

Spark

Fast, Interactive, Language-Integrated
Cluster Computing

Project Goals

Extend the MapReduce model to better support two common classes of analytics apps:

- >> Iterative algorithms (machine learning, graph)
- >> Interactive data mining

Enhance programmability:

- >> Integrate into Scala programming language
- >> Allow interactive use from Scala interpreter

Motivation

- MapReduce greatly simplified “big data” analysis on large, unreliable clusters
- But as soon as it got popular, users wanted more:
 - More **complex**, multi-stage applications (e.g. iterative machine learning & graph processing)
 - More **interactive** ad-hoc queries

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

Motivation

- Complex apps and interactive queries both need one thing that MapReduce lacks:
 - Efficient primitives for **data sharing**

In MapReduce, the only way to share data across jobs is stable storage → slow!

Memory vs Disk

If Memory = **Minute**
Network = **Weeks**
Flash = **Months**
Disk = **Decades**

RAM Latency 83 nanoseconds



Length of a Manhattan City Block

Disk Latency 13 milliseconds



11x the distance from San Francisco to NYC



Craigslist Revenue



United States Gross Domestic Product

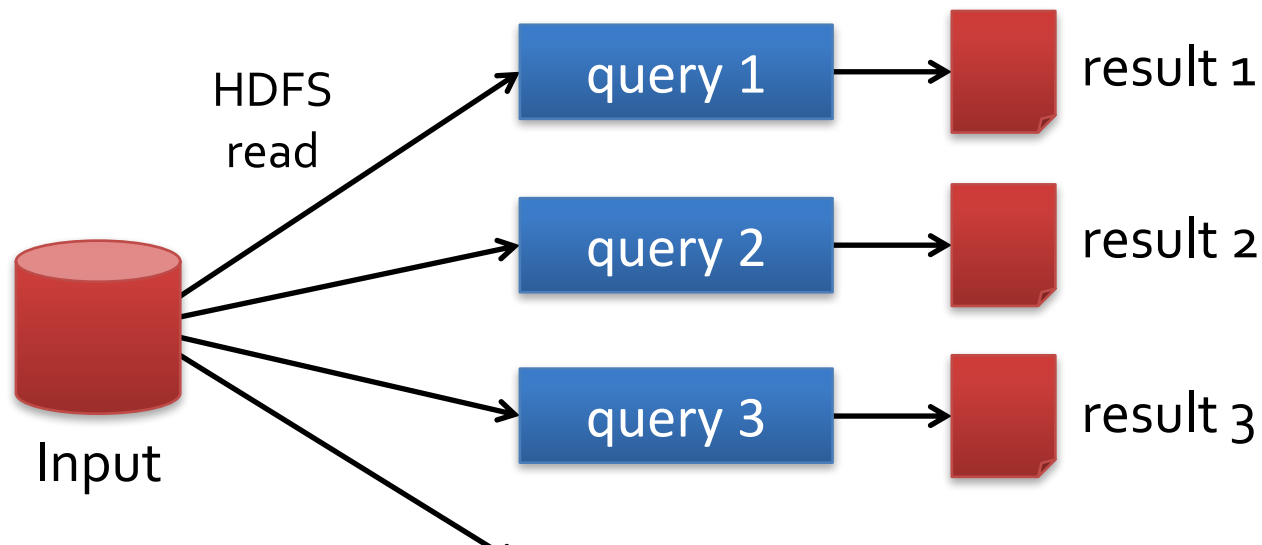
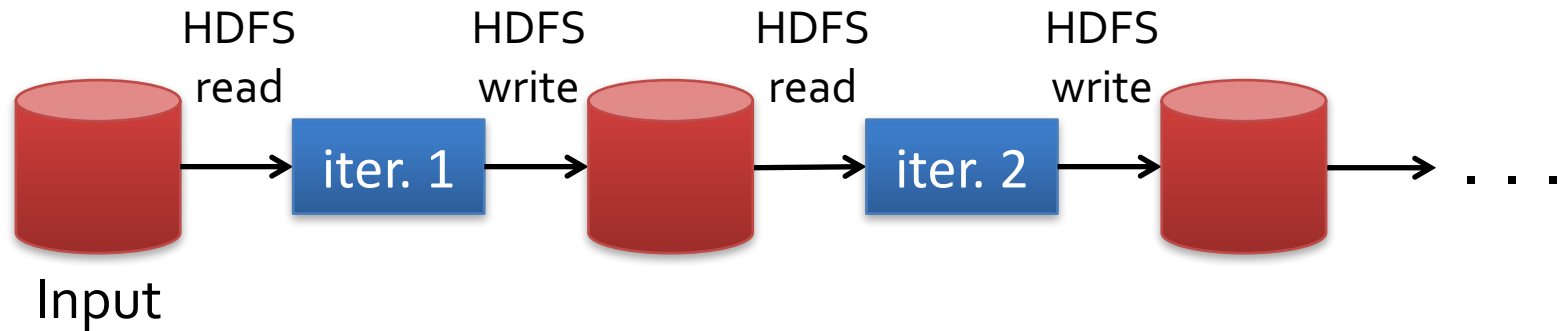


F-18 Hornet Max Speed



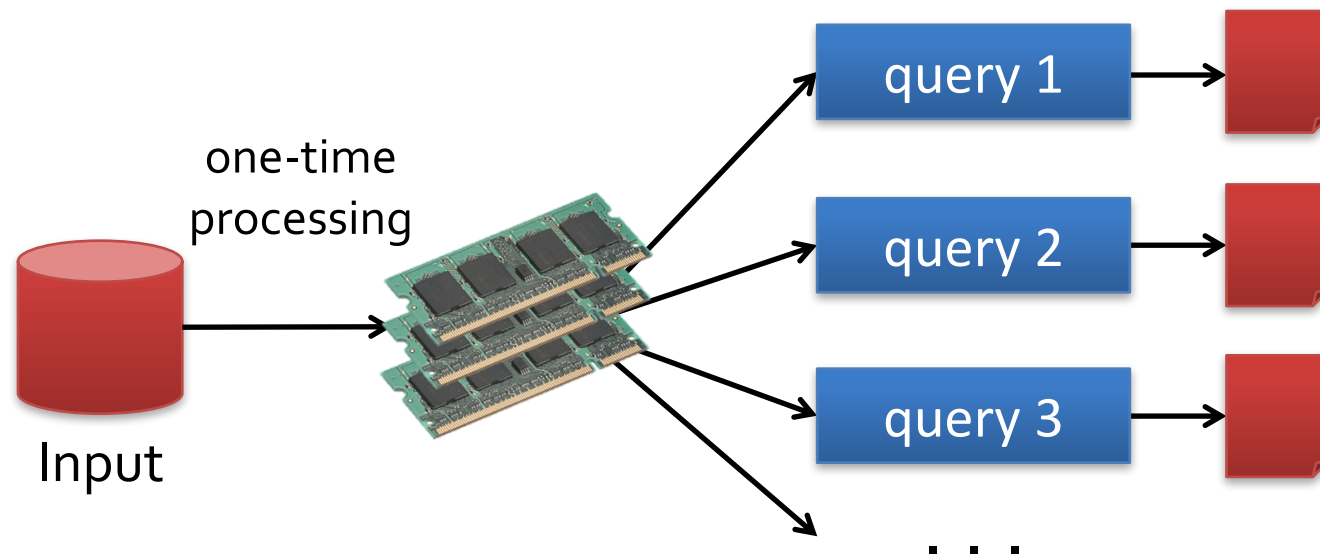
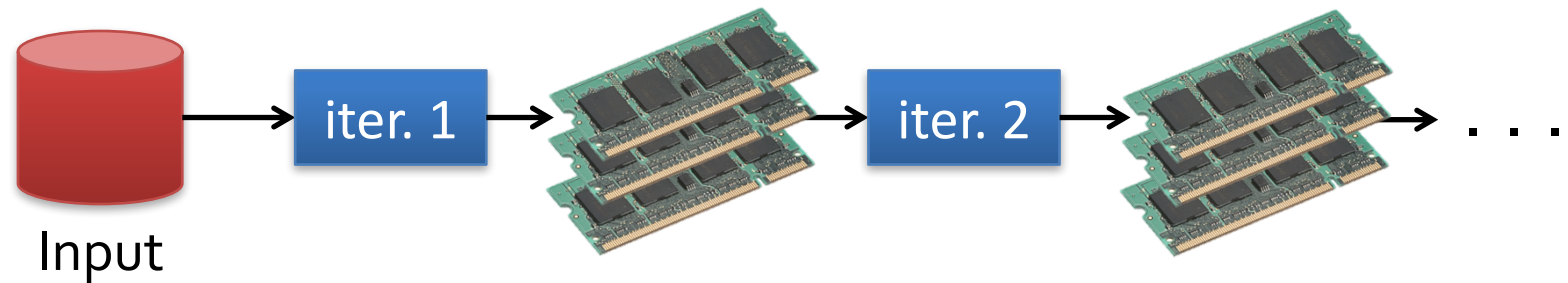
Banana Slug Max Speed

Examples



Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-Memory Data Sharing



10-100× faster than network/disk, but how to get FT?

Challenge

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

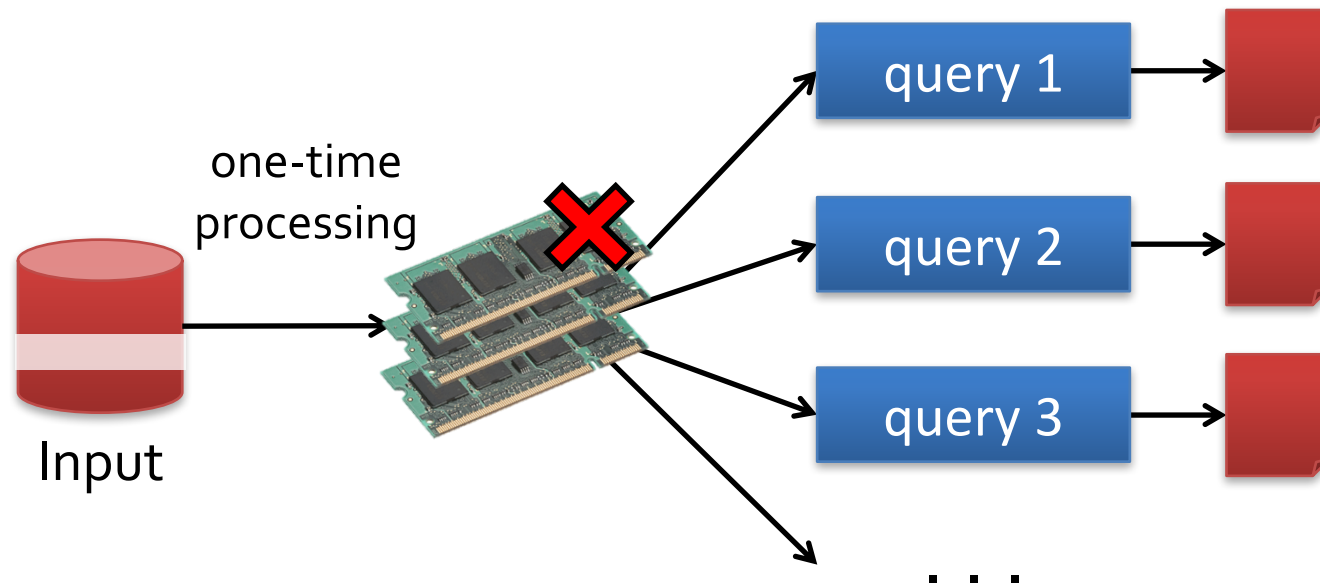
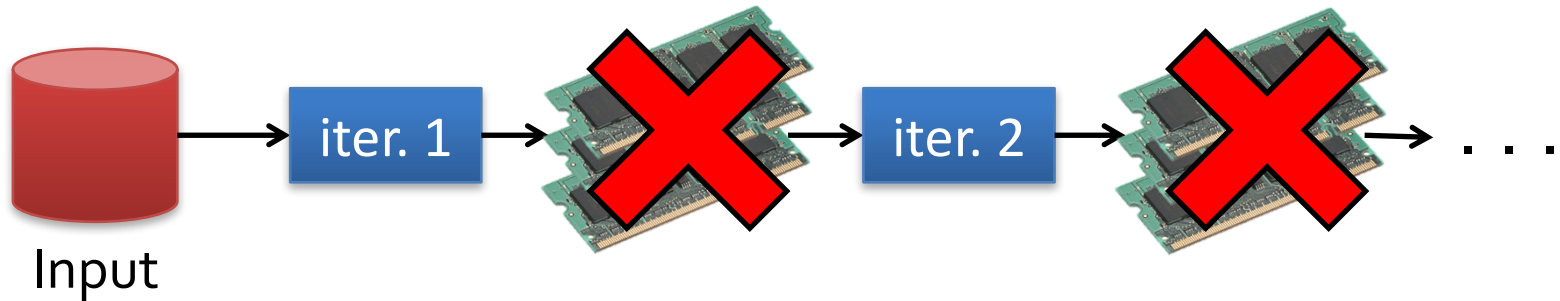
Challenge

- Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state
 - RAMCloud, databases, distributed mem, Piccolo
- Requires replicating data or logs across nodes for fault tolerance
 - Costly for data-intensive apps
 - 10-100x slower than memory write

Solution: Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
 - Immutable, partitioned collections of records
 - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, ...)
- Efficient fault recovery using *lineage*
 - Log one operation to apply to many elements
 - Recompute lost partitions on failure
 - No cost if nothing fails

RDD Recovery



Generality of RDDs

- Despite their restrictions, RDDs can express surprisingly many parallel algorithms
 - These naturally *apply the same operation to multiple items*
- Unify many current programming models
 - *Data flow models*: MapReduce, Dryad, SQL, ...
 - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, ...
- Support *new apps* that these models don't

Outline

- Introduction to Scala & functional programming
- What is Spark
- Resilient Distributed Datasets (RDDs)
- Implementation
- Demo
- Conclusion

About Scala

High-level language for JVM

- >> Object-oriented + Functional programming (FP)

Statically typed

- >> Comparable in speed to Java

- >> no need to write types due to type inference

Interoperates with Java

- >> Can use any Java class, inherit from it, etc;

- >> Can also call Scala code from Java

Quick Tour

Declaring variables:

```
var x: Int = 7
var x = 7 // type inferred
val y = "hi" // read-only
```

Java equivalent:

```
int x = 7;
final String y = "hi";
```

Functions:

```
def square(x: Int): Int = x*x
def square(x: Int): Int = {
  x*x
}
```

Last expression in block returned

Java equivalent:

```
int square(int x) {
  return x*x;
}

void announce(String text) {
  System.out.println(text);
}
```

Quick Tour

Generic types:

```
var arr = new Array[Int](8)
```

```
var lst = List(1, 2, 3)
```

```
// type of lst is List[Int]
```

Java equivalent:

```
int[] arr = new int[8];
```

```
List<Integer> lst =  
    new ArrayList<Integer>();  
lst.add(...)
```

Indexing:

```
arr(5) = 7
```

```
println(lst(5))
```

Java equivalent:

```
arr[5] = 7;
```

```
System.out.println(lst.get(5));
```


Quick Tour

Processing collections with functional programming:

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

Function expression (closure)

```
list.foreach(x => println(x)) // prints 1, 2, 3  
list.foreach(println)         // same
```

```
list.map(x => x + 2) // => List(3, 4, 5)  
list.map(_ + 2)      // same, with placeholder notation
```

```
list.filter(x => x % 2 == 1) // => List(1, 3)  
list.filter(_ % 2 == 1)     // => List(1, 3)
```

```
list.reduce((x, y) => x + y) // => 6  
list.reduce(_ + _)           // => 6
```

All of these leave the list unchanged (List is Immutable)

Scala Closure Syntax

```
(x: Int) => x + 2    // full version
```


```
x => x + 2    // type inferred
```

```
_ + 2        // when each argument is used exactly once
```

```
x => {          // when body is a block of code  
  val numberToAdd = 2  
  x + numberToAdd  
}
```

```
// If closure is too long, can always pass a function  
def addTwo(x: Int): Int = x + 2
```

```
list.map(addTwo)
```



Scala allows defining a “local function” inside another function

Other Collection Methods

Scala collections provide many other functional methods; for example, Google for “Scala Seq”

Method on Seq[T]	Explanation
<code>map(f: T => U): Seq[U]</code>	Pass each element through f
<code>flatMap(f: T => Seq[U]): Seq[U]</code>	One-to-many map
<code>filter(f: T => Boolean): Seq[T]</code>	Keep elements passing f
<code>exists(f: T => Boolean): Boolean</code>	True if one element passes
<code>forall(f: T => Boolean): Boolean</code>	True if all elements pass
<code>reduce(f: (T, T) => T): T</code>	Merge elements using f
<code>groupBy(f: T => K): Map[K, List[T]]</code>	Group elements by f(element)
<code>sortBy(f: T => K): Seq[T]</code>	Sort elements by f(element)
<code>. . .</code>	

Map vs FlatMap

```
scala> def f(x: Int) = if (x > 2) Some(x) else None

scala> l.map(x => f(x))
res63: List[Option[Int]] = List(None, None, Some(3), Some(4), Some(5))
```

```
scala> def g(v: Int) = List(v-1, v, v+1)
g: (v: Int)List[Int]

scala> l.map(x => g(x))
res64: List[List[Int]] = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4), List(3, 4, 5), Li

scala> l.flatMap(x => g(x))
res65: List[Int] = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
```

flatMap flattens the list i.e. converts multiple lists to a single list.

Key thing to know about Spark

- What is the entry point for using Spark functionality?
 - SparkContext (sc) object
 - It represents the **connection to a Spark cluster**, and can be **used to create RDDs, accumulators and broadcast variables on that cluster.**
 - **Almost every line of code starts with sc object.**

RDD in detail

- What are RDD?
 - fault-tolerant in-memory collection of elements that can be operated on in parallel.
- 2 ways of creating RDDs?
 - parallelize an existing collection
 - reference an existing dataset in external storage system e.g. HDFS, Hbase, etc

Parallelize RDD

```
val data = Array(1, 2, 3, 4, 5)
```

```
val distData = sc.parallelize(data)
```

Create RDD from HDFS file

```
val distFile = sc.textFile("data.txt")
```

By default, Spark creates one partition for each block of the file (blocks being 64MB by default in HDFS).

What other files are supported by Spark

- wholeTextFiles -> entire directory can be read
- SequenceFiles -> maps to Hadoop's sequenceFiles, which are highly efficient binary representation of Hadoop data.
- sc.hadoopRDD -> takes arbitrary inputformat class object and converts it into RDD

Operations

RDDs support two types of operations:

1. Transformations -> which create a new dataset from existing one.

e.g. map

2. Action -> returns a value to the driver program after running a computation on the dataset.

e.g. reduce

Operations

All transformations in Spark are **lazy**

⇒ They do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file).

⇒ The transformations are only computed when an action requires a result to be returned to the driver program.

Illustration

```
val lines = sc.textFile("data.txt")  
val lineLengths = lines.map(s => s.length)  
val totalLength = lineLengths.reduce((a, b) => a + b)  
=> When is the dataset loaded in memory?
```

Illustration

```
val lines = sc.textFile("data.txt")  
val lineLengths = lines.map(s => s.length)  
val totalLength = lineLengths.reduce((a, b) => a + b)
```

⇒ When is the dataset loaded in memory?

⇒ First line:

This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file.

Illustration

```
val lines = sc.textFile("data.txt")  
val lineLengths = lines.map(s => s.length)  
val totalLength = lineLengths.reduce((a, b) => a + b)
```

⇒ When is the dataset loaded in memory?

⇒ Second line:

The second line defines lineLengths as the result of a map transformation. Again, lineLengths is not immediately computed, due to laziness.

Illustration

```
val lines = sc.textFile("data.txt")  
val lineLengths = lines.map(s => s.length)  
val totalLength = lineLengths.reduce((a, b) => a + b)
```

⇒ When is the dataset loaded in memory?

⇒ Third line:

We run reduce, which is an action. At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction,

MapReduce with Spark RDD

```
val textFile = sc.textFile("README.md")  
textFile.map(line => line.split(" ").size).reduce((a, b)  
=> if (a > b) a else b)
```

*NOTE: Input to map is a closure function closure.

MapReduce conversion to Spark

```
public class LineLengthMapper
    extends Mapper<LongWritable,Text,IntWritable,IntWritable> {
    @Override
    protected void map(LongWritable lineNumber, Text line, Context context)
        throws IOException, InterruptedException {
        context.write(new IntWritable(line.getLength()), new IntWritable(1));
    }
}
```

The Spark equivalent is:

```
lines.map(line => (line.length, 1))
```

MapReduce conversion to Spark

```
public class LineLengthReducer
    extends Reducer<IntWritable,IntWritable,IntWritable,IntWritable> {
    @Override
    protected void reduce(IntWritable length, Iterable<IntWritable> counts, Context context
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable count : counts) {
            sum += count.get();
        }
        context.write(length, new IntWritable(sum));
    }
}
```

```
val lengthCounts = lines.map(line => (line.length, 1)).reduceByKey(_ + _)
```

MapReduce conversion to Spark

Spark's RDD API has a `reduce()` method, but it will reduce the entire set of key-value pairs to one single value. This is not what Hadoop MapReduce does. Instead, Reducers reduce all values for a key and emit a key along with the reduced value. `reduceByKey()` is the closer analog. But, that is not even the most direct equivalent in Spark; see `groupByKey()` below.

MapReduce Example

- If you want to map only on those words that begin with uppercase.

```
public class CountUppercaseMapper
    extends Mapper<LongWritable,Text,Text,IntWritable> {
    @Override
    protected void map(LongWritable lineNumber, Text line, Context context)
        throws IOException, InterruptedException {
        for (String word : line.toString().split(" ")) {
            if (Character.isUpperCase(word.charAt(0))) {
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

```
lines.flatMap(
    _.split(" ").filter(word => Character.isUpperCase(word(0))).map(word => (word,1))
)
```

Outline

- Introduction to Scala & functional programming
- What is Spark
- Resilient Distributed Datasets (RDDs)
- Implementation
- Demo
- Conclusion

Spark Overview

Goal: work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)

- >> Immutable collections of objects spread across a cluster
- >> Built through parallel *transformations* (*map*, *filter*, etc)
- >> Automatically rebuilt on failure
- >> Controllable *persistence* (e.g. caching in RAM) for reuse
- >> *Shared variables* that can be used in parallel operations

Outline

- Introduction to Scala & functional programming
- What is Spark
- Resilient Distributed Datasets (RDDs)
- Implementation
- Demo
- Conclusion

RDD Abstraction

An RDD is a read-only , partitioned collection of records

Can only be created by :

(1) Data in stable storage

(2) Other RDDs (transformation , lineage)

An RDD has enough information about how it was derived from other datasets(its lineage)

Users can control two aspects of RDDs:

1) *Persistence* (in RAM, reuse)

2) *Partitioning* (hash, range, [k, v])

RDDs in More Detail

- An RDD is an immutable, partitioned, logical collection of records
 - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Built using bulk transformations on other RDDs
- Can be cached for future reuse

RDD Types: parallelized collections

By calling SparkContext's parallelize method on an existing Scala collection (a Seq obj)

```
scala> val data = Array(1,2,3,4,5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@3b9c5ce6
```

Once created, the distributed dataset can be operated on in parallel

RDD Types: Hadoop Datasets

Spark supports text files, SequenceFiles, and any other Hadoop inputFormat

Local path or hdfs://, s3n://, kfs://

```
val distFiles = sc.textFile(URI)
```

Other Hadoop inputFormat

```
val distFile = sc.hadoopRDD(URI)
```

RDD Operations

Transformations

>> create a new dataset from an existing one

Actions

>> Return a value to the driver program

Transformations are *lazy*, they don't compute right away. Just remember the transformations applied to datasets(*lineage*). Only compute when an action require.

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Transformations

Transformations	Meaning
<i>map(func)</i>	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
<i>flatMap(func)</i>	Return a new datasets formed by selecting those elements of the source on which <i>func</i> returns true
<i>union(otherDataset)</i>	Return a new dataset that contains the union of the elements in the source dataset and the argument
...	...

Actions

Actions	Meaning
<i>reduce(func)</i>	Aggregate the elements of the dataset using a function <i>func</i>
<i>collect()</i>	Return all the elements of the dataset as an array at the driver program
<i>count()</i>	Return the number of elements in dataset
<i>first()</i>	Return the first element of the dataset
<i>saveAsTextFile(path)</i>	Write the elements of the dataset as text file (or set of text file) in a given dir in the local file system, HDFS or any other Hadoop-supported file system
.....

Differences between map and flatMap – converting String to Int

The following examples show more differences between `map` and `flatMap` for a simple `String` to `Int` conversion example. Given this `toInt` method:

```
def toInt(s: String): Option[Int] = {  
  try {  
    Some(Integer.parseInt(s.trim))  
  } catch {  
    // catch Exception to catch null 's'  
    case e: Exception => None  
  }  
}
```

Here are a few examples to show how `map` and `flatMap` work on a simple list of strings that you want to convert to `Int`:

```
scala> val strings = Seq("1", "2", "foo", "3", "bar")  
strings: Seq[java.lang.String] = List(1, 2, foo, 3, bar)  
  
scala> strings.map(toInt)  
res0: Seq[Option[Int]] = List(Some(1), Some(2), None, Some(3), None)  
  
scala> strings.flatMap(toInt)  
res1: Seq[Int] = List(1, 2, 3)  
  
scala> strings.flatMap(toInt).sum  
res2: Int = 6
```


Memory Management

Spark provides three options for persist RDDs:

(1) in-memory storage as deserialized Java Objs

>> fastest, JVM can access RDD natively

(2) in-memory storage as serialized data

>> space limited, choose another efficient representation, lower performance cost

(3) on-disk storage

>> RDD too large to keep in memory, and costly to recompute

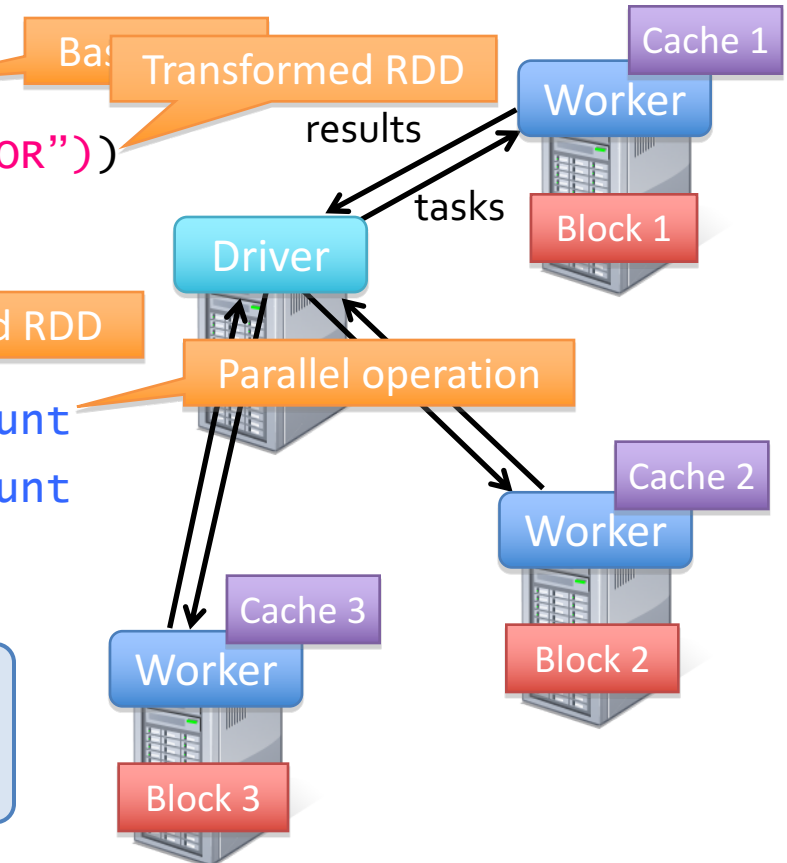
Example: Log Mining

- Load error messages from a log into 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: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

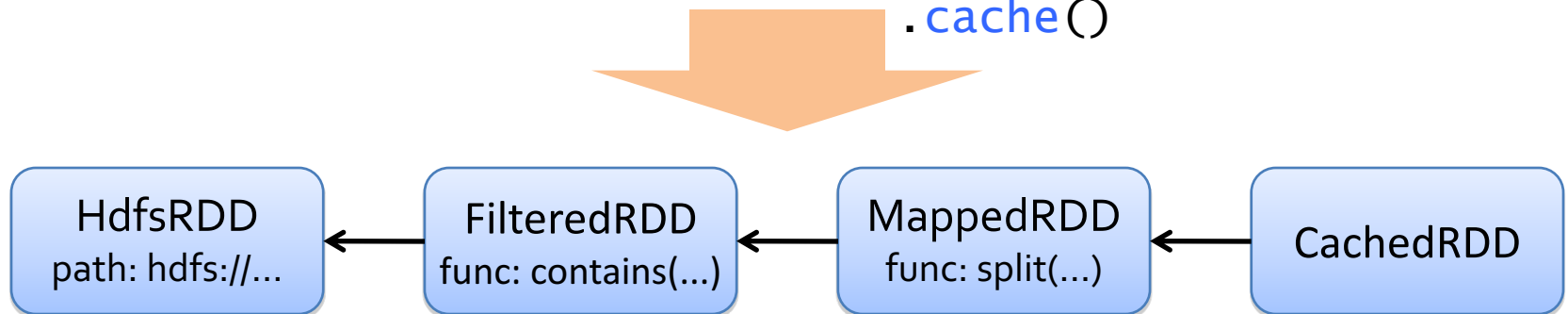


RDD Fault Tolerance

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions

- Ex:

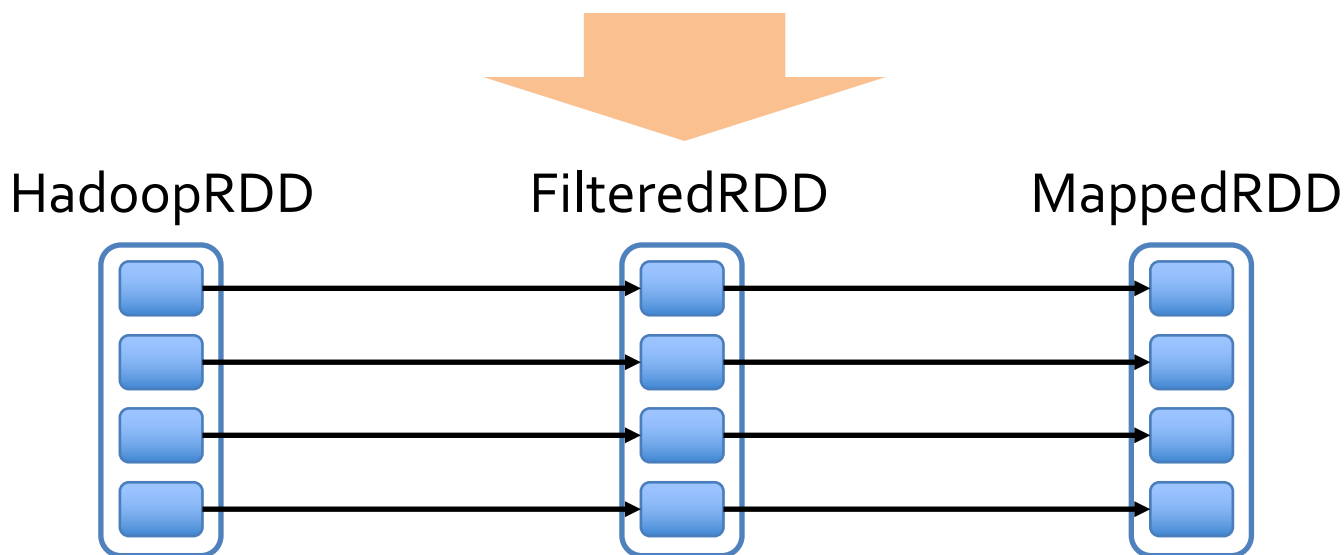
```
cachedMsgs =  
  textFile(...).filter(_.contains("error"))  
                  .map(_.split('\t')(2))  
                  .cache()
```



Fault Recovery

- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.: `messages = textFile(...).filter(_.contains("error")).map(_.split('\t')(2))`



Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

RDDs vs Distributed Shared Memory

Concern	RDDs	Distr. Shared Mem.
Reads	Fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using speculative execution	Difficult
Work placement	Automatic based on data locality	Up to app (but runtime aims for transparency)

Representing RDDs

Challenge: choosing a representation for RDDs that can track lineage across transformations

Each RDD include:

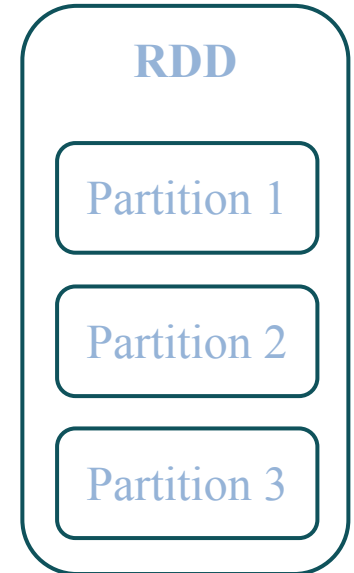
- 1) A set of partitions(atomic pieces of datasets)
- 2) A set of dependencies on parent RDDs
- 3) A function for computing the dataset based its parents
- 4) Metadata about its partitioning scheme
- 5) Data placement

Interface used to represent RDDs

Operation	Meaning
<i>partitions()</i>	Return a list of partition objects
<i>preferredLocations(p)</i>	List nodes where partition p can be accessed faster due to data locality
<i>dependencies()</i>	Return a list of dependencies
<i>iterator(p, parentIter)</i>	Compute the elements of partition p given iterators for its parent partitions
<i>partitioner()</i>	Return metadata specifying whether the RDD is hash/range partitioned

Internals of the RDD Interface

- 1) List of **partitions**
- 2) Set of **dependencies** on parent RDDs
- 3) Function to **compute** a partition, given parents
- 4) Optional **partitioning info** for k/v RDDs (Partitioner)



Example: Hadoop RDD

Partitions = 1 per HDFS block

Dependencies = None

compute(partition) = read corresponding HDFS block

Partitioner = None

```
> rdd = spark.hadoopFile("hdfs://click_logs/")
```

Example: Filtered RDD

Partitions = parent partitions

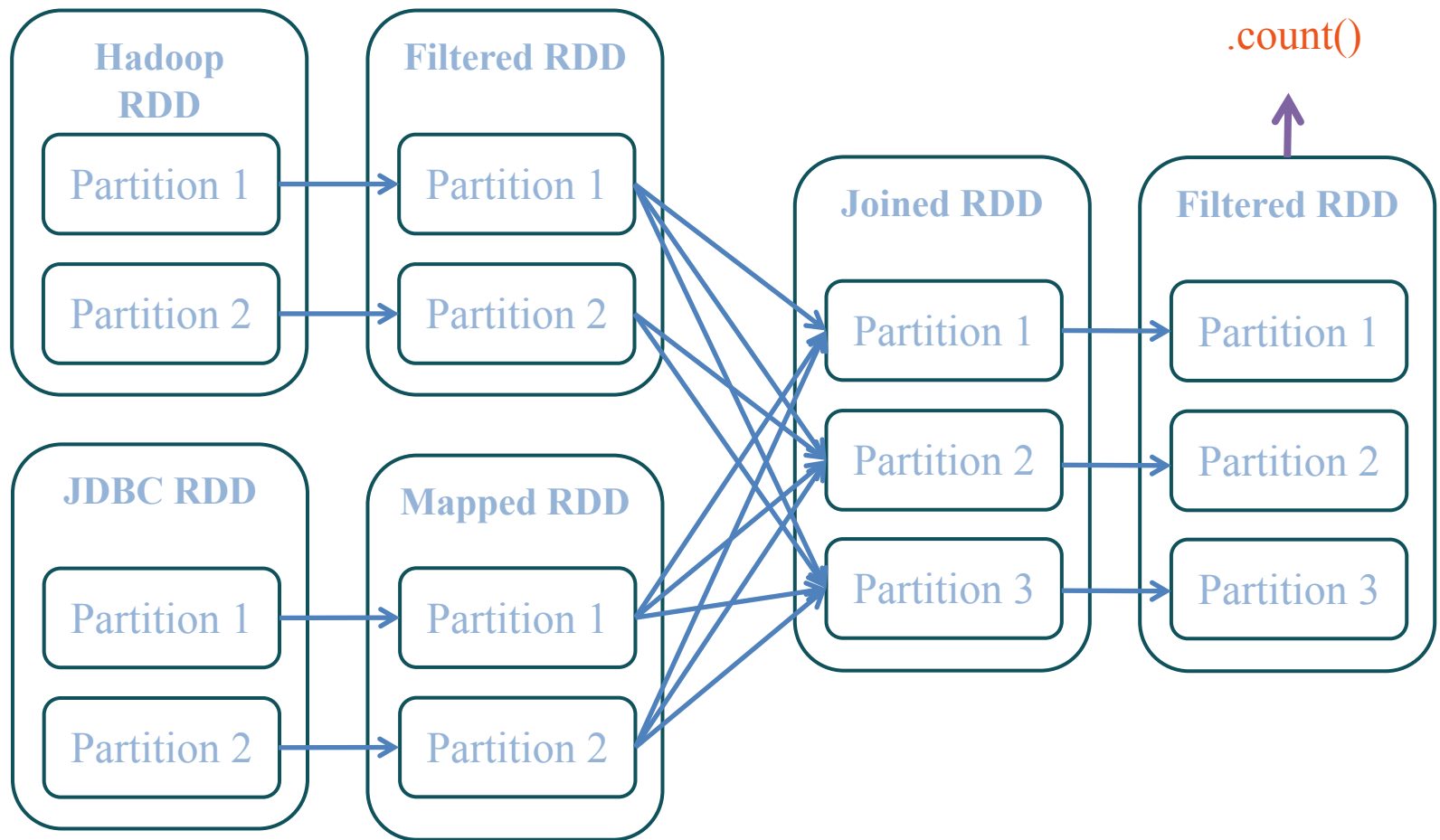
Dependencies = a single parent

compute(partition) = call parent.compute(partition) and filter

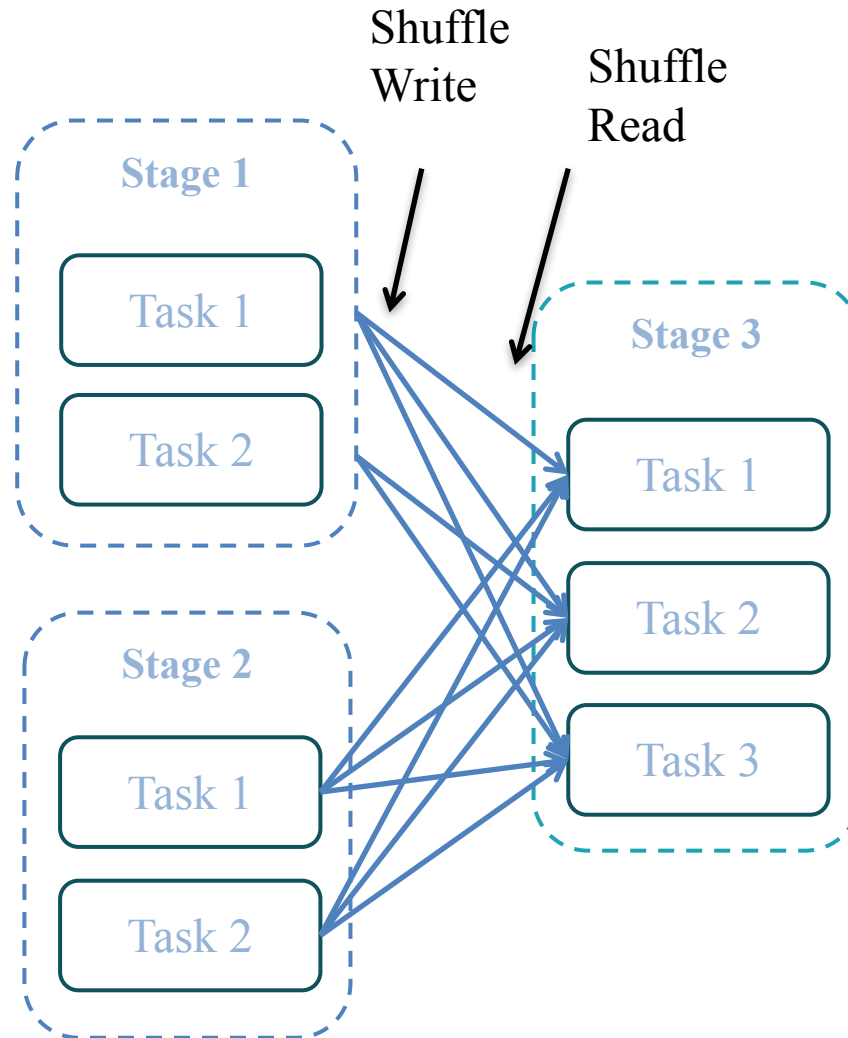
Partitioner = parent partitioner

> `filtered = rdd.filter(lambda x: x contains “ERROR”)`

A More Complex DAG

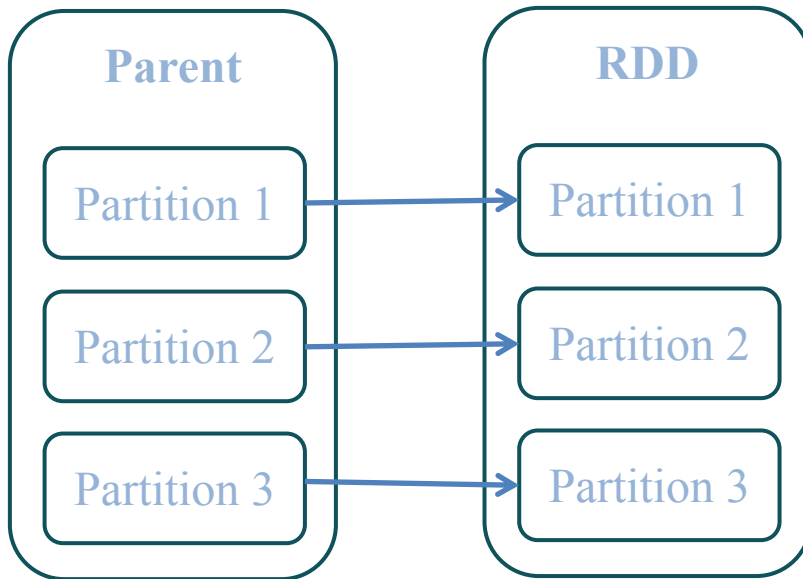


A More Complex DAG

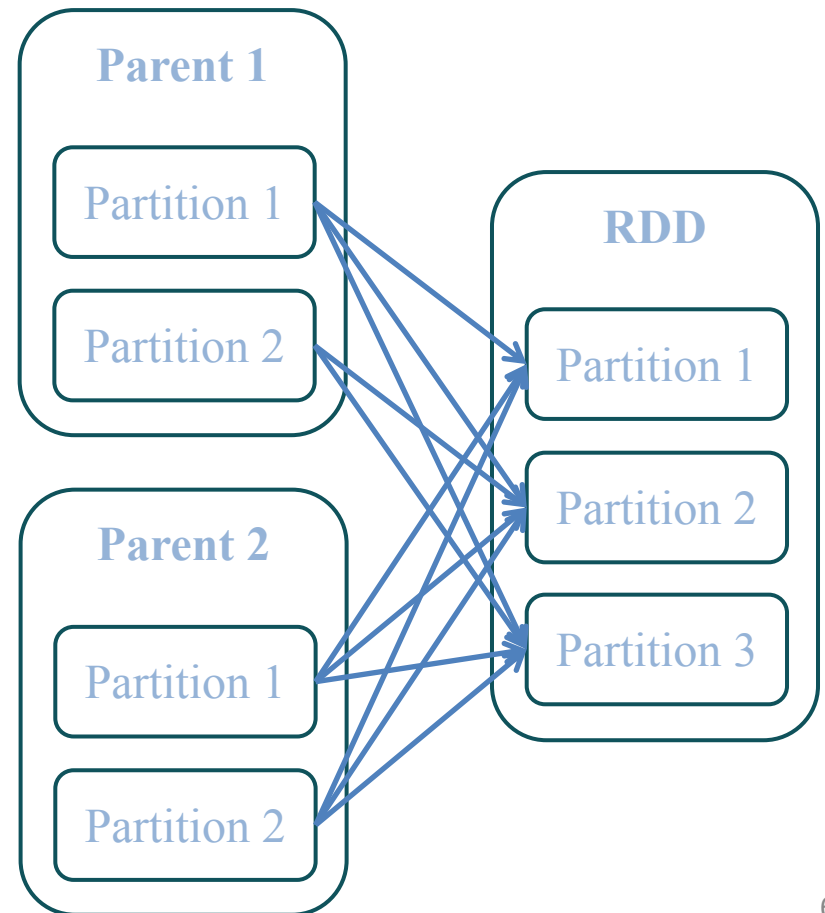


Narrow and Wide Transformations

FilteredRDD

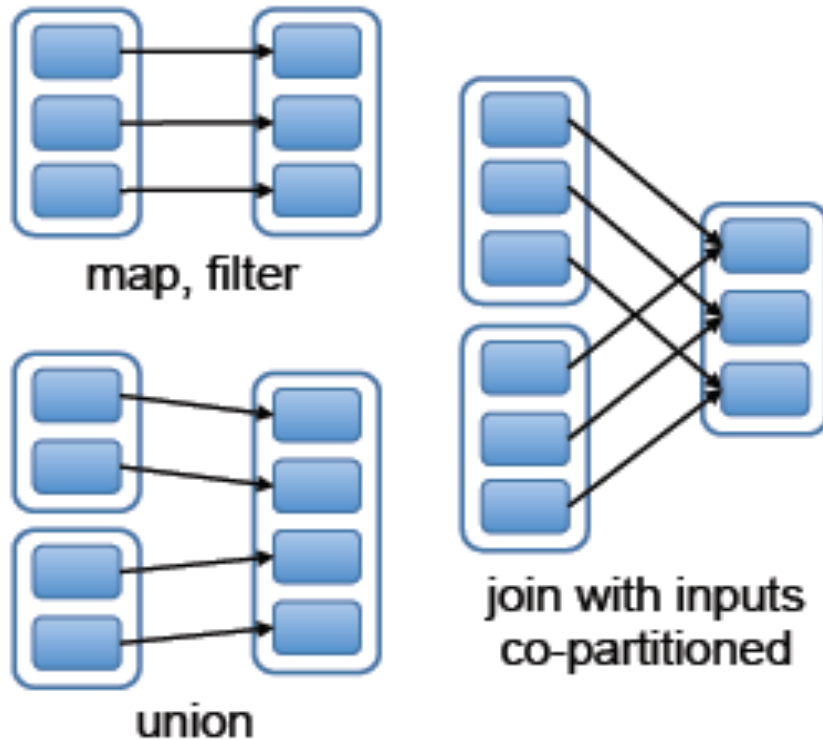


JoinedRDD

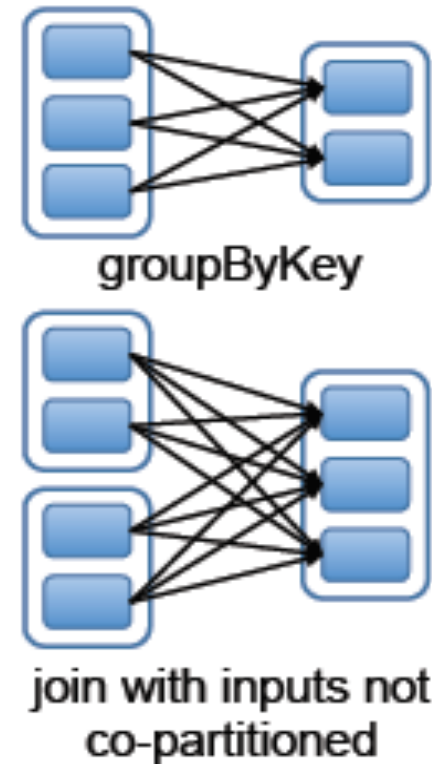


RDD Dependencies

Narrow Dependencies:



Wide Dependencies:



Each box is an RDD, with partitions shown as shaded rectangles

Outline

- Introduction to Scala & functional programming
- What is Spark
- Resilient Distributed Datasets (RDDs)
- Implementation
- Demo
- Conclusion

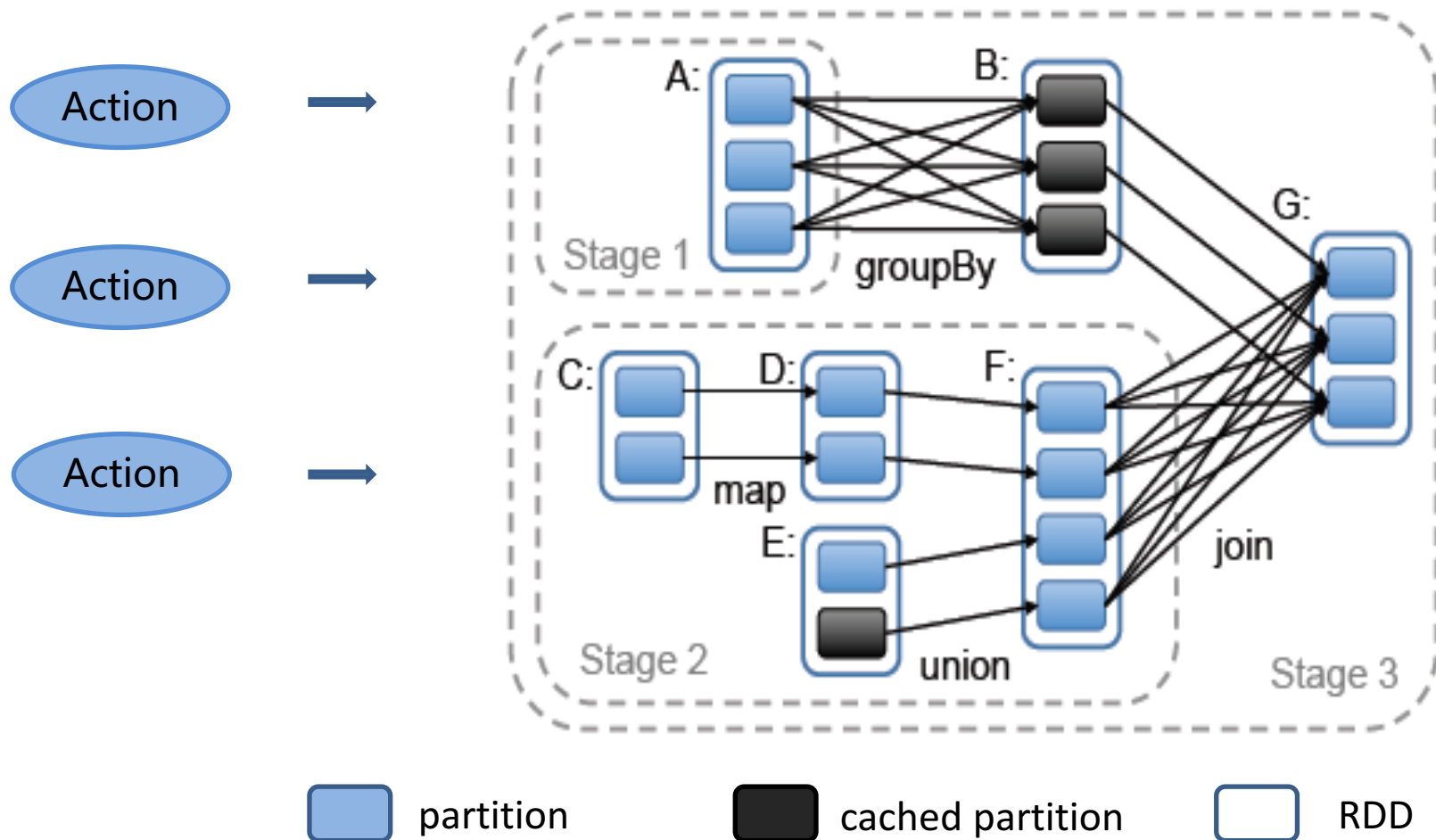
Implementation

Implement Spark in about 14,000 lines of Scala
Sketch three of the technically parts of the system:

- >> Job Scheduler
- >> Fault Tolerance
- >> Memory Management

Job Scheduler

Build a DAG according to RDD's lineage graph



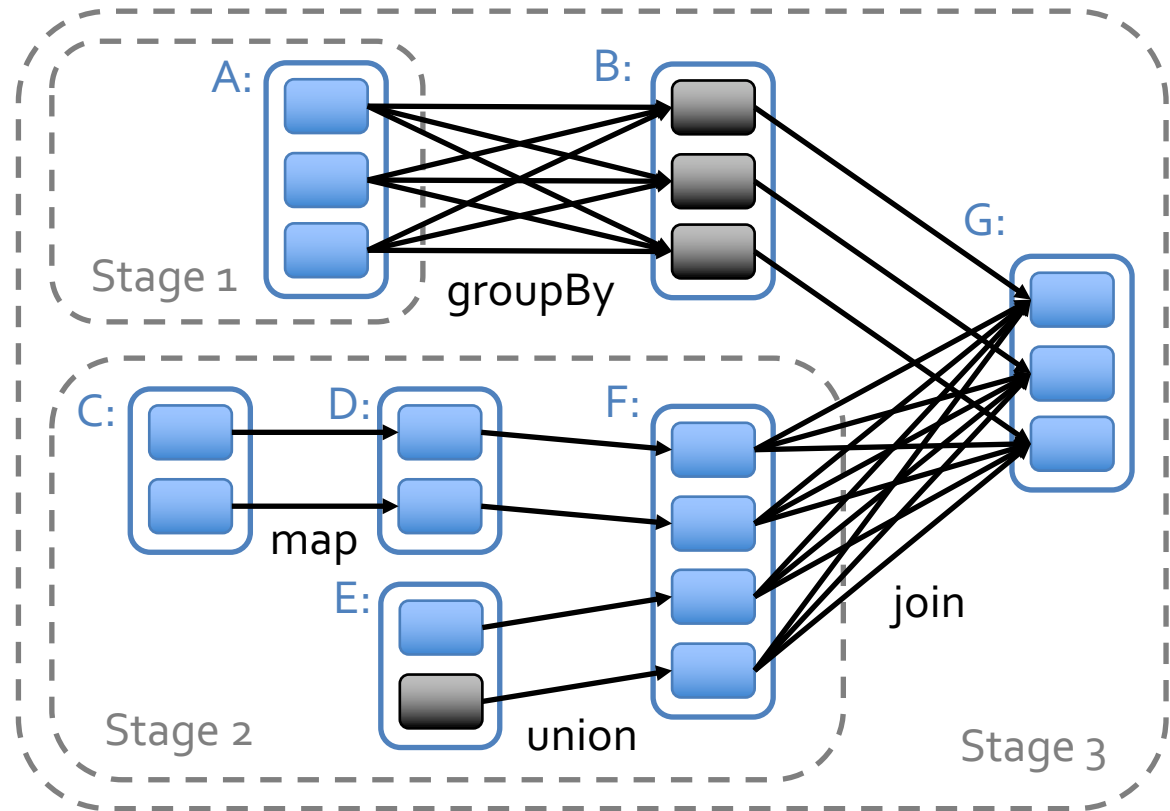
Task Scheduler

Dryad-like DAGs

Pipelines functions
within a stage

Locality & data
reuse aware

Partitioning-aware
to avoid shuffles



Advanced Features

- Controllable partitioning
 - Speed up joins against a dataset
- Controllable storage formats
 - Keep data serialized for efficiency, replicate to multiple nodes, cache on disk
- Shared variables: broadcasts, accumulators
- See online docs for details!

Spark Architecture

Architecture

Architecture

SPARK Technology Stack

SPARK SQL

SPARK
Streaming
(Streaming)

MLib
(Machine
Learning)

GraphX
(Graph
Computation)

Spark R
(R on
Spark)

SPARK Core Engine

Standalone
Scheduler

YARN

MESOS

Spark Architecture

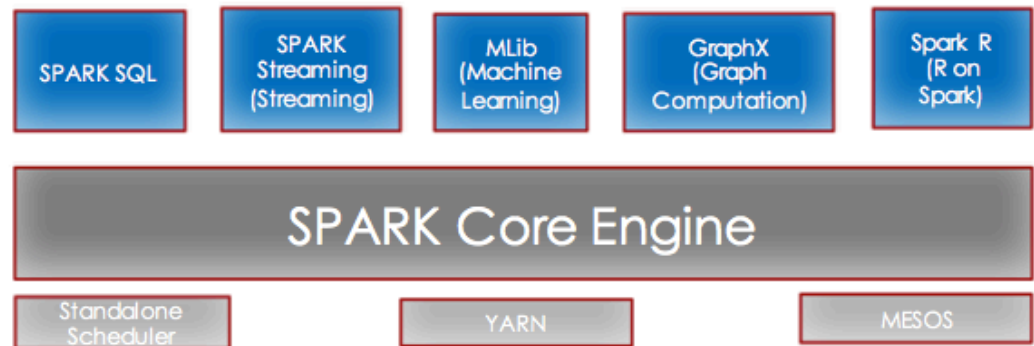
Architecture

Architecture

SPARK Technology Stack

SPARK Core Engine

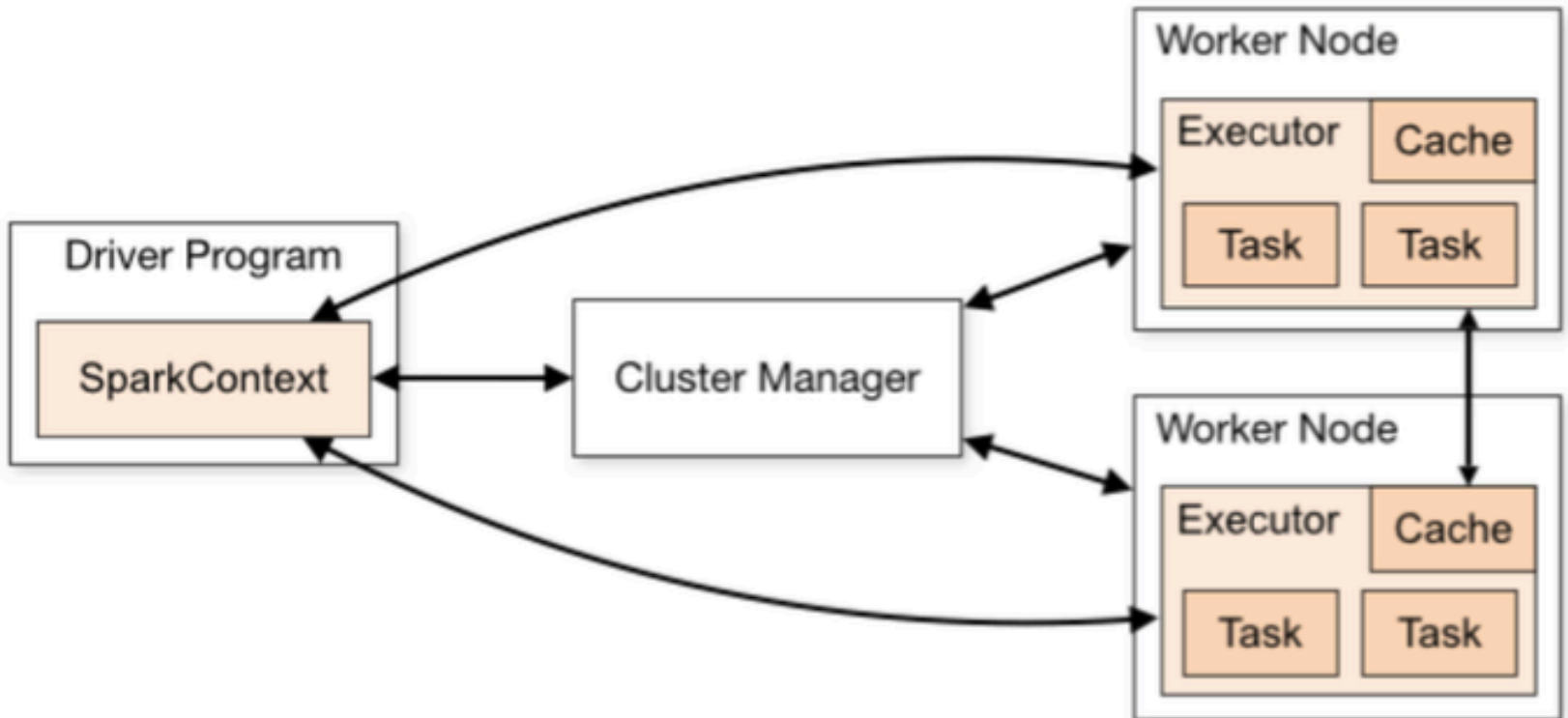
- Basic functionality of Spark
- Uses RDDs (Resilient Distributed Datasets)
- Contains APIs for manipulating RDDs



Spark RDDs are a collection of items distributed across compute nodes. Spark core APIs allow manipulation of these RDDs in parallel.

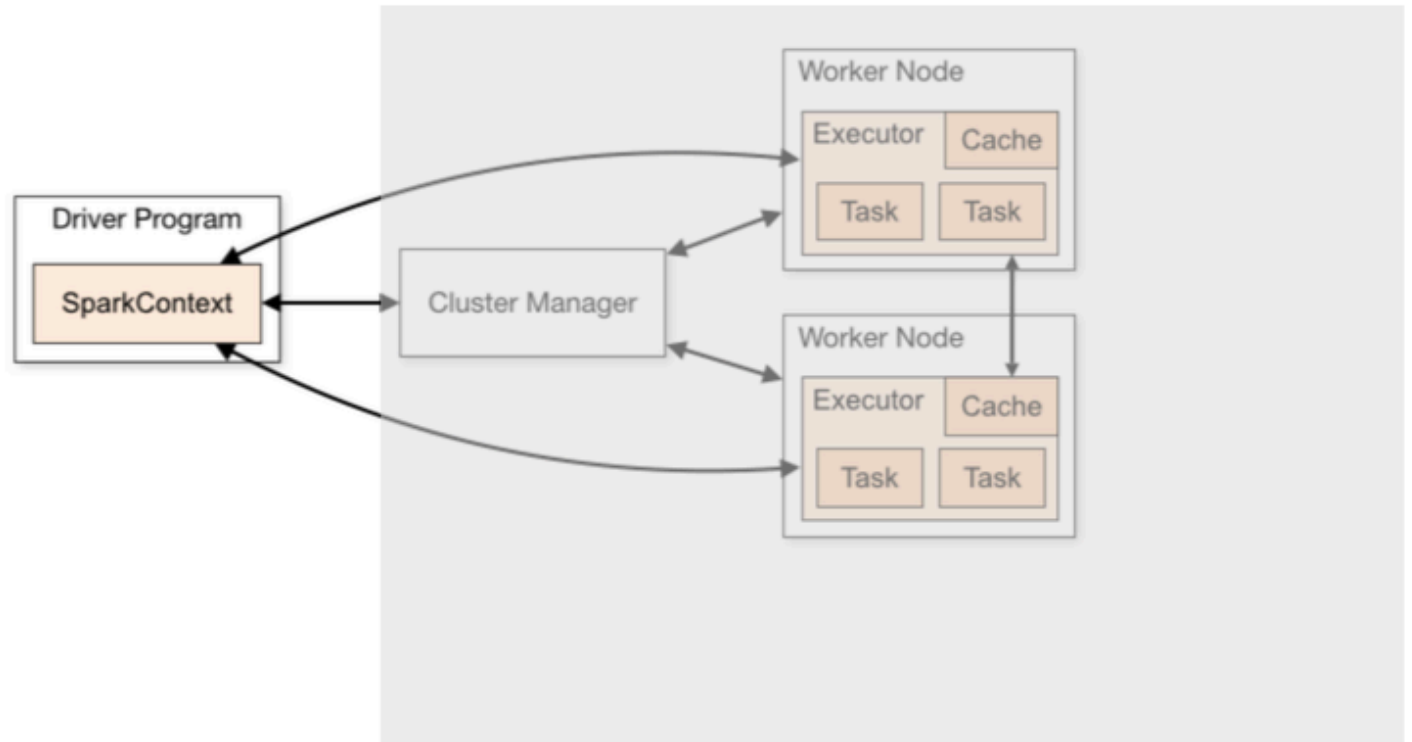
Spark Processing

SPARK Processing



Spark Processing

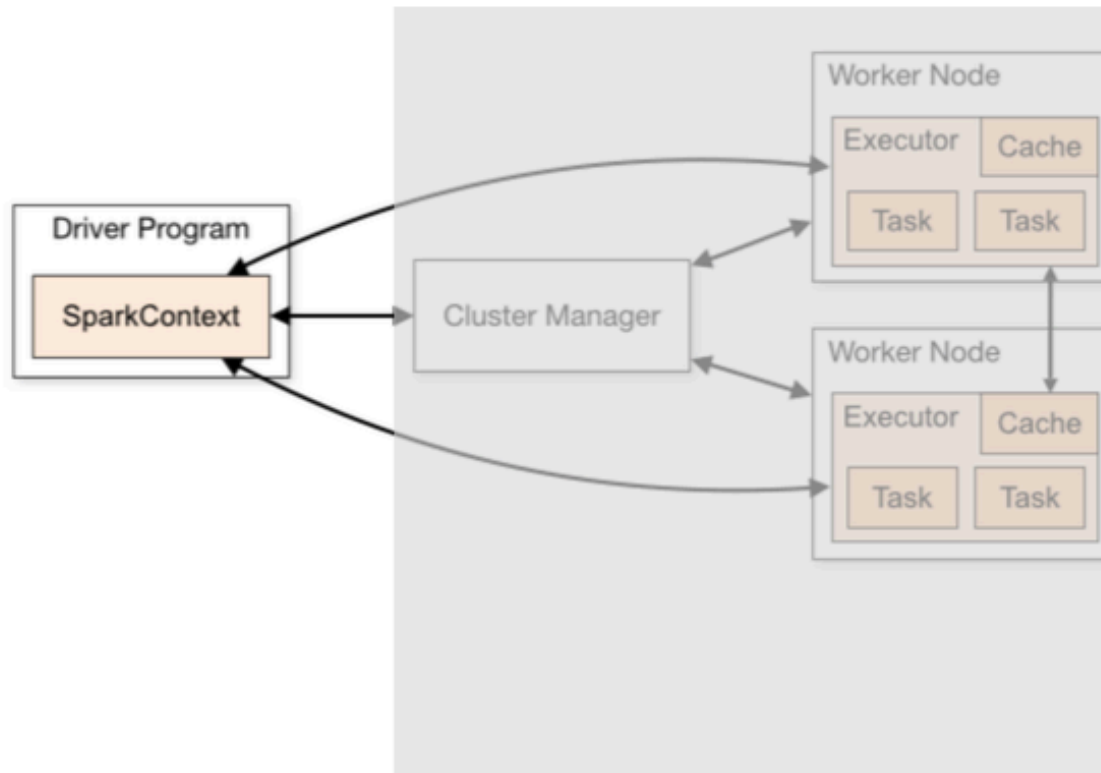
Driver program accesses Spark through a SparkContext object.



Source: <https://spark.apache.org/docs/latest/cluster-overview.html>

Spark Processing

Spark Context represents a connection to a computing cluster
Once created, it can be used to build RDDs

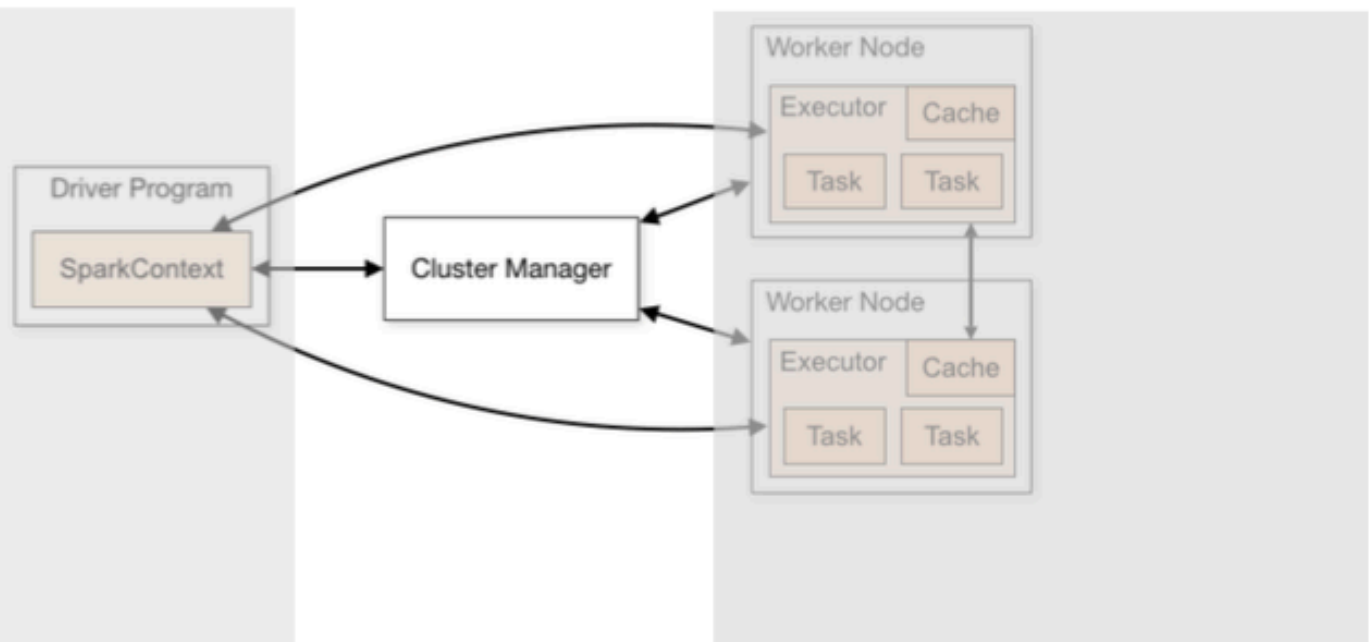


Source: <https://spark.apache.org/docs/latest/cluster-overview.html>

Spark Processing

Cluster Manager is an external service

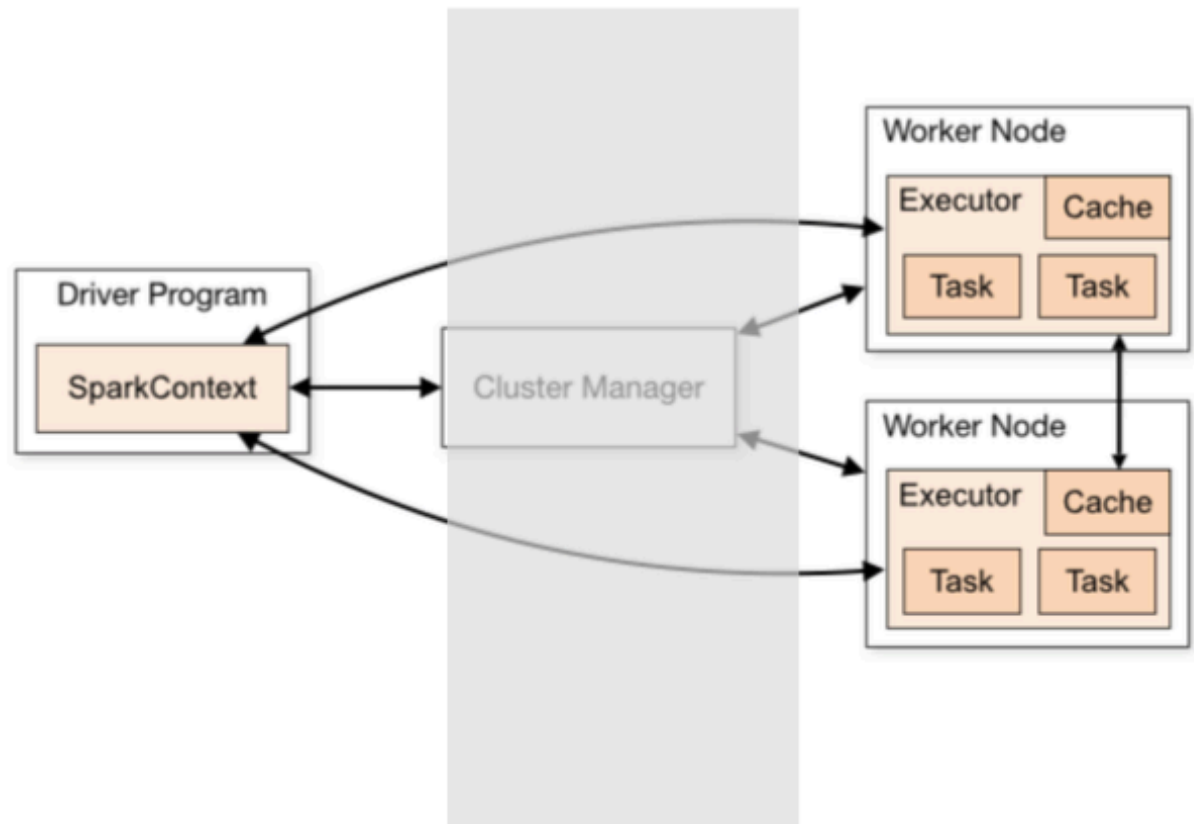
- A default built-in cluster manager called Standalone Cluster manager is pre-packaged with Spark
- Hadoop YARN and Apache Mesos are two popular cluster managers
- Driver requests cluster manager to provide resources for launching executors
- Cluster manager launches executors which are then used by driver to run tasks



Spark Processing

Executors are processes that execute tasks

- Executors run the tasks and return results to the driver
- Also provide in-memory storage for RDDs



WORKING WITH SPARK

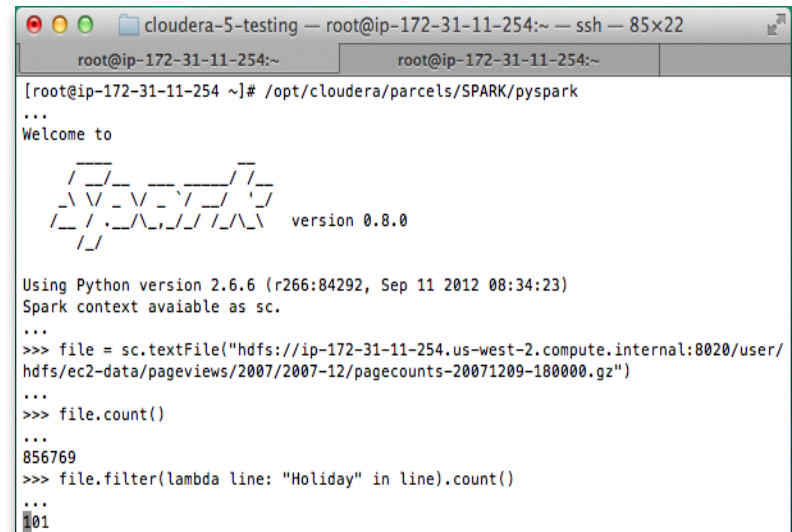
Using the Shell

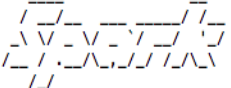
Launching:

```
spark-shell  
pyspark (IPYTHON=1)
```

Modes:

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



```
root@ip-172-31-11-254:~  
[root@ip-172-31-11-254 ~]# /opt/cloudera/parcels/SPARK/pyspark  
...  
Welcome to  
 version 0.8.0  
Using Python version 2.6.6 (r266:84292, Sep 11 2012 08:34:23)  
Spark context available as sc.  
...  
>>> file = sc.textFile("hdfs://ip-172-31-11-254.us-west-2.compute.internal:8020/user/  
hdfs/ec2-data/pageviews/2007/2007-12/pagecounts-20071209-180000.gz")  
...  
>>> file.count()  
...  
856769  
>>> file.filter(lambda line: "Holiday" in line).count()  
...  
101
```

SparkContext

- Main entry point to Spark functionality
- Available in shell as variable **SC**
- In standalone programs, you'd make your own (see later for details)

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)`

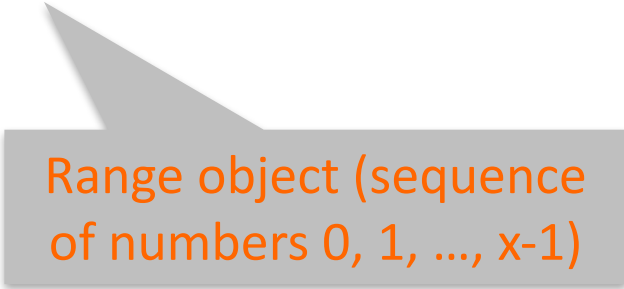
Basic Transformations

```
> val nums = sc.parallelize(Array(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(x))
  > # => {0, 0, 1, 0, 1, 2}
```



Range object (sequence
of numbers 0, 1, ..., x-1)

Basic Transformations/Actions: Scala

- `scala>val number = List(1, 2, 3)`
- `number: List[Int] = List(1, 2, 3)`
- `scala> squares = number.map(x=>x*x)`
- `scala> val squares = number.map(x=>x*x)`
- `squares: List[Int] = List(1, 4, 9)`
- `scala> val even = squares.filter(x=>x%2==0)`
- `even: List[Int] = List(4)`
- `scala>val range = even.flatMap(x => Range (1,x,1))`
- `range: List[Int] = List(1, 2, 3)`
- `scala> val range = even.flatMap(x => Range (1,x,2))`
- `range: List[Int] = List(1, 3)`
- `scala> val rangeRDD = sc.parallelize(range)`
- `scala> val counter = rangeRDD.collect()`
- `Array[Int] = Array(1, 3)`

<https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ch04.html>

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")
```

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
```

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)}
```

reduceByKey also automatically implements
combiners on the map side

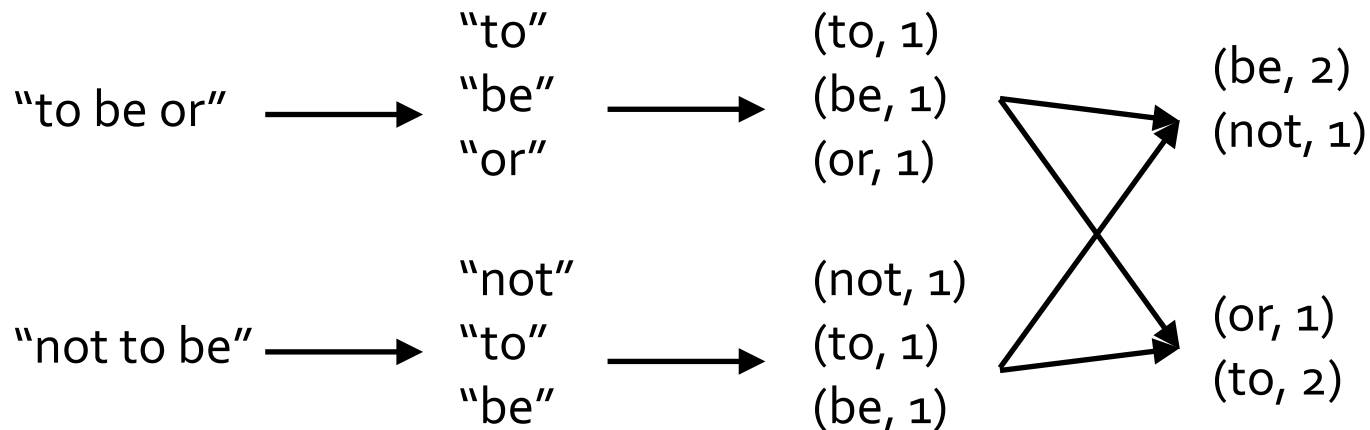
Example: Word Count

Python

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

Scala

```
> lines = sc.textFile("hamlet.txt")
> counts = lines.flatMap(line => line.split(" "))
.map(word => (word, 1))
> .reduceByKey(_+_)
```



Other Key-Value Operations

- > `visits = sc.parallelize([("index.html", "1.2.3.4"),
("about.html", "3.4.5.6"),
("index.html", "1.3.3.1")])`
- > `pageNames = sc.parallelize([("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", (["1.2.3.4", "1.3.3.1"], ["Home"]))
("about.html", (["3.4.5.6"], ["About"]))

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)
```

Using Local Variables

Any external variables you use in a closure will automatically be shipped to the cluster:

```
> query = sys.stdin.readline()  
> pages.filter(lambda x: query in  
x).count()
```

Some caveats:

- Each task gets a new copy (updates aren't sent back)
- Variable must be Serializable / Pickle-able
- Don't use fields of an outer object (ships all of it!)

Closure Mishap Example

This is a problem:

```
class MyCoolRddApp {  
  val param = 3.14  
  val log = new Log(...)  
  ...  
  
  def work(rdd: RDD[Int]) {  
    rdd.map(x => x + param)  
      .reduce(...)  
  }  
}
```

NotSerializableException:
MyCoolRddApp (or Log)

How to get around it:

```
class MyCoolRddApp {  
  ...  
  ...  
  
  def work(rdd: RDD[Int]) {  
    val param_ = param  
    rdd.map(x => x +  
    param_)  
      .reduce(...)  
  }  
}
```

References only local variable
instead of this.param

More RDD Operators

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- Cogroup
- sample
- take
- first
- partitionBy
- save ...

EXAMPLE APPLICATION: PAGERANK

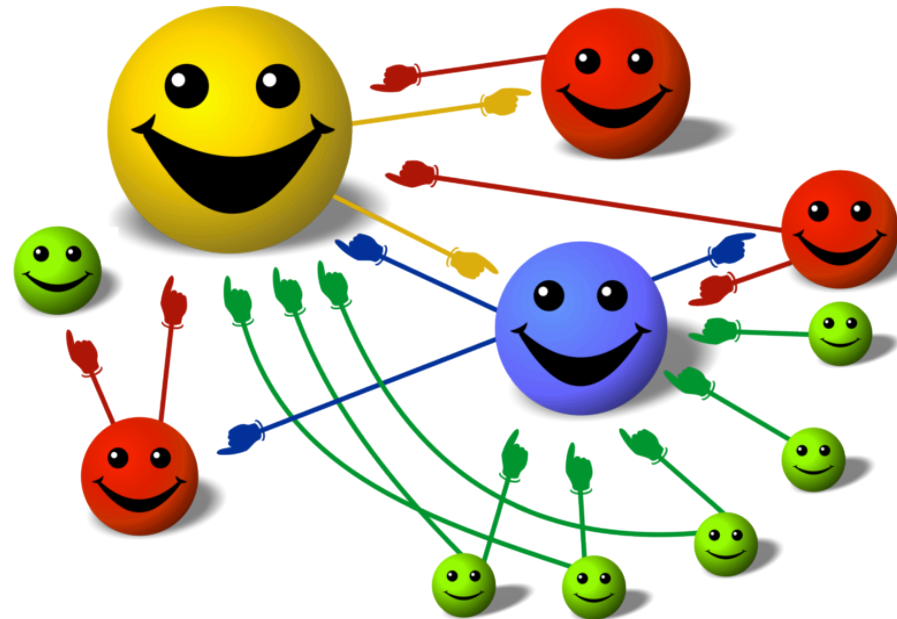
Example: PageRank

- Good example of a more complex algorithm
 - Multiple stages of map & reduce
- Benefits from Spark's in-memory caching
 - Multiple iterations over the same data

Basic Idea

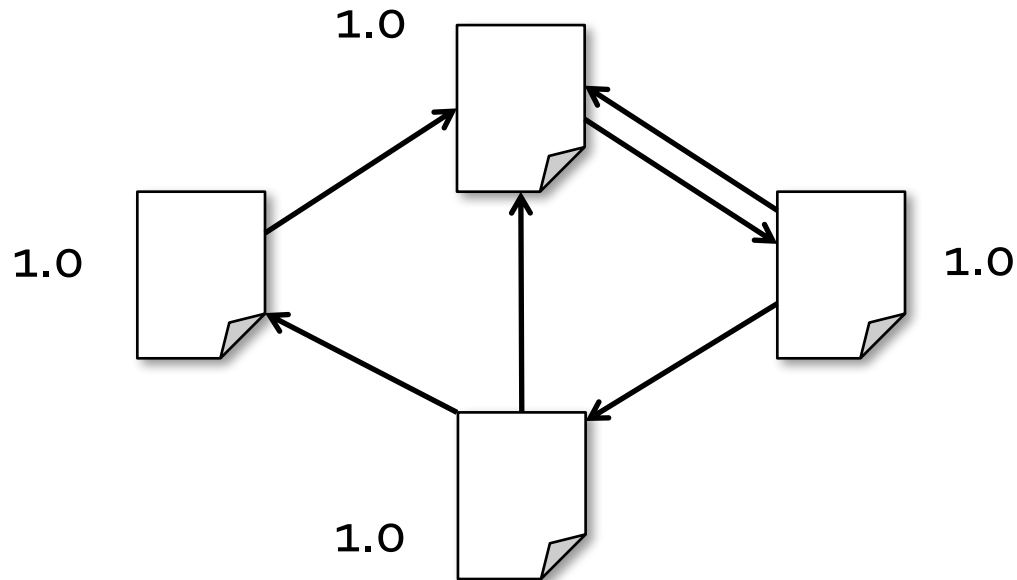
Give pages ranks (scores) based on links to them

- Links from many pages → high rank
- Link from a high-rank page → high rank



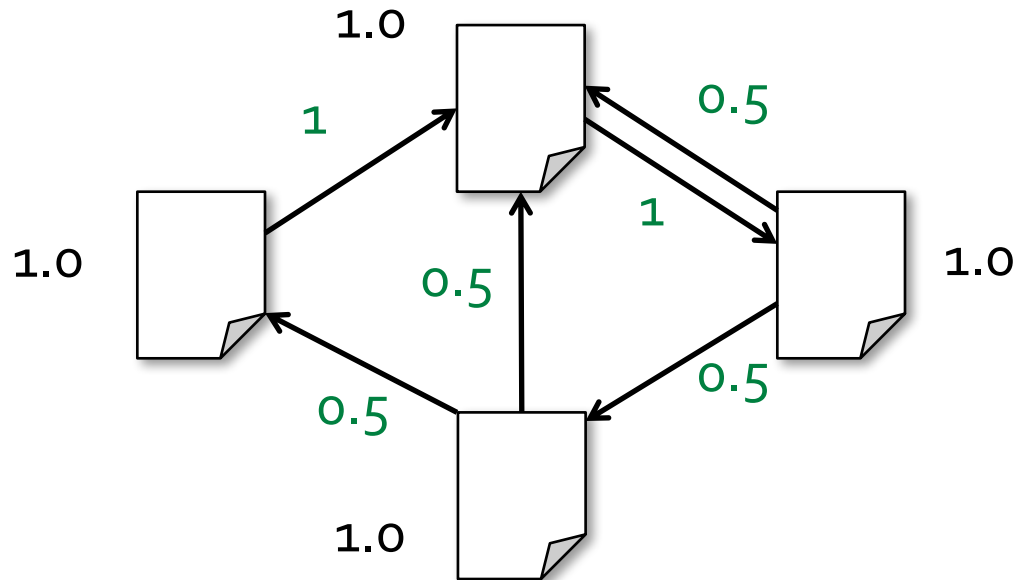
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



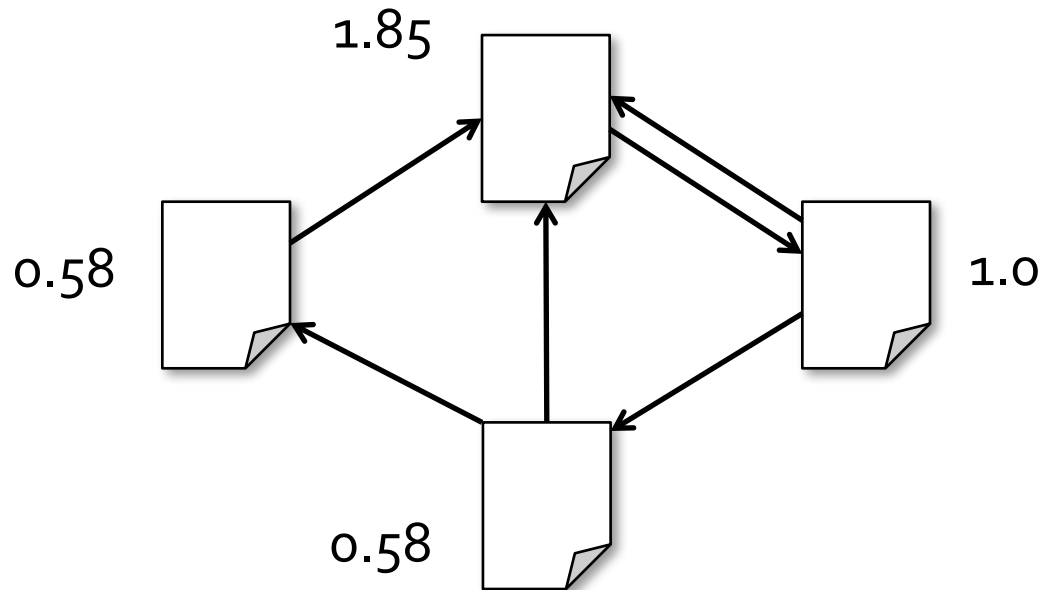
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



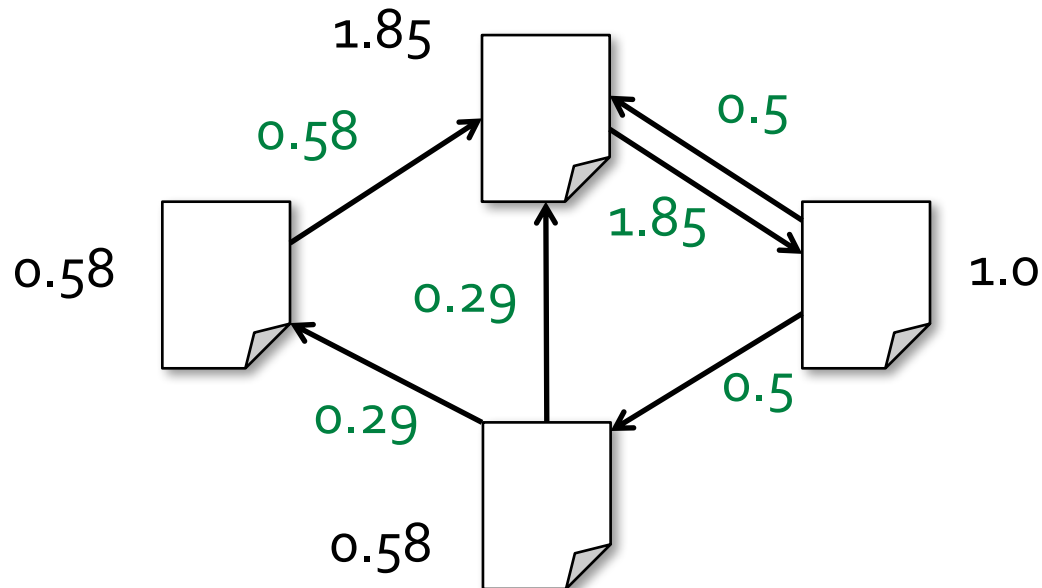
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



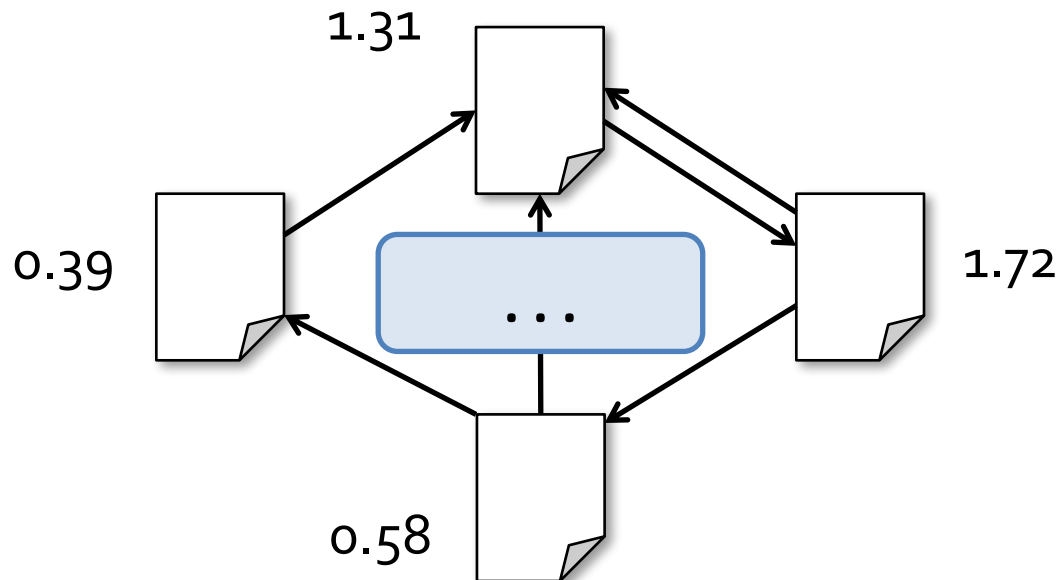
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

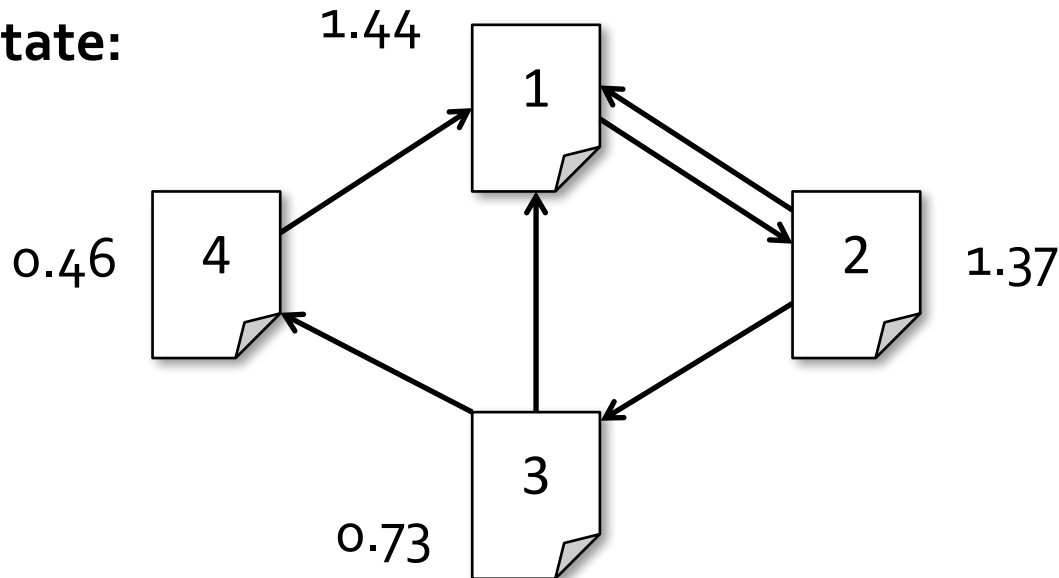
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

Final state:



Scala Implementation

```
package org.apache.spark.examples
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}

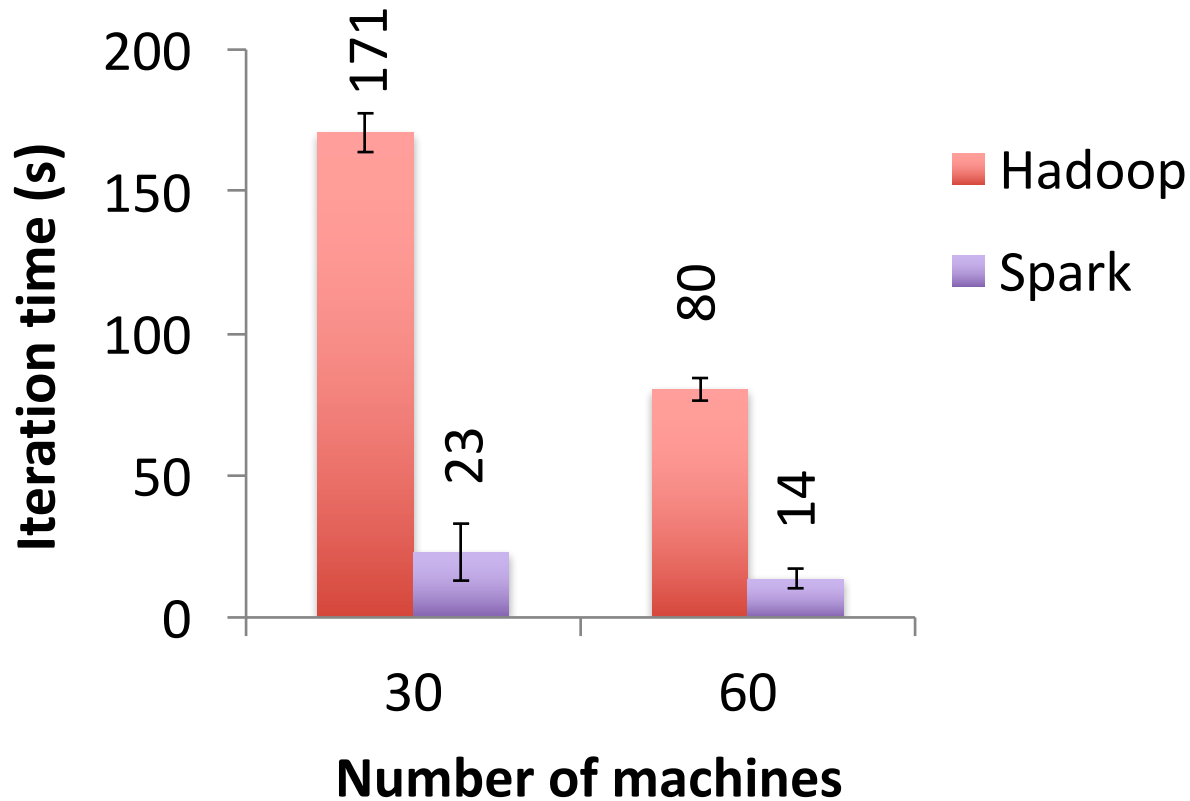
/**
 * Computes the PageRank of URLs from an input file. Input file should
 * be in format of:
 * URL      neighbor URL
 * URL      neighbor URL
 * URL      neighbor URL
 * ...
 * where URL and their neighbors are separated by space(s).
 */
object SparkPageRank {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("PageRank")
    var iters = args(1).toInt
    val ctx = new SparkContext(sparkConf)
    val lines = ctx.textFile(args(0))
    val links = lines.map{ s => val parts = s.split("\\s+")
      (parts(0), parts(1))
    }.distinct().groupByKey().cache()
    var ranks = links.mapValues(v => 1.0)
```

```
    for (i <- 1 to 15) {
      val contribs = links.join(ranks).values.flatMap{ case (urls,
        rank) =>
        val size = urls.size
        urls.map(url => (url, rank / size))
      }
      ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85
        * _)
    }

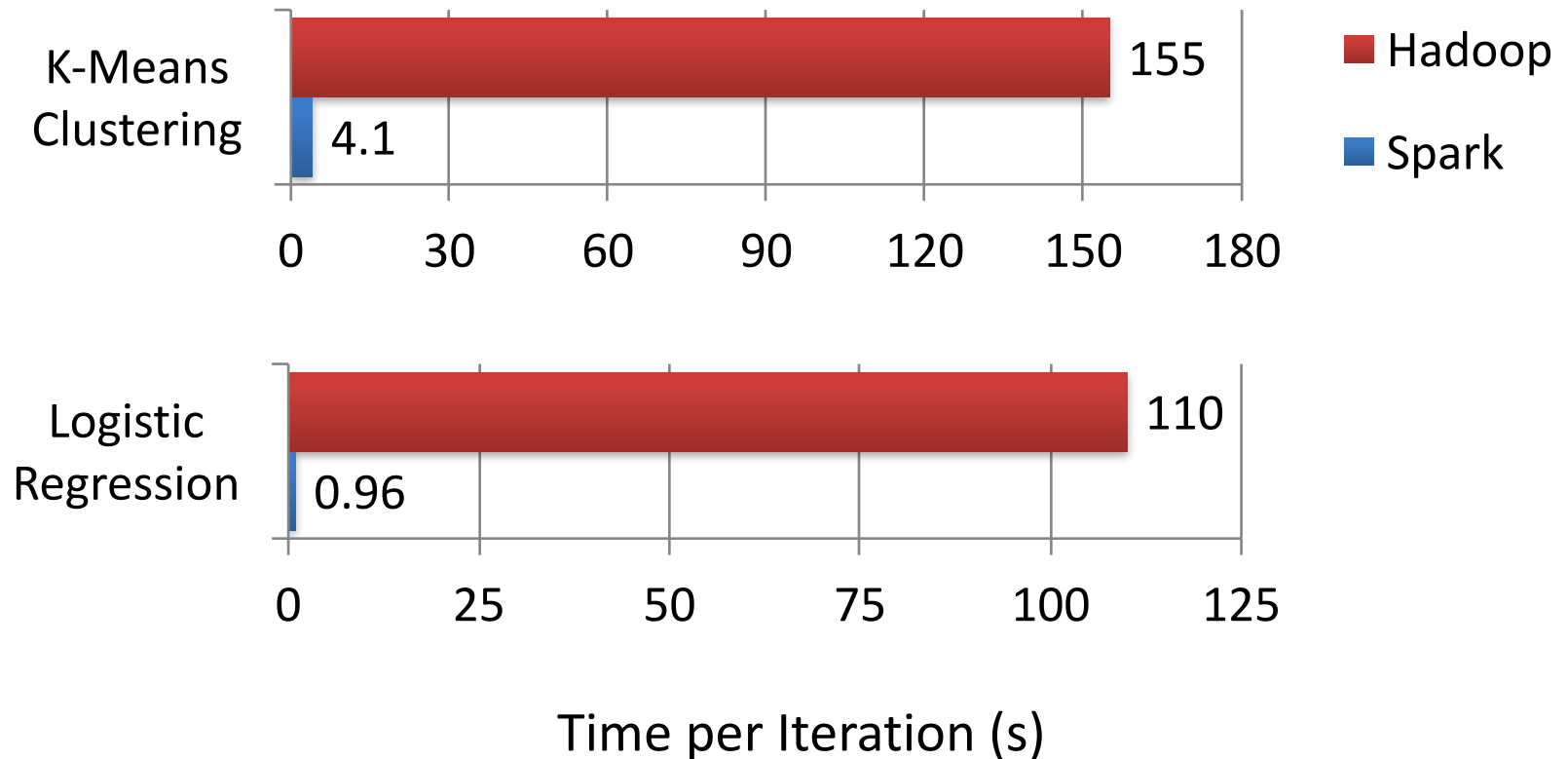
    val output = ranks.collect()
    output.foreach(tup => println(tup._1 + " has rank: " + tup._2 +
      "."))

    ctx.stop()
  }
}
```

PageRank Performance



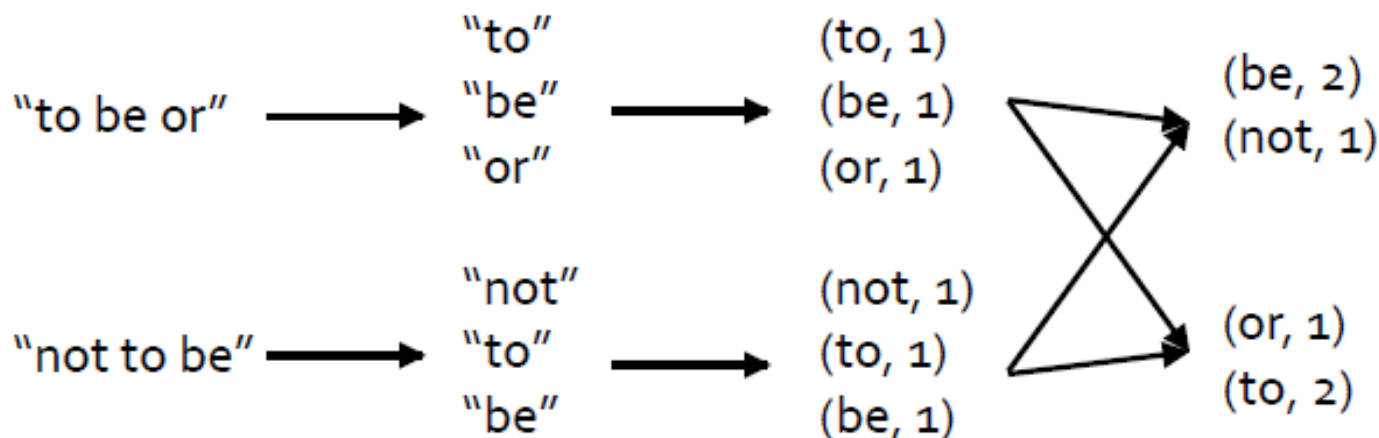
Other Iterative Algorithms



Example: Word Count

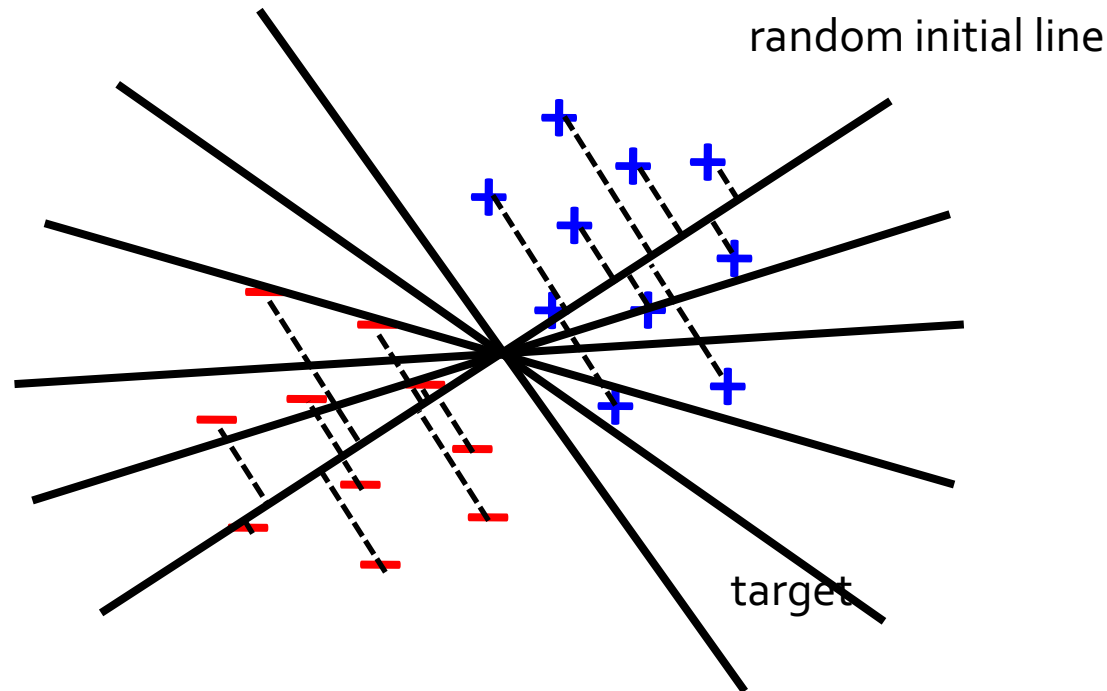
```
val lines = sc.textFile("hamlet.txt")
```

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



Example: Logistic Regression

- Goal: find best line separating two sets of points



Logistic Regression Code

- `val data =
 spark.textFile(...).map(readPoint).cache()`
- `var w = Vector.random(D)`
- `for (i <- 1 to ITERATIONS) {`
- `val gradient = data.map(p =>`
- `(1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y *`
- `p.x`
- `).reduce(_ + _)`
- `w -= gradient`
- `}`
- `println("Final w: " + w)`