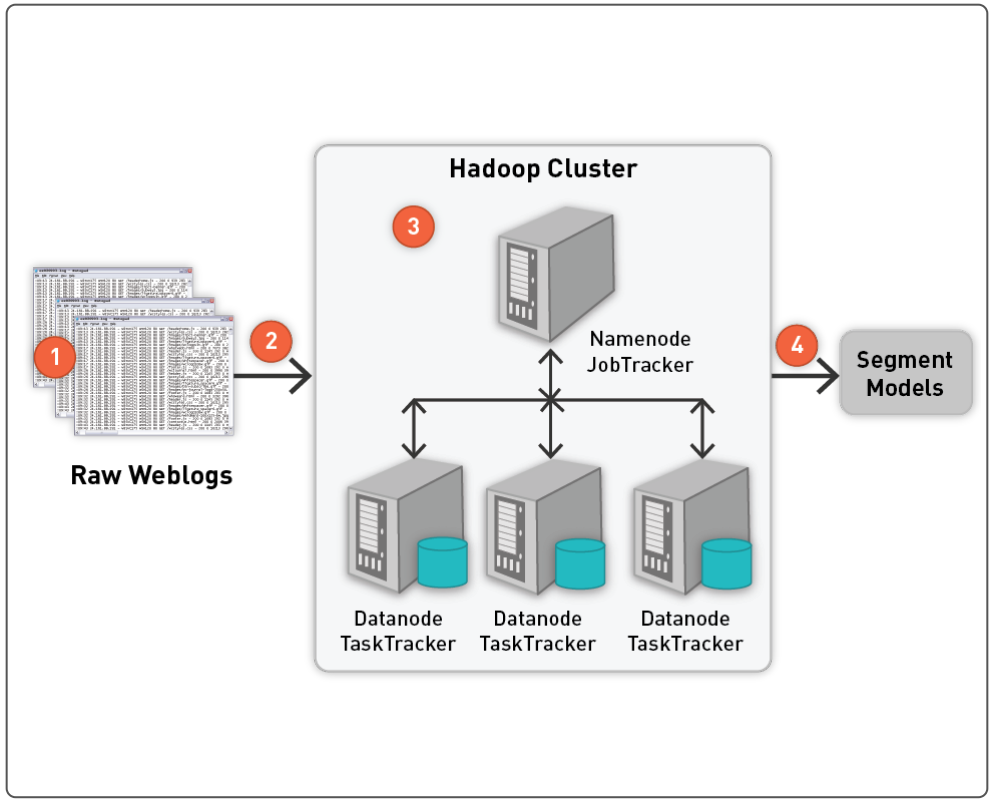
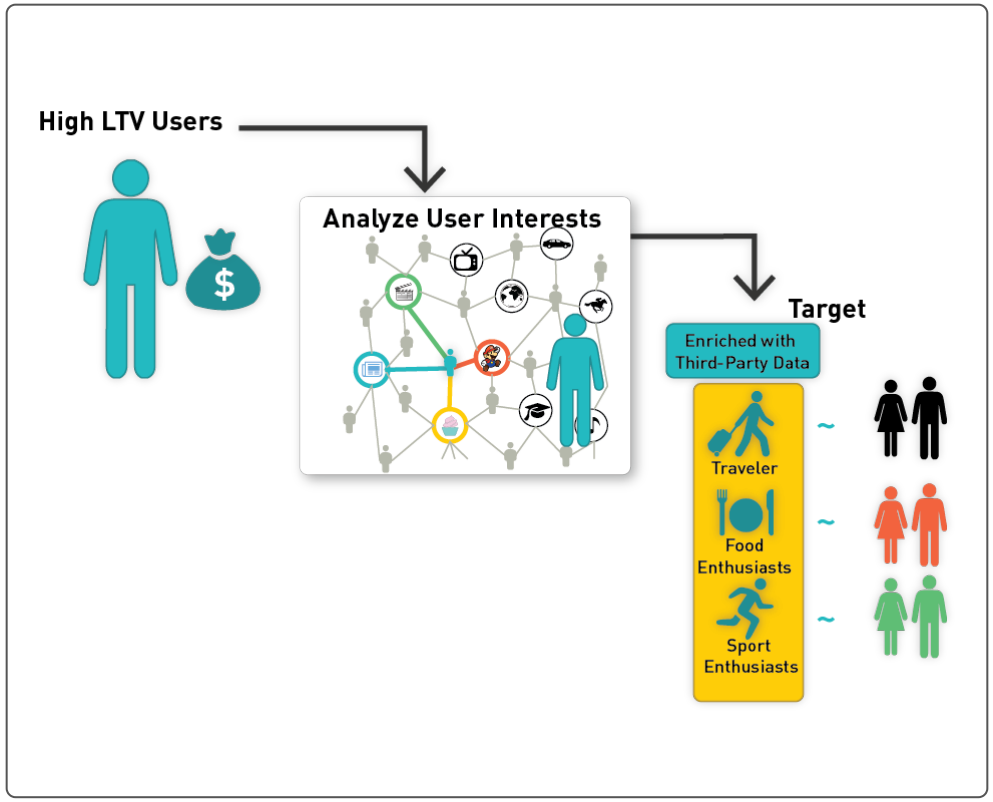
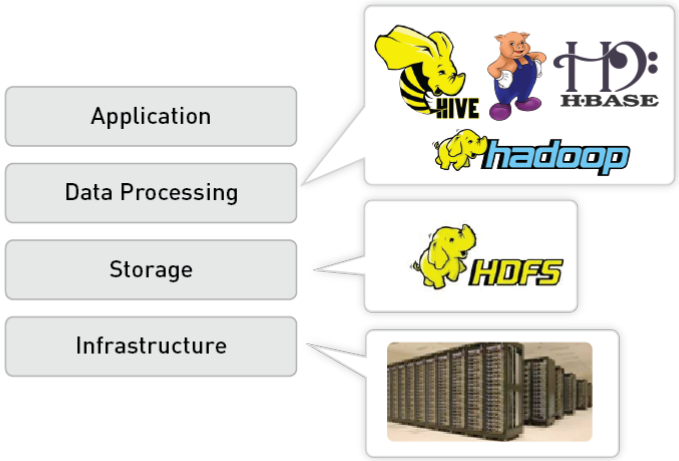

Hadoop Cluster
1
2
3
4
Raw Weblogs
Namenode
JobTracker
Segment
Models
Datanode
TaskTracker
Datanode
TaskTracker
Datanode
TaskTracker


High LTV Users
$
Analyze User Interests
Target
Enriched with Third-Party Data
Traveler
Food Enthusiasts
Sport Enthusiasts

Data Processing Goals

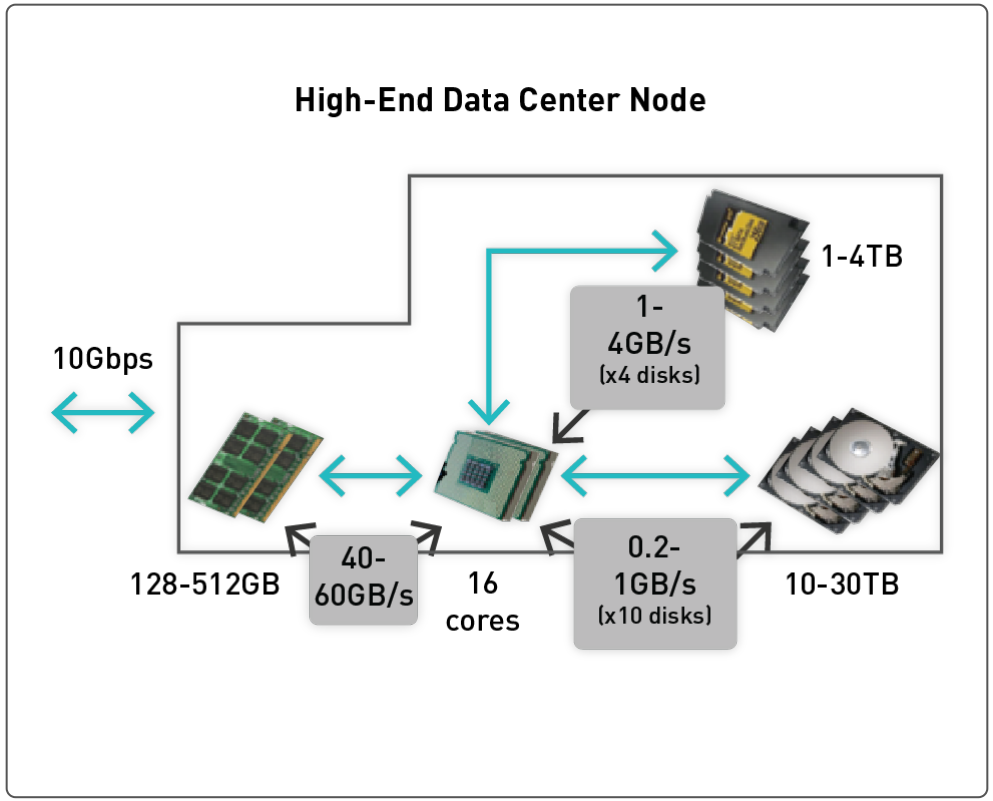
- Low latency (interactive) queries on historical data
  - Enable faster decision-making
  - E.g., identify why a site is slow and fix it
- Low latency queries on live data (streaming)
  - Enable decisions on real-time data
- Sophisticated data processing
  - Enable better decision-making
  - E.g., machine learning

Today's Open Analytics Stack



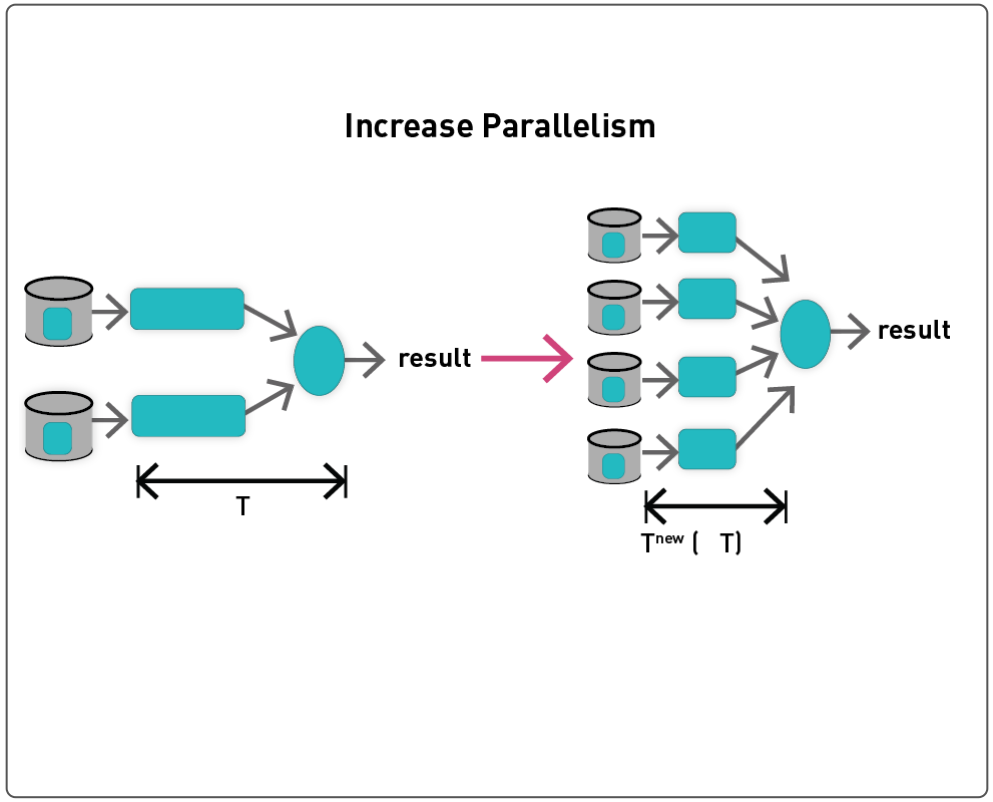
Goals

- Easy to combine batch, streaming, and interactive computations
- Easy to develop sophisticated algorithms
- Compatible with existing open-source ecosystem (Hadoop/HDFS)


High-End Data Center Node
1-4TB
1-
4GB/s
(x4 disks)
10Gbps
40-
60GB/s
128-512GB
16
cores
0.2-
1GB/s
(x10 disks)
10-30TB

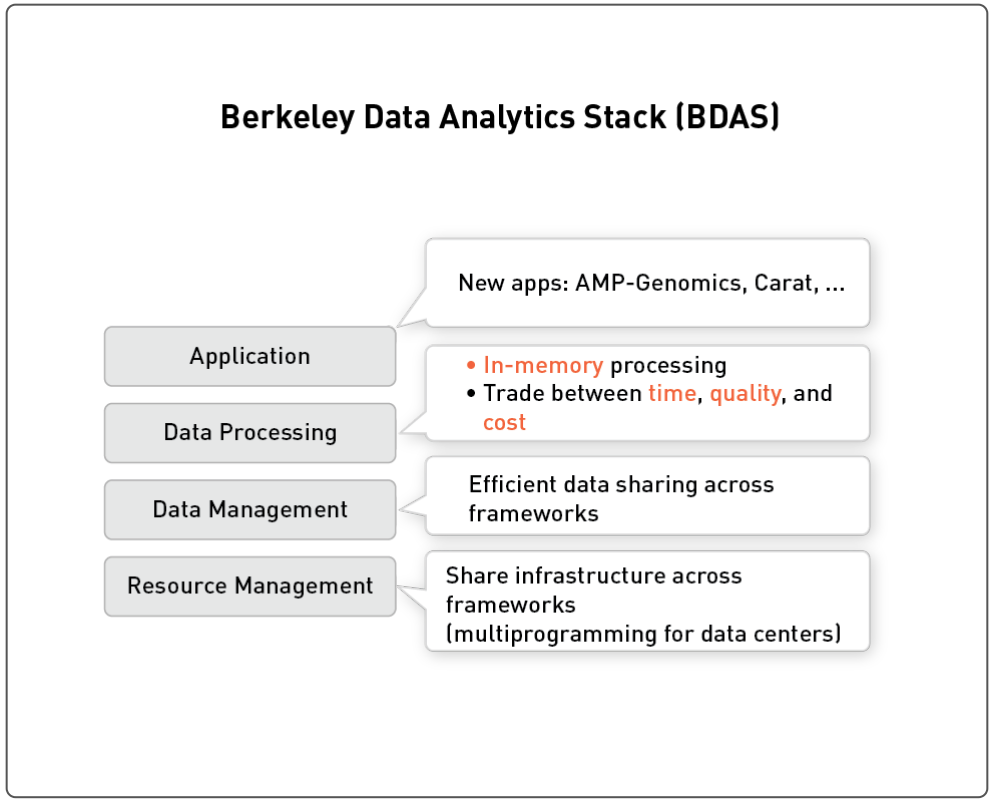
Improving Interactivity and Streaming: Memory

- Memory transfer rates are 100x faster than disk or SSDs.
- Low cost makes increasing memory a viable option.
    - E.g., 1 TB = 1 billion records at 1 KB each
    - Inputs of over 90% of the jobs in Facebook, Yahoo!, and Bing clusters fit into memory.


Increase Parallelism
result
T
result
Tnew (T)

Improving Interactivity and Streaming: Increase Parallelism

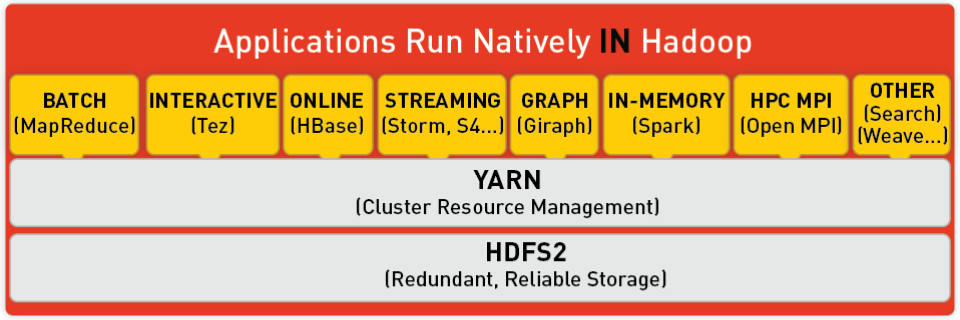
- Spark provides:
  - Low latency
  - Optimized parallel communication patterns
  - Improved shuffle (TimSort, broadcast)
  - Efficient recovery from failures and straggler mitigation


Berkeley Data Analytics Stack (BDAS)
Application
New apps: AMP-Genomics, Carat, ...
Data Processing
• In-memory processing
• Trade between time, quality, and cost
Data Management
Efficient data sharing across frameworks
Resource Management
Share infrastructure across frameworks (multiprogramming for data centers)

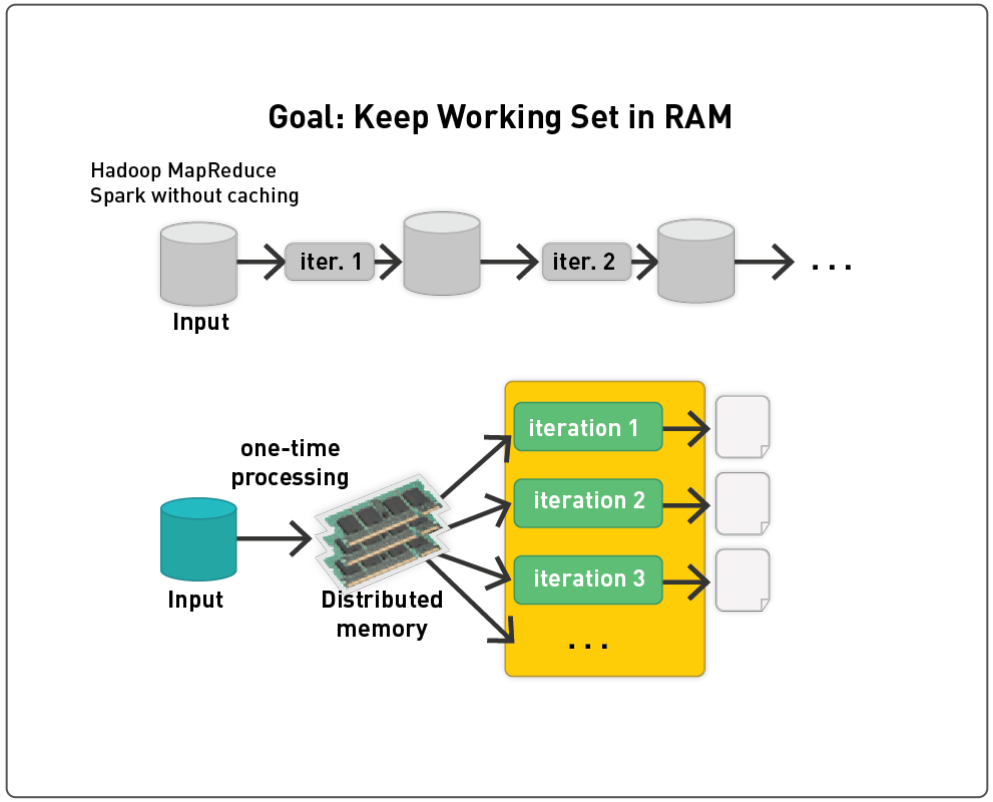
Datacenter Resource Management

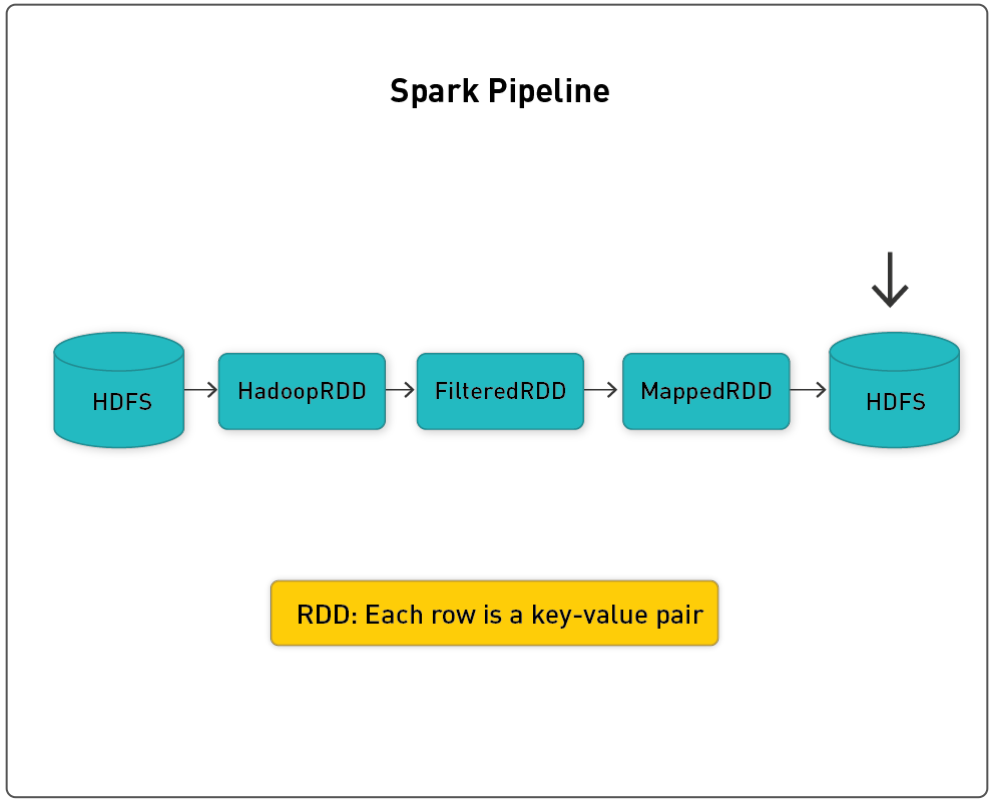
- YARN (successor to MapReduce)
- Mesos has proven performance at Twitter, Airbnb, and Netflix.

Hadoop 2 and YARN



YARN supports multiple processing models in addition to MapReduce.


Goal: Keep Working Set in RAM
Hadoop MapReduce
Spark without caching
iter. 1
iter. 2
. . .
Input
one-time
processing
Input
Distributed
memory
iteration 1
iteration 2
iteration 3
. . .


Spark Pipeline
HDFS
HadoopRDD
FilteredRDD
MappedRDD
HDFS
RDD: Each row is a key-value pair

Disadvantages of Hadoop

- Hadoop is a black box, lacks read-eval-print loop (REPL).
  - Cannot execute REPL
- Complex jobs are not possible.

Spark: Advantages

- High-level data management
  - Split data
  - Ship data
  - Replicate data
  - Run tasks
- Fault tolerance
- First-class framework for distributing programming over a large volume of data
  - Leverages memory, disk, network resources, and constraints
- REPL

Spark Framework

- 100x faster than Hadoop
- Specialized libraries for machine learning, graph processing, and database management
- APIs include:
  - Java
  - Scala
  - Python
  - R

What Is Spark?

- Spark (written in Scala) is an optimized engine that supports general execution graphs over an RDD.
  - 100x faster than in memory
  - 10x faster on disk
- Fast and expressive cluster computing system
- Compatible with Apache Hadoop
- Efficient
  - Two to five times less code
  - Builds off of existing Hadoop systems (e.g., Giraph, Mahout, Storm)

Why Spark?

- Unified big data pipeline for:
  - Batch and interactive (Spark Core vs. MR/Tez)
  - SQL (Shark/Spark SQL vs. Hive)
  - Streaming (Spark Streaming vs. Storm)
  - Machine learning (MLlib vs. Mahout)
  - Graph (GraphX vs. Giraph)

Introduction

- When we introduced Hadoop, we talked about functional programming.
  - Basics: Map and reduce functions
- Spark borrows a lot of the principles behind functional programming.
  - Does not support the full functionality

Functional Programming: Background

- Treats computation as the evaluation of mathematical functions
  - Avoids changing-state and mutable data
- Is a declarative programming paradigm
- Has no side effects
  - I.e., changes in state that do not depend on the function inputs

Functional Programming: Map and Reduce

- The idea of map and reduce is 40-plus years old.
  - Present in all functional programming languages
  - E.g., APL, Lisp, and ML
- A key feature of functional languages is the concept of higher order functions:
  - I.e., functions that can accept other functions as arguments.
  - Map and reduce are higher order functions.

Map: A Higher Order Function

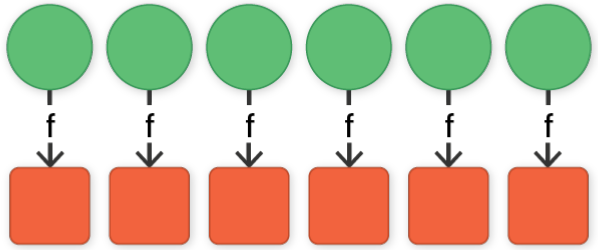
- `Let`
  - `F(x: int)` returns `r: int`
  - `V` be an array of integers
- `W = map(F, V)`
  - `W[i] = F(V[i])` for all `I`
  - I.e., apply `F` to every element of `V`
- Map examples in Haskell
  - `map (+1) [1,2,3,4,5] == [2, 3, 4, 5, 6]`
  - `map (toLower) "abcDEFG12!@#" == "abcdefg12!@#"`
  - `map ('mod' 3) [1..10] == [1, 2, 0, 1, 2, 0, 1, 2, 0, 1]`

Reduce: A Higher Order Function

- Definition (recursive function)
  - `foldl f z [] = z`
  - `foldl f z (x:xs) = foldl f (f z x) xs`
- Examples:
  - `foldl (+) 0 [1..5] == 15`
  - `foldl (+) 10 [1..5] == 25`
- Reduce is also known as fold, accumulate, compress, or inject.
- Reduce as fold takes in a trace function and folds it in between the elements of a list.

Map: Transform the Input List

map f lst: ('a->'b) -> ('a list) -> ('b list)



Folding: Recursion

- Function to sum a list of integers:

```
fun sum (l:int list) :int =
case l of [] =>0
| x::xs =>x + (sum xs)
```

- Function to concatenate a list of strings:

```
fun concat (l: string list) :string
case l of [] =>""
| x::xs =>x ^ (concat xs)
```

Folding: Recursion (with Accumulator)

- Function to sum a list of integers:

```
fun sum' (acc:int) (l:int list):int =
case l of [] =>acc
|x::xs =>sum' (acc+x) xs
```

- Function to concatenate a list of strings:

```
fun concat' (acc:string) (l:string list) :string =
case l of [] =>acc
|x::xs =>concat' (acc+x) xs
```

Tail Recursive Function

- Tail recursive definitions can be compiled into a loop.
  - Faster over data
- `foldr` is not tail recursive (`foldl` is).
  - `fun concat (l:string list) = foldl String.^ "" (rev l)`

Version 1 – With Generators

```
# Fibonacci numbers, imperative style
# https://docs.python.org/2.7/tutorial/modules.html
def fibonacci (n, first = 0, second = 1):
    for i in range(n):
        yield first # Return current iteration
        first, second = second, first + second

print ( x for x in fibonacci(10) )
```

Version 2 – Normal

```
def fibonacci(n):
    first, second = 0, 1
    for i in range(n):
        print first # Print current interation
        first, second = second, first + second #Calculate next value

fibonacci(10)
```

Version 3 – Recursive

```
def fibonacci(n, first = 0, second = 1):
    if n == 1:
        return ( first )
    else:
        return ( first ) + fibonacci(n - 1, second, first + second)

print fibonacci(10)
```

Reduce Operation in Spark

```
def reduce(self, f):
" " "
Reduces the elements of this ROD using the specified
commutative and associative binary operator.
Currently reduces partitions locally.
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add) 15
>>> sc.parallelize((2 for _ in range(10))).map(lambda x:
1).cache().reduce(add) 10
" " "
  def func(iterator):
    acc = None
    for obj in iterator:
      if acc is None:
        acc = obj
      else:
        acc = f(obj, acc)
    if acc is not None:
      yield acc
vals = self.mapPartitions(func).collect()
return reduce(f, vals)
```

Functional Programming vs. Spark

- Functional programming is "purely functional."
  - No side effects for operators
- Spark is not purely functional.
  - Allows side effects
- Spark observes many properties of functional programming.

Functional Programming Review

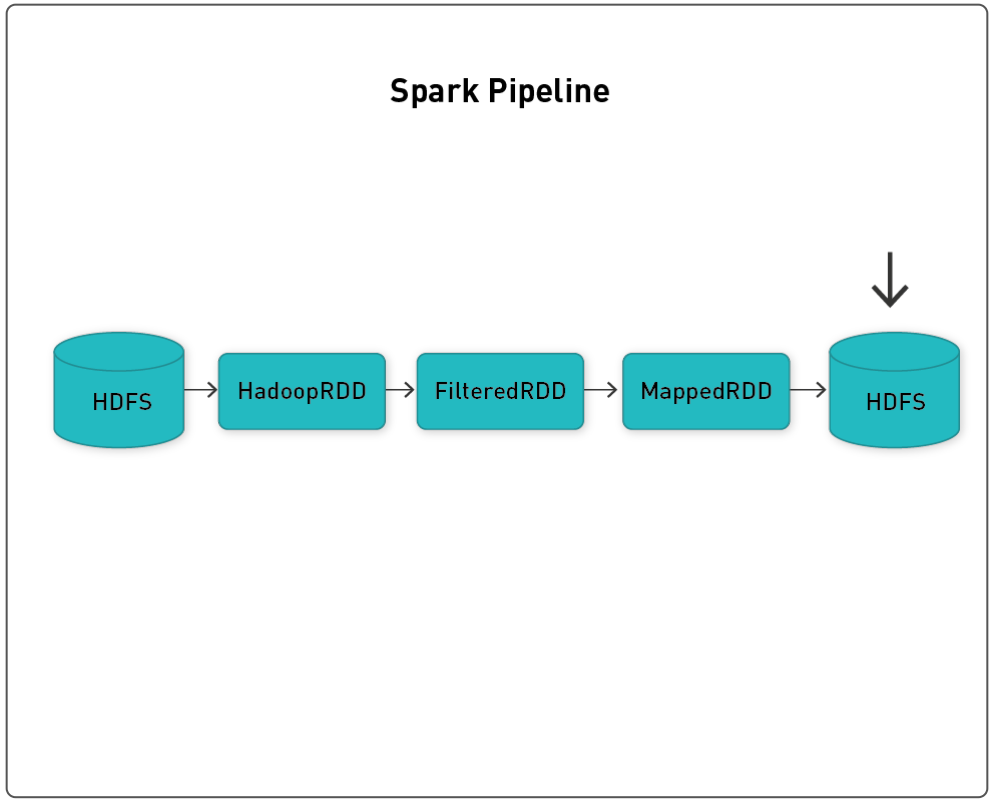
- Functional operations do not modify data.
  - They always create new structures.
- Data flows are implicit in the program design.
- The order of operations does not matter.
- Functional programming is lazy.
  - Nothing happens until you need the data.

Introduction

- Begin learning the specifics of programming the Spark distributed framework.
- Examine basic map and reduce (count) operations over the distributed key-value store (RDD).
    - Resilient distributed data set (RDD)

Spark Basics

- An RDD is simply a distributed collection of elements.
- In Spark, all work is expressed as:
  - Creating new RDDs.
  - Transforming existing RDDs.
  - Calling operations on RDDs to compute a result.
- Spark automatically distributes the data contained in RDDs across your cluster.
  - Also parallelizes the operations performed on them.


Spark Pipeline
HDFS
HadoopRDD
FilteredRDD
MappedRDD
HDFS

Spark Pipeline

- Data structure is distributed over a cluster of nodes.
- Each record is a key-value pair.
- Iterate over each of these records (similar to MapReduce).

Spark Programming Interface

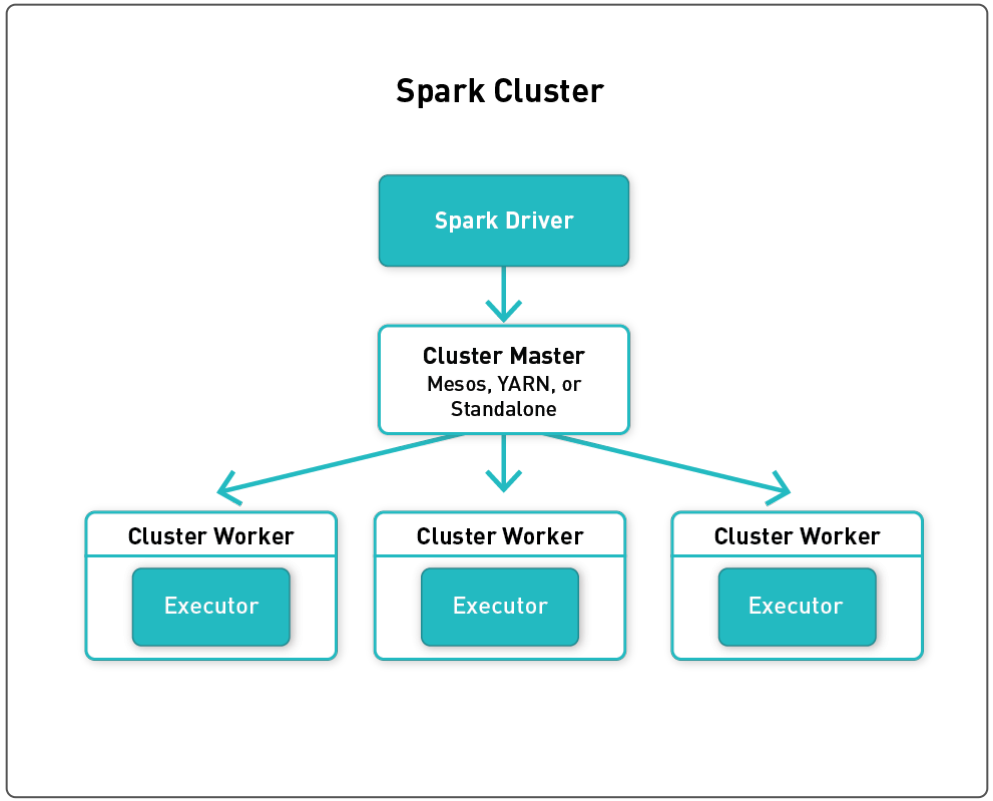
- Language-integrated API:
  - Scala
  - Java
  - Python
  - R
- RDDs provide fault tolerance.
- Operations on RDDs:
  - Transformations: (think *map*), actions (think *reduce*)
- Restricted shared variables (e.g., broadcast, accumulators)

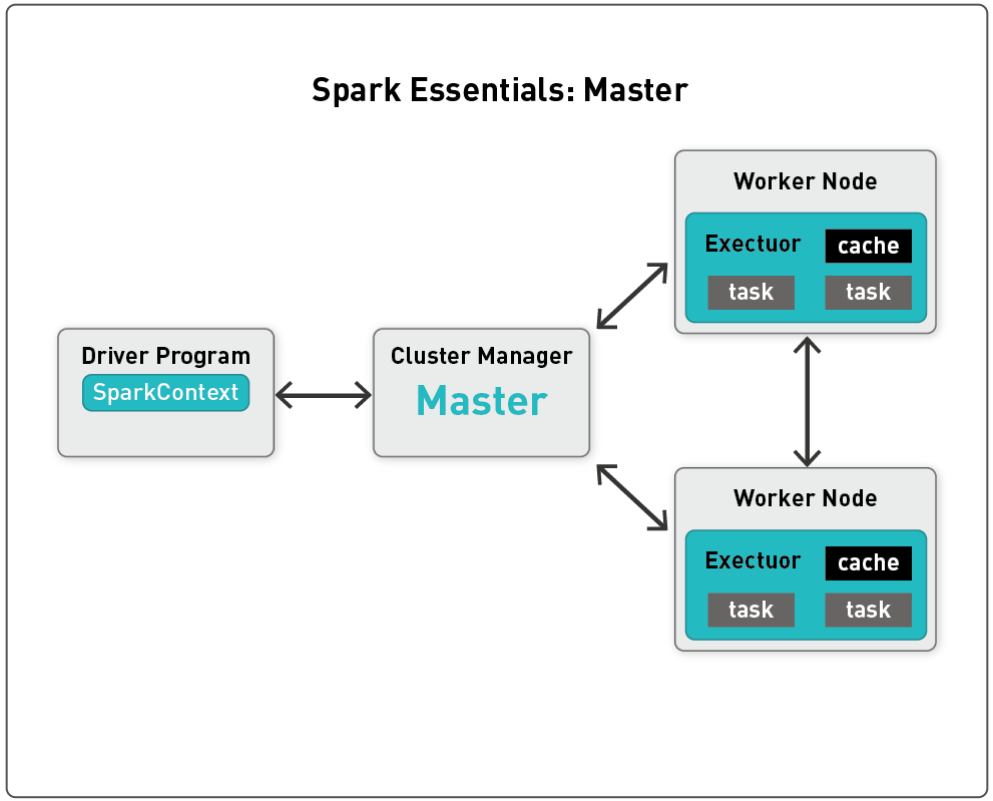
Goals

- Make parallel programs look like local ones.
  - Think about the data frame in a local capacity.

Spark Programming

- Apps in both Scala and Python
  - E.g., IPython Notebook
- Coding from a command line
- Large-scale jobs deployed on clusters


Spark Cluster
Spark Driver
Cluster Master
Mesos, YARN, or
Standalone
Cluster Worker
Executor
Cluster Worker
Executor
Cluster Worker
Executor


Spark Essentials: Master
Worker Node
Exectuor
cache
task
task
Driver Program
SparkContext
Cluster Manager
Master
Worker Node
Exectuor
cache
task
task

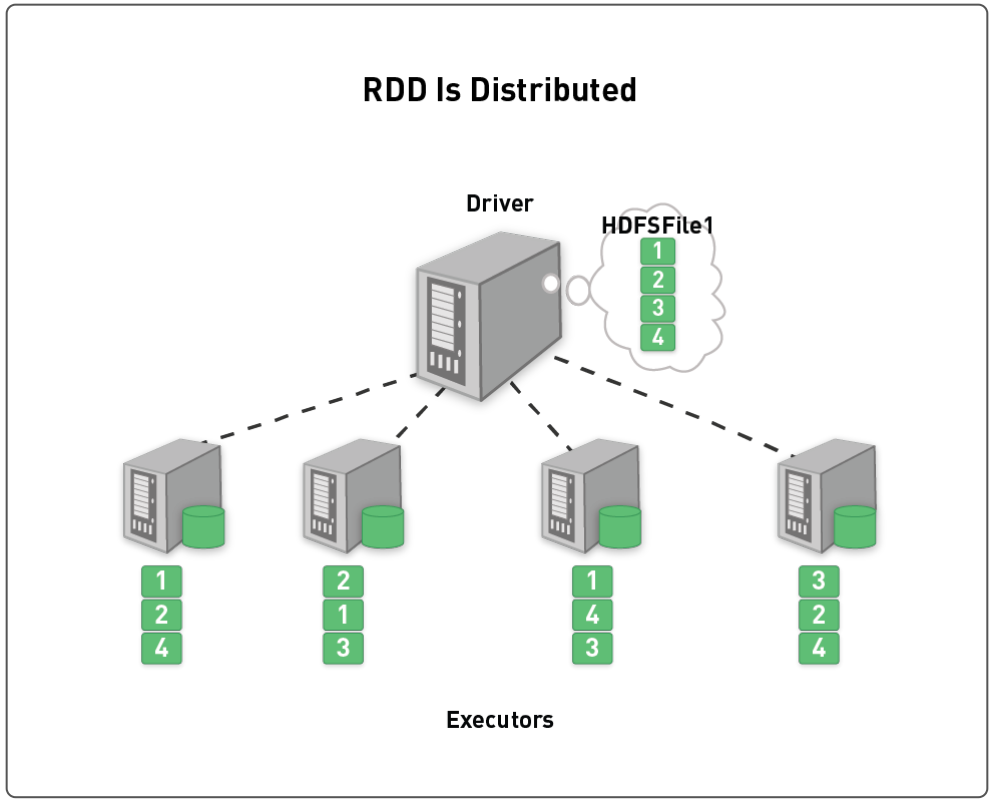
Spark: Connecting to a Cluster

The `master` parameter for a `SparkContext` determines which cluster to use.

Master	Description
`local`	Run Spark locally with one worker thread (no parallelism)
`local |K|`	Run Spark locally with K worker threads (ideally set to number of cores)
`spark: //HOST: PORT`	Connect to a Spark standalone cluster; PORT depends on conf.j (7077 by default)
`im-sos : //HOST : PORT`	Connect to a Mesos cluster; PORT depends on config (5050 by default)

Master: Cluster Manager

- Connects to a cluster manager that allocates resources across applications
- Acquires executors on cluster nodes
  - App code is sent out to the executors
  - Tasks are also sent for the executors to run
- Spark context is responsible for getting code to the workers
- Spark optimizes the pipeline of operations

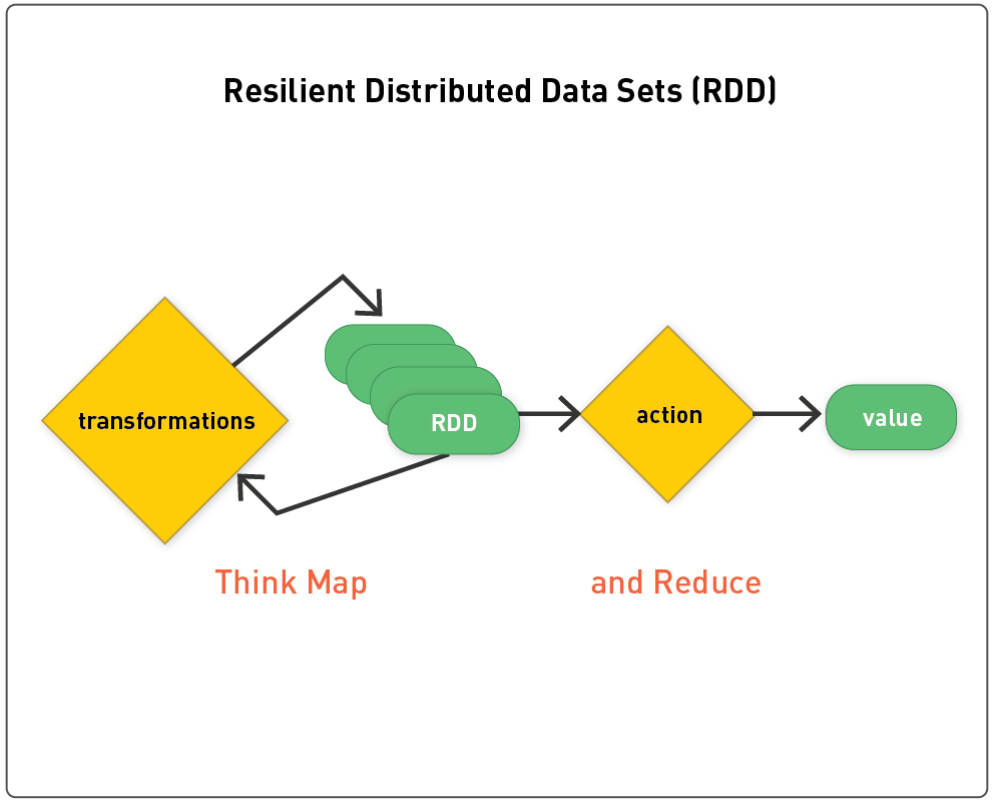

RDD Is Distributed
Driver
HDFSFile1
1
2
3
4
1
2
4
2
1
3
1
4
3
3
2
4
Executors

Resilient Distributed Data Set (RDD)

- The RDD is a data frame.
  - It contains records consisting of key-value pairs
- The RDD can be distributed over a cluster of computers.
- Data are stored in a text file, with one observation on each line.
  - E.g., JSON, zipped, AVRO, Parquet

Creating an RDD

- Spark can create RDDs from any file system supported by Hadoop:
  - E.g., local file system, Amazon S3, Hypertable, HBase.
- Spark supports text files, SequenceFiles, and any other Hadoop `inputFormat`.


Resilient Distributed Data Sets (RDD)
transformations
RDD
action
value
Think Map
and Reduce

RDD Operations: Transformations and Actions

- Transformations:
  - A transformation is an operator applied to each record of an RDD.
  - It generates a new record in a new RDD.
  - Input data are never modified.
- Actions:
  - A results-oriented process
    - E.g., reduce

RDDs: Summary

- An RDD is an immutable, partitioned, logical collection of records.
- It need not be materialized.
  - Contains information to rebuild a data set from stable storage
- Materialize RDDs through actions (e.g., count).
- Partitioning can be based on a key in each record.
- It can be cached for future reuse.

Spark and RDDs

- Transformations are lazy (not computed immediately).
- The transformed RDD gets recomputed when an action is run on it (default).
- An RDD can be persisted into storage in memory or disk.

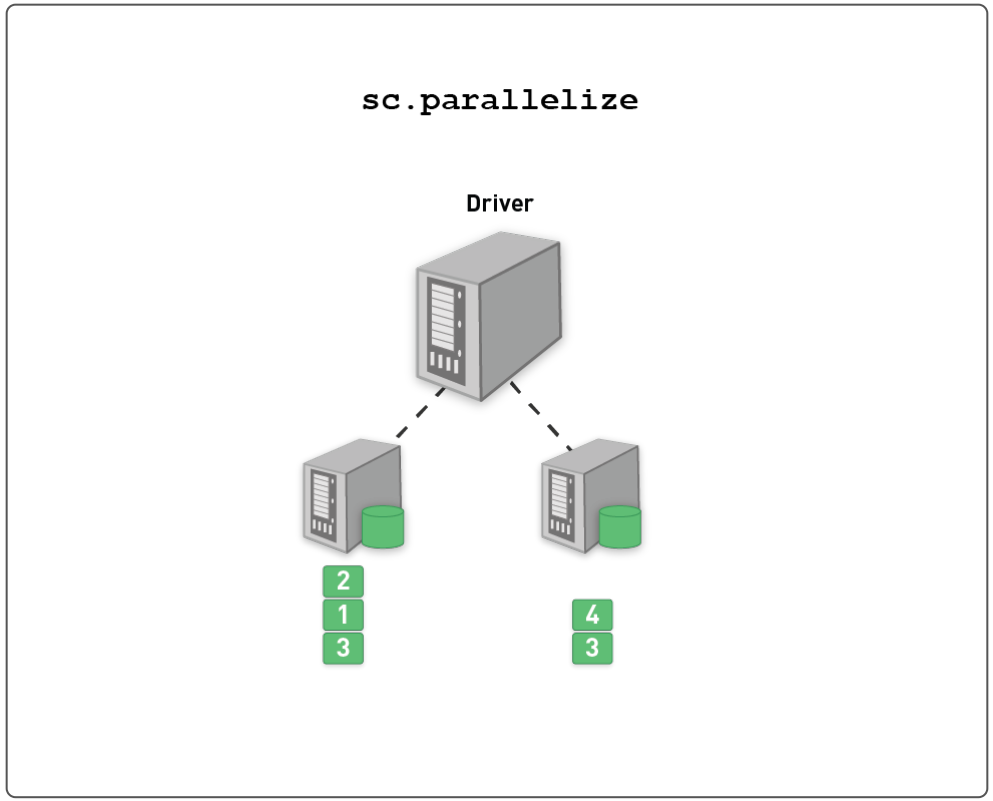
Creating an RDD

- Scala:

```
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@10dl3e3e
```

- Python:

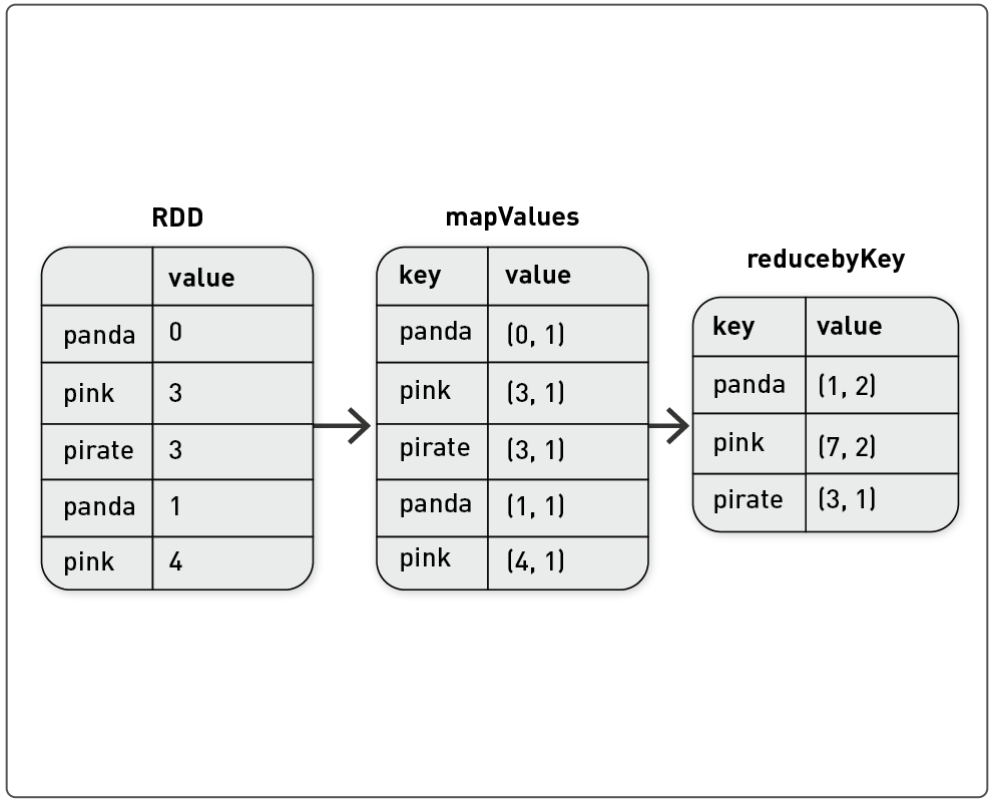
```
>>> data =[1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]
>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD(0) at parallelize at PythonRDD.scala:229
```


sc.parallelize
Driver
2
1
3
4
3

Creating an RDD: Text File

- Python:

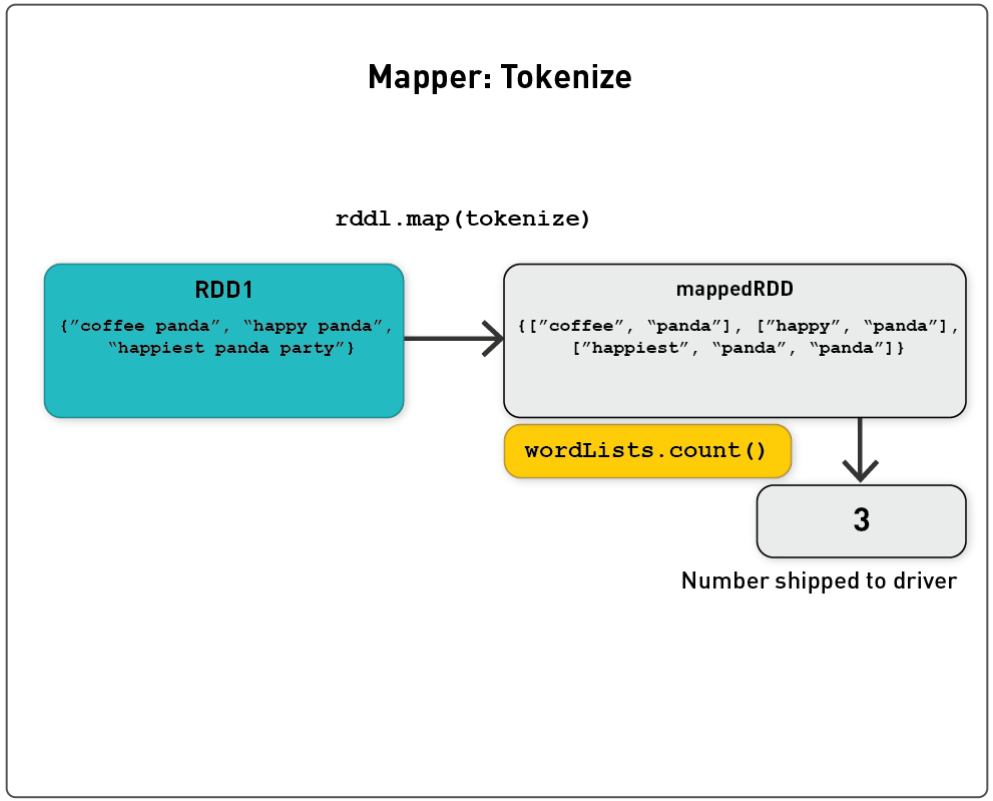
```
>>> distFile = sc.textFile("README.md")
14/04/19 23:42:40 INFO storage.MemoryStore: ensureFreeSpace(36827)
called
with curMem=0, maxMem=318111744
14/04/19 23:42:40 INFO storage.MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 36.0 KB, free 303.3 MB)
>>> distFile
MappedRDD [2] at textFile at NativeMethodAccessorImpl.java:-2
```

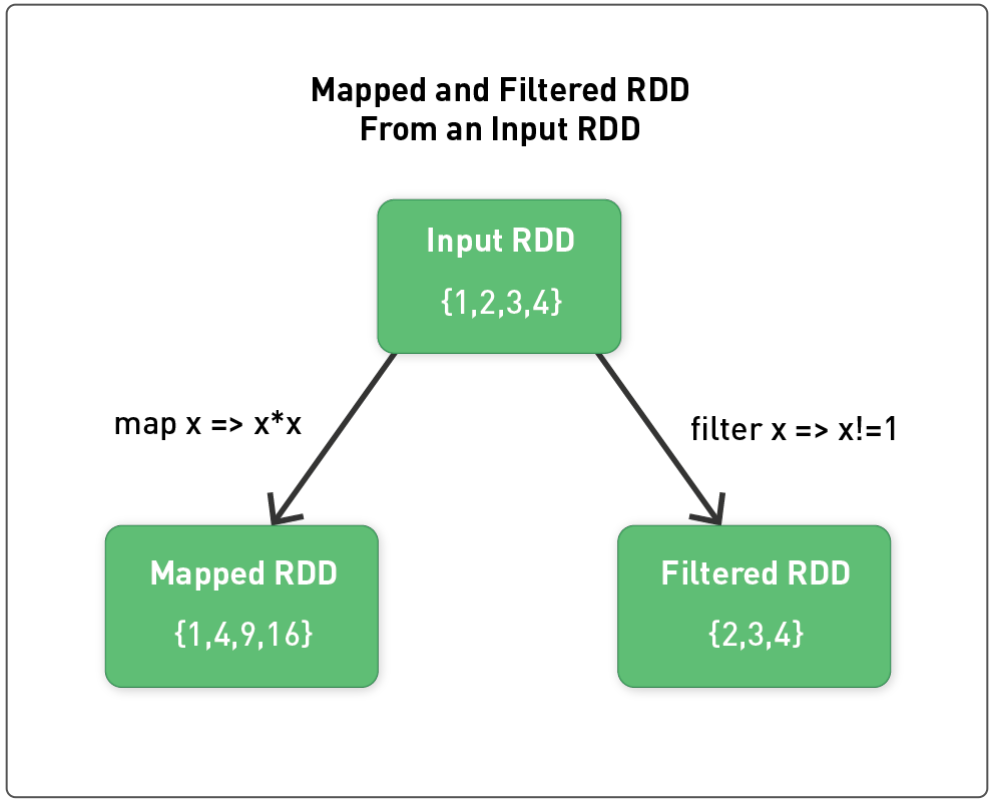

RDD
value
panda 0
pink 3
pirate 3
panda 1
pink 4
mapValues
key value
panda (0, 1)
pink (3, 1)
pirate (3, 1)
panda (1, 1)
pink (4, 1)
reducebyKey
key value
panda (1, 2)
pink (7, 2)
pirate (3, 1)

Mapper: Tokenize

- Apply tokenize to each input record.

```
def tokenize(line):
  return(line.split(" ")))

rdd1= sc.parallelize(["coffee panda", "happy panda", "happiest panda
party"])
words = rdd.map(tokenize)
```


Mapper: Tokenize
rddl.map(tokenize)
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "panda"]}
wordLists.count()
3
Number shipped to driver

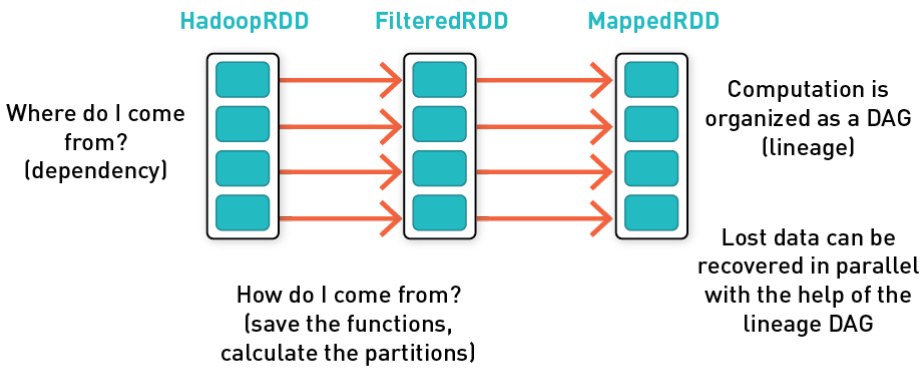

Mapped and Filtered RDD
From an Input RDD
Input RDD
{1,2,3,4}
map x => x*x
filter x => x!=1
Mapped RDD
{1,4,9,16}
Filtered RDD
{2,3,4}

Lazy Evaluation

- Example: Distribution of a file across the network
  - Examine just the first line of this file.
    - `lines = sc.textFile("exampleFile") #if not lazy would read whole file in`
  - Spark:
    - `lines = sc.textFile("exampleFile").first()`

Computation Is Organized as a DAG

```
errorMessages= sc.parallelize.filter(lambda x: 'ERROR' in x) \
      .map(lambda line: line.split(" ")[1]) #second word
```



Computation is organized as a DAG (lineage)

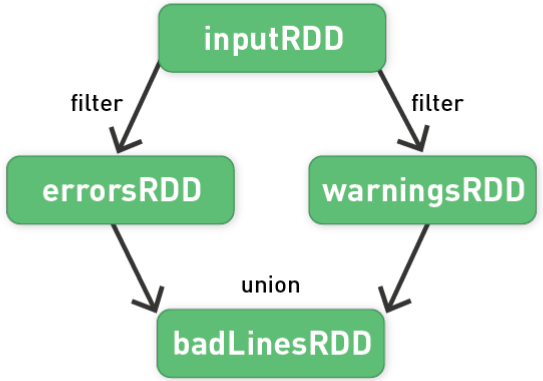
Lost data can be recovered in parallel with the help of the lineage DAG

Spark: Directed Acyclic Graph (DAG)

- Operations assembled into a pipeline are organized as a DAG:
  - Instructions on how to build an RDD.
- We can go from data to computation using **lineage**:
  - A set of instructions that are used to construct a particular pipeline.
  - This acts as a means of fault tolerance.
    - A "lost" RDD can be recomputed.

RDD Lineage Graph

```
errorsRDD = inputRDD.filter(lanbda x:  "error" In x)
warningsRDD = inputRDD.filter(lanbda x:    "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```



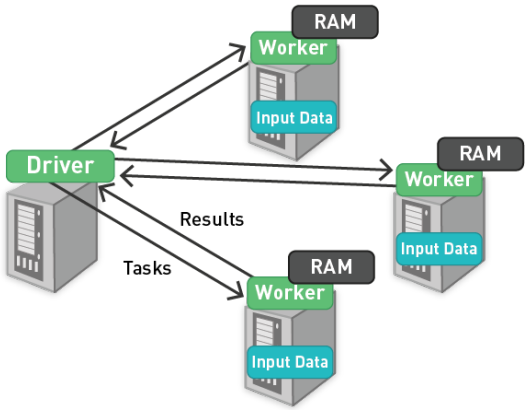
counts.toDebugString()

```
in [5]: lines = sc.parallelize(["Data line 1", "Mining line 2", "data
line 3", "Data line 4", "Data Mining line 5" ))
counts = lines.flatMap(lambda line: line.split(" ")) \
.map(lambda word: (word, 1)) \
.reduceByKey(lambda a, b: a + b)
counts.collect()

Out[5]: [('1', 1),
  ('line', 5),
  ('Mining', 2),
  ('3', 1),
  ('2', 1),
  ('data', 1),
  ('5', 1),
  ('Data', 3),
  ('4', 1)]

in [7]: counts.toDebugString()

Out[7]: '(8) PythonRDD[7] at collect at :4 []\n
| MapPartitionsRDD[6] at mapPartitions at PythonRDD.scala:346 []\n
| ShuffledRDD[5] at partitionBy at NativeMethodAccessorImpl.java:-2 []\n
+-(8) PairwiseRDD[4] at reduceByKey at :2 []\n
| PythonRDD[3] at reduceByKey at :2 []\n
| ParallelCollectionRDD[2] at parallelize at PythonRDD.scalar396 []'
```

Introduction

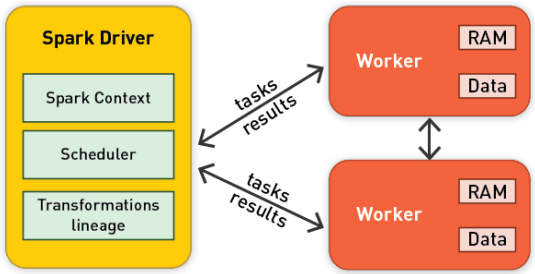
- We used basic map and reduce operations over the distributed key-value store (RDD).
- We'll look at base RDDs (no keys, just values).
- We'll look at pair RDDs in future sections.
- We'll write some code.

Spark Runtime



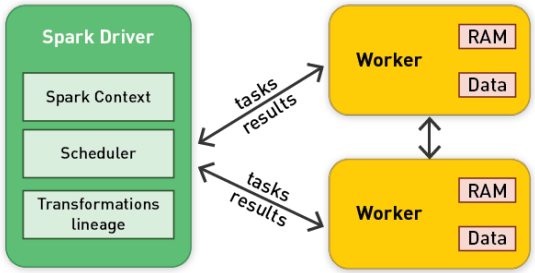
- The user's driver program launches multiple workers.
- Workers read data blocks from a distributed file system.
- Workers persist computed RDD partitions in memory.

Drivers



- The driver defines and launches operations on RDDs.
- The driver tracks RDD lineages so lost partitions can be recovered.
- The driver uses RDD DAGs in the scheduler to define stages and tasks to be executed.
- The driver ships tasks to workers.

Workers



- Workers read, transform, and write RDD partitions.
- Workers communicate with other workers to share cache and shuffle data.

Closure

- The Spark driver is responsible for creating, transforming, and saving RDDs based on the programmer's instruction.
- The driver needs to communicate both what tasks need to be performed and which RDDs they need to be performed on.
- A **closure** is a data structure storing a function together with an environment.
- It is a mapping associating each free variable of the function (used locally but defined in an enclosing scope) with the value or storage location the name was bound to at the time the closure was created.

Closure Usage

- Imagine a mapper function called `split` that separates each line of input into tokens.
- We pass just the name of the function.
- If the function is user defined, we'll have to encode it in byte form (serialize it).
- We'll serialize our RDD operative functions with closures.

Python:

```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lamda x: x.split(' ')).collect()
```

Writing a Program in Spark

- We'll use IPython Notebooks.
- Begin by opening a connection to a computing cluster using SparkContext.

```
from pyspark import SparkConf, SparkContext
conf =
SparkConf().setMaster("local").setAppName("App")
sc = SparkContext(conf = conf)
```

IP[y]: Notebook Chapter-3-RDDS Last Checkpoint: Mar 16 15:24 (unsaved changes)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [1]:
```
import os
import sys

#set up Spark and give us a spark context sc
spark_home = os.environ['SPARK_HOME']='/Users/jshanahan/Software/spark-1.2.1-bin-hadoop2.4/'  #desktop

print "["+ spark_home+"]"
if not spark_home:
    raise ValueError('SPARK_HOME environment variable is not set')

sys.path.insert(0, os.path.join(spark_home, 'python'))
sys.path.insert(0, os.path.join(spark_home, 'python/lib/py4j-0.8.2.1-src.zip'))
execfile(os.path.join(spark_home, 'python/pyspark/shell.py'))
```

```
[/Users/jshanahan/Software/spark-1.2.1-bin-hadoop2.4/]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

Using Python version 2.7.8 (default, Aug 21 2014 15:21:46)
SparkContext available as sc.
```

In [28]:
```
import numpy as np
import pylab

dataRDD = sc.parallelize(np.random.random_sample(1000))
data2X= dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2X.filter(lambda x: x > 1.0)
cachedRDD = dataGreaterThan1.cache()
```

In [29]:
```
cachedRDD.filter(lambda x: x<1).count()
```

In [28]:
```
import numpy as np
import pylab

dataRDD = sc.parallelize(np.random.random_sample(1000))
data2X= dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2X.filter(lambda x: x > 1.0)
cachedRDD = dataGreaterThan1.cache()
```

In [29]:
```
cachedRDD.filter(lambda x: x<1).count()
```

Out[29]: 0

In [31]:
```
cachedRDD.filter(lambda x: x>1).count()
```

Out[31]: 514

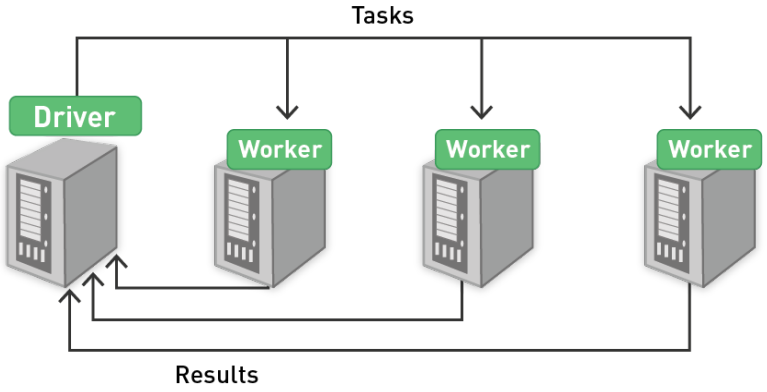
In [32]:
```
cachedRDD.filter(lambda x: x>1).count()
```

Out[32]: 514

Transformations vs. Parallel Operations

Transformations	Parallel operations (actions)
Define a new RDD.	Return a result to driver.
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	⋮
join	
cache	
⋮	

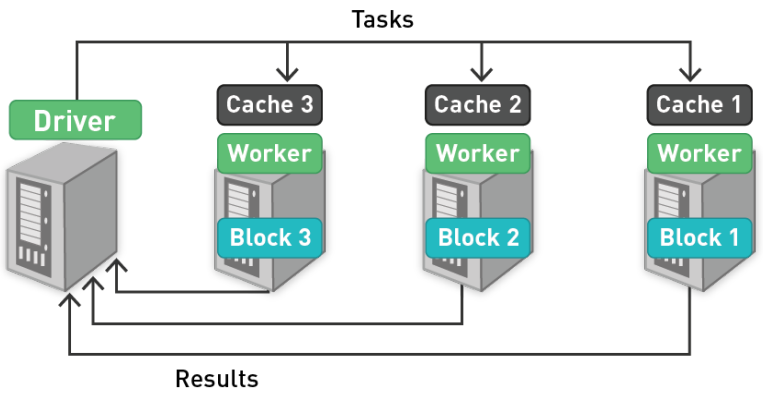
Random Numbers Example: Laziness



- We have a Spark context (connection to cluster driver).
- Load random numbers into our RDD.

```
dataRDD =
sc.parallelize(np.random.random_sample(1000))
data2x = dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2x.filter(lambda x: x > 1)
cachedRDD = dataGreaterThan1.cache()
```

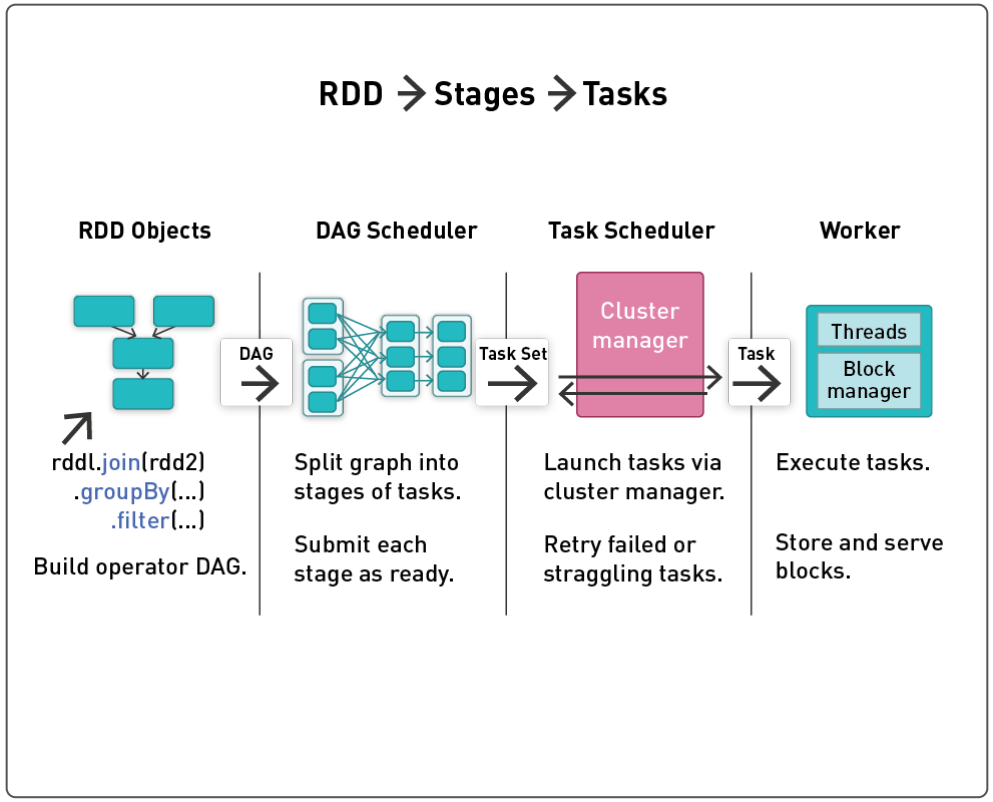
Random Numbers Example: Execution




```
dataRDD =
sc.parallelize(np.random.random_sample(1000))
data2x = dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2x.filter(lambda x: x > 1)
cachedRDD = dataGreaterThan1.cache()

> cachedRDD.filter(lambda x: x > 0.5).count()
508
```

- Additional actions can be performed from RDDs in memory.

It's preferable to cache, rather than recompute.


RDD → Stages → Tasks
RDD Objects
DAG Scheduler
Task Scheduler
Worker
DAG
Task Set
Cluster manager
Task
Threads
Block manager
rddl.join(rdd2)
.groupBy(...)
.filter(...)
Build operator DAG.
Split graph into stages of tasks.
Submit each stage as ready.
Launch tasks via cluster manager.
Retry failed or straggling tasks.
Execute tasks.
Store and serve blocks.

RDD Creation

- By array:

```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
```
…
```
rdd: org.apache.spark.rdd.RDD[Int] = …
```

- From file:

```
val rdd2 = sc.textFile("hdfs:///some/path.txt")
```
…
```
rdd2: org.apache.spark.rdd.RDD[String] = …
```

- From directory

Transformations

Transformation	Description
`map(`*`func`***`)`**	Return a new distributed data set formed by passing each element of the source through a function *`func`*.
`filter(`*`func`***`)`**	Return a new data set formed by selecting those elements of the source on which *`func`* returns true.
`flatMap(`*`func`***`)`**	Similar to `map`, but each input item can be mapped to zero or more output items. (So *`func`* should return a `Seq` rather than a single item.)
`sample(`*`withReplacement, fraction, seed`***`)`**	Sample a fraction *`fraction`* of the data, with or without replacement, using a given random-number generator *`seed`*.
`union(`*`otherDataset`***`)`**	Return a new data set that contains the union of the elements in the source data set and the argument.
`distinct([`*`numTasks`***`])`**	Return a new data set that contains the distinct elements of the source data set.

Transformations (cont.)

Transformation	Description
groupByKey([*numTasks*])	When called on a data set of `(K, V)` pairs, returns a data set of `(K, Seq[V])` pairs.
reduceByKey(*func*, [*numTasks*])	When called on a data set of `(K, V)` pairs, returns a data set of `(K, V)` pairs, where the values for each key are aggregated using the given reduce function.
sortByKey([*ascending*], [*numTasks*])	When called on a data set of `(K, V)` pairs where `K` implements `Ordered`, returns a data set of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
join(*otherDataset*, [*numTasks*])	When called on data sets of type `(K, V)` and `(K, W)`, returns a data set of `(K, (V, W))` pairs, with all pairs of elements for each key.
cogroup(*otherDataset*, [*numTasks*])	When called on data sets of type `(K, V)` and `(K, W)`, returns a data set of `(K, Seq[V], Seq[W])` tuples. Also called `groupWith`.
cartesian(*otherDataset*)	When called on data sets of types `T` and `U`, returns a data set of `(T, U)` pairs (all pairs of elements).

Transformations vs. Actions

- Transformations take an RDD as input and give a new RDD as output.
- Transformations are lazy.
- Nothing happens without actions.
- Actions take an RDD as input and give a result back to the driver or save it to external storage.

```
>>> pythonLines = lines.filter(lambda line:
"Python" in line)
>>> pythonLines.first()
u'## Interactive Python Shell'
```

Actions

Action	Description
reduce(*func*)	Aggregate the elements of the data set using a function *func* (which takes two arguments and returns one); should also be commutative and associative so it can be computed correctly in parallel.
collect()	Return all the elements of the data set as an array at the driver program; usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the data set.
first()	Return the first element of the data set; similar to take(*n*).
take(*n*)	Return an array with the first *n* elements of the data set; currently computed by the driver program rather than in parallel.
takeSample (*withReplacement*, *fraction*, *seed*)	Return an array with a random sample of *num* elements of the data set, with or without replacement, using the given random-number generator *seed*.

Actions (cont.)

Action	Description
`saveAsTextFile(`*`path`***`)`**	Write the elements of the data set as a text file (or set of text files) in a given directory in the local file system, HDFS, or any other Hadoop-supported file system. Spark calls `toString` on each element to convert it to a line of text in the file.
`saveAsSequenceFile(`*`path`***`)`**	Write the elements of the data set as a Hadoop `SequenceFile` in a given path in the local file system, HDFS, or any other Hadoop-supported file system; available only on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable`. (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc.)
`countByKey()`	Available only on RDDs of type `(K, V)`. Returns a "map" of `(K, Int)` pairs with the count of each key.
`foreach(`*`func`***`)`**	Run a function *`func`* on each element of the data set; usually done for side effects such as updating an accumulator variable or interacting with external storage systems.

Example in Scala and Python

Scala

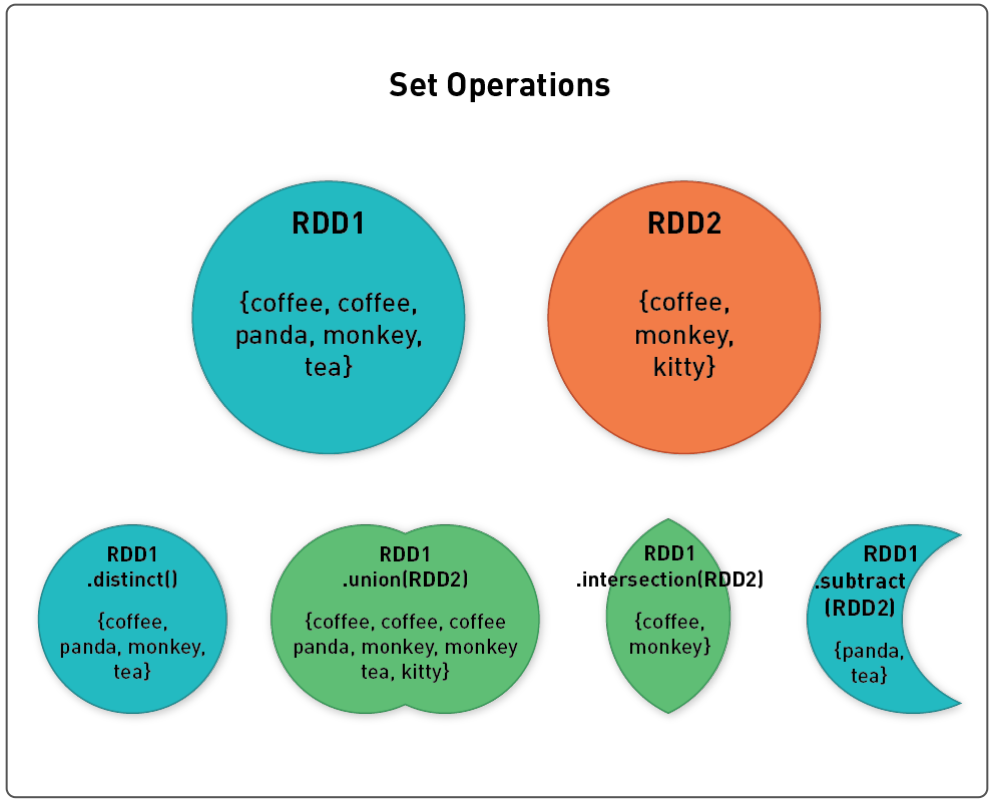
```
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word
=> (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Python

```
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split('
')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

Persistence

Transformation	Description
`MEMORY_ONLY`	Store RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, some partitions may not be cached and will be recomputed on the fly whenever needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store additional partitions on disk, and read them from disk whenever needed.
`MEMORY_ONLY_SER`	Store RDD as serialized Java objects (one byte array per partition). Generally more space efficient than deserialized objects, especially when using a fast serializer, but more CPU intensive to read.
`MEMORY_AND_DISK_SER`	Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk instead of recomputing on the fly whenever needed.
`DISK_ONLY`	Store RDD partitions only on disk.
`MEMORY_ONLY_2`, `MEMORY_AND_DISK_2`, `etc.`	Same as levels above, but replicate each partition on two cluster nodes.


Set Operations
RDD1
{coffee, coffee, panda, monkey, tea}
RDD2
{coffee, monkey, kitty}
RDD1 .distinct()
{coffee, panda, monkey, tea}
RDD1 .union(RDD2)
{coffee, coffee, coffee panda, monkey, monkey tea, kitty}
RDD1 .intersection(RDD2)
{coffee, monkey}
RDD1 .subtract (RDD2)
{panda, tea}

Be aware of operation cost.

Pitfalls: Memory Overload

- Actions may cause the movement of an enormous amount of data.
- The local driver computer can crash if it's shipped too much information.

```
errorsRDD = inputRDD.filter(lambda x: "error" in
x)
warningsRDD = inputRDD.filter(lambda x: "warning"
in x)
badLinesRDD = errorsRDD.union(warningsRDD)

print "Input had " + badLinesRDD.count() + "
concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
  print line
```

Summary

- That was our first dive into Spark.
- We saw how to create a program and run it.
- We saw the difference between transformations and actions.
- We worked on base RDDs.
- We'll work on pair RDDs momentarily.

Example: How Spark processes a HDFS space file, works, and saves output back to HDFS.

Filtering Error Lines from Logfile

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<input path or local file>") \
    .filter(lambda x: 'ERROR' in x) \        # Contains 'ERROR'
    .map(lambda line: line.split(" ")[1])    # 2nd word & line
    .saveAsTextFile("hdfs://<output path or local directory>")
```

Scala

```
sc.textFile("hdfs://<input>")
    .filter(_.startsWith("ERROR"))
    .map(_.split(" ")(1))
    .saveAsTextFile("hdfs://<output>")
```

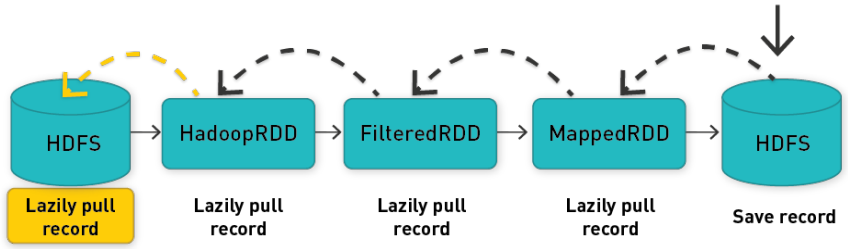
Filtering Error Lines from Logfile

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<input path or local file>") \
    .filter(lambda x: 'ERROR' in x) \        # Contains 'ERROR'
    .map(lambda line: line.split(" ")[1])    # 2nd word & line
    .saveAsTextFile("hdfs://<output path or local directory>")
```



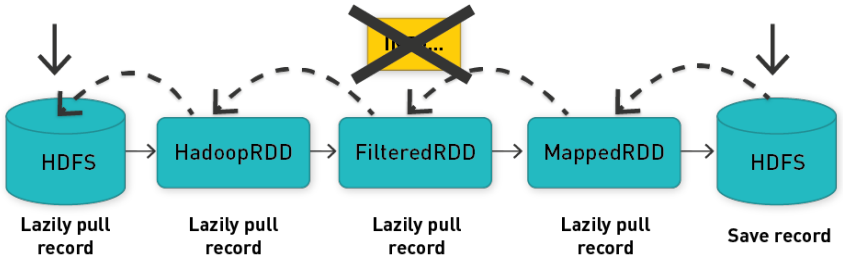
Filtering Error Lines from Logfile

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<input path or local file>") \
    .filter(lambda x: 'ERROR' in x) \        # Contains 'ERROR'
    .map(lambda line: line.split(" ")[1])    # 2nd word & line
    .saveAsTextFile("hdfs://<output path or local directory>")
```



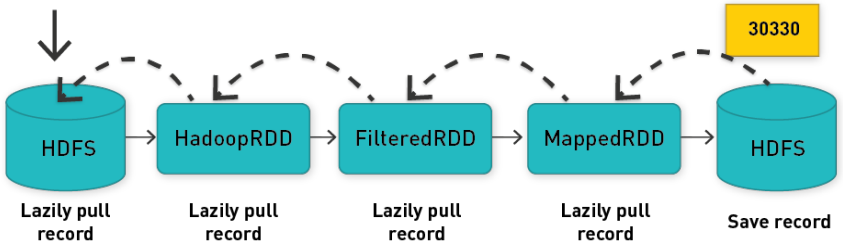
Filtering Error Lines from Logfile

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
sc.textFile("hdfs://<input path or local file>") \
    .filter(lambda x: 'ERROR' in x) \        # Contains 'ERROR'
    .map(lambda line: line.split(" ")[1])    # 2nd word & line
    .saveAsTextFile("hdfs://<output path or local directory>")
```



This process is repeated for every record in the source file.

Caching

- One block per RDD partition
- LRU cache eviction
- Locality aware
- Evicted blocks *recomputable in parallel* with help of RDD lineage DAG

Filtering Error Lines with Cached Intermediate

HDFS file

```
INFO not much going on here
ERROR 30330 task aborted with no status
ERROR 44404 task aborted with no status
INFO doodle
```

pySpark

```
cached = sc.textFile("hdfs://<input path>") \
     .filter(lambda x: 'Error' in x).cache()
     \                                          # Contains 'ERROR'

cached.map(lambda line: line.split(" ")[1])   # 2nd word & line
     .saveAsTextFile("hdfs://<output path>")
```

Scala

```
val cached = sc
     .textFile("hdfs://<input>")
     .filter(_.startsWith("ERROR"))
     .cache()

cached
     .map(_.split(" ")(1))
     .saveAsTextFile("hdfs://<output>")
```

Spark Is Complex

```
>>> help(pyspark)
```

- `help(pyspark)` provides refinable, command-line documentation.
- Additional resources can be found on the Spark home page.

Pair RDDs

- Composed of key-value pairs
  - Common data type in many Spark operations
- Used for aggregations
- Initially formatted by ETL processes
- Offer new operations

We'll also discuss partitioning: controlling data distribution to minimize communication costs.

Key-Value Pairs

Allow:

- Acting on all keys in parallel
- Regrouping data across network
- Aggregating data separately for each key (`reduceByKey()`)
- Extracting existing fields for use as keys
    - E.g., event time or customer ID

Pair RDD Creation

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

- Create a pair RDD from an in-memory collection in Scala and Python by calling `SparkContext.parallelize()` on a collection of pairs.
- Pair RDDs use all transformations available to standard RDDs, with some additions.

Example: Filtering

Key	Value
holden	likes coffee
panda	likes long strings and coffee

Filter →

Key	Value
holden	likes coffee

Python

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Scala

```
pairs.filter{case (key, value) => value.length < 20}
```

Example: First Word as Key

```
#Example 4-1. Creating a pair RDD
# using the first word as the key in Python

lines = sc.parallelize(["Data line 1", "Mining
line 2", \
"data line 3", "Data line 4", "Data Mining line
5"])
#first word and the original line
pairs = lines.map(lambda x: (x.split(" ")[0], x))
pairs.collect()

[('Data', 'Data line 1'),
('Mining', 'Mining line 2'),
('data', 'data line 3'),
('Data', 'Data line 4'),
('Data', 'Data Mining line 5')]
```

Example: Something More Elaborate

```
#Paired RDD examples
logFileName='log.txt'
#Word count for error messages exercise
# READ Lines And PRINT
recs = sc.textFile(logFileName)
#recs = sc.textFile(logFileName).filter(lambda l: "ERROR" in l)
wcErrors = recs.flatMap(lambda l: l.split(" ")) \
       .map(lambda word: (word, 1)) \
       .reduceByKey(lambda x, y: x + y)
for (key, value) in wcErrors.collect():
   print "WordCount-->> %s:%d"%(key, value)
```

```
WordCount-->> :1
WordCount-->> dying:1
WordCount-->> for:1
WordCount-->> reasons:1
WordCount-->> ERROR    mysql:1
WordCount-->> angry:1
WordCount-->> cluster::1
WordCount-->> context:1
WordCount-->> spark:2
WordCount-->> with:1
WordCount-->> you:1
WordCount-->> me?:1
WordCount-->> WARN    dave,:1
WordCount-->> ERROR    php::1
WordCount-->> unknown:1
WordCount-->> it:1
WordCount-->> replace:1
WordCount-->> cluster:1
WordCount-->> missing...darn:1
WordCount-->> WARN    xylons:1
WordCount-->> at:1
WordCount-->> ERROR    :1
WordCount-->> approaching:1
WordCount-->> are:1
```

All transformations available for base RDDs are also available for pair RDDs.

Transformations: Single Pair RDD

```
rdd = {(1, 2), (3, 4), (3, 6)}
```

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey( createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See examples 4-12 through 4-14.	
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues( x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, producing a key-value entry with the old key for each element returned; often used for tokenization	`rdd.flatMapValues( x => (x to 5))`	`{(1, 2), (1, 3), (1, 4) (1, 5), (3, 4), (3, 5)}`

Transformations: Single Pair RDD (cont.)

```
rdd = {(1, 2), (3, 4), (3, 6)}
```

Function name	Purpose	Example	Result
`keys()`	Return an RDD of only keys.	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of only values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

Transformations: Two Pair RDDs

`rdd = {(1, 2), (3, 4), (3, 6)}, other = {(3, 9)}`

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.rightOuterJoin(other)`	`{(3, (Some(4), 9)), (3, (Some(6), 9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1, (2, None)), (3, (4, Some(9))), (3, ([4, 6], [9]))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1, ([2], [])), (3, ([4, 6], [9]))}`

Combination

```
rdd.mapValues(lambda x: (x, 1)) \
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

Key	Value
panda	0
pink	3
pirate	3
panda	1
pink	4

mapValues

→

Key	Value
panda	(0, 1)
pink	(3, 1)
pirate	(3, 1)
panda	(1, 1)
pink	(4, 1)

↓ reduceByKey

Key	Value
panda	(1, 2)
pink	(7, 2)
pirate	(3, 1)

- Spark will automatically perform combining locally before computing global key totals.
- You can customize the combiner with `combineByKey()`.

Example: Word Count

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)) \
.reduceByKey(lambda x, y: x + y)
```

All actions available for base RDDs are available for pair RDDs.

Actions

```
rdd = {(1, 2), (3, 4), (3, 6)}
```

Function name	Purpose	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the results as a map to provide easy look-up.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

Parallelism

- Communication is very expensive in distributed programs.
- Organizing data to minimize network traffic can greatly improve performance.
- Spark's partitioning is available on all key-value pair RDDs and groups elements by a function of each key.

```
# Default
sc.parallelize(data).reduceByKey(lambda x, y: x + y)
```

- We can intervene and enforce our own partitioning.

```
# Custom
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)
```

Conclusion

- MapReduce has only two operators.
- Spark has more than 80.

Overview

- Word frequency program in Spark
  - Plus some variations
- Develop and debug in Spark

Word Frequency in Python

```
def map(key, value):
       for word in value.split():
              emit(word, 1)

def reduce(key, values):
       count = 0
       for val in values:
              count += val
       emit(key, count)
```

Word Frequency in Spark

```
from operator import add

def tokenize(text):
     return text.split

# Create RDD
text = sc.textFile("tolstoy.txt")

# Transform
wc = text.flatMap(tokenize)
wc = wc.map(lambda x: (x,1)).reduceByKey(add)

wc.saveAsTextFile("counts")
```

Paired RDD Example

Efficient Version (One Less Map)

```
lines = sc.parallelize(["Data line 1", "Mining
       line 2", "data line 3", "Data line 4",
       "Data Mining line 5"])
def emitWordCounts(line):
       wordCounts = []
       for w in line.split(" "):
              wordCounts.append((w,1))
       return (wordCounts)

counts = lines.flatMap(emitWordCounts)

counts.collect()
```

In-Memory Combiner at Record Level

Most Efficient Version

```
lines = sc.parallelize(["Data line 1", "Mining
       line 2", "data line 3", "Data line 4",
       "Data Mining line 5"])
def emitWordCounts(line):
       wordCounts = []
       for w in line.split(" "):
              wordCounts.append((w,1))
       return (wordCounts)

counts = lines.flatMap(emitWordCounts)
       .reduceByKey(lambda a, b: a + b)
counts.collect()
```

Locally aggregating words and counts saves network resources.

Debugging in Spark: Example

```
lines = sc.parallelize(["Data line 1", "Mining
       line 2", "data line 3", "Data line 4",
       "Data Mining line 5"])
def emitWordCounts(line):
       wordCounts = {}
       for w in line.split(" "):
              if wordCounts.has_key(w):
                     wordCounts[w] = wordCounts[w]
                     + 1
              else:
                     wordCounts[w] = 1
       for key, value in wordCounts.items():
              print key, ":", value
       return (wordCounts.items())

---------------------

emitWordCounts("Mining line 2")
counts = lines.flatMap(emitWordCounts)
            .reduceByKey(lambda a, b: a + b)
counts.collect()
```

Data Organization in Cluster

- Caching
  - Either in memory or on disk
  - Typically only one copy

Map Partitions

Average with mapPartitions()

```
def partitionCtr(nums):
     # Compute sumCounter for partition
     sumCount = [0, 0]       for num in nums:
          sumCount[0] += num
          sumCount[1] += 1
     return [sumCount]

def fastAvg(nums):
     # Compute the average
     sumCount = nums.mapPartitions(partitionCtr)
                    .reduce(combineCtrs)
     return sumCount[0] / float(sumCount[1])
```

Goal: Do as much calculation locally as possible.

Map Partitions

Average with mapPartitions()

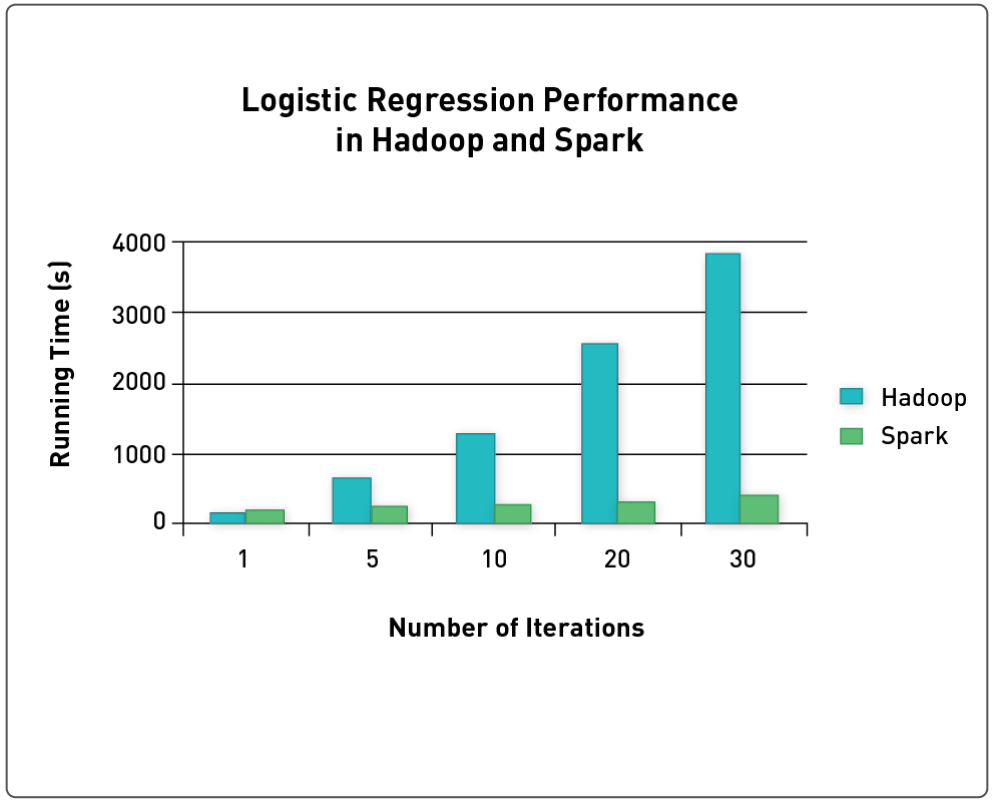
```
def partitionCtr(nums):
     # Compute sumCounter for partition
     sumCount = [0, 0]       for num in nums:
          sumCount[0] += num
          sumCount[1] += 1
     return [sumCount]

def fastAvg(nums):
     # Compute the average
     sumCount = nums.mapPartitions(partitionCtr)
                    .reduce(combineCtrs)
     return sumCount[0] / float(sumCount[1])
```

Goal: Do as much calculation locally as possible.

How can *mapPartitions()* be used to make a better word count?

Overview of Spark

- Fast and expressive cluster computing system
- Compatible with Apache Hadoop
- Efficient
    - 100x faster than Hadoop in memory
    - 10x faster on disk
    - In-memory storage
    - General execution graphs
    - Two to five times less code
- Rich APIs (Java, Scala, Python, R)
- Interactive shell


Logistic Regression Performance
in Hadoop and Spark
Running Time (s)
4000
3000
2000
1000
0
1
5
10
20
30
Number of Iterations
Hadoop
Spark

Key Idea: RDDs

- **R**esilient **D**istributed **D**ata structures
- Abstraction representing a read-only collection of objects across a set of machines
- RDDs can be:
  - Rebuilt from lineage (fault tolerance)
  - Accessed via MapReduce-like (functional) parallel operations
  - Cached in memory for immediate reuse
  - Written to storage
- Meet Hadoop requirements for a distributed computation framework

Programming with RDDs

- Two types of operations:
  - Transformations
    - *Lazily* evaluated
    - Not executed until action occurs
    - Allow for task optimization
  - Actions

	ML Haskell	Hadoop MapReduce	Spark (Scala)	PySpark
Purely functional prog.	Yes	Minimally	Partially	Partially
Lazy evaluation	Yes	No	Yes	Yes
Higher order functions	Yes	Yes	Yes	Yes
Side effects	No	Yes	Yes	Yes
Parallel Computation				
Fault tolerance	No	Yes	Yes	Yes
Serialization	No	Yes for Java, no for streaming	Yes	Partially
Communication	Barrier	Barrier, shuffle and broadcast, pipeline	Barrier, shuffle and broadcast, BSP, pipeline	Barrier, shuffle and broadcast, BSP, pipeline
Software Engineering				
Programming	Yes	Limited (e.g., no loops)	Yes	Yes
REPL	Yes	No	Yes	Yes
Big data processing	No	Yes	Yes	Yes
Unified big data framework	No	No	Yes	Limited
Custom libraries	No	No	Yes	Limited

Conclusion

- We will cover some more detailed examples in Spark in our synchronous time and also over the coming lectures.

- In the meantime, welcome to Spark! Enjoy.