

# Apache Spark

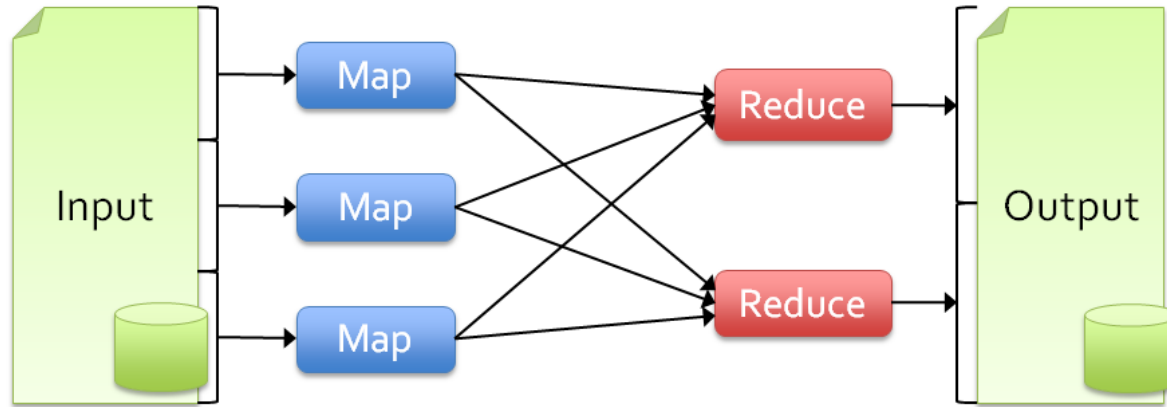
COMP9313: Big Data Management

Thanks to Dr. Xin Cao for sharing useful materials for  
the preparation of this course

# Motivation of Spark

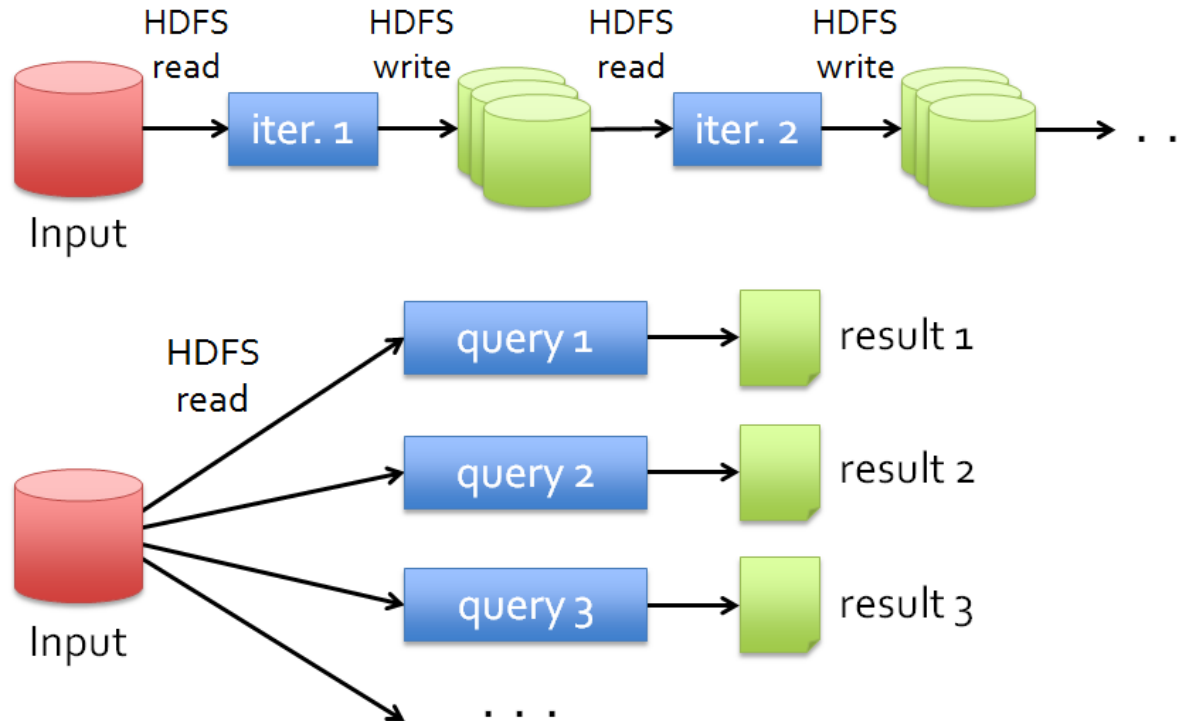
- MapReduce greatly simplified big data analysis on large, unreliable clusters. It is great at one-pass computation
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graphs)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All 3 need faster **data sharing** across parallel jobs
  - One reaction: specialized models for some of these apps, e.g.,
    - Pregel (graph processing)
    - Storm (stream processing)

# Limitations of MapReduce



- As a general programming model:
  - It is more suitable for one-pass computation on a large dataset
  - Hard to compose and nest multiple operations
  - No means of expressing iterative operations
- As implemented in Hadoop
  - All datasets are read from disk, then stored back on to disk
  - All data is (usually) triple-replicated for reliability
  - Not easy to write MapReduce programs using Java

# Data Sharing in MapReduce



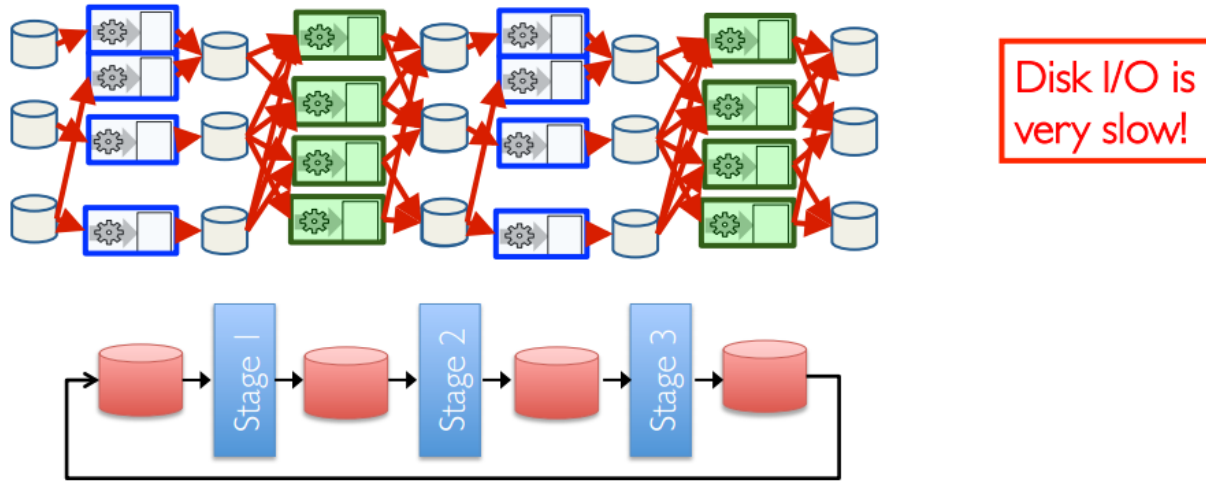
**Slow** due to replication, serialization, and disk IO

- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

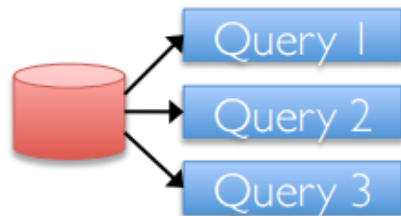
Efficient primitives for **data sharing**

# Data Sharing in MapReduce

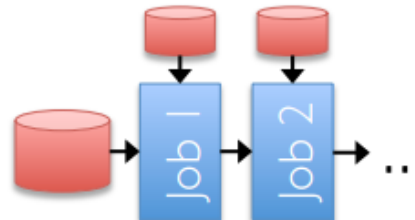
- Iterative jobs involve a lot of disk I/O for each repetition



- Interactive queries and online processing involves lots of disk I/O



Interactive mining

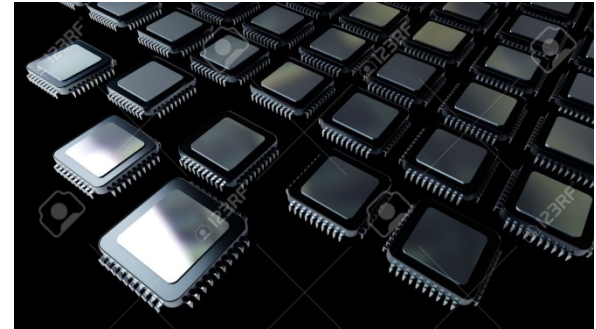


Stream processing

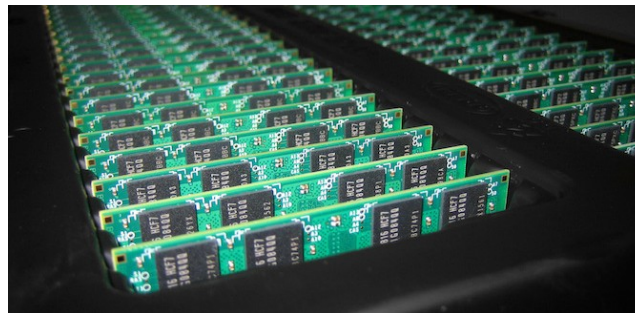
# Hardware for Big Data



Lots of hard drives



Lots of CPUs

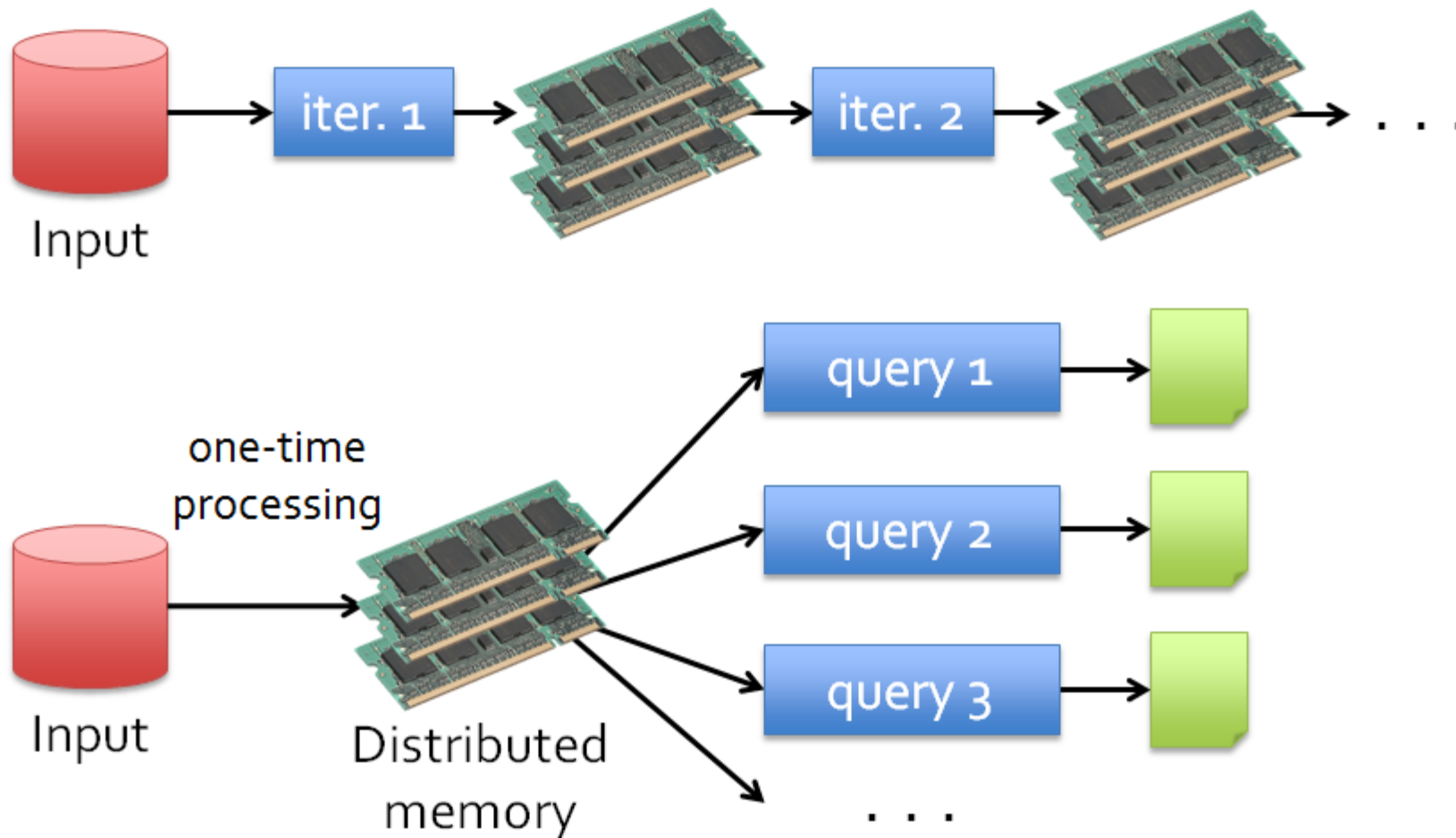


And lots of memory!

# Goals of Spark

- Keep more data in-memory to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

# Data Sharing in Spark with RDDs (Resilient Distributed Dataset)

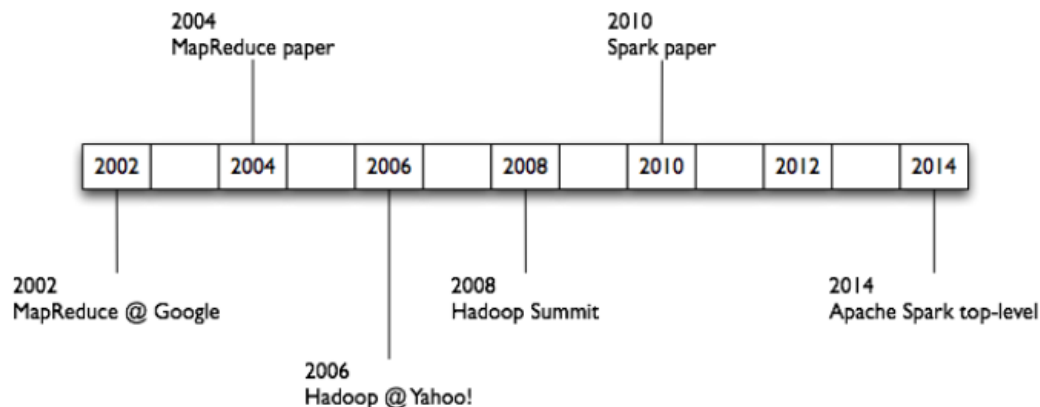


**10-100 × faster than network and disk**





# What is Spark?

- One popular answer to “What’s beyond MapReduce?”
- Open-source engine for large-scale data processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java, Python and R
- Brief history:
  - Developed at UC Berkeley AMPLab in 2009
  - Open-sourced in 2010
  - Became top-level Apache project in February 2014
  - Commercial support provided by DataBricks

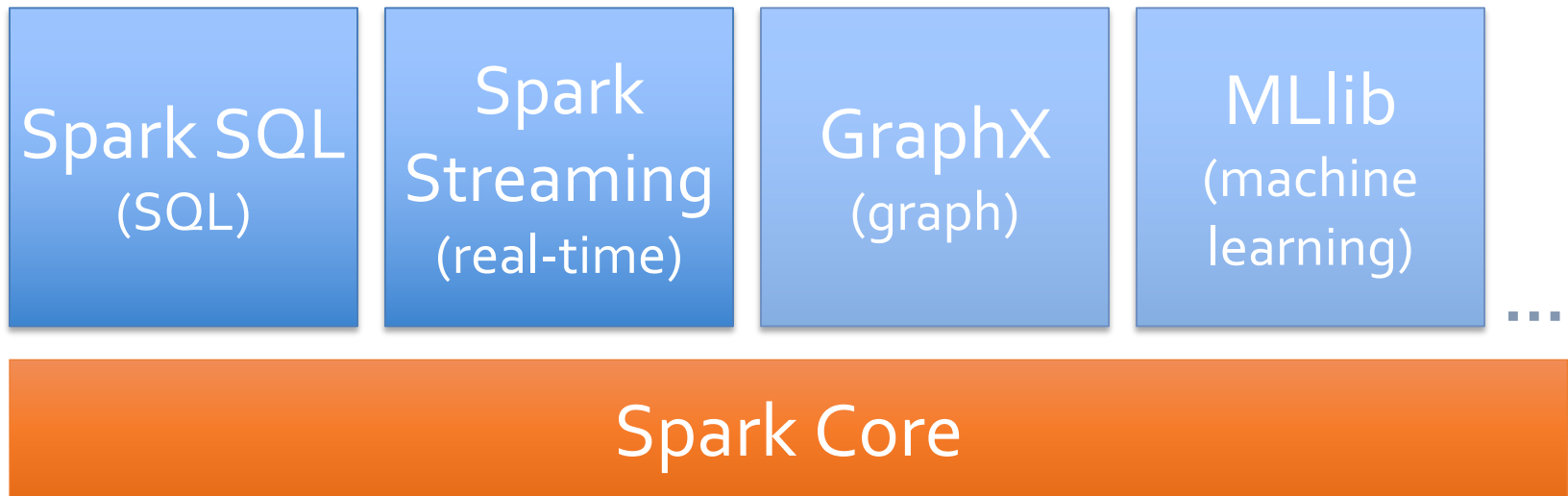


# What is Spark?

- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - **In-memory** computing primitives
  - General computation graphs Up to 100 × faster  
(10 × on disk)
- Improves usability through:
  - Rich APIs in Scala, Java, Python, R
  - Interactive shell Often 5 × less code
- **Spark is not**
  - a modified version of Hadoop
  - dependent on Hadoop because it has its own cluster management (Spark uses Hadoop for storage purpose only)

# What is Spark?

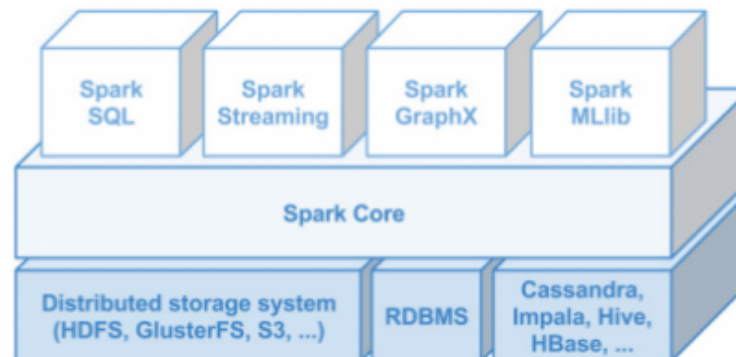
- Spark is the basis of a wide set of projects in the Berkeley Data Analytics Stack (BDAS)



- Spark SQL (SQL on Spark)
- Spark Streaming (stream processing)
- GraphX (graph processing)
- MLlib (machine learning library)

# Data Sources

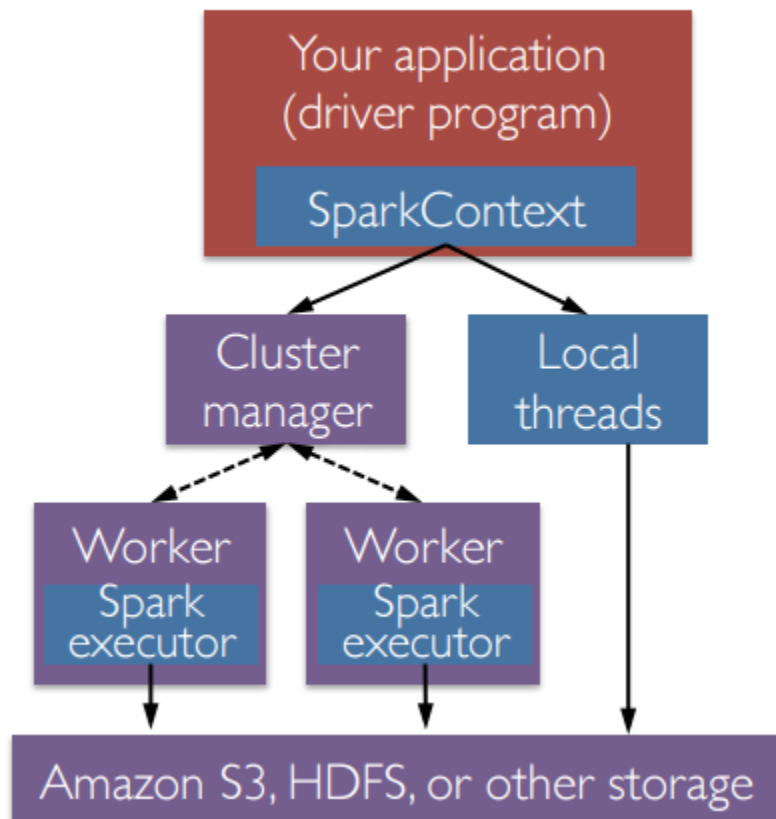
- Local Files
  - `file:///opt/httpd/logs/access_log`
- Amazon S3
- Hadoop Distributed Filesystem
  - Regular files, sequence files, any other Hadoop InputFormat
- HBase, Cassandra, etc.



# Spark Ideas

- Expressive computing system, 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)
- Layer an in-memory system on top of Hadoop
- Achieve fault-tolerance by re-execution instead of replication

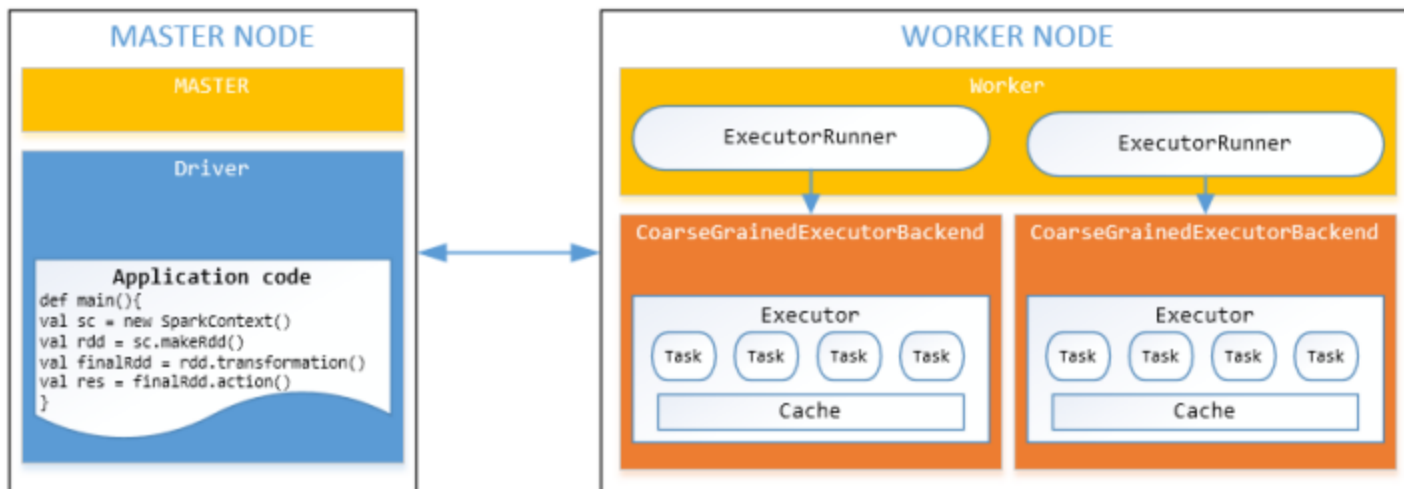
# Spark Workflow



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Connects to several types of cluster managers (e.g. YARN, Mesos, or its own manager)
- Cluster manager:
  - Allocates resources across applications
- Spark executor:
  - Runs computations
  - Accesses data storage

# Workers Nodes and Executors

- Worker nodes are machines that run executors
  - Host one or multiple Workers
  - One JVM (1 process) per Worker
  - Each Worker can spawn one or more Executors
- Executors run tasks
  - Run in child JVM (1 process)
  - Execute one or more task using threads in a ThreadPool

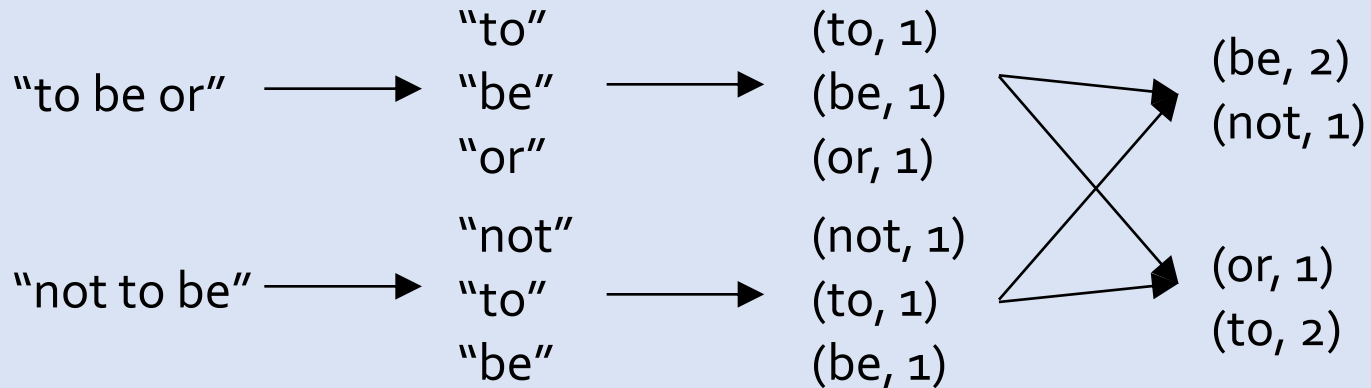


# Word Count in Spark

```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                   .map(word => (word, 1))
                   .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```





# Introduction to Scala

# Scala

- Scala is a *general-purpose programming language* designed to express common programming patterns in a concise, elegant, and type-safe way
- Scala supports both Object Oriented Programming and Functional Programming
- Scala:
  - Can be used as drop-in replacement for Java
    - Mixed Scala/Java projects
  - Uses existing Java libraries
  - Uses existing Java tools (Ant, Maven, JUnit, etc...)
  - Has IDE Support (NetBeans, IntelliJ, Eclipse)



# Why Scala

- Scala supports object-oriented programming. Conceptually, every value is an object and every operation is a method call. The language supports advanced component architectures through classes and traits
- Scala is also a functional language. Supports functions, immutable data structures and preference for immutability over mutation
- Seamlessly integrated with Java
- Being used heavily for Big data, e.g., Spark, Kafka, etc.

# Scala Basic Syntax

- When considering a Scala program, it can be defined as a collection of objects that communicate via invoking each other's methods
- **Object** – same as in Java
- **Class** – same as in Java
- **Methods** – same as in Java
- **Fields** – Each object has its unique set of instance variables, which are called fields. An object's state is created by the values assigned to these fields
- **Traits** – Like Java Interface. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes
- **Closure** – A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function

closure = function + environment

# Object-oriented Programming in Scala

- Scala is object-oriented, and is based on Java's model
- An **object** is a **singleton object** (there is only one of it)
  - Variables and methods in an **object** are somewhat similar to Java's **static** variables and methods
  - Reference to an **object**'s variables and methods have the syntax ***ObjectName.methodOrVariableName***
  - The name of an **object** should be capitalized
- A **class** may take parameters, and may describe any number of objects
  - The class body *is* the constructor, but you can have additional constructors
  - With correct use of **val** and **var**, Scala provides getters and setters for class parameters

# Scala is Statically Typed

- You don't have to specify a type in most cases
- Type Inference

```
val sum = 1 + 2 + 3
```

```
val nums = List(1, 2, 3)
```

```
val map = Map("abc" -> List(1,2,3))
```

Explicit Types

```
val sum: Int = 1 + 2 + 3
```

```
val nums: List[Int] = List(1, 2, 3)
```

```
val map: Map[String, List[Int]] = ...
```

# Scala is High Level

**// Java - Check if string has uppercase character**

```
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
```

**// Scala**

```
val hasUpperCase = name.exists(_.isUpper)
```

# Scala is Concise

## // Java

```
public class Person {
    private String name;
    private int age;
    public Person(String name, Int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {                // name getter
        return name;
    }
    public int getAge() {                    // age getter
        return age;
    }
    public void setName(String name) {       // name setter
        this.name = name;
    }
    public void setAge(int age) {            // age setter
        this.age = age;
    }
}
```

## // Scala

```
class Person(var name: String, private var _age: Int) {
    def age = _age                // Getter for age
    def age_=(newAge: Int) {      // Setter for age
        println("Changing age to: "+newAge)
        _age = newAge
    }
}
```



# Variables and Values

- **Variables**: values stored can be changed

```
var foo = "foo"
```

```
foo = "bar" // okay
```

- **Values**: immutable variable

```
val foo = "foo"
```

```
foo = "bar" // nope
```

# Scala is Pure Object Oriented

```
// Every value is an object
1.toString
// Every operation is a method call
1 + 2 + 3 → (1).+(2).+(3)
// Can omit . and ( )
"abc" charAt 1 → "abc".charAt(1)
// Classes (and abstract classes) like Java
abstract class Language(val name:String) {
  override def toString = name
}
// Example implementations
class Scala extends Language("Scala")
// Anonymous class
val scala = new Language("Scala") { /* empty */ }
```

# Scala Traits

```
// Like interfaces in Java
trait JVM {
  // But allow implementation
  override def toString = super.toString +
    "runs on JVM" }
trait Static {
  override def toString = super.toString +
    "is Static" }

// Traits are stackable
class Scala extends Language with JVM with Static {
  val name = "Scala"
}
println(new Scala) → "Scala runs on JVM is Static"
```

# Scala is Functional

- First-Class Functions. Functions are treated like objects:
  - passing functions as arguments to other functions
  - returning functions as the values from other functions
  - assigning functions to variables or storing them in data structures

```
// Lightweight anonymous functions  
(x:Int) => x + 1
```

```
// Calling the anonymous function  
val plusOne = (x:Int) => x + 1  
plusOne(5) → 6
```

# Scala is Functional

- Closures: a function whose return value depends on the value of one or more variables declared outside this function.

// plusFoo can reference any **values/variables** in scope

**var foo** = 1

**val** plusFoo = (x:Int) => x + **foo**

plusFoo(5) → 6

// Changing foo changes the return value of plusFoo

**foo** = 5

plusFoo(5) → 10

# Scala is Functional

- Higher Order Functions

- A function that does at least one of the following:

- takes one or more functions as arguments
    - returns a function as its result

```
val plusOne = (x:Int) => x + 1
```

```
val nums = List(1,2,3)
```

```
// map takes a function: Int => T
```

```
nums.map(plusOne)      → List(2,3,4)
```

```
// Inline Anonymous
```

```
nums.map(x => x + 1)    → List(2,3,4)
```

```
// Short form
```

```
nums.map(_ + 1)        → List(2,3,4)
```

# More Examples on Higher Order Functions

```
val nums = List(1,2,3,4)
// A few more examples for List class
nums.exists(_ == 2)      → true
nums.find(_ == 2)       → Some(2)
nums.indexWhere(_ == 2) → 1

// functions as parameters, apply f to the
// value "1"
def call(f: Int => Int) = f(1)

call(plusOne)           → 2
call(x => x + 1)         → 2
call(_ + 1)             → 2
```

# The Usage of “\_” in Scala

- In anonymous functions, the “\_” acts as a placeholder for parameters

`nums.map(x => x + 1)`

is equivalent to:

`nums.map(_ + 1)`

`List(1,2,3,4,5).foreach(print(_))`

is equivalent to:

`List(1,2,3,4,5).foreach(a => print(a))`

- You can use two or more underscores to refer different parameters.

`val sum = List(1,2,3,4,5).reduceLeft(_+_)`

is equivalent to:

`val sum = List(1,2,3,4,5).reduceLeft((a, b) => a + b)`

- The `reduceLeft` method works by applying the function/operation you give it, and applying it to successive elements in the collection



# Introduction to RDDs

# Challenge

- Existing Systems

- Existing in-memory storage systems have interfaces based on fine-grained updates
  - Reads and writes to cells in a table
  - E.g., databases, key-value stores, distributed memory
- Requires replicating data or logs across nodes for fault tolerance
  - > expensive!
    - 10-100x slower than memory write

- How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

# Solution: Resilient Distributed Datasets

- *Resilient Distributed Datasets (RDDs)*
  - Distributed collections of objects that can be cached in memory across cluster
  - Manipulated through parallel operators
  - Automatically recomputed on failure based on lineage
- RDDs can express many parallel algorithms, and capture many current programming models
  - Data flow models: MapReduce, SQL, ...
  - Specialized models for iterative apps: Pregel, ...

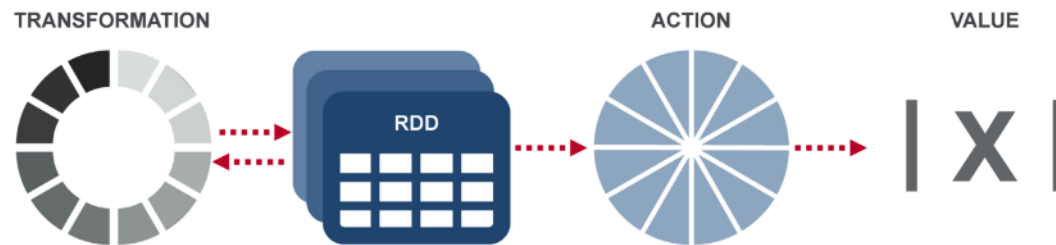
# What is RDD?

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12 ([paper](#))
  - RDD is a **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures
- **Distributed**
  - Data residing on multiple nodes in a cluster
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with)
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel

# RDD Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed)
- **Parallel**, i.e. process data in parallel
- **Typed**, i.e. values in a RDD have types, e.g. `RDD[Long]` or `RDD[(Int, String)]`
- **Partitioned**, i.e. the data inside an RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node)

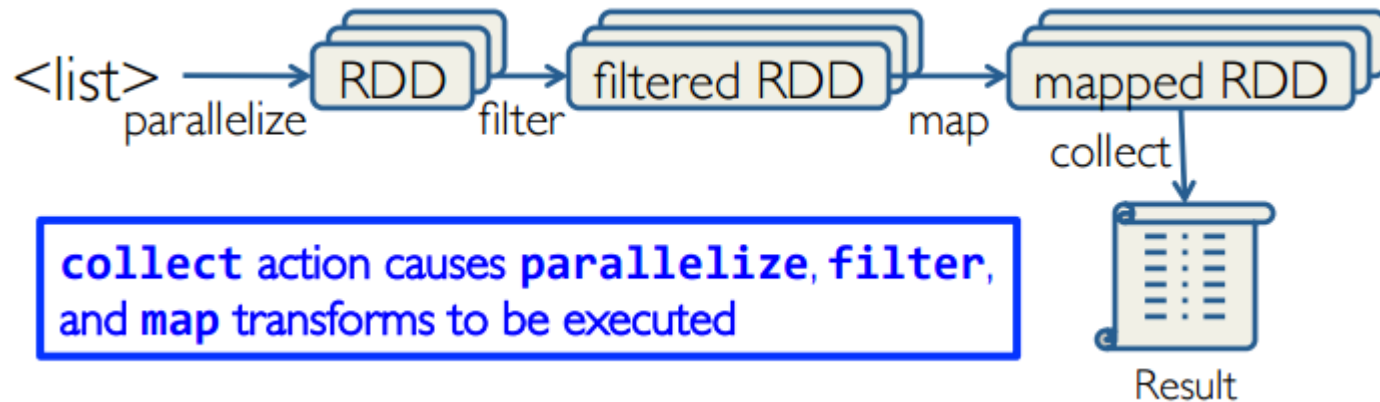
# RDD Operations



- **Transformation:** returns a new RDD
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD
  - Transformation functions include *map*, *filter*, *flatMap*, *groupByKey*, *reduceByKey*, *aggregateByKey*, *filter*, *join*, etc.
- **Action:** evaluates and returns a new value
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned
  - Action operations include *reduce*, *collect*, *count*, *first*, *take*, *countByKey*, *foreach*, *saveAsTextFile*, etc.

# Working with RDDs

- Create an RDD from a data source
  - by parallelizing existing collections (lists or arrays)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., map, filter
- Apply actions to an RDD: e.g., collect, count



- Users can control two other aspects:
  - Persistence
  - Partitioning

# Creating RDDs

- From HDFS, text files, Amazon S3, Apache HBase, SequenceFiles, any other Hadoop InputFormat

- Creating an RDD from a File

➤ `val inputfile = sc.textFile("...", 4)`

- RDD distributed in 4 partitions
- Elements are lines of input
- Lazy evaluation means no execution happens now

```
scala> val inputfile = sc.textFile("pg100.txt")
inputfile: org.apache.spark.rdd.RDD[String] = pg100.txt MapPartitionsRDD[17] at
textFile at <console>:24
```

- Turn a collection into an RDD

➤ `sc.parallelize(Array("hello", "spark"))`, creating from a Scala Array

- Creating an RDD from an existing Hadoop InputFormat

➤ `sc.hadoopFile(keyClass, valClass, inputFmt, conf)`



# Spark Transformations

- Create new datasets from an existing one
- Use lazy evaluation: Results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - Spark optimizes the required calculations
  - Spark recovers from failures
- Some transformation functions

Transformation	Description
<code>map(func)</code>	return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
<code>filter(func)</code>	return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true
<code>distinct([numTasks])</code>	return a new dataset that contains the distinct elements of the source dataset
<code>flatMap(func)</code>	similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item)

# Spark Actions

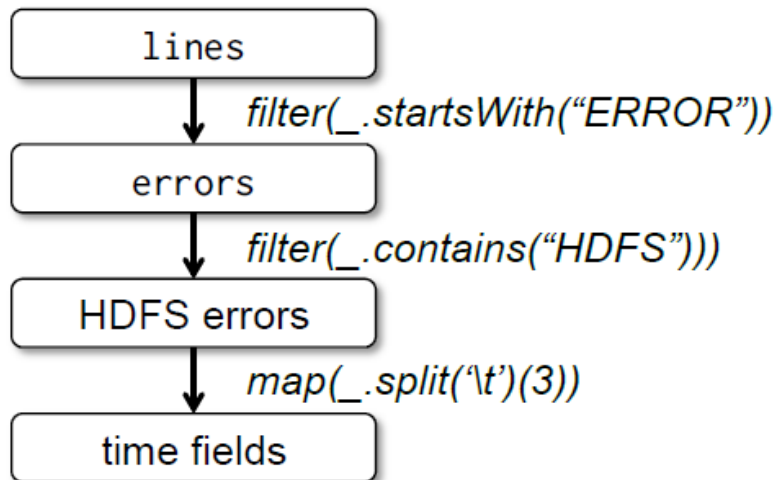
- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
- Some action functions

Action	Description
<code>reduce(func)</code>	aggregate dataset's elements using function <i>func</i> . <i>func</i> takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
<code>take(n)</code>	return an array with the first <i>n</i> elements
<code>collect()</code>	return all the elements as an array <b>WARNING: make sure will fit in driver program</b>
<code>takeOrdered(n, key=func)</code>	return <i>n</i> elements ordered in ascending order or as specified by the optional key function

- Example: `words.collect().foreach(println)`

# Example

- Web service is experiencing errors and an operators want to search terabytes of logs in the Hadoop file system to find the cause



*//base RDD*

```
val lines = sc.textFile("hdfs://...")
```

*//Transformed RDD*

```
val errors = lines.filter(_.startsWith("Error"))
```

```
errors.persist()
```

```
errors.count()
```

```
errors.filter(_.contains("HDFS"))
```

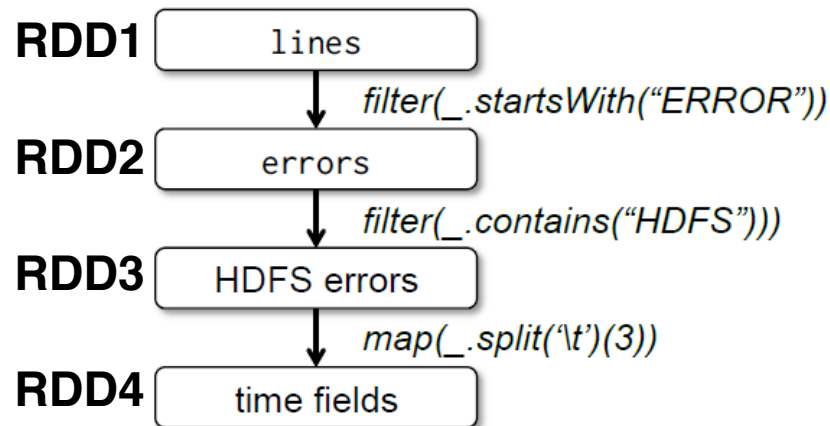
```
.map(_.split("\t")(3))
```

```
.collect()
```

- **Line1:** RDD backed by an HDFS file (base RDD lines not loaded in memory)
- **Line3:** Asks for errors to persist in memory (errors are in RAM)

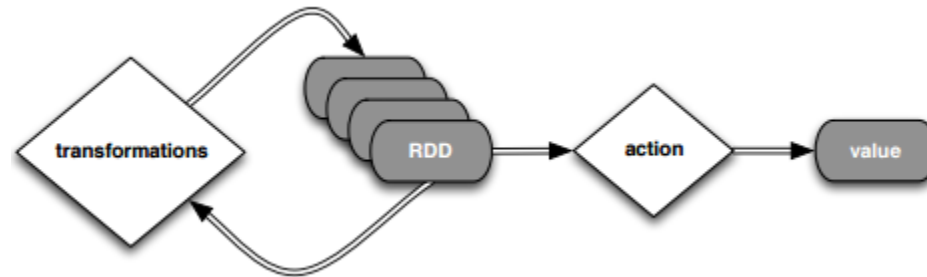
# Lineage Graph

- RDDs keep track of lineage
- RDD has enough information about how it was derived from to compute its partitions from data in stable storage



- Example:
  - If a partition of errors is lost, Spark rebuilds it by applying a filter on only the corresponding partition of lines
  - Partitions can be recomputed in parallel on different nodes, without having to roll back the whole program

# Deconstructed



*//base RDD*

```
val lines = sc.textFile("hdfs://...")
```

*//Transformed RDD*

```
val errors =
```

```
lines.filter(_.startsWith("Error"))
```

```
errors.persist()
```

```
errors.count()
```

```
errors.filter(_.contains("HDFS"))
```

```
.map(_.split('\t')(3))
```

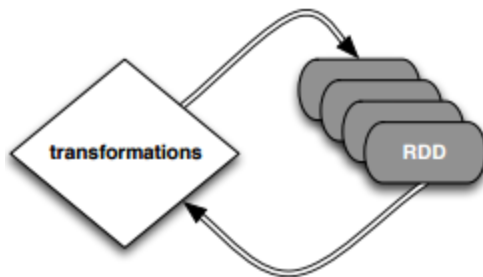
```
.collect()
```

# Deconstructed



*//base RDD*

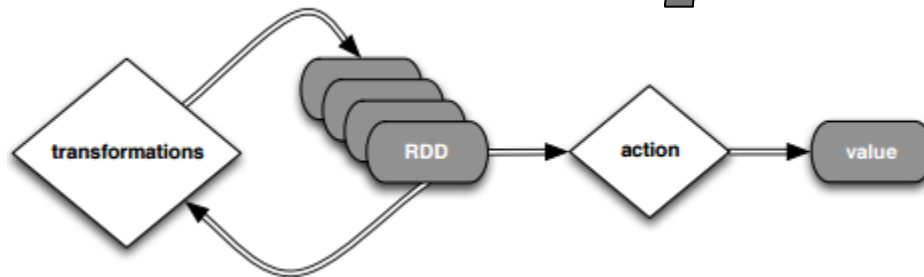
*val lines = sc.textFile("hdfs://...")*



*//Transformed RDD*

*val errors = lines.filter(\_.startsWith("Error"))*

*errors.persist()*



*errors.count()*

count() causes Spark to: 1)  
read data; 2) sum within  
partitions; 3) combine sums in  
driver

Put transform and action together:

*errors.filter(\_.contains("HDFS")).map(\_split('\t')(3)).collect()*

# SparkContext

- SparkContext is the entry point to Spark for a Spark application
- Once a SparkContext instance is created you can use it to
  - Create RDDs
  - Create accumulators
  - Create broadcast variables
  - Access Spark services and run jobs
- A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application*
- The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster
- In the Spark shell, a special interpreter-aware SparkContext is already created for you, in the variable called `sc`

## RDD Persistence: Cache/Persist

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations
- When you persist an RDD, each node stores any partitions of it. You can reuse it in other actions on that dataset
- Each persisted RDD can be stored using a different *storage level*, e.g.
  - MEMORY\_ONLY:
    - Store RDD as deserialized Java objects in the JVM
    - If the RDD does not fit in memory, some partitions will not be cached and will be recomputed when they're needed
    - This is the default level
  - MEMORY\_AND\_DISK:
    - If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed
- `cache()` = `persist(StorageLevel.MEMORY_ONLY)`



# Why Persisting RDD?

```
val lines = sc.textFile("hdfs://...")
```

```
val errors = lines.filter(_.startsWith("Error"))
```

```
errors.persist()
```

```
errors.count()
```

- If you do `errors.count()` again, the file will be loaded again and computed again
- `Persist` will tell Spark to cache the data in memory, to reduce the data loading cost for further actions on the same data
- `errors.persist()` will do nothing. It is a lazy operation. But now the RDD says "read this file and then cache the contents". The action will trigger computation and data caching

# Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a *Pair RDD* is a pair tuple
- Some Key-Value transformation functions:

Key-Value Transformation	Description
<code>reduceByKey(func)</code>	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function <i>func</i> , which must be of type $(V,V) \rightarrow V$
<code>sortByKey()</code>	return a new dataset (K,V) pairs sorted by keys in ascending order
<code>groupByKey()</code>	return a new dataset of (K, Iterable<V>) pairs

# More Examples on Pair RDD

- Create a pair RDD

```
val pairs = sc.parallelize( List( ("This", 2), ("is", 3), ("Spark", 5), ("is", 3) ) )  
pairs.collect().foreach(println)
```

Output?

- reduceByKey() function: reduce key-value pairs by key using give *func*

```
val pairs1 = pairs.reduceByKey((x,y) => x + y)  
pairs1.collect().foreach(println)
```

Output?

- mapValues() function: work on values only

```
val pairs2 = pairs.mapValues( x => x - 1 )  
pairs2.collect().foreach(println)
```

Output?

- groupByKey() function: When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs

```
pairs.groupByKey().collect().foreach(println)
```

# Setting the Level of Parallelism

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

> words.reduceByKey((x,y) => x + y, 5)

> words.groupByKey(5)

# Spark Programming Model

# How Spark Works?

- User application create RDDs, transform them, and run actions
- This results in a DAG (Directed Acyclic Graph) of operators
- DAG is compiled into stages
- Each stage is executed as a series of Task (one Task for each Partition)

```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                  .map(word => (word,1))
                  .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

# Word Count in Spark

```
val file = sc.textFile("hdfs://...", 4)
val words = file.flatMap(line =>
    line.split(" "))
val pairs = words.map(t => (t, 1))
val count = pairs.reduceByKey(_+_ )
count.collect()
```

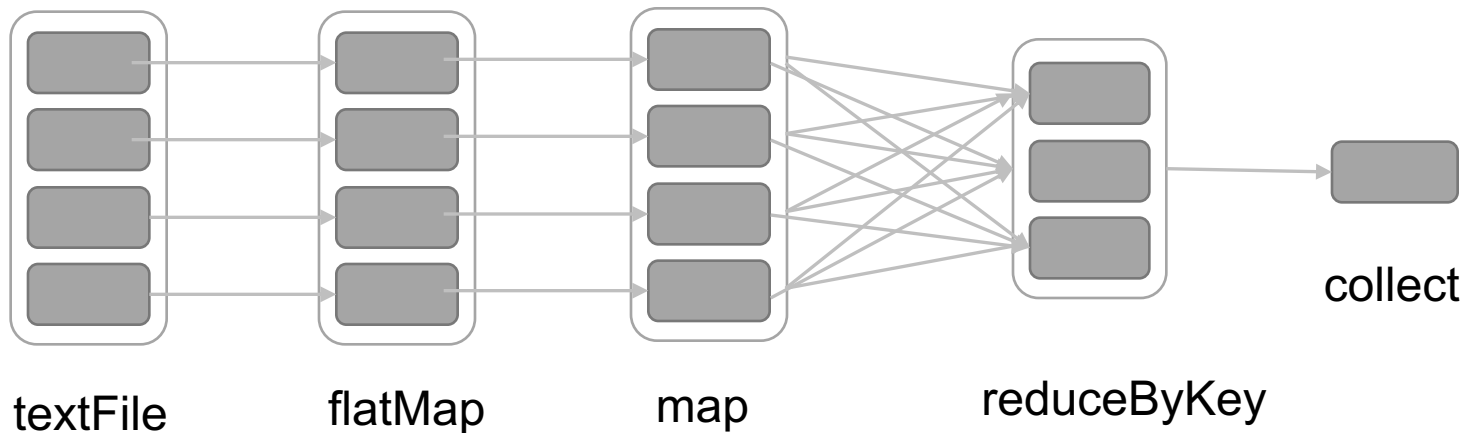
RDD[String]

RDD[List[String]]

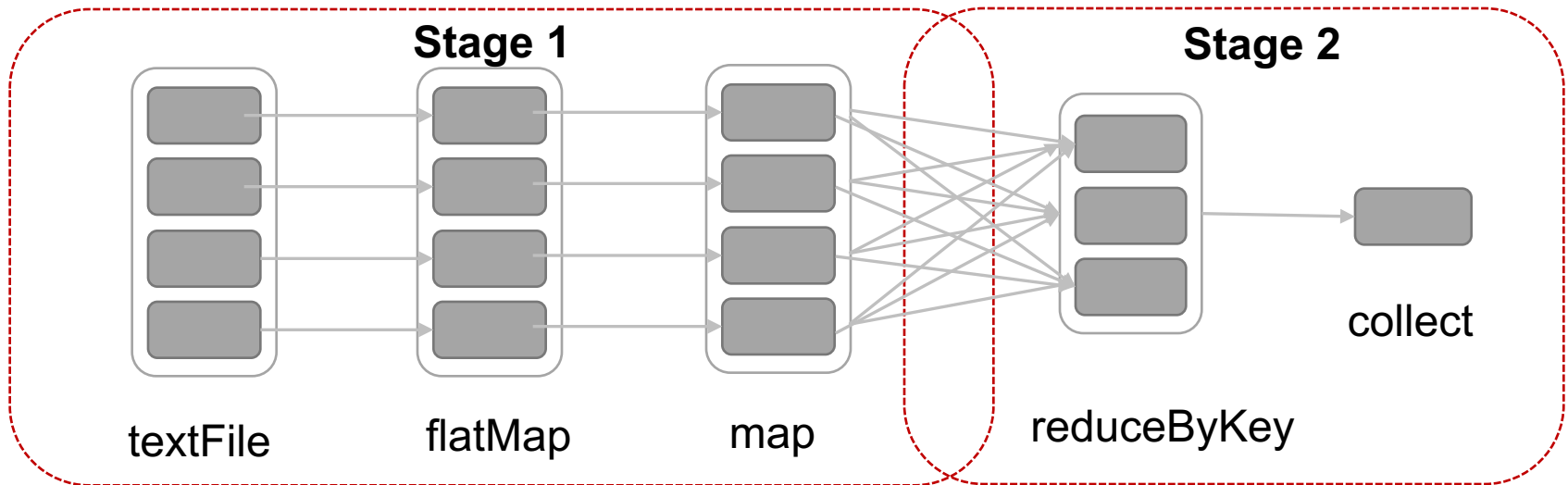
RDD[(String, Int)]

RDD[(String, Int)]

Array[(String, Int)]



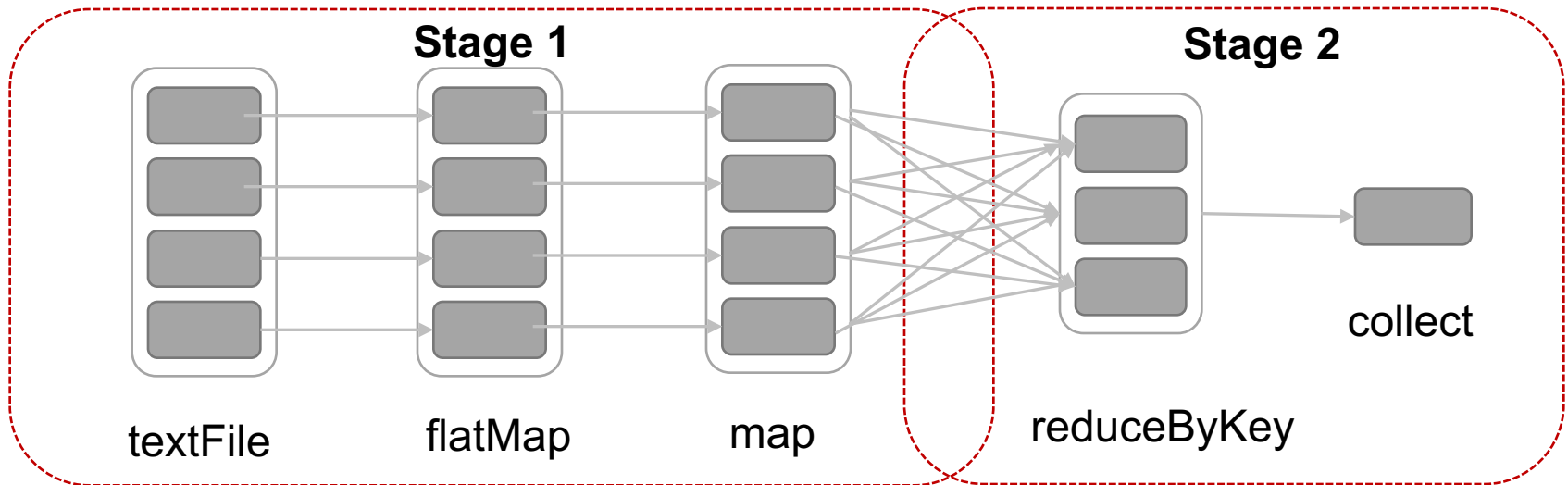
# Execution Plan



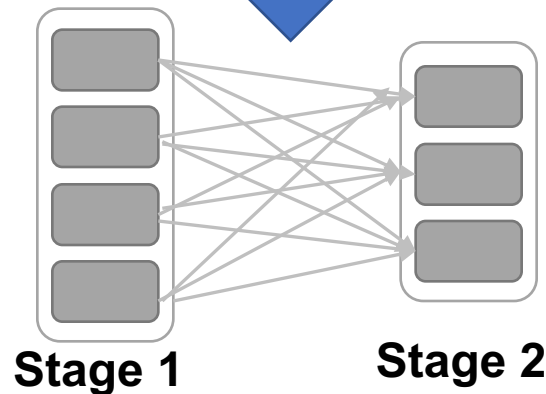
- The scheduler examines the RDD's lineage graph to build a DAG of stages
- Stages are sequences of RDDs, that don't have a Shuffle in between
- The boundaries are the shuffle stages



# Execution Plan



1. Read HDFS split
2. Apply both the maps
3. Start Partial reduce
4. Write shuffle data



1. Read shuffle data
2. Final reduce
3. Send result to driver program

# Stage Execution



- Create a task for each Partition in the new RDD
- Serialize the Task
- Schedule and ship Tasks to Slaves
- All this happens internally

# Word Count in Spark (As a Whole View)

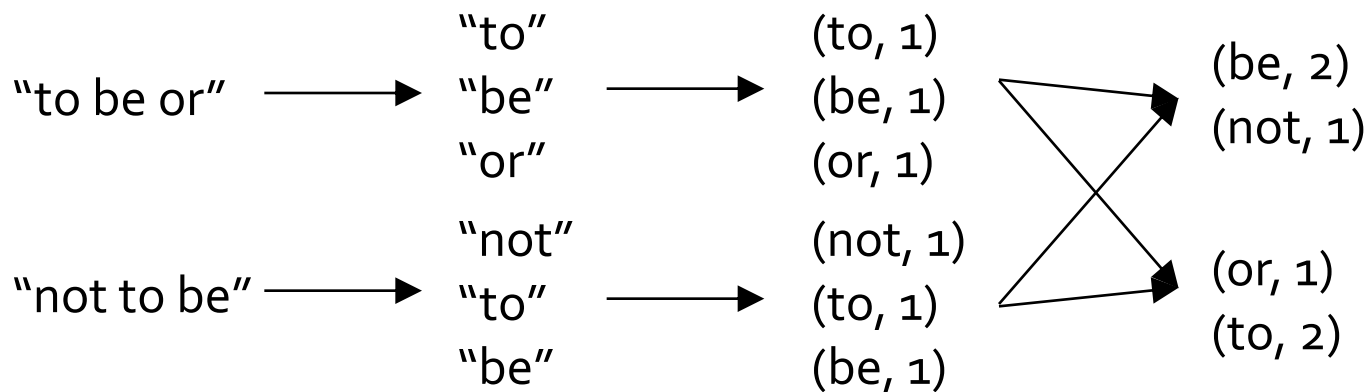
- Word Count using Scala in Spark

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

Transformation

```
counts.saveAsTextFile("hdfs://...")
```

Action



# map vs. flatMap

- Sample input file:

```
comp9313@comp9313-VirtualBox:~$ hdfs dfs -cat inputfile
This is a short sentence.
This is a second sentence.
```

```
scala> val inputfile = sc.textFile("inputfile")
inputfile: org.apache.spark.rdd.RDD[String] = inputfile MapPartitionsRDD[1] at t
extFile at <console>:24
```

- map: Returns a new distributed dataset formed by passing each element of the source through a function *func*

```
scala> inputfile.map(x => x.split(" ")).collect()
res3: Array[Array[String]] = Array(Array(This, is, a, short, sentence.), Array(T
his, is, a, second, sentence.))
```

- flatMap: Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a **Seq** rather than a **single item**)

```
scala> inputfile.flatMap(x => x.split(" ")).collect()
res4: Array[String] = Array(This, is, a, short, sentence., This, is, a, second,
sentence.)
```

# RDD Operations

<b>Transformations</b>	$map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
<b>Actions</b>	$count() : RDD[T] \Rightarrow Long$ $collect() : RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T) : RDD[T] \Rightarrow T$ $lookup(k : K) : RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String) : \text{Outputs RDD to a storage system, e.g., HDFS}$

Spark RDD API Examples:

<http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html>

# Self-Contained Applications

# WordCount (Scala)

- Standalone code

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object WordCount {
  def main(args: Array[String]) {
    val inputFile = args(0)
    val outputFolder = args(1)
    val conf = new SparkConf().setAppName("wordCount").setMaster("local")
    // Create a Scala Spark Context.
    val sc = new SparkContext(conf)
    // Load our input data.
    val input = sc.textFile(inputFile)
    // Split up into words.
    val words = input.flatMap(line => line.split(" "))
    // Transform into word and count.
    val counts = words.map(word => (word, 1)).reduceByKey(_+_ )
    counts.saveAsTextFile(outputFolder)
  }
}
```

# WordCount (Scala)

- Linking with Apache Spark

- The first step is to explicitly import the required spark classes into your Spark program

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

- Initializing Spark

- Create a Spark context object with the desired spark configuration that tells Apache Spark on how to access a cluster

```
val conf = new SparkConf().setAppName("wordCount").setMaster("local")
val sc = new SparkContext(conf)
```

- SparkConf: Spark configuration class
- setAppName: set the name for your application
- setMaster: set the cluster master URL



# setMaster

- Set the cluster master URL to connect to
- Parameters for setMaster:
  - local(default) - run locally with only one worker thread (no parallel)
  - local[k] - run locally with k worker threads
  - spark://HOST:PORT - connect to Spark standalone cluster URL
  - mesos://HOST:PORT - connect to Mesos cluster URL
  - yarn - connect to Yarn cluster URL
    - Specified in SPARK\_HOME/conf/yarn-site.xml
- setMaster parameters configurations:
  - In source code
    - SparkConf().setAppName("wordCount").setMaster("local")
  - spark-submit
    - spark-submit --master local
  - In SPARK\_HOME/conf/spark-default.conf
    - Set value for spark.master

# WordCount (Scala)

- Creating a Spark RDD

- Create an input Spark RDD that reads the text file input.txt using the Spark Context created in the previous step

```
val input = sc.textFile(inputFile)
```

- Spark RDD Transformations in Wordcount Example

- flatMap() is used to tokenize the lines from input text file into words
- map() method counts the frequency of each word
- reduceByKey() method counts the repetitions of word in the text file

- Save the results to disk

```
counts.saveAsTextFile(outputFolder)
```

# Passing Functions to RDD

- Spark's API relies heavily on passing functions in the driver program to run on the cluster
  - Anonymous function. E.g.,
    - `val words = input.flatMap(line => line.split(" "))`
  - Static methods in a global singleton object. E.g.,
    - `object MyFunctions { def func1(s: String): String = { ... } }`  
`myRdd.map(MyFunctions.func1)`

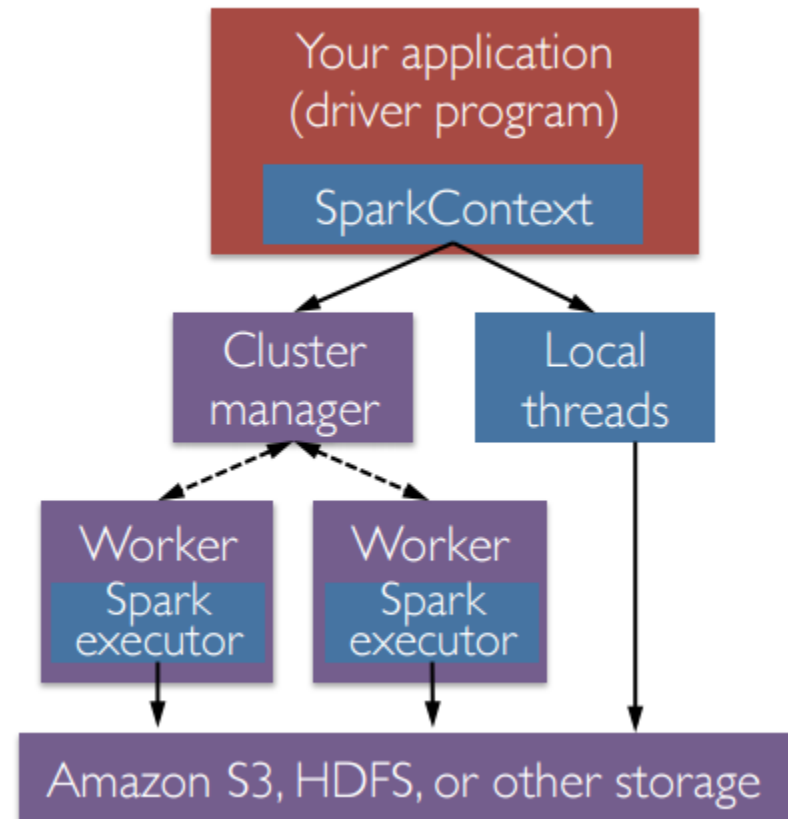
# Run the Application on a Cluster

- A Spark application is launched on a set of machines using an external service called a cluster manager

- Local threads
- Standalone
- Mesos
- Yarn

- Driver

- Executor



# Launching a Program

- Spark provides a single script you can use to submit your program to it called **spark-submit**
  - The user submits an application using spark-submit
  - spark-submit launches the driver program and invokes the `main()` method specified by the user
  - The driver program contacts the cluster manager to ask for resources to launch executors
  - The cluster manager launches executors on behalf of the driver program
  - The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks
  - Tasks are run on executor processes to compute and save results
  - If the driver's `main()` method exits or it calls `SparkContext.stop()`, it will terminate the executors and release resources from the cluster manager

# Package Your Code and Dependencies

- Ensure that all your dependencies are present at the runtime of your Spark application
- Java Application (Maven)
- Scala Application (sbt)
  - a newer build tool most often used for Scala projects

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" %
"2.3.1"
```

- libraryDependencies: list all dependent libraries (including third party libraries)
- A jar file simple-project\_2.11-1.0.jar will be created after compilation

# Deploying Applications in Spark


- spark-submit

Common flags	Explanation
--master	Indicates the cluster manager to connect to
--class	The “main” class of your application if you’re running a Java or Scala program
--name	A human-readable name for your application. This will be displayed in Spark’s web UI.
--executor-memory	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as “512m” (512 megabytes) or “15g” (15 gigabytes)
--driver-memory	The amount of memory to use for the driver process, in bytes.

```
➤ spark-submit --master spark://hostname:7077 \  
  --class YOURCLASS \  
  --executor-memory 2g \  
  YOURJAR "options" "to your application" "go here"
```

# Spark Web Console

- You can browse the web interface for the information of Spark Jobs, storage, etc. at: <http://localhost:4040>

 2.3.0

Jobs | Stages | Storage | Environment | Executors

Spark shell application UI

## Spark Jobs (?)

User: comp9313  
Total Uptime: 8.7 min  
Scheduling Mode: FIFO  
Completed Jobs: 4  
[▶ Event Timeline](#)

### Completed Jobs (4)

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	reduce at <console>:26 <a href="#">reduce at &lt;console&gt;:26</a>	2018/04/14 17:01:23	38 ms	1/1	1/1
2	reduce at <console>:26 <a href="#">reduce at &lt;console&gt;:26</a>	2018/04/14 17:00:08	45 ms	1/1	1/1
1	first at <console>:26 <a href="#">first at &lt;console&gt;:26</a>	2018/04/14 16:55:54	21 ms	1/1	1/1
0	count at <console>:26 <a href="#">count at &lt;console&gt;:26</a>	2018/04/14 16:55:38	0.5 s	1/1	1/1



Thanks