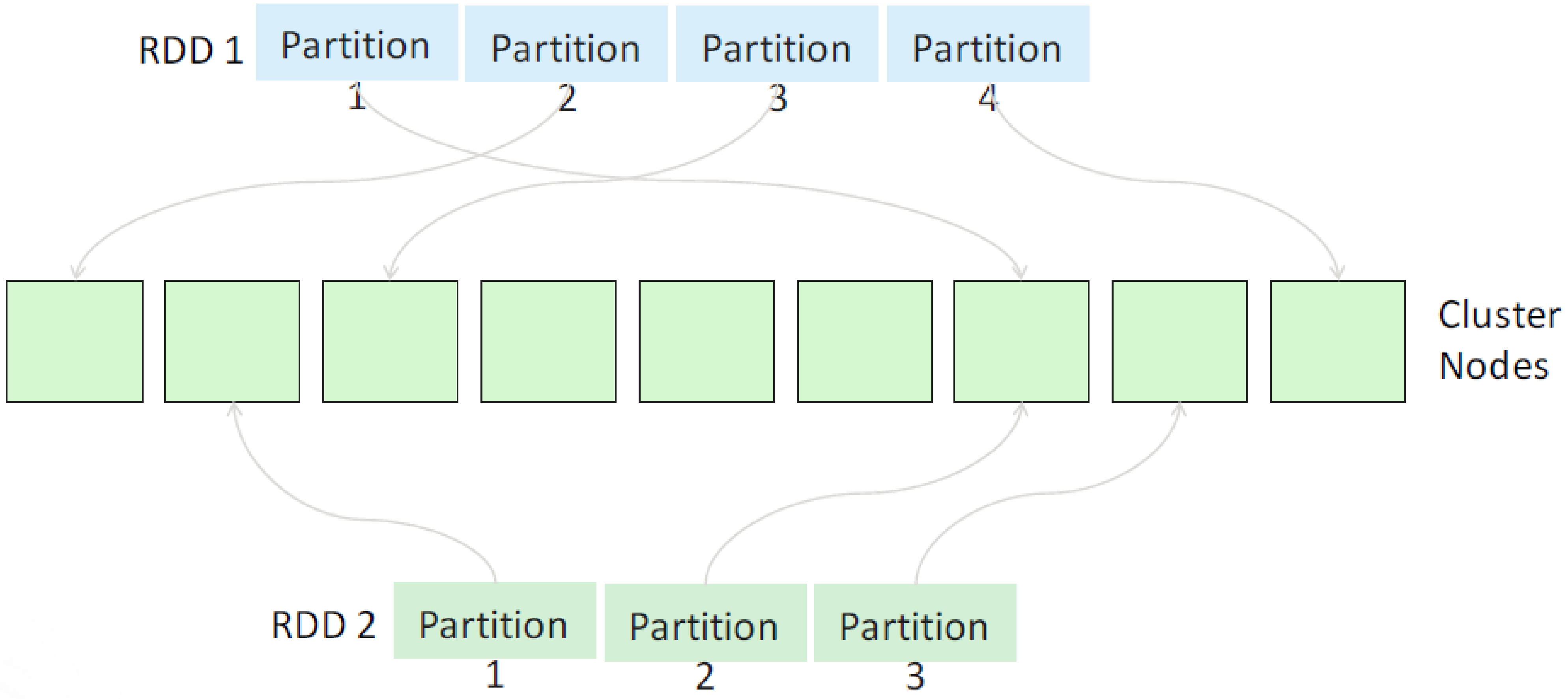
## Key Idea: Elastic parallelism

RDDs operations are designed to offer embarrassing parallelism.

Spark will spread the task over the nodes where data resides, offers a highly concurrent execution that minimizes delays. Term: "partitioned computation".

If some component crashes or even is just slow, Spark simply kills that task and launches a substitute.

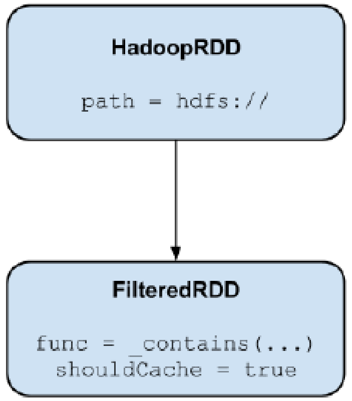
## RDD and Partitions (Parallelism example)

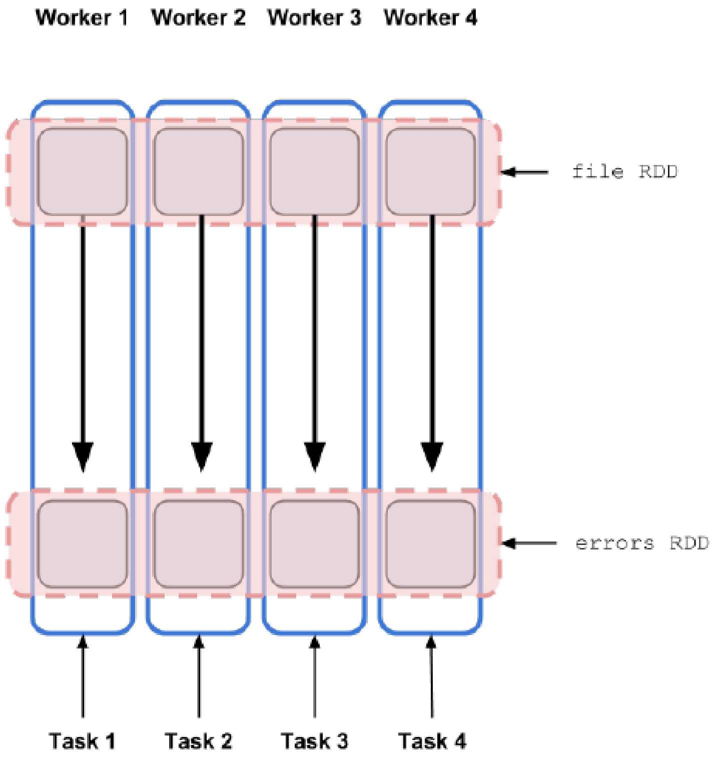


## RDD Graph: Data Set vs Partition Views

Much like in Hadoop MapReduce, each RDD is associated to (input) partitions

```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```





## RDDs: Data Locality

- Data Locality Principle
  - ➢ Keep high-value RDDs precomputed, in cache or SDD
  - ➢ Run tasks that need the specific RDD with those same inputs on the node where the cached copy resides.
  - ➢ This can maximize in-memory computational performance.

Requires cooperation between your hints to Spark when you build the RDD, Spark runtime and optimization planner, and the underlying YARN resource manager.

## RDDs -- Summary

RDD are partitioned, locality aware, distributed collections

- RDD are immutable

RDD are data structures that:

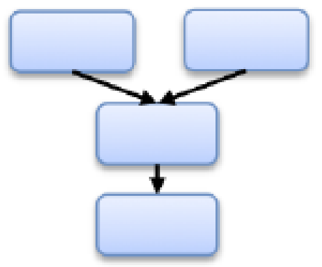
- Either point to a direct data source (e.g. HDFS)
- Apply some transformations to its parent RDD(s) to generate new data elements

Computations on RDDs

- Represented by lazily evaluated lineage DAGs composed by chained RDDs

# Lifetime of a Job in Spark

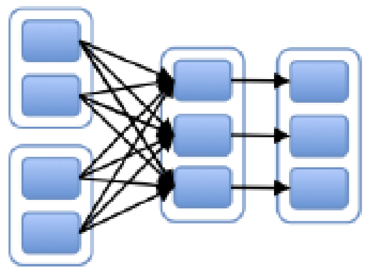
**RDD Objects**



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

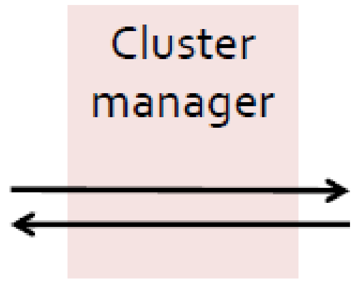
**DAG Scheduler**



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

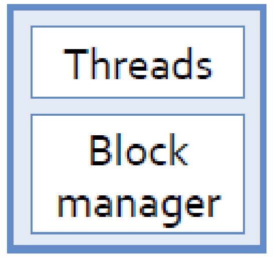
**Task Scheduler**



Launch tasks via Master

Retry failed and straggler tasks

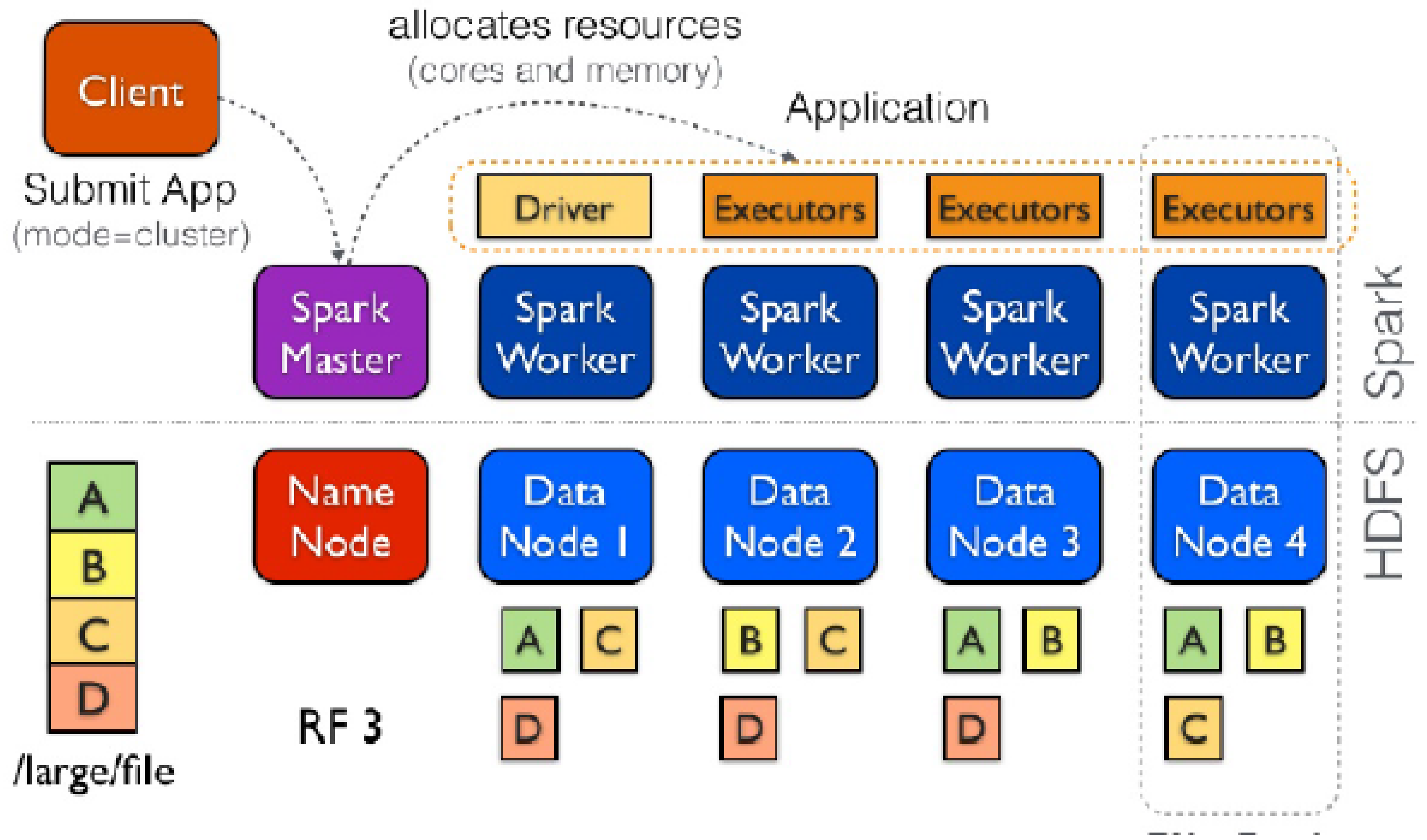
**Worker**



Execute tasks

Store and serve blocks

# Anatomy of a Spark Application



Cluster Manager
(YARN/Mesos)

## Typical RDD pattern of use

Instead of doing a lot of work in each RDD, developers split tasks into lots of small RDDs

These are then organized into a DAG.

Developer anticipates which will be costly to recompute and hints to Spark that it should cache those.

## Why is this a good strategy?

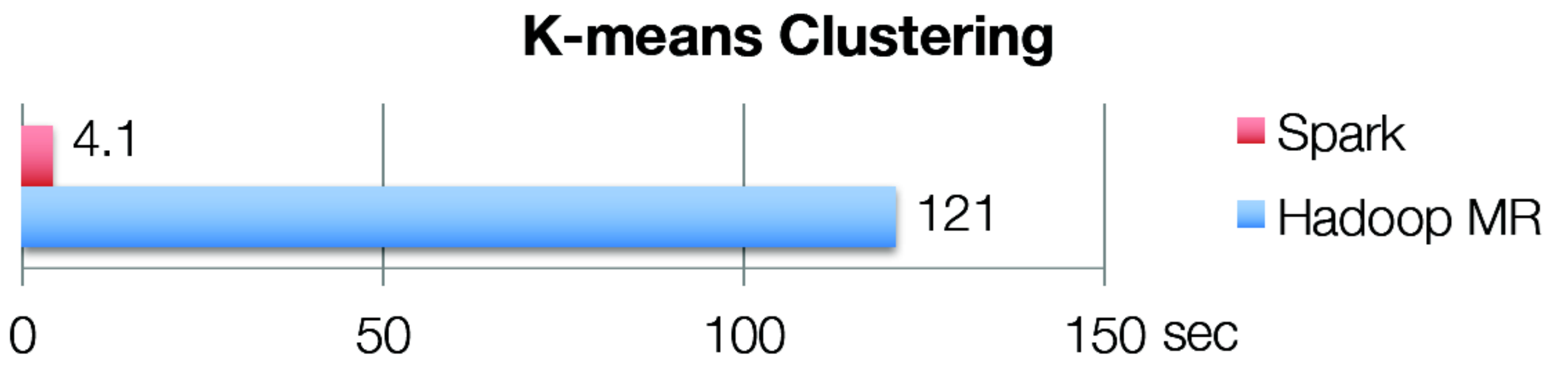
Spark tries to run tasks that will need the same intermediary data on the same nodes.

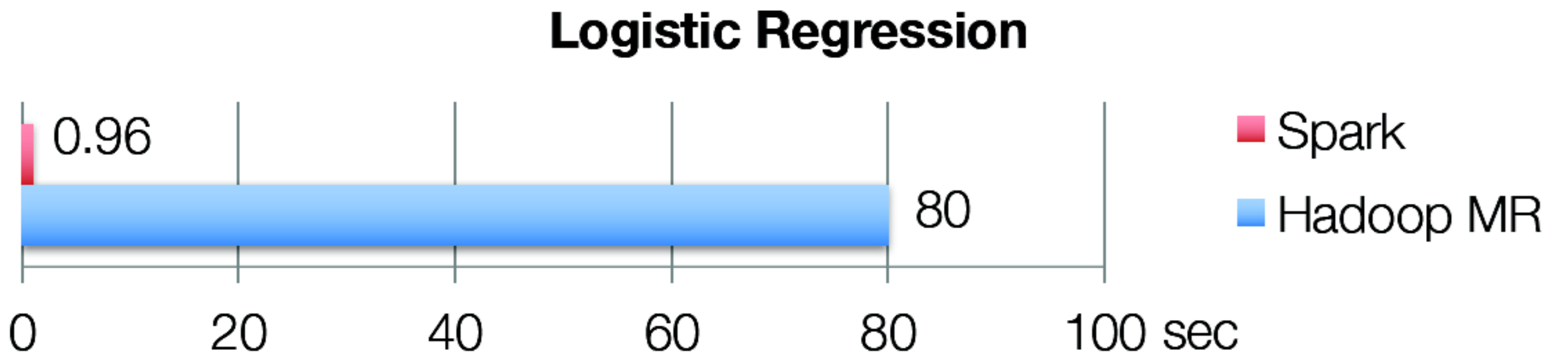
If MapReduce jobs were arbitrary programs, this wouldn't help because reuse would be very rare.

But in fact the MapReduce model is very repetitious and iterative, and often applies the same transformations again and again to the same input files.

- Those particular RDDs become great candidates for caching.
- MapReduce programmer may not know how many iterations will occur, but Spark itself is smart enough to evict RDDs if they don't actually get reused.

## Iterative Algorithms: Spark vs MapReduce





## Today’s Topics

- Motivation
- Spark Basics
- Spark Programming

## Spark Programming (1)

### Creating RDDs

```
# Turn a Python 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 existing Hadoop InputFormat (Java/Scala only)
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

## Spark Programming (2)

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}
```

# Spark Programming (3)

## 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
```

# Spark Programming (4)

## Working with Key-Value Pairs

Spark’s “distributed reduce” transformations operate 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);
          pair._1 // => a
          pair._2 // => b
```

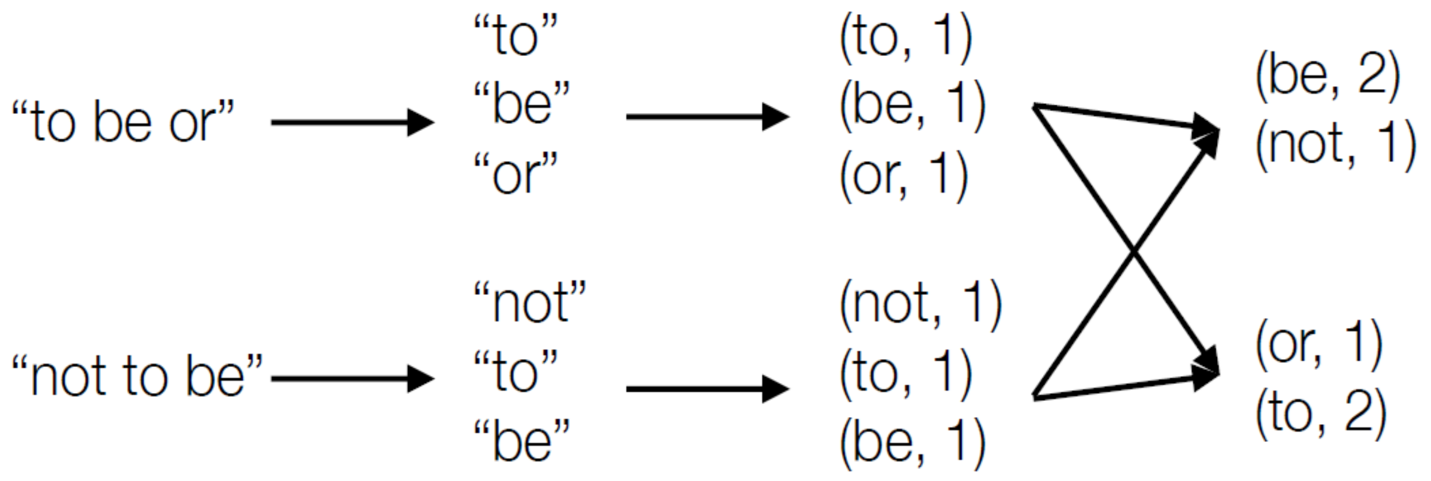
# Spark Programming (5)

Some Key-Value Operations

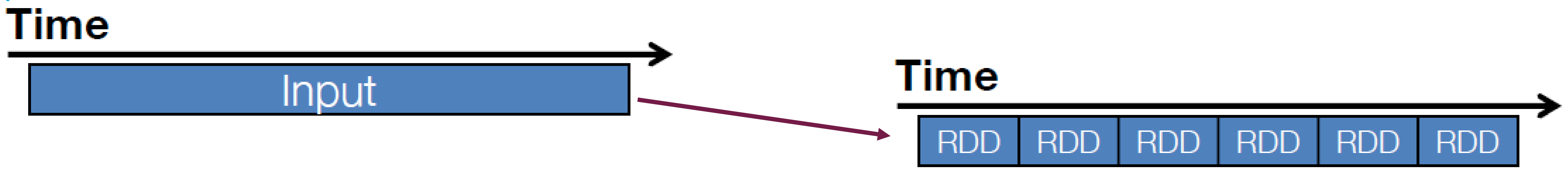
```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y)    # => {(cat, 3), (dog, 1)}

pets.groupByKey()      # => {(cat, [1, 2]), (dog, [1])}

pets.sortByKey()       # => {(cat, 1), (cat, 2), (dog, 1)}
```

# 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)
```



# Example: Spark Streaming



Represents streams as a series of RDDs over time (typically sub second intervals, but it is configurable)

```
val spammers = sc.sequenceFile(“hdfs://spammers.seq”)
sc.twitterStream(...)
   .filter(t => t.text.contains(“Santa Clara University”))
   .transform(tweets => tweets.map(t => (t.user, t)).join(spammers))
   .print()
```

## Spark: Combining Libraries (Unified Pipeline)

```
# Load data using Spark SQL
points = spark.sql(“select latitude, longitude from tweets”)

# Train a machine learning model
model = KMeans.train(points, 10)

# Apply it to a stream
sc.twitterStream(...)
   .map(lambda t: (model.predict(t.location), 1))
   .reduceByWindow(“5s”, lambda a, b: a + b)
```

# Spark: Setting 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)
```

# Summary

Spark is a powerful "manager" for big data computing.

It centers on a job scheduler for Hadoop (MapReduce) that is smart about where to run each task: co-locate task with data.

The data objects are "RDDs": a kind of recipe for generating a file from an underlying data collection. RDD caching allows Spark to run mostly from memory-mapped data, for speed.

- Online tutorials: spark.apache.org/docs/latest