IEMS5730
Spring 2023

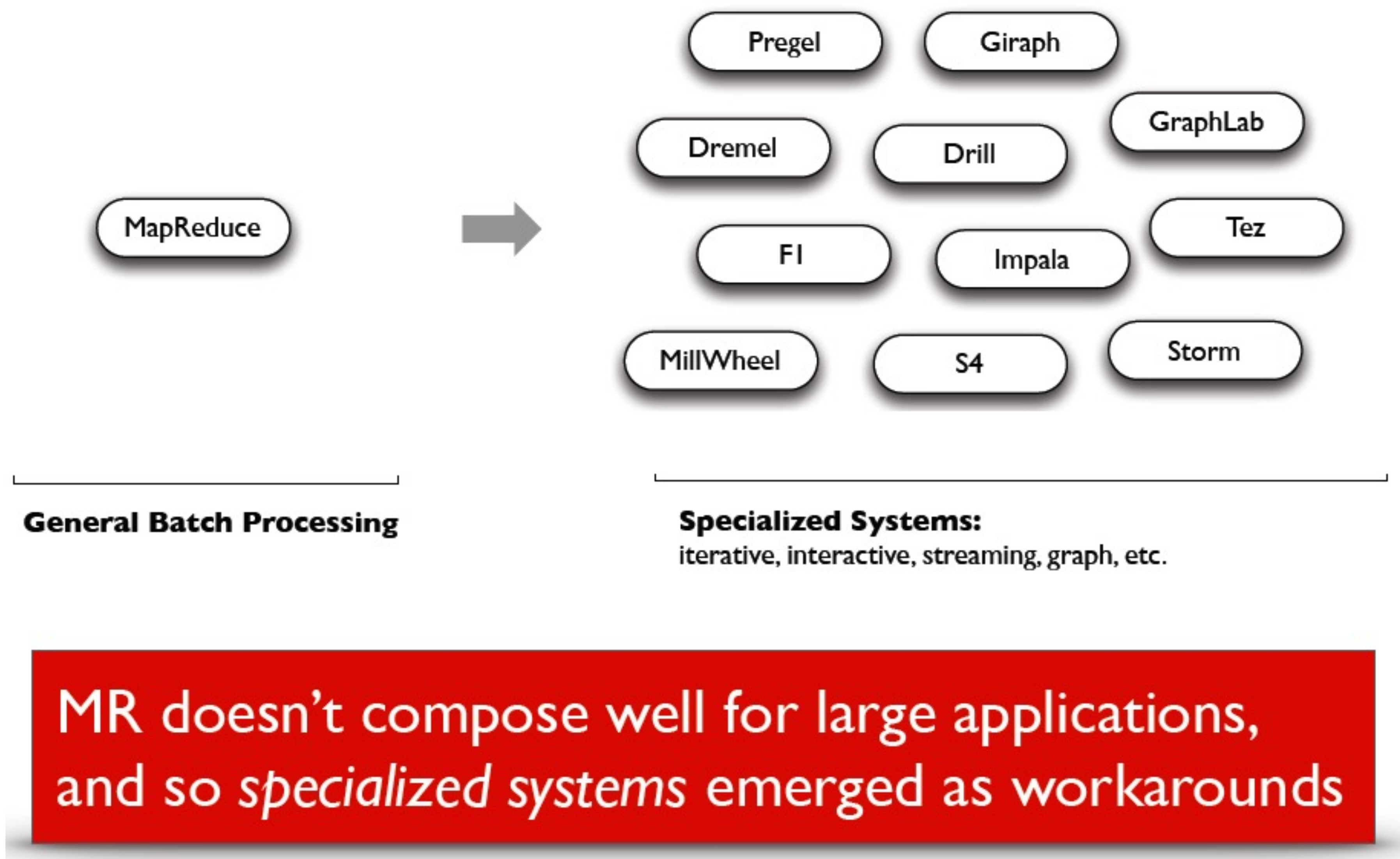
BDAS and Spark

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

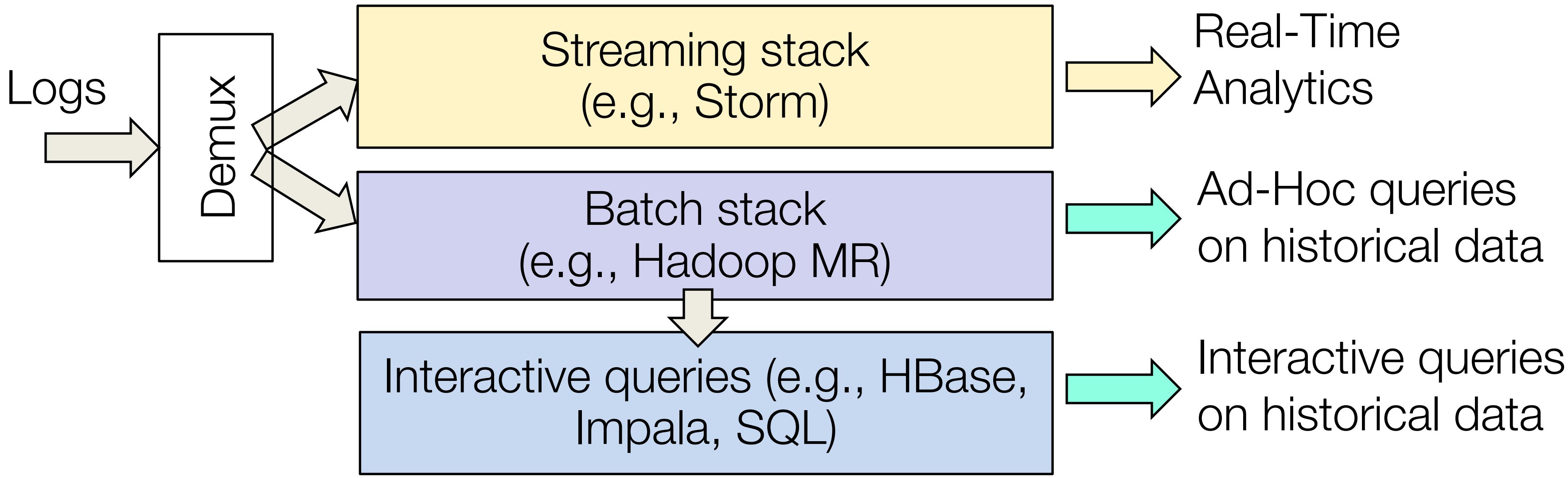
- Slides in this chapter are adapted from the following sources:
  - Matei Zaharia et al, “Spark: In-Memory Cluster Computing for Iterative and Interactive Applications,” UC Berkeley AMPlabs talk, 2011.
  - Matei Zaharia, “Advanced Spark Features,” AMPCAMP talk, 2012.
  - Matei Zaharia, “Parallel Programming with Spark,” Talks for O’Reilly Strata Conference and AMPCAMP, 2013.
  - Reynold Xin, “Spark,” Stanford CS347 Guest Lecture, May 2015.
  - Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, “Learning Spark,” Published by O’Reilly, 2015.
  - Tathagata Das, “Spark Streaming: Large-scale near-real-time stream processing,” O’Reilly Strata Conference talk, 2013.
  - Joseph Gonzalez et al, “GraphX: Graph Analytics on Spark,” talk at AMPCAMP 3, 2013.
  - Ion Stoica, “Intro to AMPLab and Berkeley Data Analytics Stack,” talk at AMPCAMP 3, 2013.
  - Ion Stoica, “State of the BDAS Union,” talk at AMPCAMP 6, Nov. 2015.
  - Paco Nathan, “Intro to Apache Spark,” GOTO; Conference 2015
  - Zhiguang Wen, “Spark: Fast, Interactive, Language-Integrated Cluster Computing,” 2012.
-

A Brief History of MapReduce



The Need for Unification (1/2)

- Big Data Analytics stack BEFORE Spark/BDAS



Challenges:

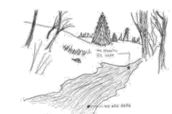
» Need to maintain three separate stacks

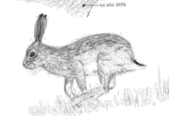
- Expensive and complex
- Hard to compute consistent metrics across stacks

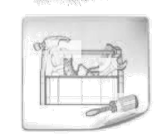
» Hard and slow to share data across stacks

The Need for Unification (2/2)

- Make real-time decisions
  - Detect DDoS, Fraud, etc
- E.g.,: what's needed to detect a DDoS attack?
  1. Detect attack pattern in real time → streaming
  2. Is traffic surge expected? → interactive queries
  3. Making queries fast → pre-computation (batch)

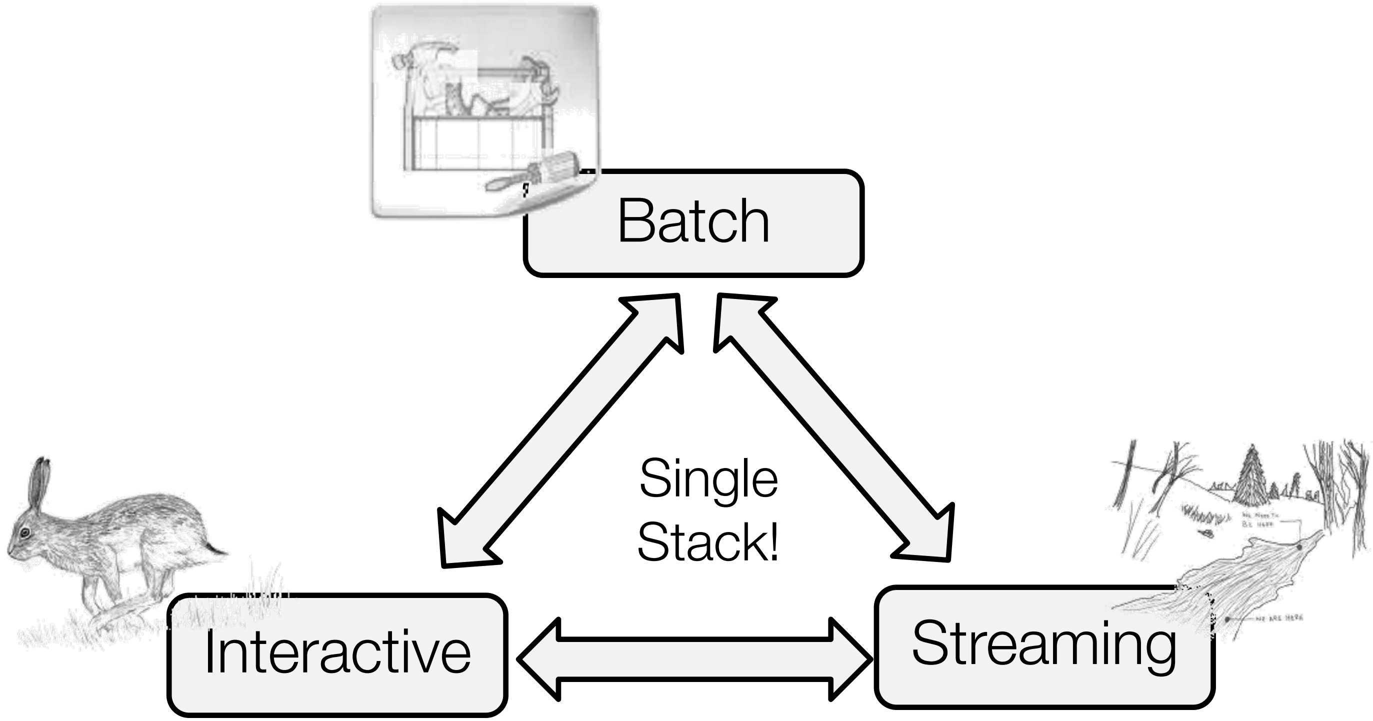






- And need to implement complex algos (e.g., ML)!

Goal of the Berkeley Data Analytics Stack (BDAS) Project by AMPLab @ UCB

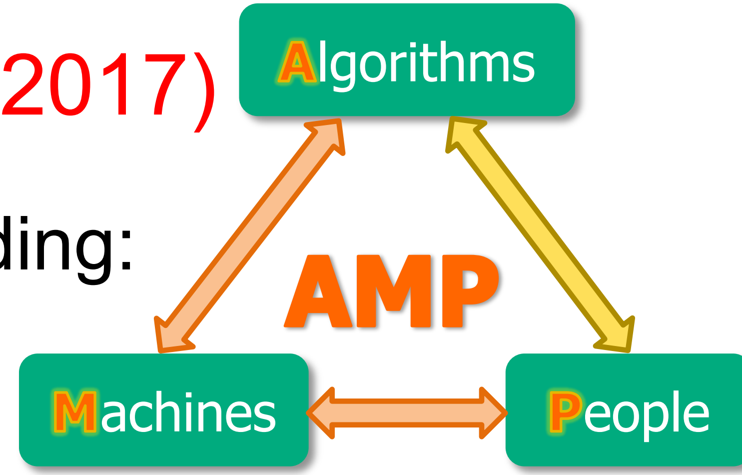


- Support *batch*, *streaming*, and *interactive* computations…

… and make it easy to compose them

- *Easy* to develop *sophisticated* algorithms (e.g., graph, ML algos)

The Berkeley AMPLab (2011-2017)



- Governmental & industrial funding:

Goal: Next generation of open source data analytics stack for industry & academia: Berkeley Data Analytics Stack (BDAS)

A Brief History of Spark

Developed in 2009 at UC Berkeley AMPLab, then open sourced in 2010, Spark has since become one of the largest OSS communities in big data, with over 200 contributors in 50+ organizations

"Organizations that are looking at big data challenges – including collection, ETL, storage, exploration and analytics – should consider Spark for its in-memory performance and the breadth of its model. It supports advanced analytics solutions on Hadoop clusters, including the iterative model required for machine learning and graph analysis."

Gartner, *Advanced Analytics and Data Science* (2014)

spark.apache.org

A Brief History of Spark

circa 2010:

a unified engine for enterprise data workflows, based on commodity hardware a decade later…

amplab UC BERKELEY

Spark: Cluster Computing with Working Sets
Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, Ion Stoica
people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica
usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

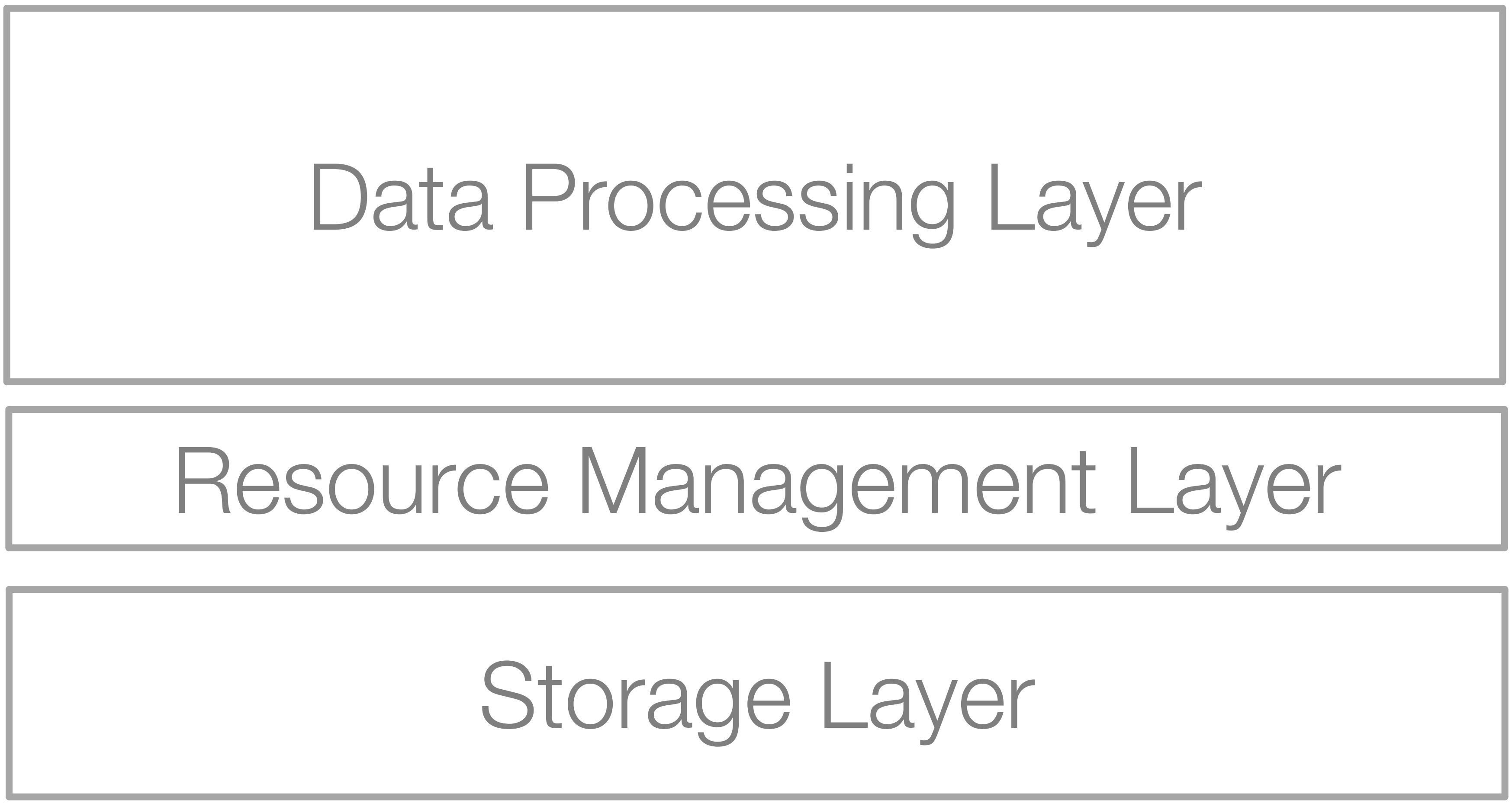
A Brief History of Spark

Unlike the various specialized systems, Spark's goal was to *generalize* MapReduce to support new apps within same engine

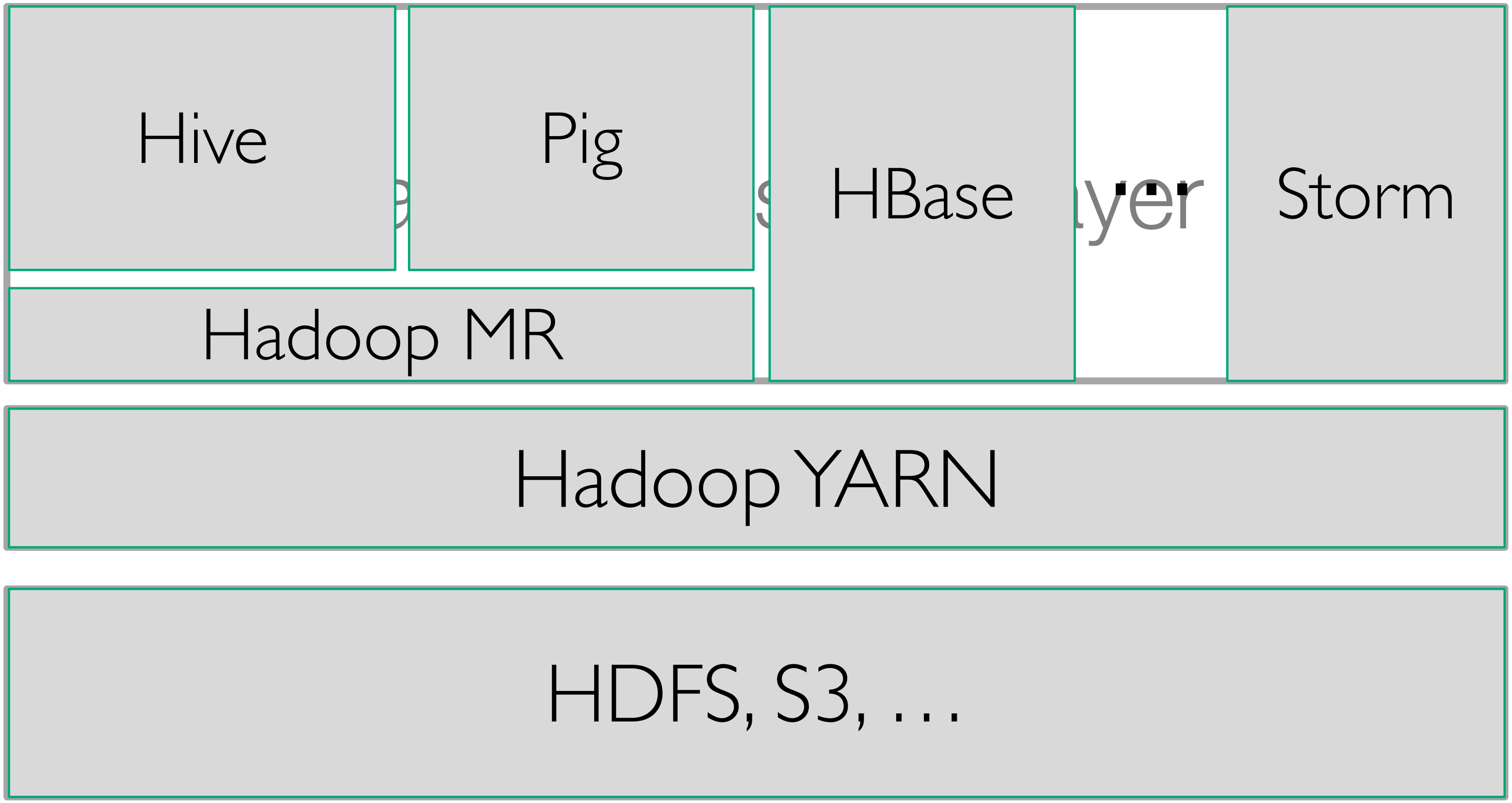
Two reasonably small additions are enough to express the previous models:

- *fast data sharing*
- *general DAGs*

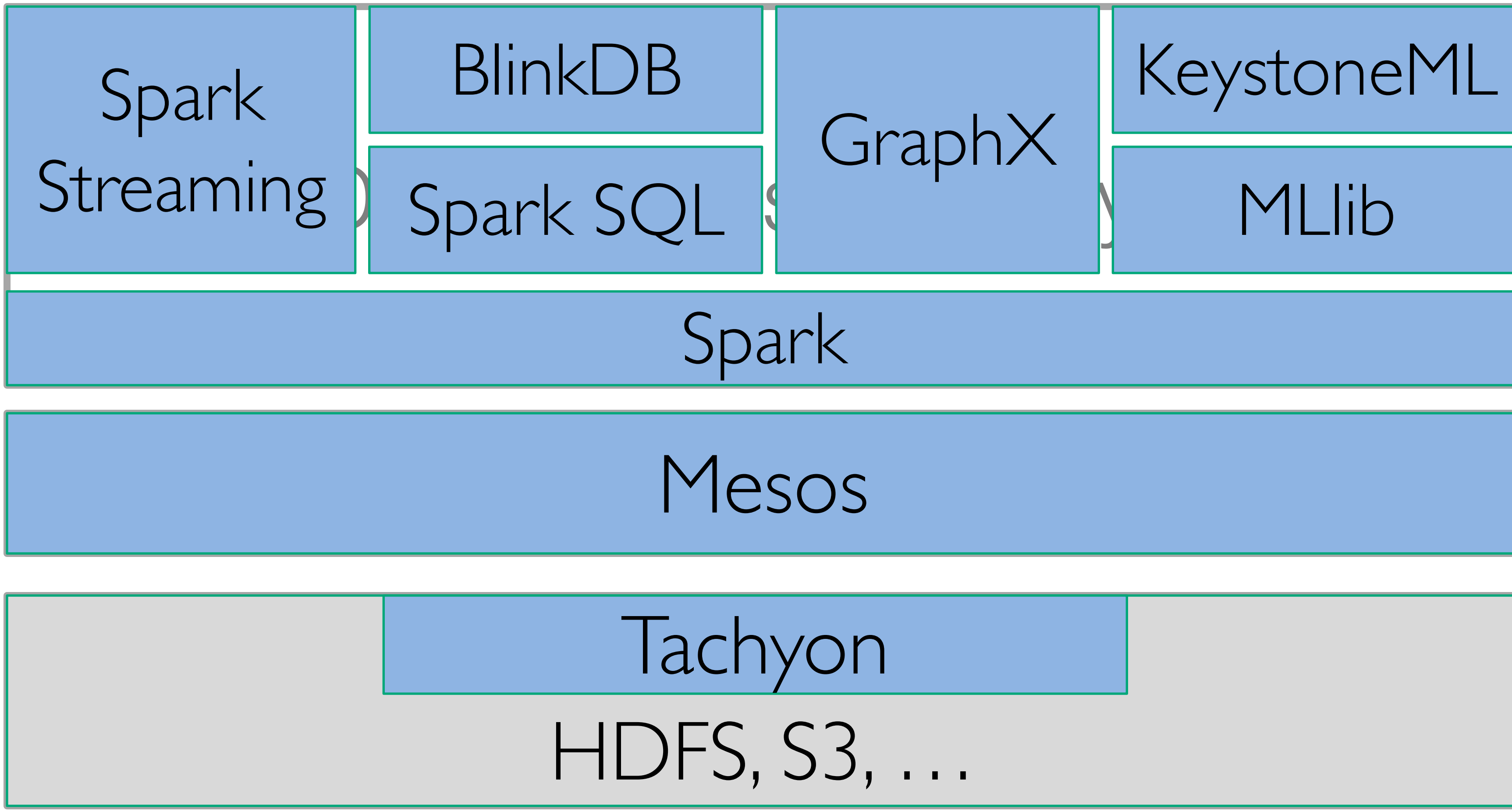
Data Processing Stack



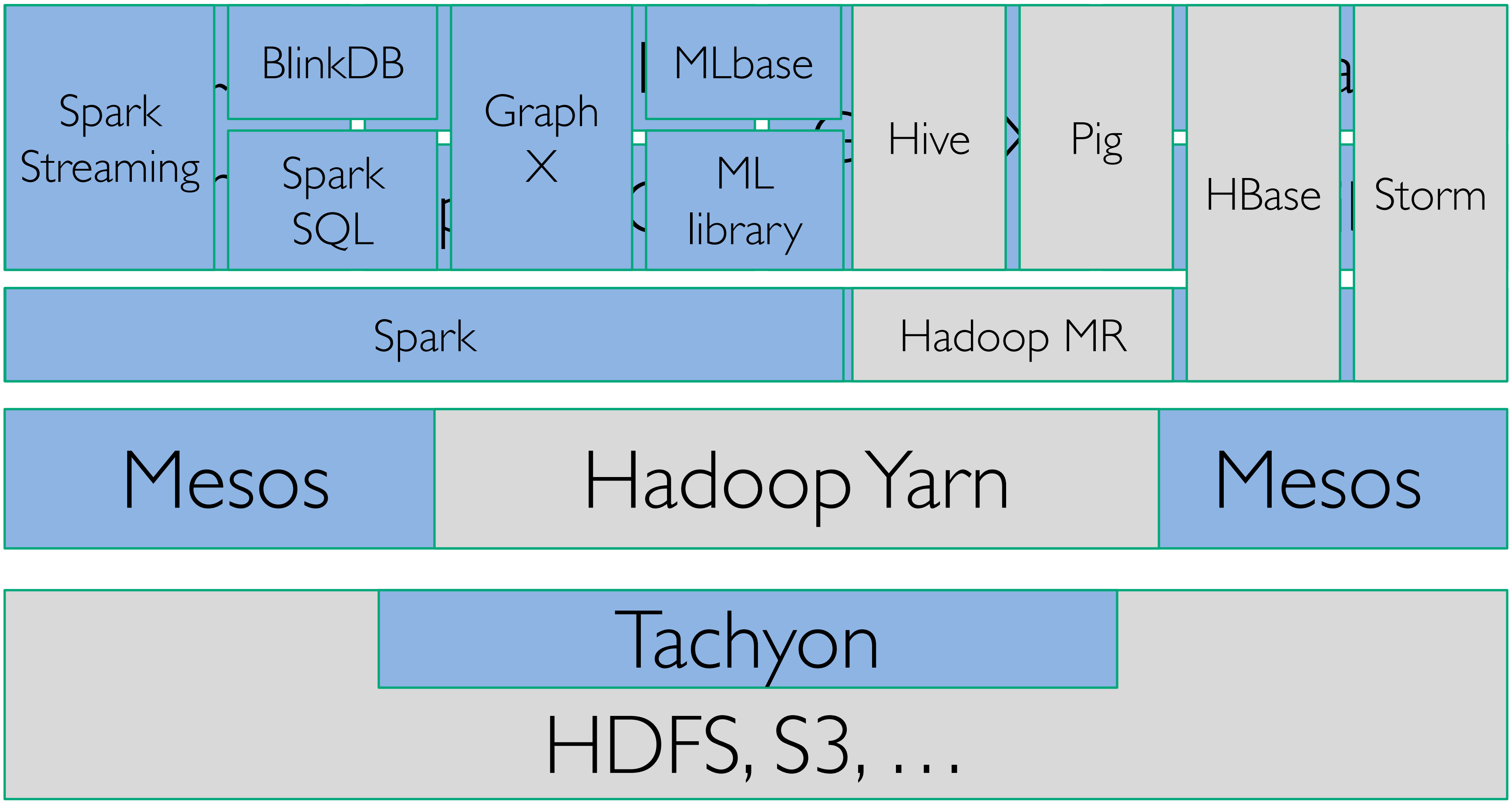
Hadoop Stack



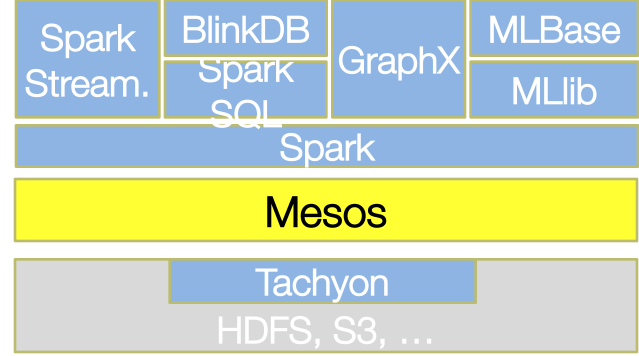
BDAS


Spark Streaming
BlinkDB
Spark SQL
GraphX
KeystoneML
MLlib
Spark
Mesos
Tachyon
HDFS, S3, …

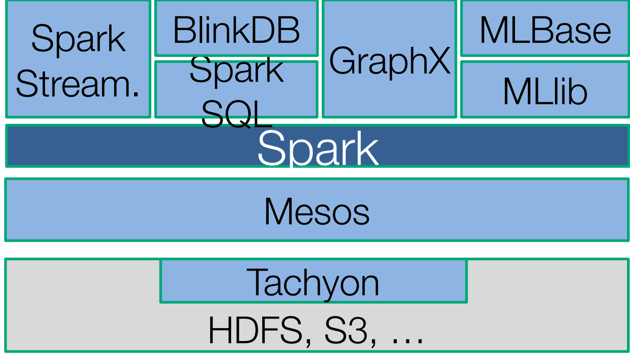
How do BDAS & Hadoop fit together?



Apache Mesos
(http://mesos.apache.org)



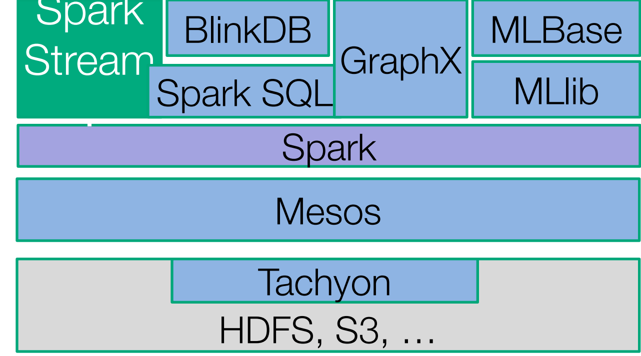
- Another competing Cluster Resource Management software
- Enable multiple frameworks to share same cluster resources (e.g., MapReduce, Storm, Spark, HBase, etc)
- Originated from UCBerkeley's BDAS project ;
  - B. Hindman et al, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", Usenix NSDI 2011.
- Hardened via Twitter's large scale in-house deployment
  - 6,000+ servers,
  - 500+ engineers running jobs on Mesos
- Third party Mesos schedulers
  - AirBnB's Chronos ; Twitter's Aurora
- Mesospehere: startup to commercialize Mesos



Apache Spark

- Distributed Execution Engine
  - Fault-tolerant, efficient in-memory storage (RDDs)
  - Powerful programming model and APIs (Scala, Python, Java)
- **Fast**: up to 100x faster than Hadoop
- **Easy** to use: 5-10x less code than MapReduce
- **General**: support interactive & iterative apps

Spark Streaming



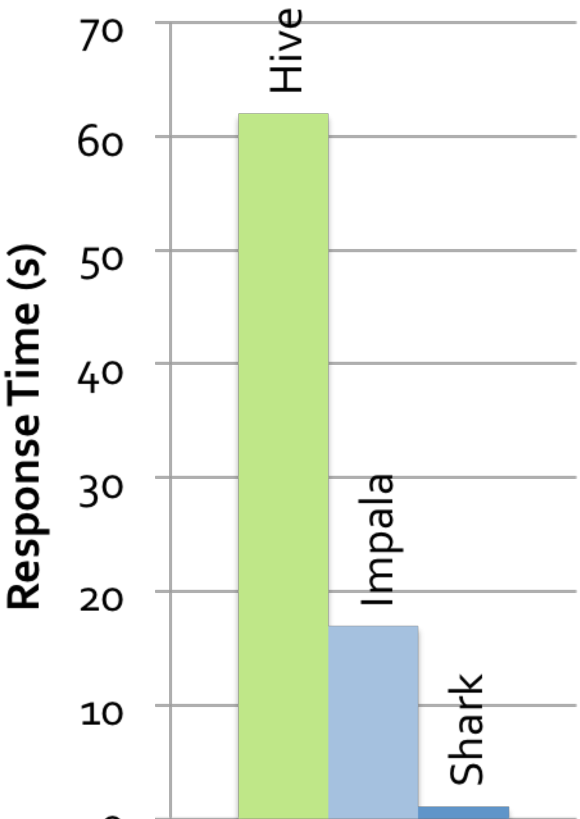
- Large scale streaming computation
- Implement streaming as a sequence of <1s jobs
  - Fault tolerant
  - Handle stragglers
  - Ensure “exactly once” semantics
- Integrated with Spark: unifies batch, interactive, and batch computations
  - Initially, Spark realized streaming in form of “micro-batched” processing and was not truly msec-type “real-time”.
  - Since 2018 (ver2.2), Spark started to support low-latency streaming under the name of “Continuous Processing Mode”.

Unified Programming Models

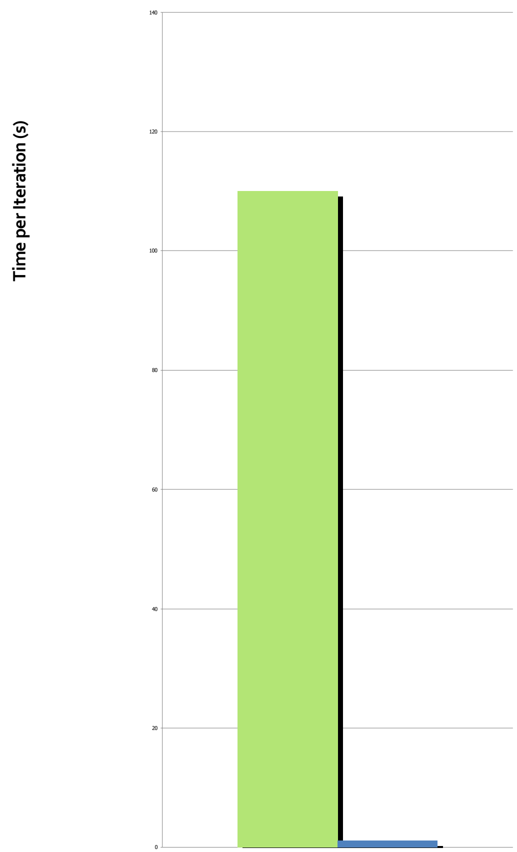
- Unified system for SQL, graph processing, machine learning
- All share the same set of workers and caches

```
def logRegress(points: RDD[Point]): Vector {
  var w = Vector(D, _ => 2 * rand.nextDouble - 1)
  for (i <- 1 to ITERATIONS) {
    val gradient = points.map { p =>
      val denom = 1 + exp(-p.y * (w dot p.x))
      (1 / denom - 1) * p.y * p.x
    }.reduce(_ + _)
    w -= gradient
  }
  w
}

val users = sql2rdd("SELECT * FROM user u
   JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row =>
  new Vector(extractFeature1(row.getInt("age")),
             extractFeature2(row.getStr("country")),
             ...)}
val trainedVector = logRegress(features.cache())
```

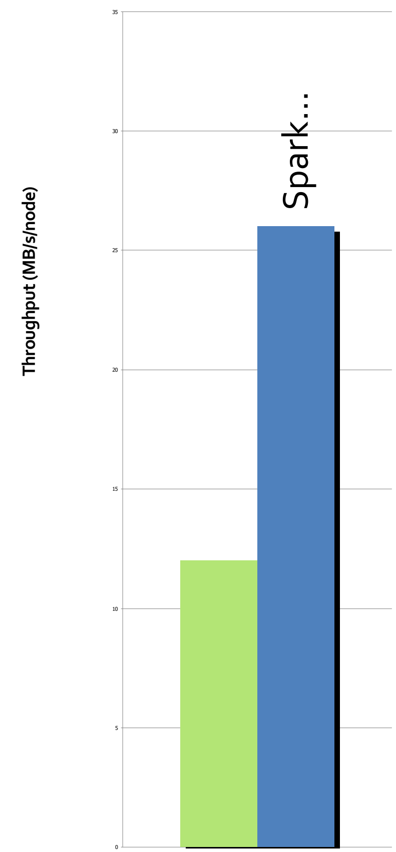
Performance and Generality
(Unified Computation Models)



Interactive
(SQL, Shark)

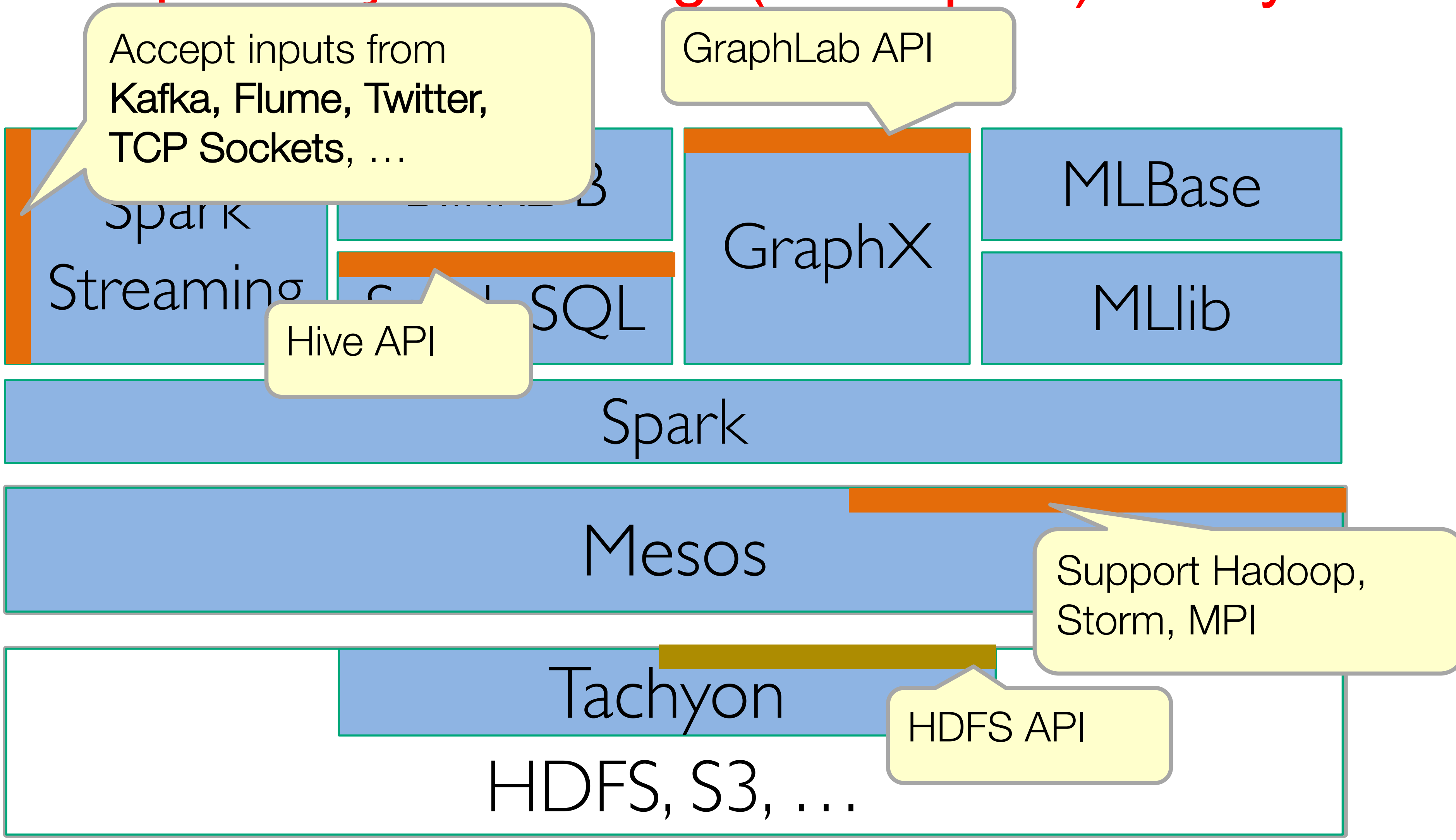


Batch
(ML, Spark)



Streaming
(SparkStreaming)

Compatibility to existing (non-Spark) Ecosystem



Highly Visible Industrial Impact

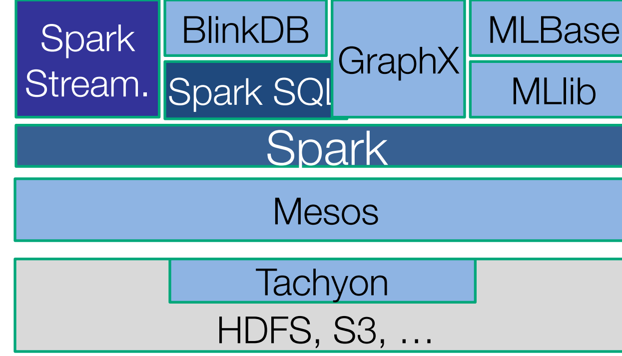
Thousands of companies using BDAS components

Three startups behind BDAS main components

- Mesos — MESOSPHERE
- Spark — databricks
- Tachyon — TACHYON Recently renamed to: ALLUXIO

Rapid Adoption

Spark
Lightning-Fast Cluster Computing



- Train > 10K people via Tutorials in AMPCamp 1-6, Strata, Spark Summits and MOOCs
- 42K+ Spark Meetup members
- 600+ Contributing Developers to codebase

YAHOO! intel webtrends ClearStory DATA AdMobius twitter Adobe CONVIVA bizo TAGGED quantiFind 阿里巴巴 Alibaba.com

Highly Visible Industrial Impact – Large Scale Usage

Largest cluster: 8000 nodes

Largest single job: 1 petabyte

Top streaming intake: 1 TB/hour

2014 on-disk sort record

Spark Ecosystems

Distributions

databricks

Hortonworks MAPR cloudera

IBM Pivotal ORACLE

DATASTAX SAP guavus

bluedata STRATIO

HUAWEI SequoiaDB

mesosphere Typesafe

Applications

tableau MicroStrategy Qlik

elasticsearch. pentaho talend

tresata TRIFACTA SKYTREE

Alpine atscale looker technicolor virdata

FAiMDATA ADATAO DiYOTTA

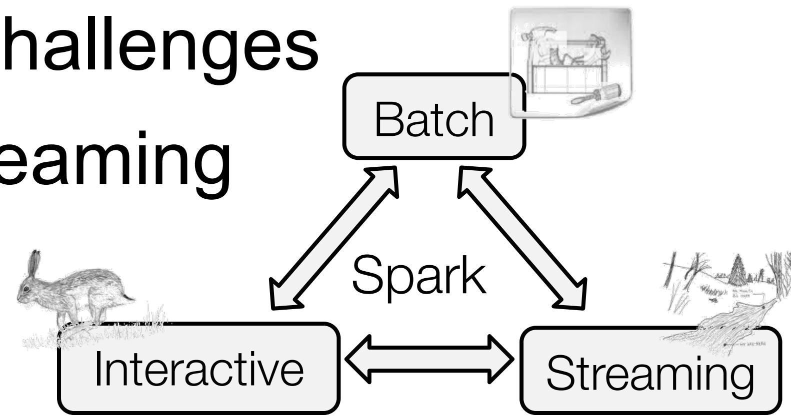
ZOOMDATA platfora APERVI NUBE

Atigeo 日志易 rizhiyi.com ZALONI Typesafe

H2O.ai Ideata lynx analytics

BDAS Summary

- BDAS: address next Big Data challenges
- Unify batch, interactive, and streaming computations



- Facilitate the development of sophisticate applications
  - Support graph & ML algorithms, approximate queries
- Witnessed significant adoption
- Many more additional systems built on the top of (and around) Spark within the BDAS:
  - Spark Streaming, GraphX, KeystoneML, MLbase, Spark SQL, BlinkDB, Tachyon, Succinct…

Key Features of Spark

- handles batch, interactive, and real–time within a single framework
- native integration with Java, Python, Scala
- programming at a higher level of abstraction
- more general: map/reduce is just one set of supported constructs

Programming Language Support by Spark

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Standalone Programs

Python, Scala, & Java

Interactive Shells

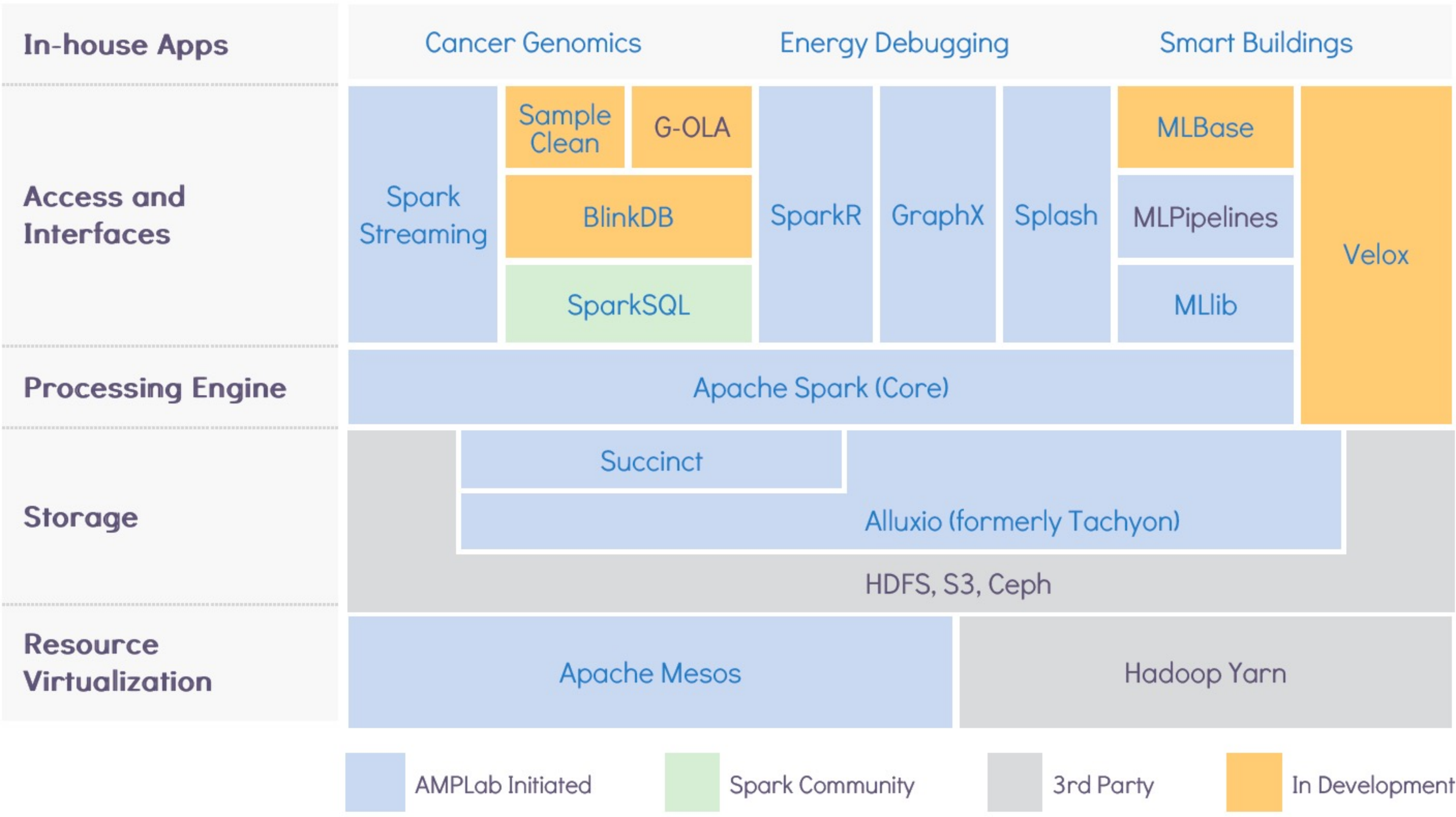
Python & Scala

Performance

Java & Scala are faster due to static typing

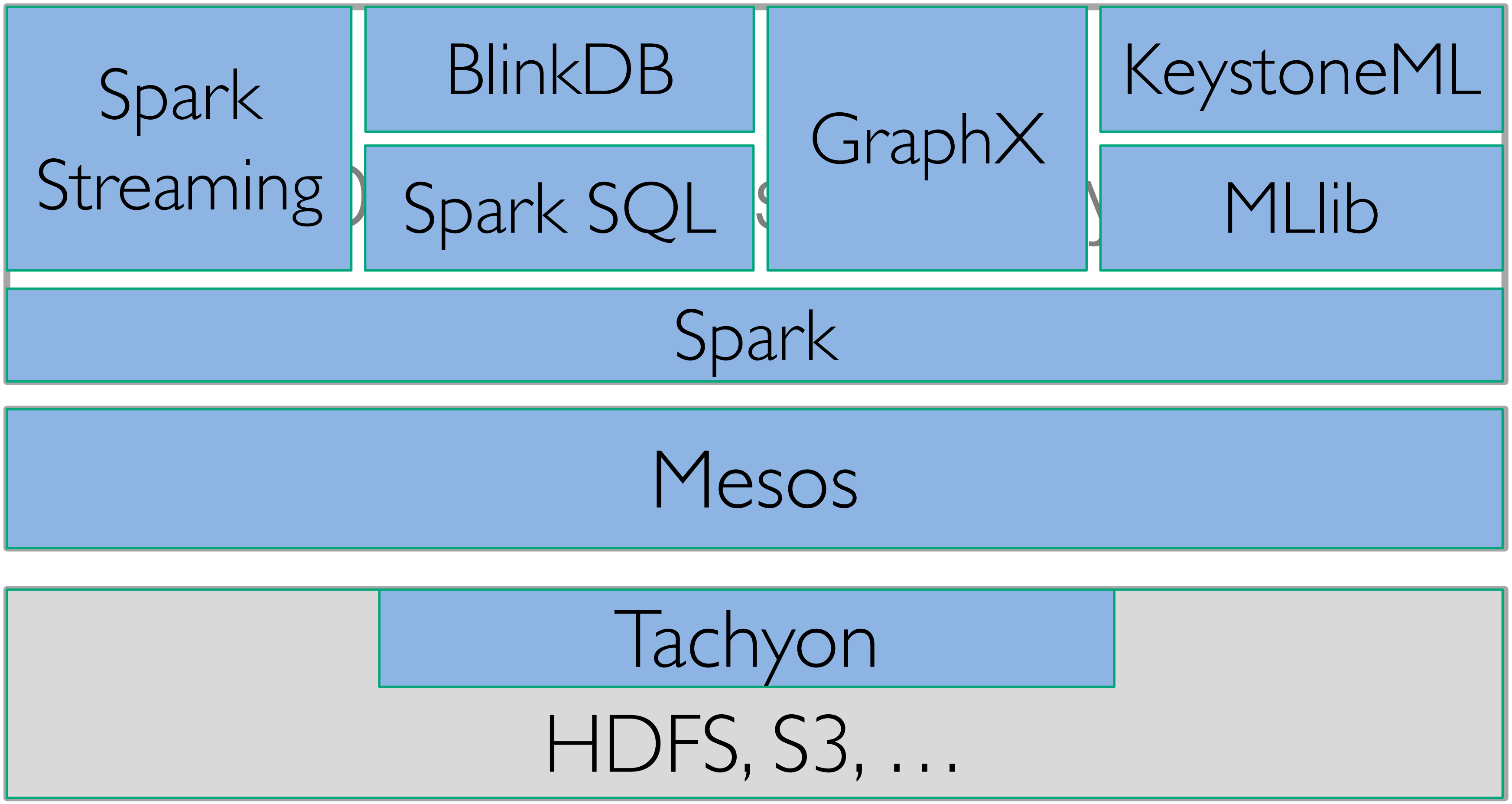
…but Python is often fine

BDAS (since Nov 2016)



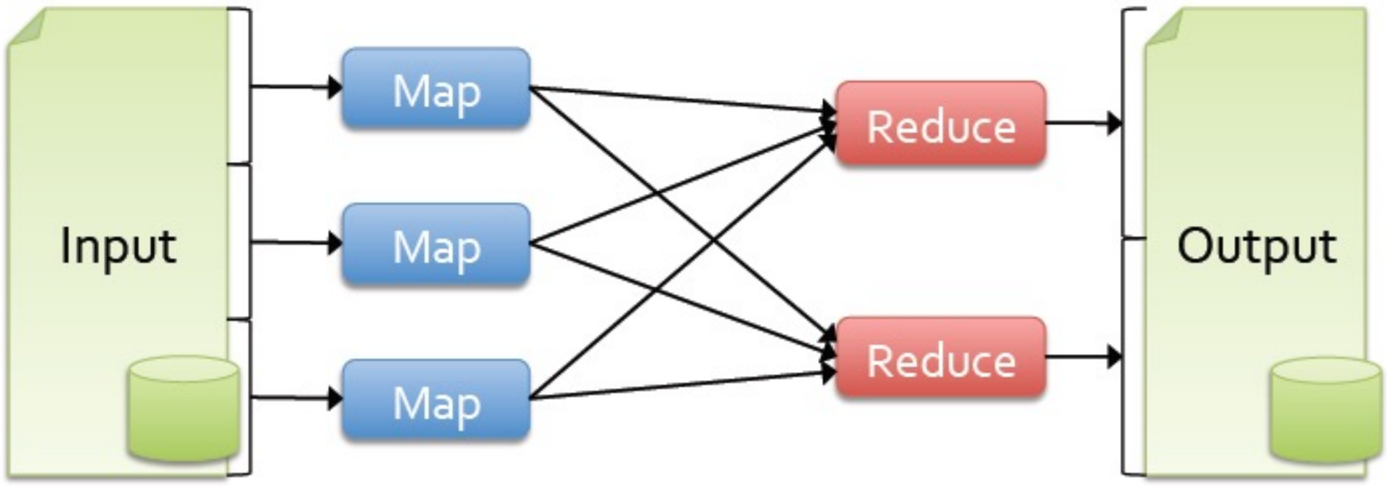
Spark

Spark as the Core Distributed Processing Engine of BDAS



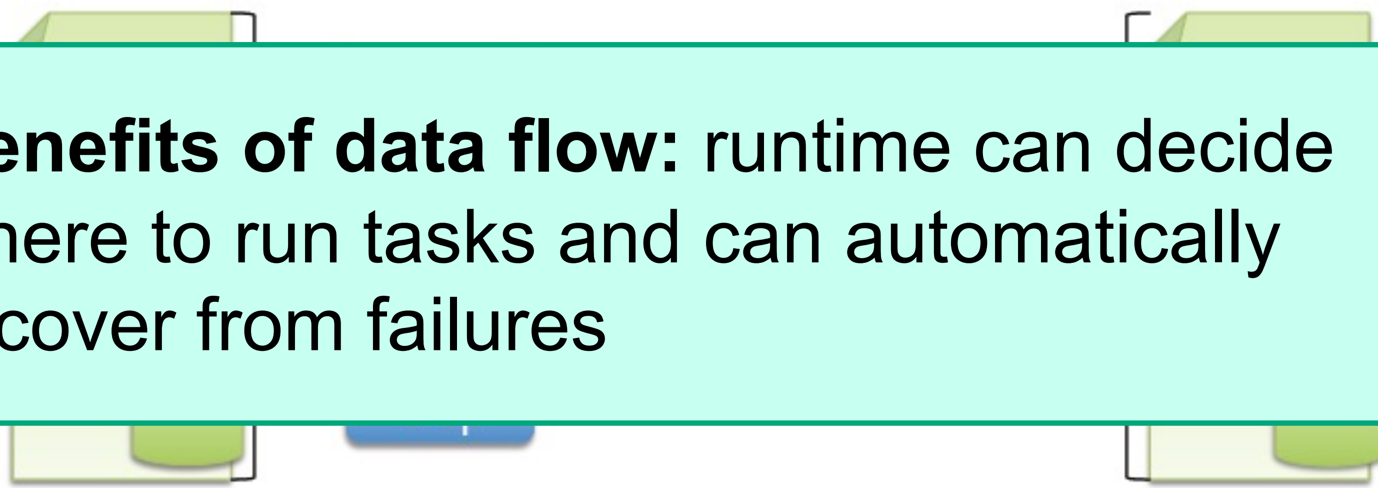
Motivation

Many of the previous cluster programming models are based on directed acyclic data flow from stable storage to stable storage, e.g. MapReduce, Dryad, Tez, SQL



Motivation

Many of the previous cluster programming models are based on directed acyclic data flow from stable storage to stable storage, e.g. MapReduce, Dryad, Tez, SQL



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation (cont'd)

- Although Acyclic data flow is a powerful abstraction, it is NOT efficient for applications that repeatedly reuse a *Working-Set* of data:

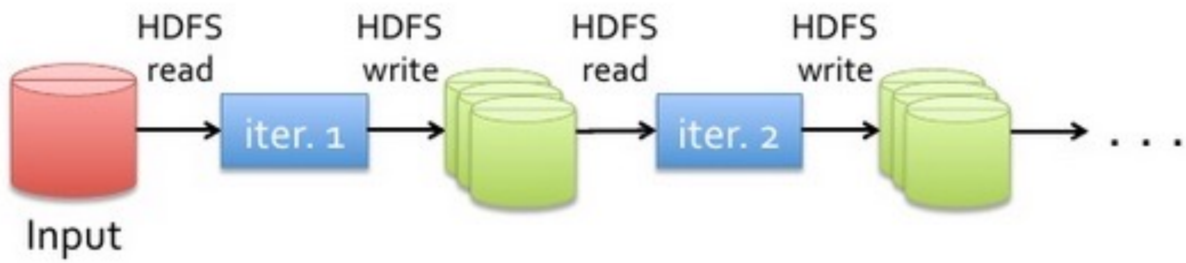
  >> Iterative algorithms (machine learning)

  >> Interactive data mining tools (R, Excel, Python)

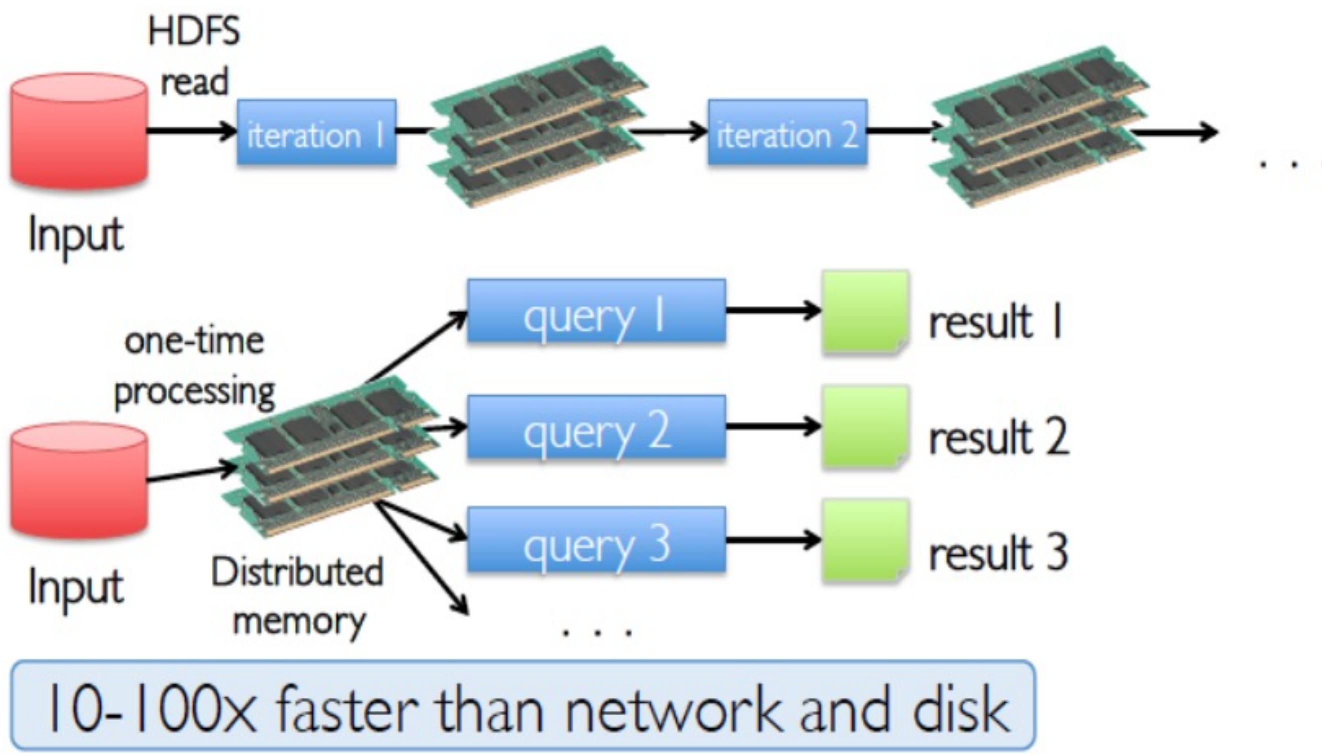
- With previous frameworks, apps reload data from stable storage on each query

Data Sharing

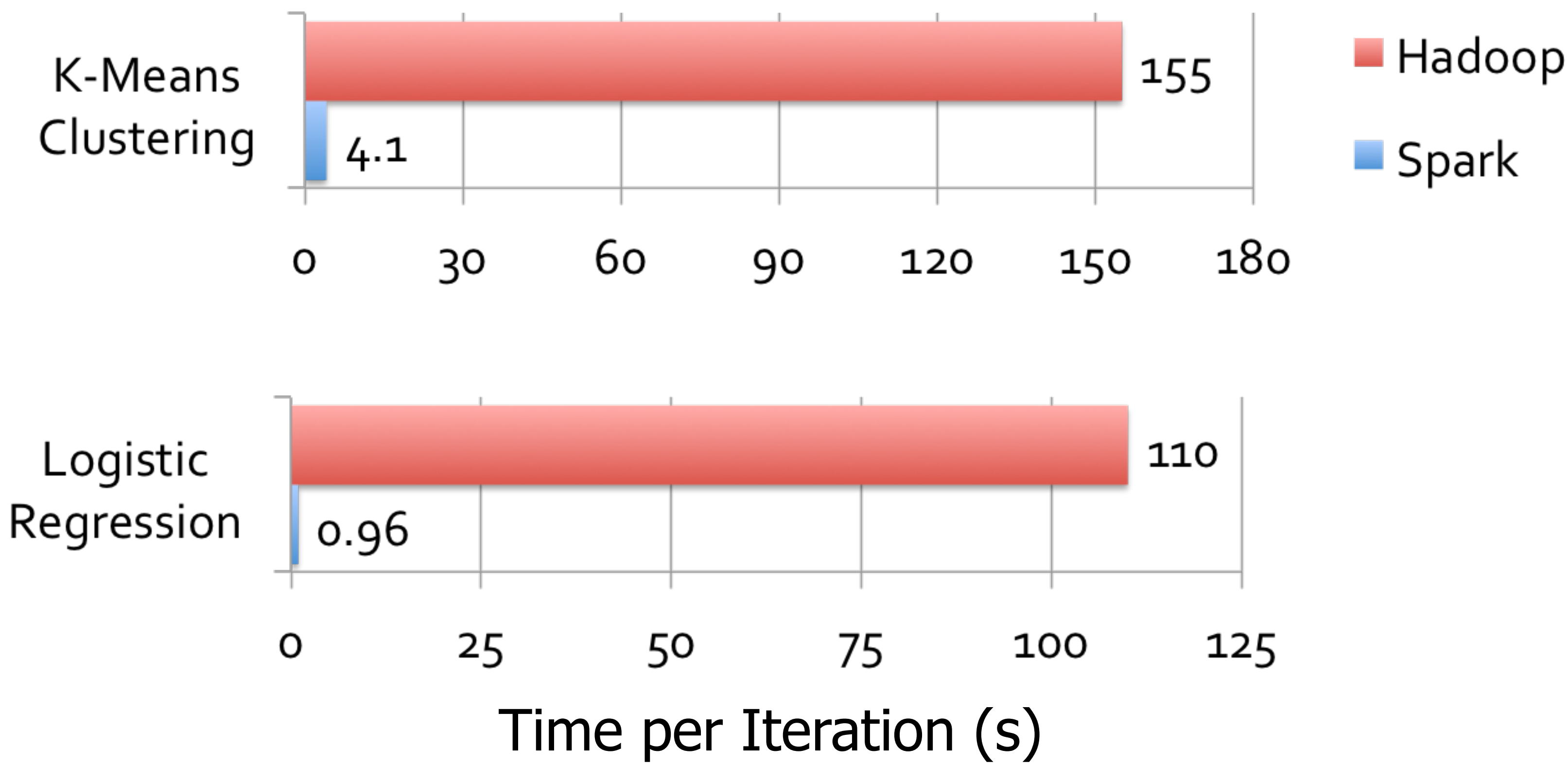
- MapReduce: Sharing via Disk I/O



- Spark: In-memory Sharing (Fast Disk-based sharing as well)

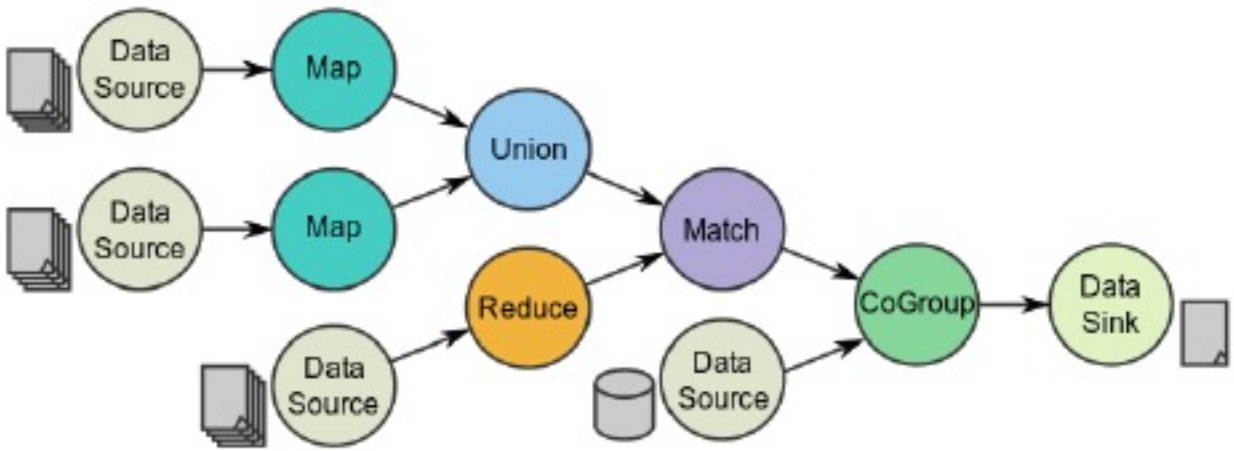


Examples on the Performance Edge of Spark over MapReduce on some common Iterative Algorithms



Key Ideas behind Spark's Solution: Data Flow Model + Resilient Distributed Datasets

- Augment Data Flow model with "Resilient Distributed Datasets" (RDDs)



- Combine Data Flow with RDDs to unify many cluster programming models
  - Instead of specialized APIs for one-type of apps, give users 1st-class control of Distributed Datasets

Key Ideas behind Spark

- Spark makes Working Datasets a first-class concept to efficiently support In-memory Data-Sharing across (different iterations/ stages of) apps
- Provide Distributed Memory Abstractions (called Resilient Distributed Datasets - RDDs) for clusters to support apps with Working Sets
  - Work with distributed collections as you would with local ones
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

Outline

- Introduction to Functional Programming & Scala
- Spark’s Resilient Distributed Datasets (RDDs)
- Implementation
- Conclusion

A Brief History:
Functional Programming for Big Data

Theory, Eight Decades Ago:
what can be computed?

Alonso Church
wikipedia.org

Haskell Curry
haskell.org

Praxis, Four Decades Ago:
algebra for applicative systems

John Backus
acm.org

David Turner
wikipedia.org

Reality, Two Decades Ago:
machine data from web apps

Pattie Maes
MIT Media Lab

amazon.com
ebay
YAHOO!
Google

A Brief History:
Functional Programming for Big Data

circa late 1990s:

explosive growth e-commerce and machine data implied that workloads could not fit on a single computer anymore…

notable firms led the shift to *horizontal scale-out* on clusters of commodity hardware, especially for machine learning use cases at scale

A Brief History: Functional Programming for Big Data

circa 2002:

mitigate risk of large distributed workloads lost due to disk failures on commodity hardware…

Google

Google File System
Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung
research.google.com/archive/gfs.html

MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean, Sanjay Ghemawat
research.google.com/archive/mapreduce.html

Why Functional Programming is a good fit for Parallel, Concurrent, Fault-Tolerant Computing ?

The Root of The Problem

- *Non-determinism* caused by *concurrent threads* accessing *shared mutable* state.
- It helps to encapsulate state in actors or transactions, but the fundamental problem stays the same.
- So,

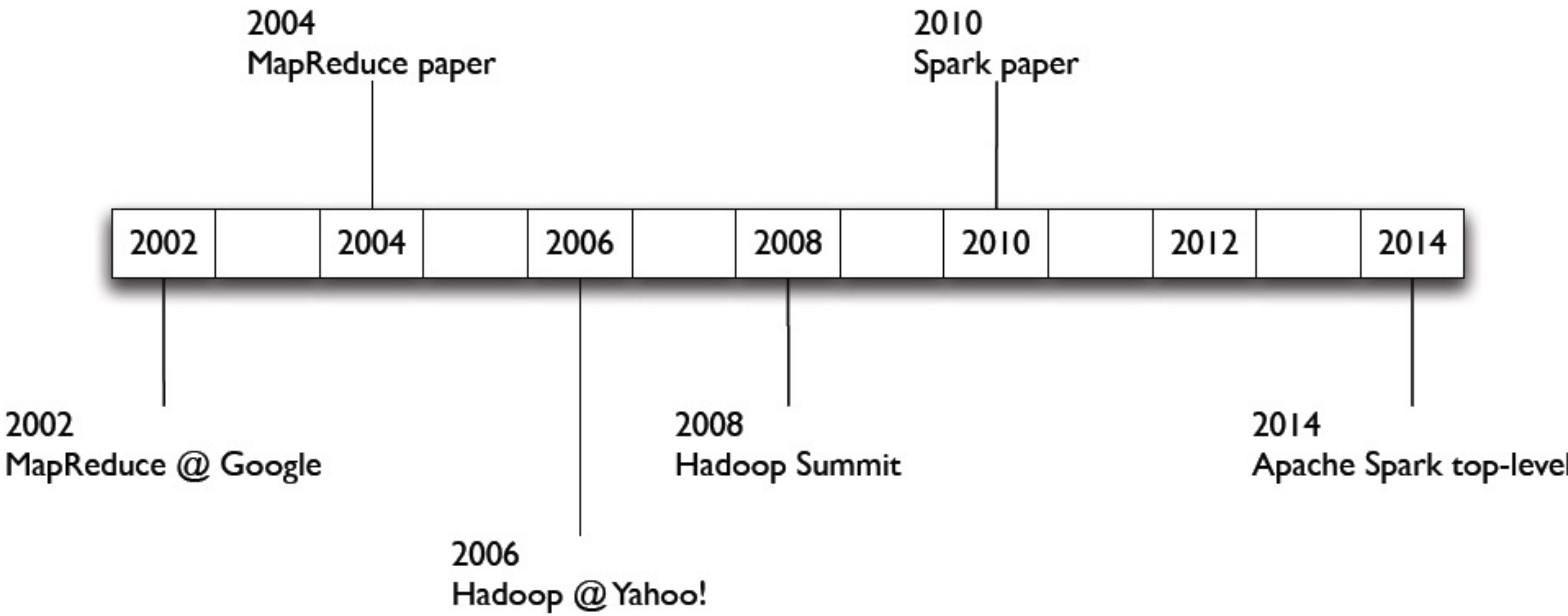
```
var x = 0
async { x = x + 1 }
async { x = x * 2 }

// can give 0, 1, 2
```

non-determinism = *parallel processing* + *mutable state*

- To get deterministic processing, avoid the mutable state!
- Avoiding mutable state means programming *functionally*.

Source: Odersky's OSCON 2011 keynote: https://www.youtube.com/watch?v=3jg1AheF4n0

A Brief History: Functional Programming for Big Data



About Scala

High-level language for JVM

>> Object-Oriented + Functional programming (FP)

>> Designed by Martin Odersky of EPFL in 2001 ;
First public release in 2004.

>> Odersky founded Typesafe in 2011 to provide commercial support of Scala

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

Where to learn more

>>Odersky's Scala course on Coursera:https://www.coursera.org/course/progfun

>>Odersky's OSCON 2011 keynote on why Functional Programming & Parallel-processing is a good fit: https://www.youtube.com/watch?v=3jg1AheF4n0

Quick Tour of Scala

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
}

def announce(text: String) {
  println(text)
}
```

Last expression in block returned

Java equivalent:

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

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

Quick Tour of Scala (cont'd)

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 of Scala (cont'd)

Processing collections with functional programming:

Function expression (closure)

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

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 (cont'd)

```
(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

Scala Cheat Sheet

Variables:

```
var x: Int = 7
var x = 7       // type inferred

val y = "hi"  // read-only
```

Functions:

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

def square(x: Int): Int = {
  x*x   // last line returned
}
```

Collections and closures:

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

nums.map((x: Int) => x + 2) // => Array(3, 4, 5)

nums.map(x => x + 2)  // => same
nums.map(_ + 2)       // => same

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

Java interop:

```
import java.net.URL

new
URL("http://cnn.com").openStre
```

More details:
scala-lang.org

Other Scala Collection Methods

More details:
scala-lang.org

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

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

Outline

- Introduction to Functional programming & Scala
- Spark's Resilient Distributed Datasets (RDDs)
- Implementation
- Conclusion

Key Ideas behind Spark

- Spark makes Working Datasets a first-class concept to efficiently support In-memory Data-Sharing across (different iterations/ stages of) apps
- Provide Distributed Memory Abstractions (called Resilient Distributed Datasets - RDDs) for clusters to support apps with Working Sets
  - Work with distributed collections as you would with local ones
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

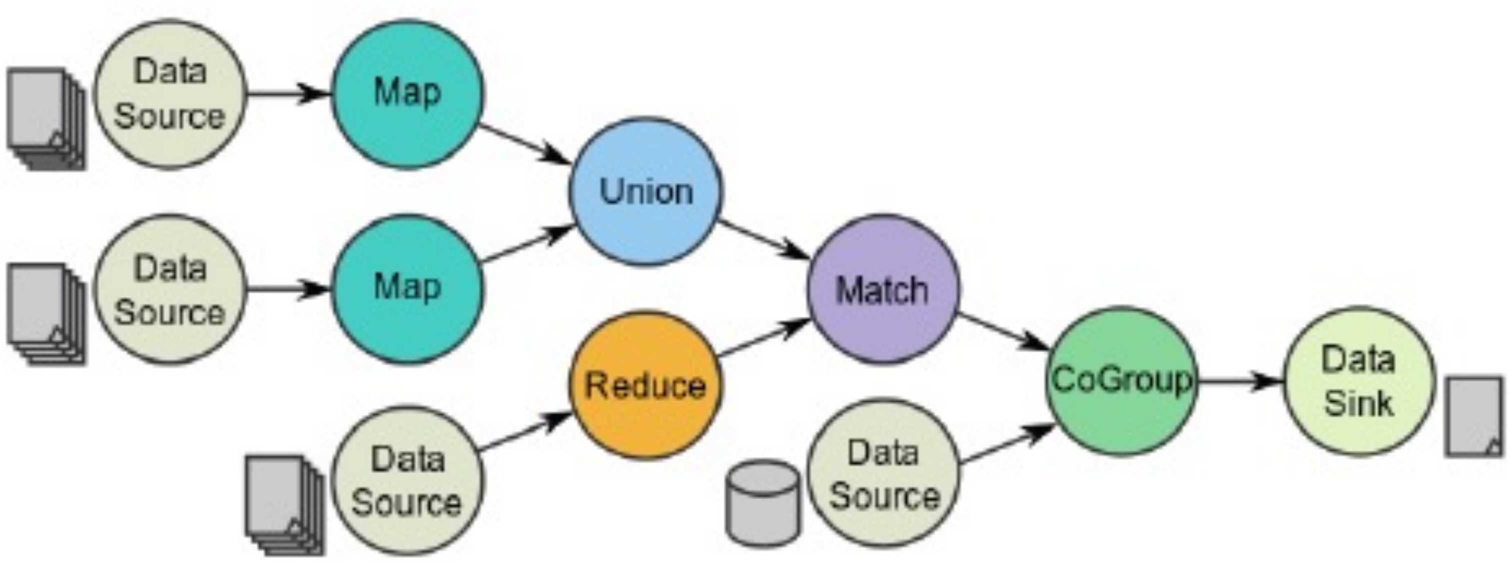
What are Resilient Distributed Datasets (RDDs) ?

- RDDs are Immutable (i.e. become read-only once they are created) collections partitioned across cluster that can be rebuilt if a partition is lost
- Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
- The elements of an RDD need not exist in physical storage;
  - Instead, a handle to an RDD contains enough information (aka lineage info) to compute the RDD starting from data in reliable storage.

=>RDDs can always be reconstructed if nodes fail.

Reap Key Ideas behind Spark's Solution: Data Flow Model + Resilient Distributed Datasets

- Augment Data Flow model with "Resilient Distributed Datasets" (RDDs)



What are RDDs (cont’d) ?

- RDDs that can be *cached* (aka *persist*) in RAM across parallel operations and to be shared by different Apps
- User can control the *Partitioning* of an RDD, e.g .one comprised of <key,value> pairs based on hash or range of the key.
  - Once partitioned, Spark will remember the way an RDD is partitioned and use the info to reduce unnecessary data shuffling when operating on RDDs
    - e.g. Functions that benefit from partitioning include: cogroup( ), groupWith( ), join( ) , groupByKey( ), reduceByKey( ), combineByKey( ), lookup( )
  - Spark knows internally which operations may affect partitioning, and will automatically set the partitioner of an RDD

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)
```

Programming Model of Spark

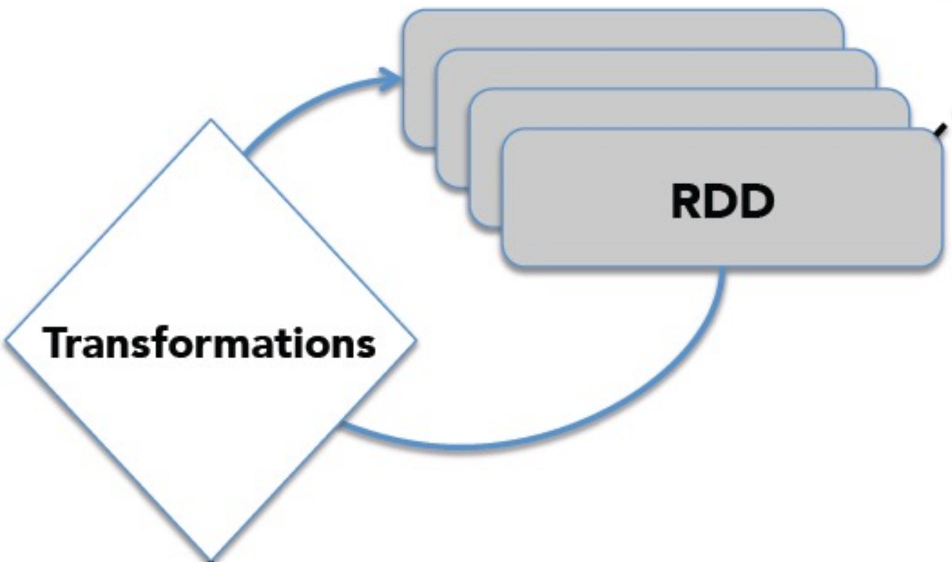
- Use Resilient Distributed Datasets (RDDs) as basic building blocks
- Perform Parallel Operations on RDDs
  - Transformations: Operations to create new RDD(s) from existing ones, e.g. map, filter, groupBy, join ;
  - Actions: Return a result (value) to a driver program after running the computation on the RDD or write it to storage, e.g. reduce, collect, count, save …
  - Transformations are *Lazy* (They don't compute right away):
    - Spark just remembers the transformations applied to datasets(lineage). Only compute when an action requires.
- Restricted Shared Variables
  - Accumulators, Broadcast variables

Working with RDDs

```
textFile = sc.textFile("SomeFile.txt")
```

Working with RDDs

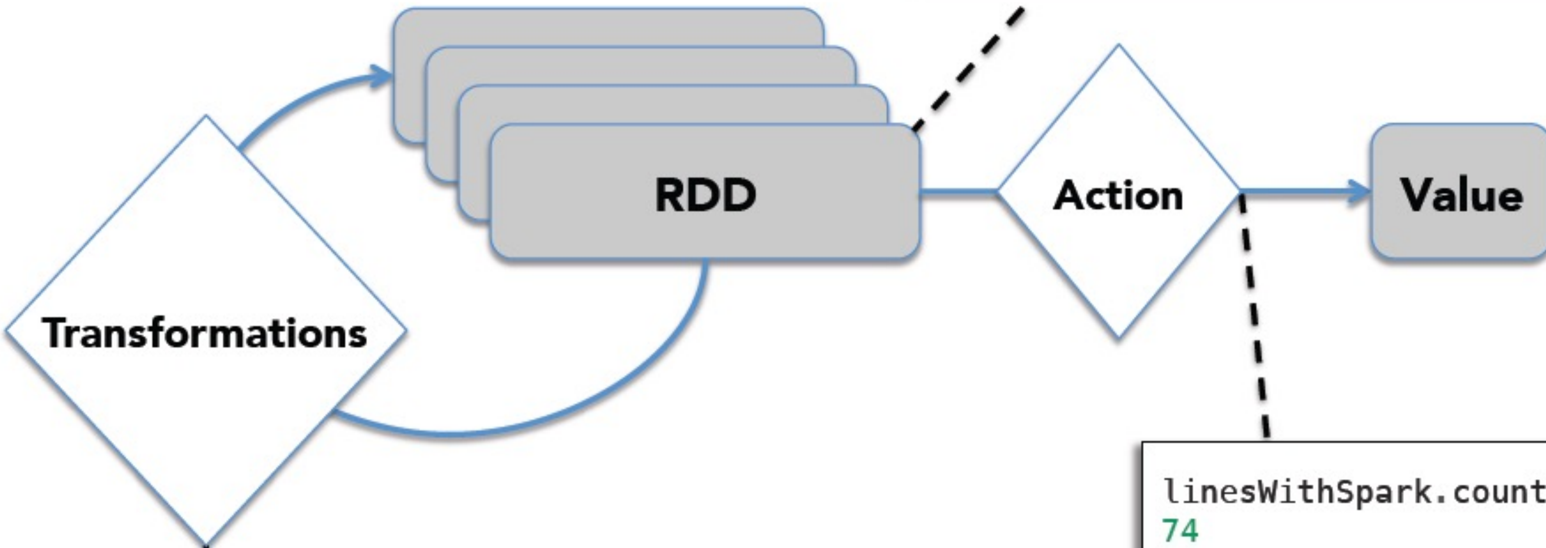
```
textFile = sc.textFile("SomeFile.txt")
```




```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

Working with RDDs

```
textFile = sc.textFile("SomeFile.txt")
```




```
linesWithSpark.count()
74

linesWithSpark.first()
# Apache Spark
```

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

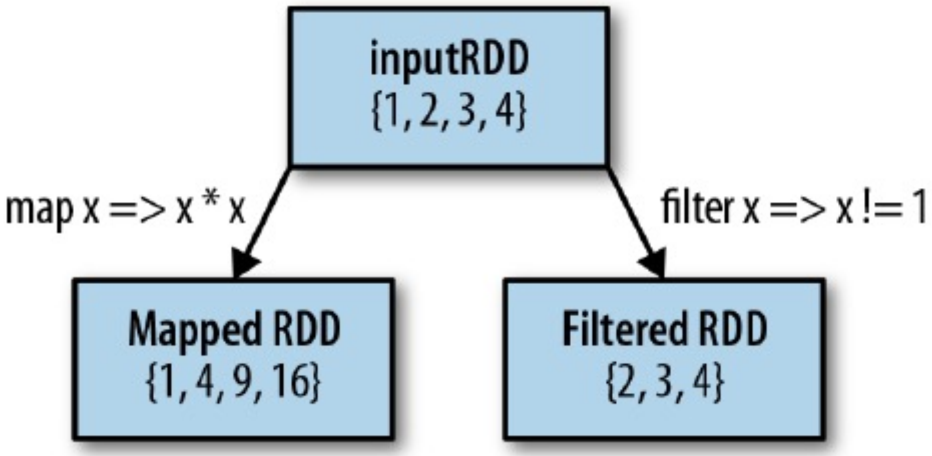
Transformations

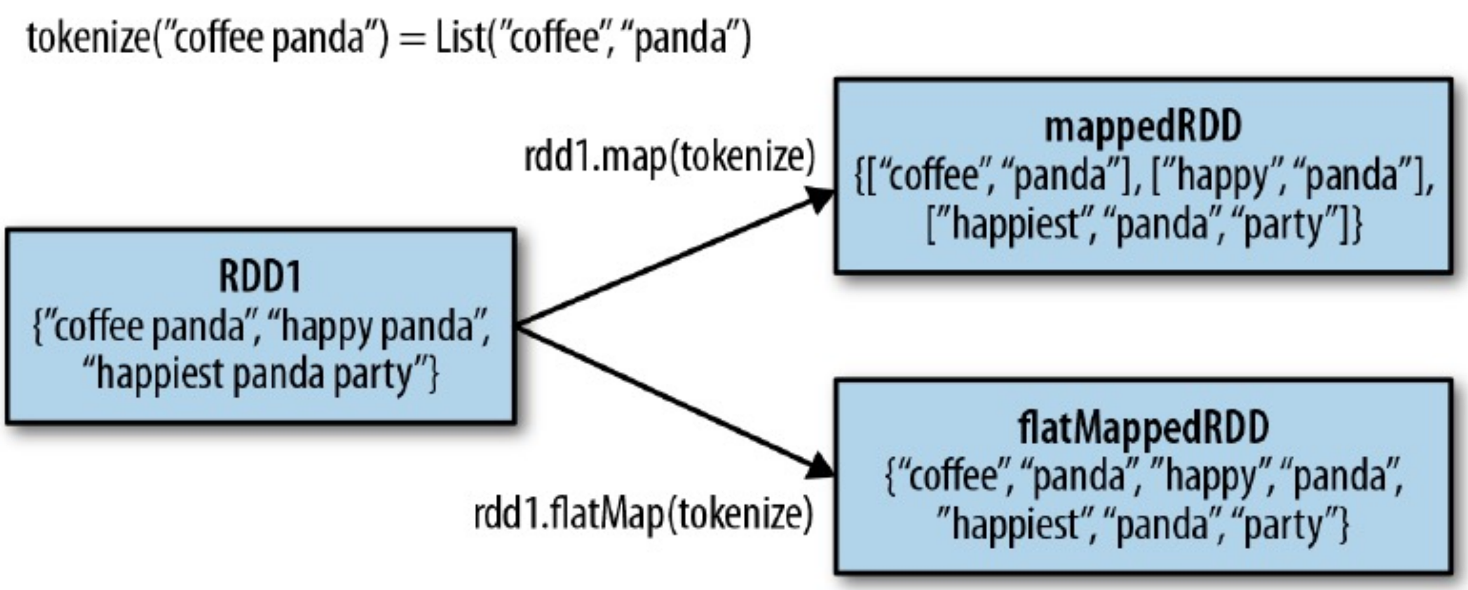
transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 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 dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Transformations (cont'd)

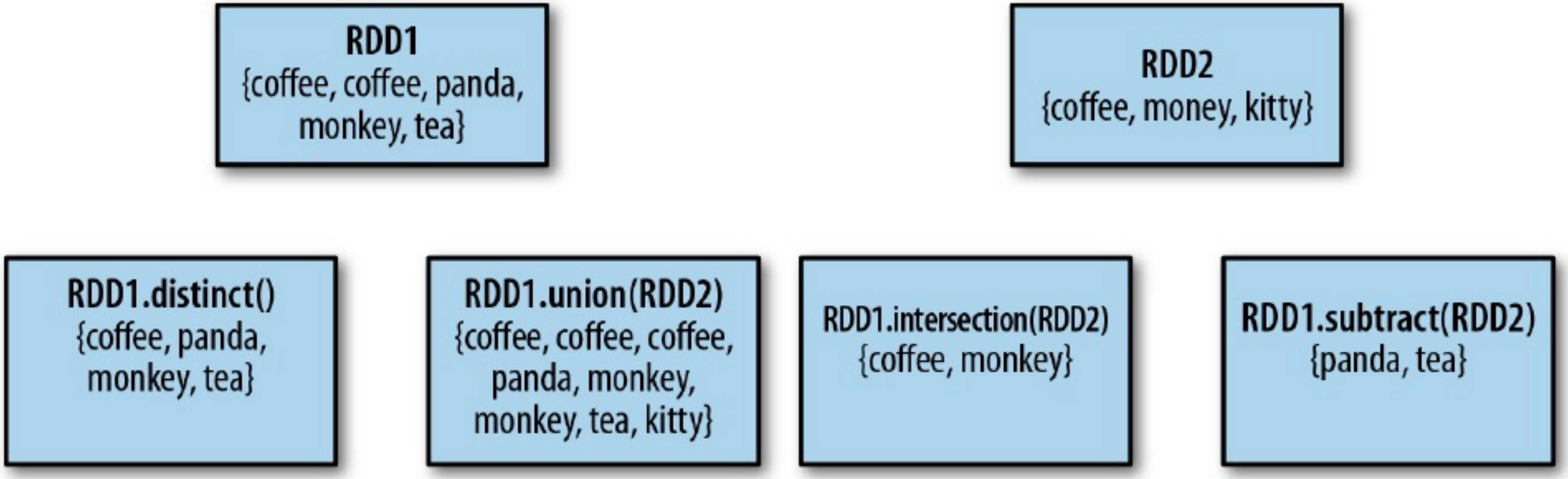
transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset 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 datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset*)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Transformations Examples

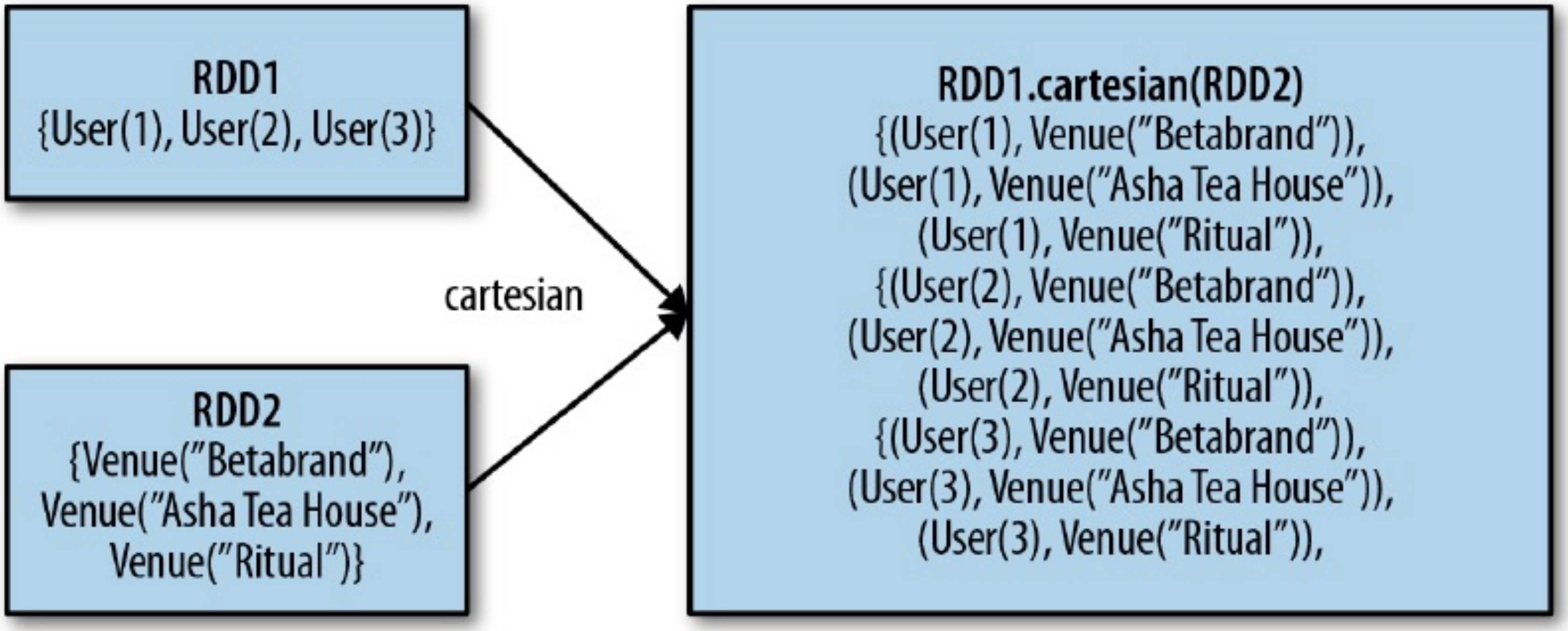




Examples on Set Operations



Examples on Cartesian product b/w two RDDs



More Examples Basic RDD Transformations

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

More Examples Basic RDD Transformations (cont’d)

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

More Transformations Example

Scala:

```
val distFile = sqlContext.table("readme").map(_(0).asInstanceOf[String])
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

distFile is a collection of lines

Python:

```
distFile = sqlContext.table("readme").map(lambda x: x[0])
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

More Transformations Example

Scala:

```
val distFile = sqlContext.table("readme").map(_(0).asInstanceOf[String])
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

closures

Python:

```
distFile = sqlContext.table("readme").map(lambda x: x[0])
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

Actions

action	*description*
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset 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 dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement*, *fraction*, *seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Actions (cont'd)

action	*description*
`saveAsTextFile(`*`path`*`)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`*`)`	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available 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()`	only available 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 dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Examples of Actions on RDDs

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`

More Examples of Actions on RDDs

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

More Action Examples

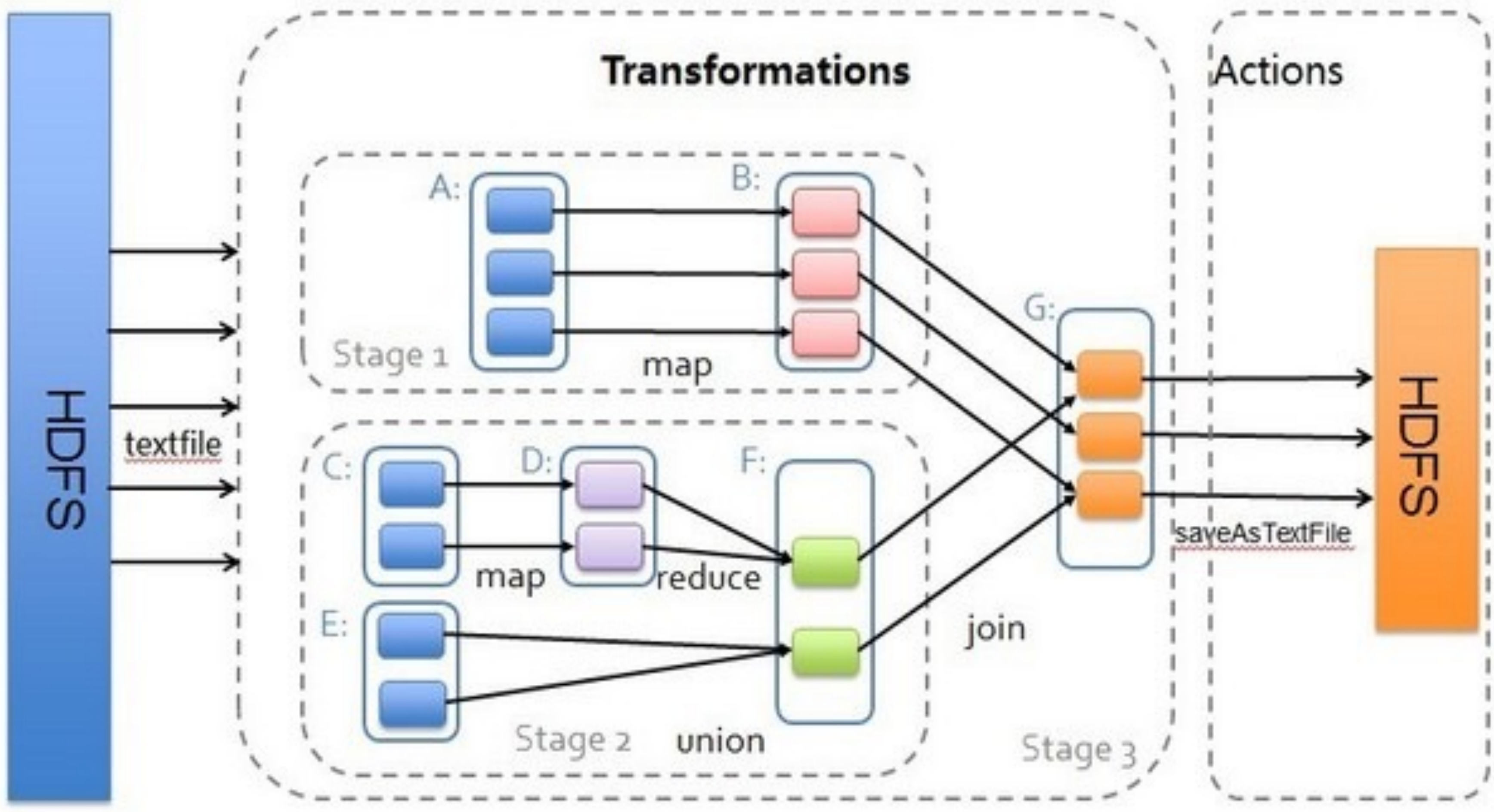
Scala:

```
val f = sqlContext.table("readme").map(_(0).asInstanceOf[String])
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```
from operator import add
f = sqlContext.table("readme").map(lambda x: x[0])
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

Transformations & Actions



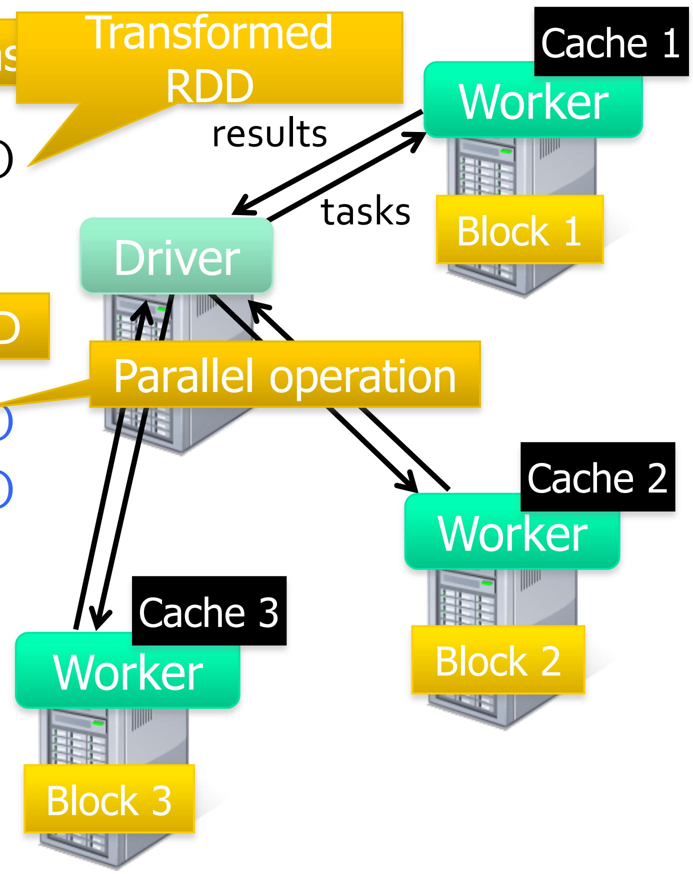
Parallel Operations

- reduce: Combines dataset elements using an associative function to produce a result at the driver program.
- collect: Sends all elements of the dataset to the driver program.

Example: Log Mining w/ Spark in Scala

- 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 <0.5 sec (vs 20 sec for on-disk data)

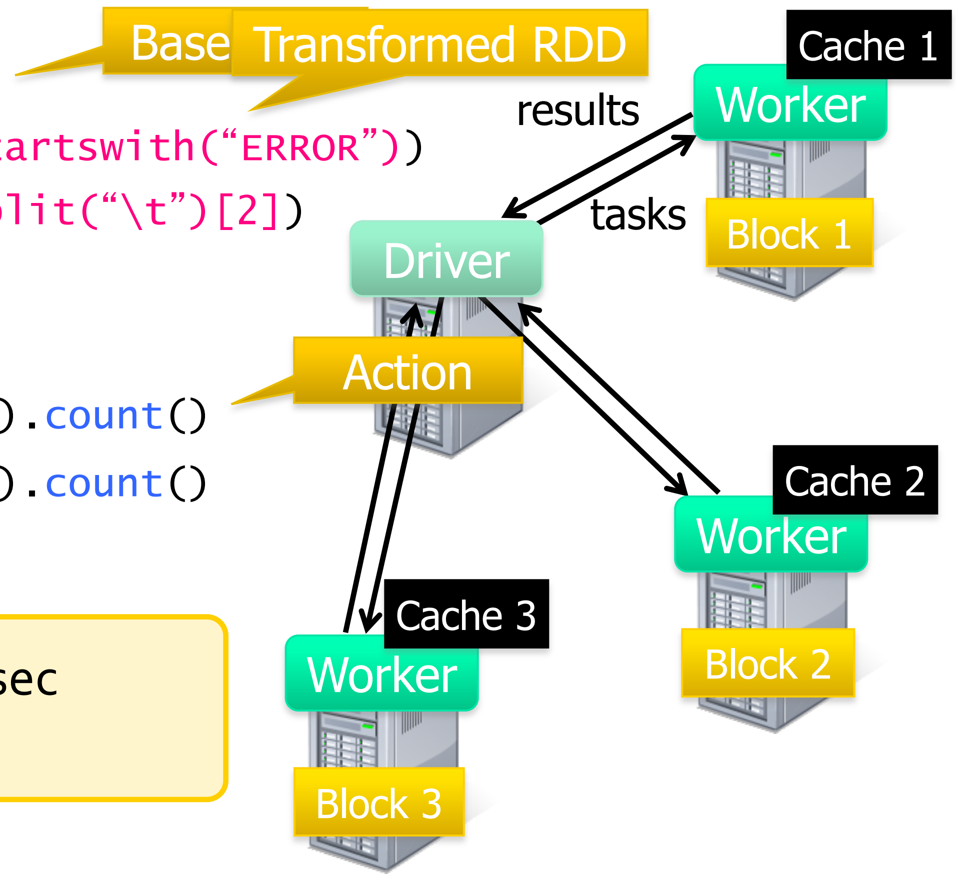
Spark in Scala and Java

```
// Scala:

val lines = sc.textFile(...)
lines.filter(x => x.contains(“ERROR”)).count()

//the line above is the long form of:

// lines.filter(_.contains(“ERROR”)).count()


// Java:

JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains(“error”);
  }
}).count();
```

Same Example in Python

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```



Result: scaled to 1 TB data in 5 sec
(vs 180 sec for on-disk data)

Working with Key-Value Pairs

- Spark's "distributed reduce" transformations operate on RDDs of key-value pairs
- Python:

```python
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

- Scala:

```scala
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

- Java:

```java
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

Examples of Transformations on Pair RDDs

Table 4-1. Transformations on one pair RDD (example: {(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])}`
`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, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`

More Examples of Transformations on Pair RDDs

Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`
`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.		

More Examples of Transformations on Pair RDDs

Table 4-2. Transformations on 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, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

See https://www.tutorialspoint.com/scala/scala_options.htm for more details on Some( )

Example of using combineByKey to compute Per-key averaging for Pair RDDs in Python or Scala

Example 4-12. Per-key average using combineByKey() in Python

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)),
                             (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

key	value
panda	0
pink	3
pirate	3
panda	1
pink	4

Example 4-13. Per-key average using combineByKey() in Scala

```
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
  ).map{ case (key, value) => (key, value._1 / value._2.toFloat) }
  result.collectAsMap().map(println(_))
```

Examples of combineByKey for Pair RDDs in Java

Example 4-14. Per-key average using combineByKey() in Java

```
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {   total_ = total;   num_ = num; }
  public int total_;
  public int num_;
  public float avg() {   return total_ / (float) num_; }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, AvgCount>() {
  public AvgCount call(Integer x) {
    return new AvgCount(x, 1);
  }
};
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
  public AvgCount call(AvgCount a, Integer x) {
    a.total_ += x;
    a.num_ += 1;
    return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total_ += b.total_;
    a.num_ += b.num_;
    return a;
  }
};
AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
  nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
  System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}
```

Examples of Filtering on Values of a Pair-RDD

Example 4-4. Simple filter on second element in Python

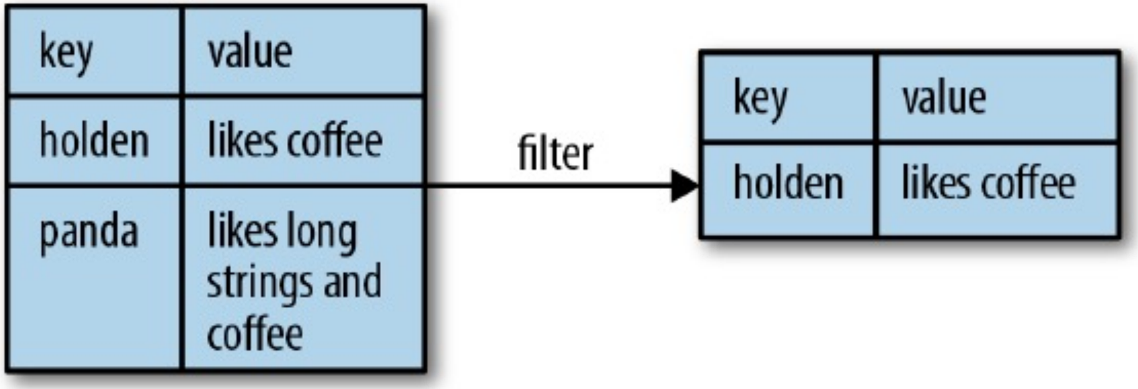
```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Example 4-5. Simple filter on second element in Scala

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

Example 4-6. Simple filter on second element in Java

```
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> keyValue) {
      return (keyValue._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```



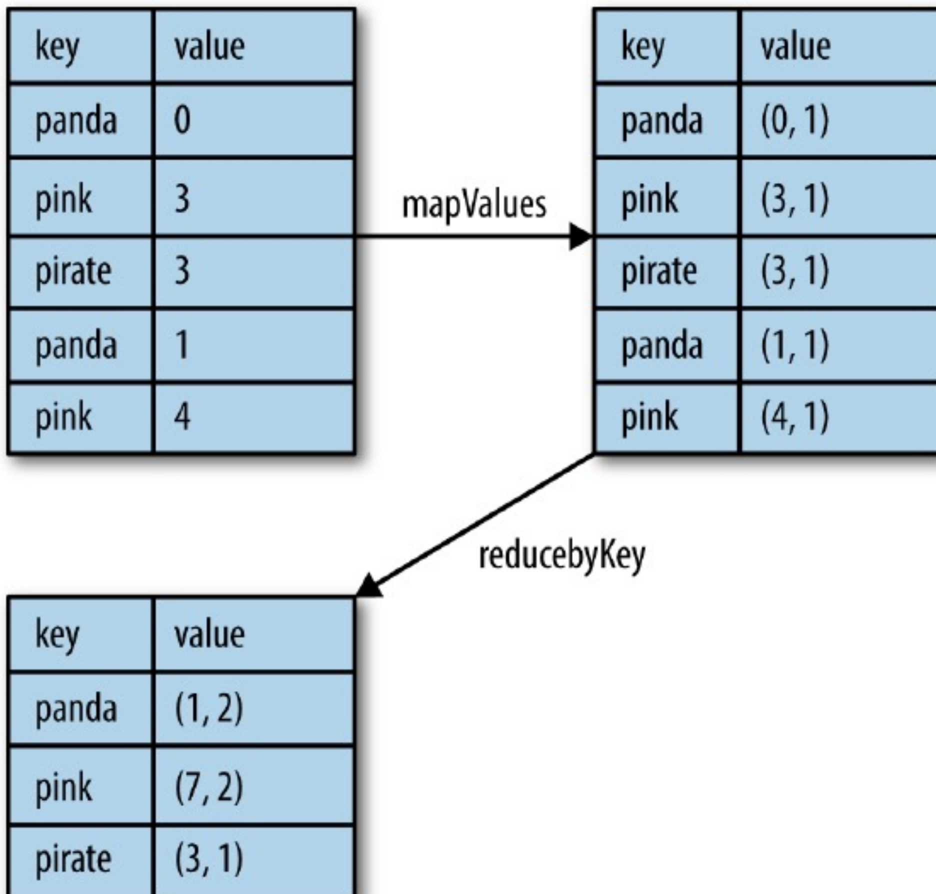
Examples of Per-key Averaging

Example 4-7. Per-key average with reduceByKey() and mapValues() in Python

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

Example 4-8. Per-key average with reduceByKey() and mapValues() in Scala

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
```



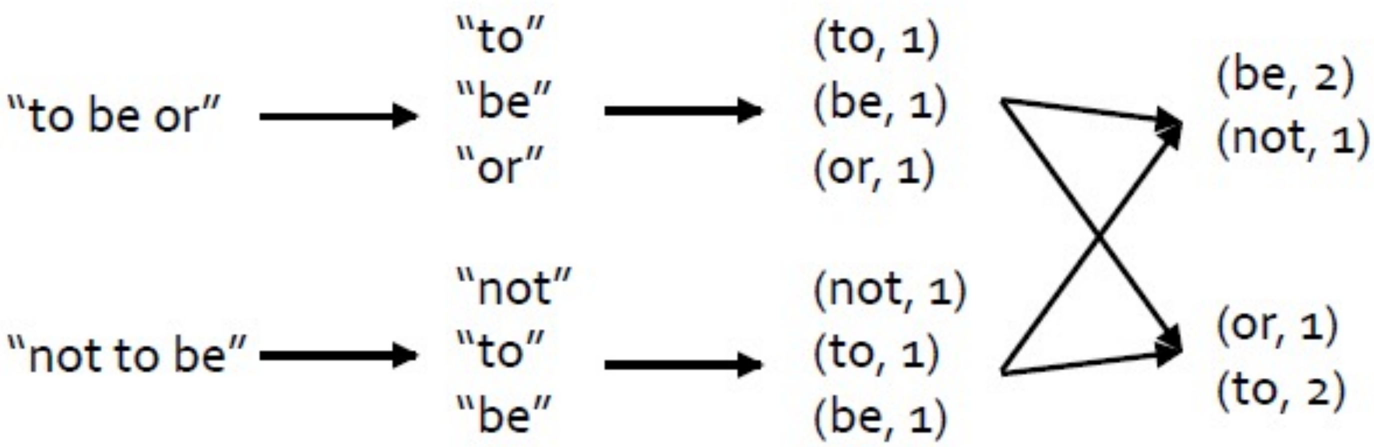
The Word Count Example in Python or Scala

Example 4-9. Word count in Python

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

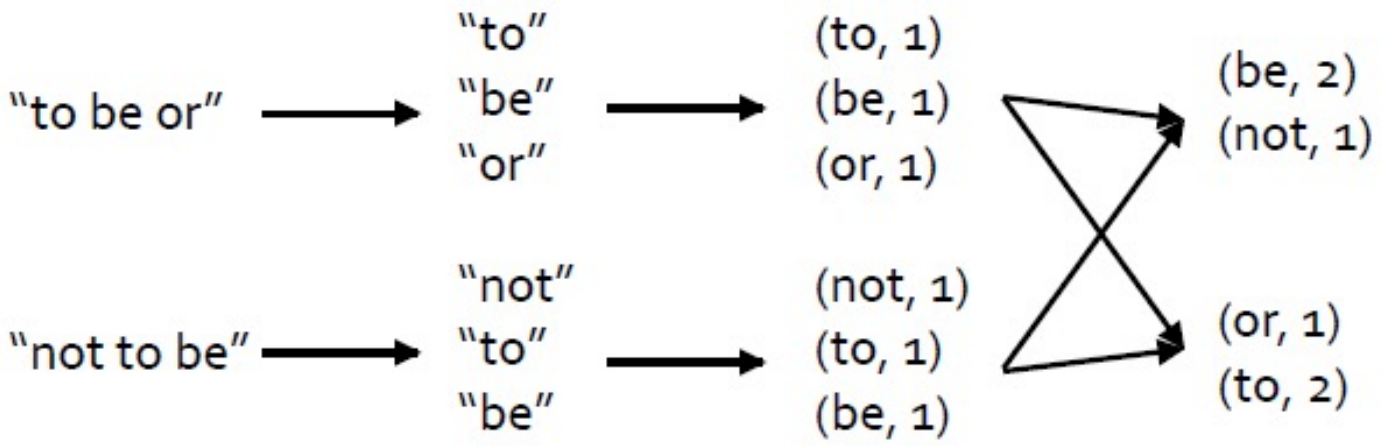
Example 4-10. Word count in Scala

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```



The Word Count Example (w/ Scala shorthand):

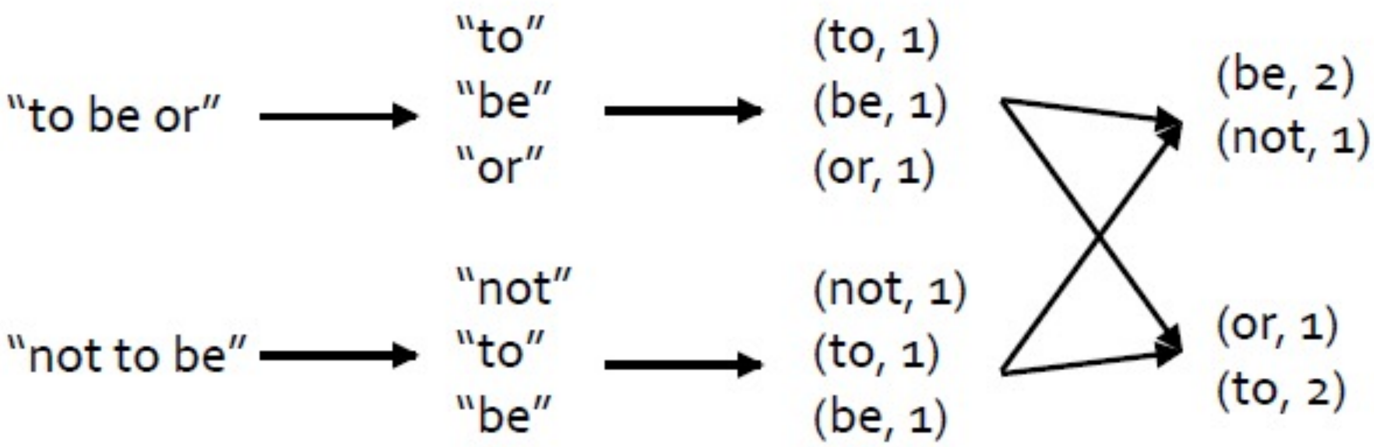
```
val lines = sc.textFile("hamlet.txt")

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



The Word Count Example in Java

Example 4-11. Word count in Java

```java
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
}).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
```



A Complete Example of Word-Count w/ Spark

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

Changing the Persistence of RDD

- By default, RDDs are lazy and ephemeral.
- User can alter the persistence of an RDD through two actions:
  - Cache action: By calling the persist( ) method, user provides the **hints** that the RDD should be kept in memory after the first time it is computed, because it will be reused.
  - Save action: evaluates the dataset and writes it to a distributed filesystem such as HDFS
- Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM.
- Users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first.

Memory Management in Spark

Spark provides three options for persist RDDs:

(1) In-memory storage as deserialized Java Objects

>> fastest, JVM can access RDD natively

(2) In-memory storage as serialized data

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

(3) On-disk storage

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

Persistence Levels in Spark

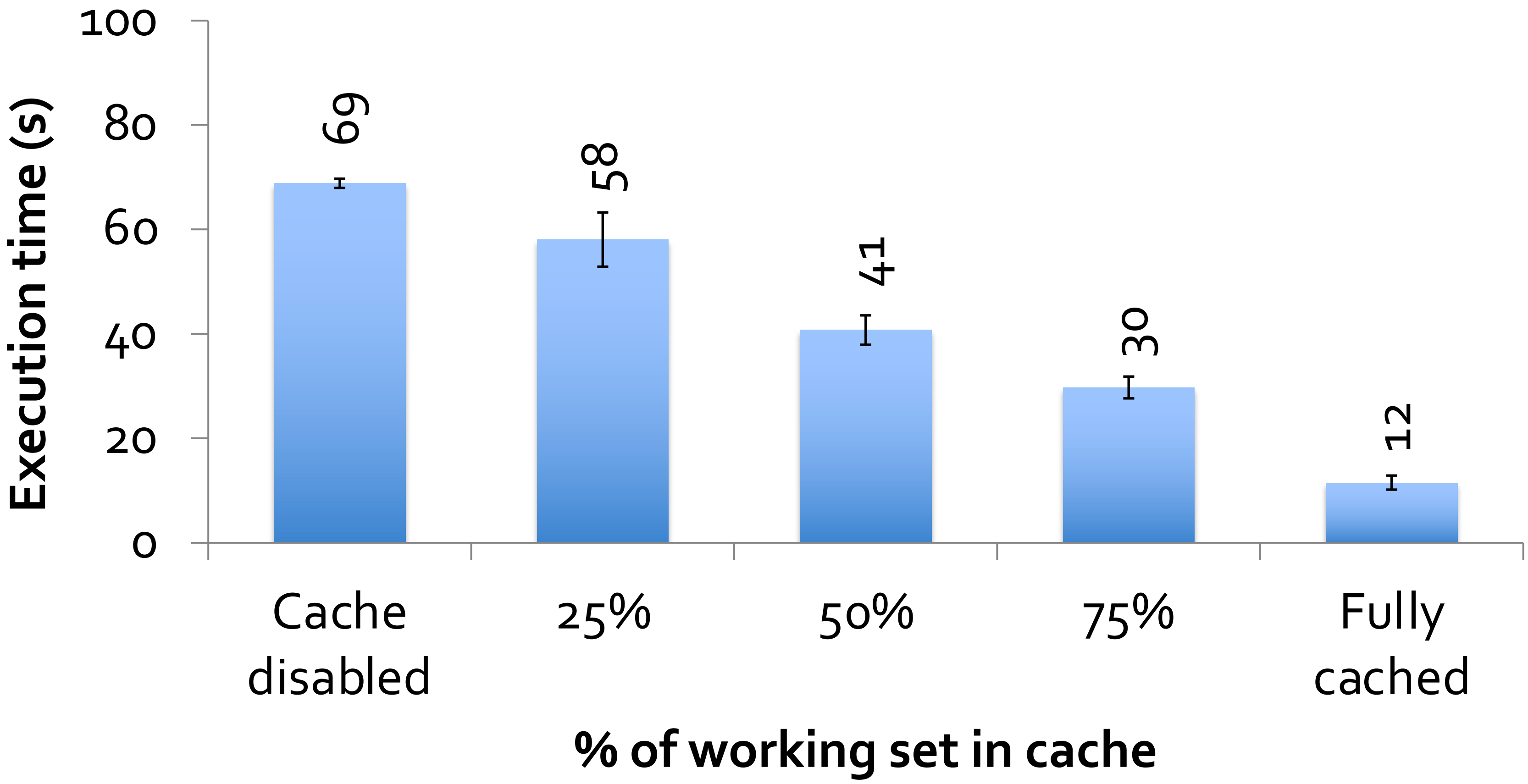
Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Example 3-40. persist() in Scala

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Behavior with Less RAM



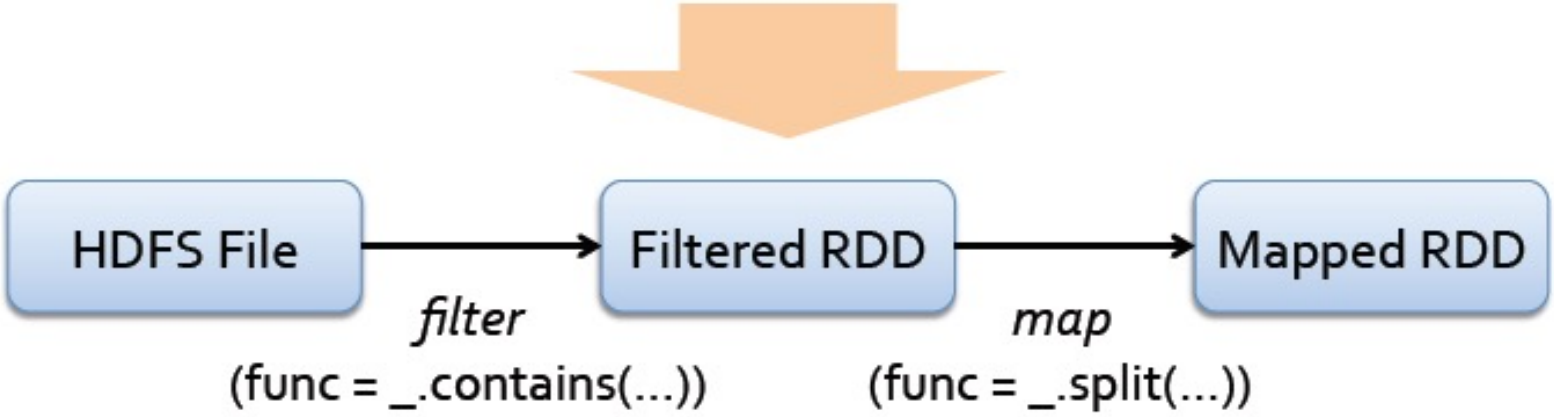
RDDs vs. Distributed Shared Memory

Aspect	RDDs	DSM
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	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 backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance(swapping ?)

RDD Fault Tolerance

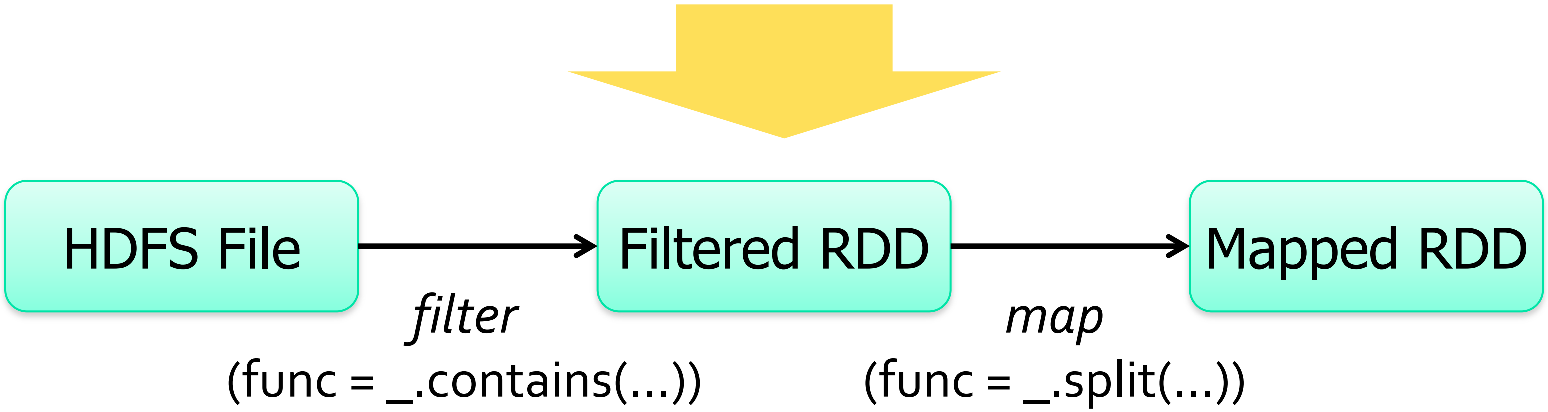
- An RDD has enough information about how it was derived from other datasets (aka its lineage).
  - RDD’s Lineage info can be used to reconstruct lost partitions

Ex:
```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



(Same Example in Python)

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
               .map(lambda s: s.split("\t")[2])
```

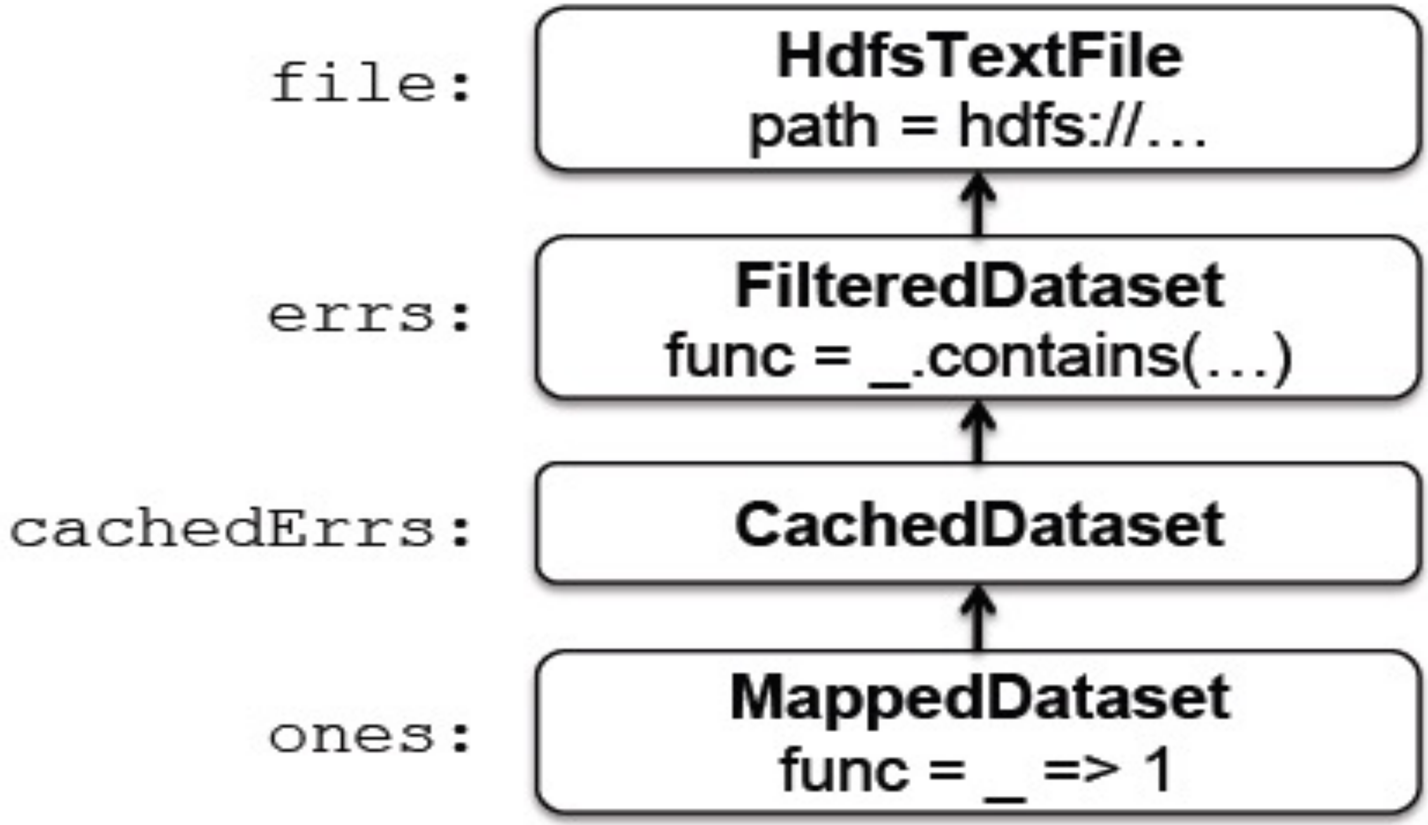


Example 2 of RDD

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

- These datasets will be stored as a chain of objects capturing the lineage of each RDD. Each dataset object contains a pointer to its parent and information about how the parent was transformed.

Lineage Chain of Example2



Example 3 of RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()


// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

Lineage Chain of Example 3

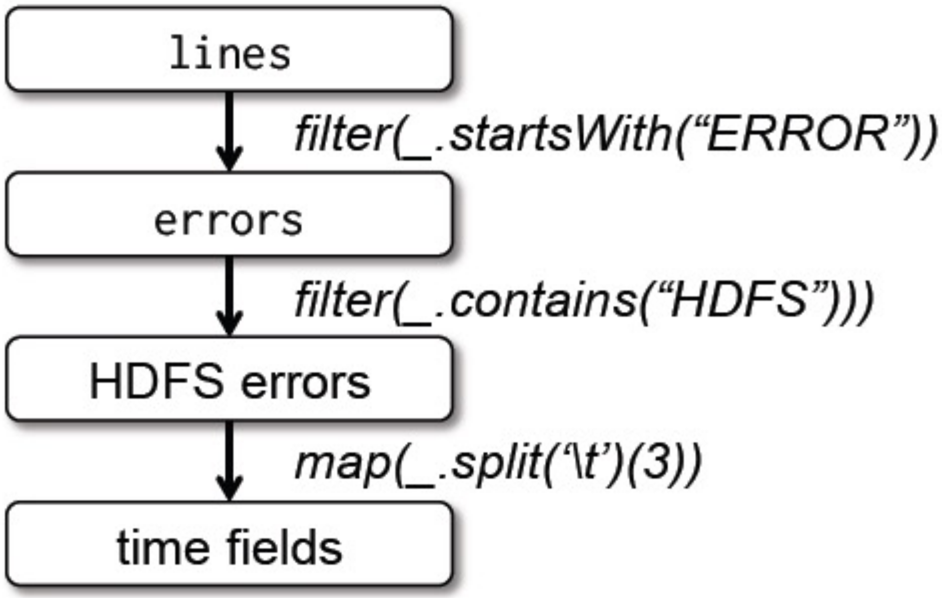


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

What is an RDD ?

A: Distributed Collection of Objects on disks

B: Distributed Collection of Objects in memory

C: Distributed Collection of Objects in Cassandra

- Answer: Could be any of the above.

What is an RDD ?

- Scientific Answer: RDD is an Interface !

1. Set of *partitions* (“splits” in Hadoop)
2. List of *dependencies* on parent RDDs
3. Function to *compute* a partition (as an Iterator) given its parent(s)

} “lineage”

4. (Optional) *partitioner* (hash, range)
5. (Optional) *preferred location*(s) for each partition

} optimized execution

Interface used to represent RDDs

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

Example: A HadoopRDD

partitions = one per HDFS block

dependencies = none

compute*(part)* = read corresponding block

preferredLocations*(part)* = HDFS block location

partitioner = none

Example: A Filtered RDD

partitions = same as parent RDD

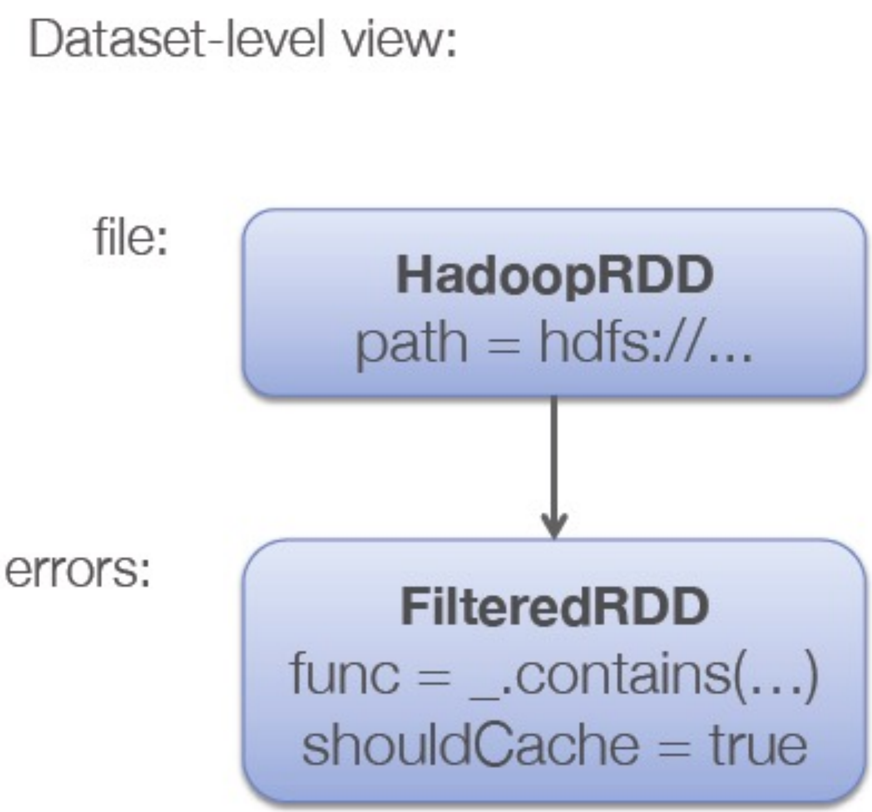
dependencies = "one-to-one" on parent

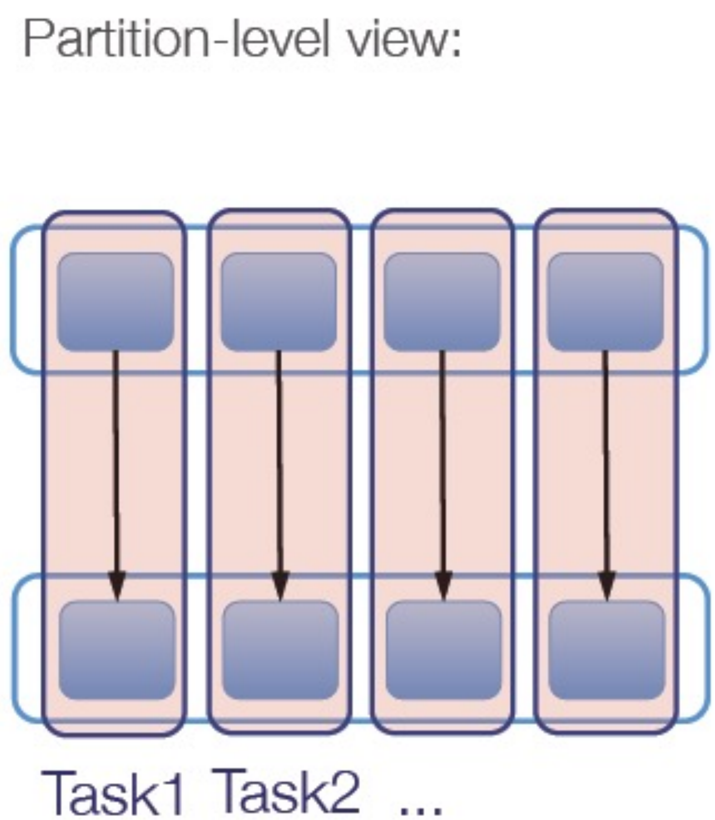
compute*(part)* = compute parent and filter it

preferredLocations*(part)* = none (ask parent)

partitioner = none

RDD Graph (DAG of tasks)





Example: A Joined RDD

partitions = one per reduce task

dependencies = "shuffle" on each parent

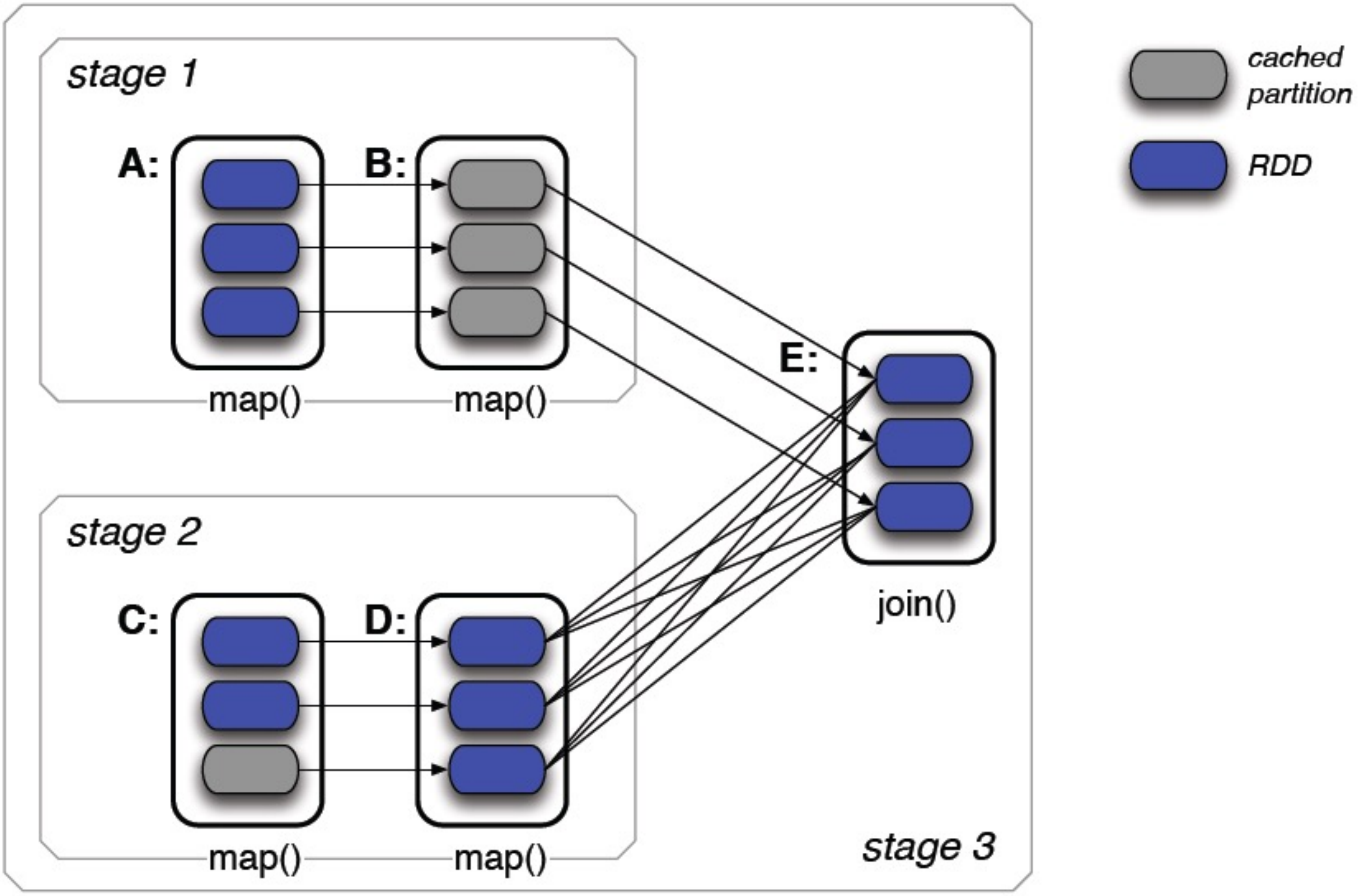
compute*(partition)* = read and join shuffled data

preferredLocations*(part)* = none

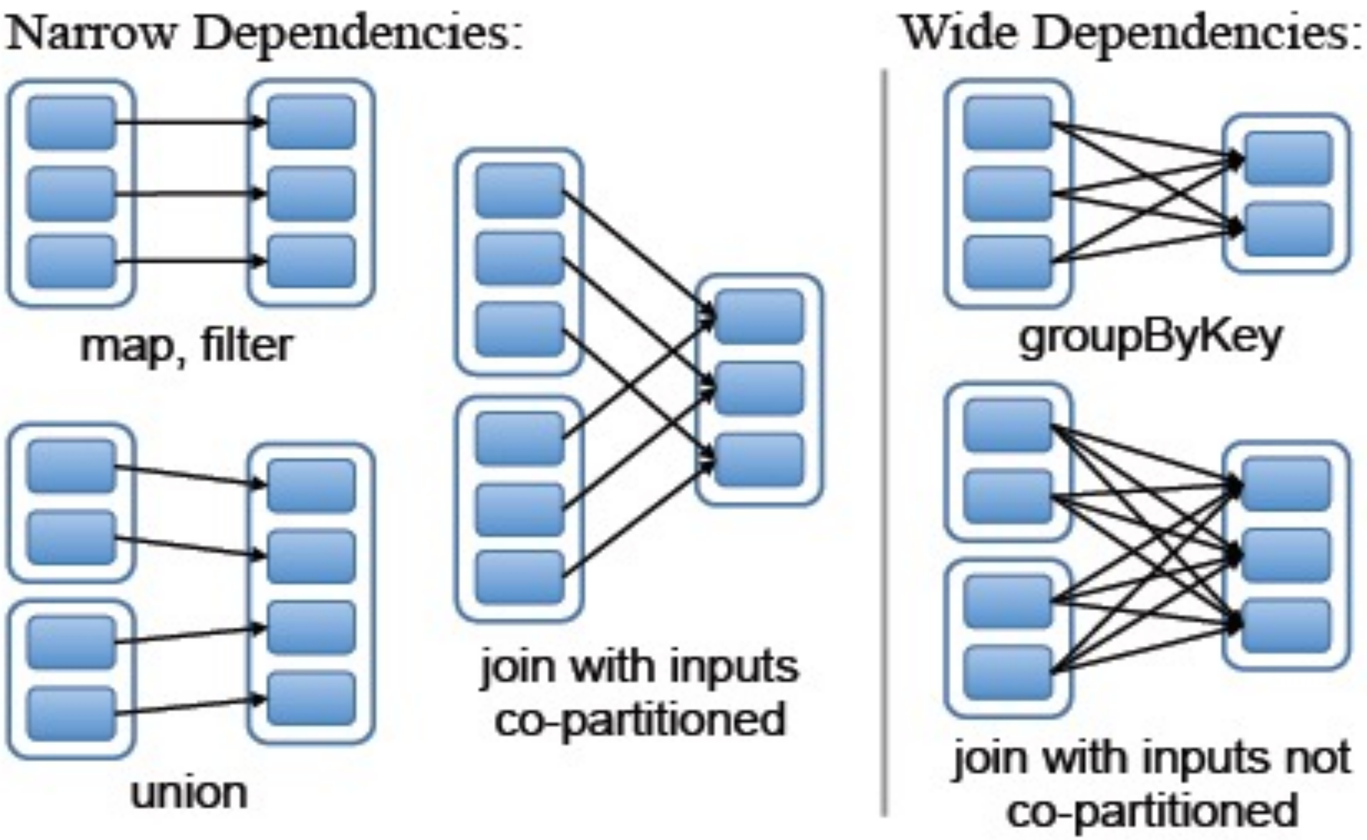
partitioner = HashPartitioner(numTasks)

Spark will now know this data is hashed!

Example: Join and its Operator Graph



RDD Dependency Types



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

Dependencies between RDDs(1)

- Narrow Dependencies: each partition of the parent RDD is used by at most one partition of the child RDD(1:1). Map leads to a narrow dependency.

- Wide Dependencies: multiple child partitions may depend on it(1:N). Join leads to wide dependencies.

Dependencies between RDDs(2)

- Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.
- Wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce like operation.
- Recovery after a node failure is more efficient with a narrow dependency than the ones with wide dependency.

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

Shared Variables

- Programmers invoke operations like map, filter and reduce by passing closures (functions) to Spark. Normally, when Spark runs a closure on a worker node, these variables are copied to the worker.
- However, Spark also lets programmers create two restricted types of shared variables to support two simple but common usage patterns.

Broadcast Variables

- When one creates a broadcast variable b with a value v, v is saved to a file in a shared file system. The serialized form of b is a path to this file. When b’s value is queried on a worker node, Spark first checks whether v is in a local cache, and reads it from the file system if it isn’t.

Accumulators

- Each accumulator is given a unique ID when it is created. When the accumulator is saved, its serialized form contains its ID and the “zero” value for its type.
- On the workers, a separate copy of the accumulator is created for each thread that runs a task using thread-local variables, and is reset to zero when a task begins. After each task runs, the worker sends a message to the driver program containing the updates it made to various accumulators.

A More Sophisticated Example: Computing PageRank w/ Spark

- 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
- Demonstrating the Importance of Controlling the Partitioning of RDDs for Performance Optimization

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

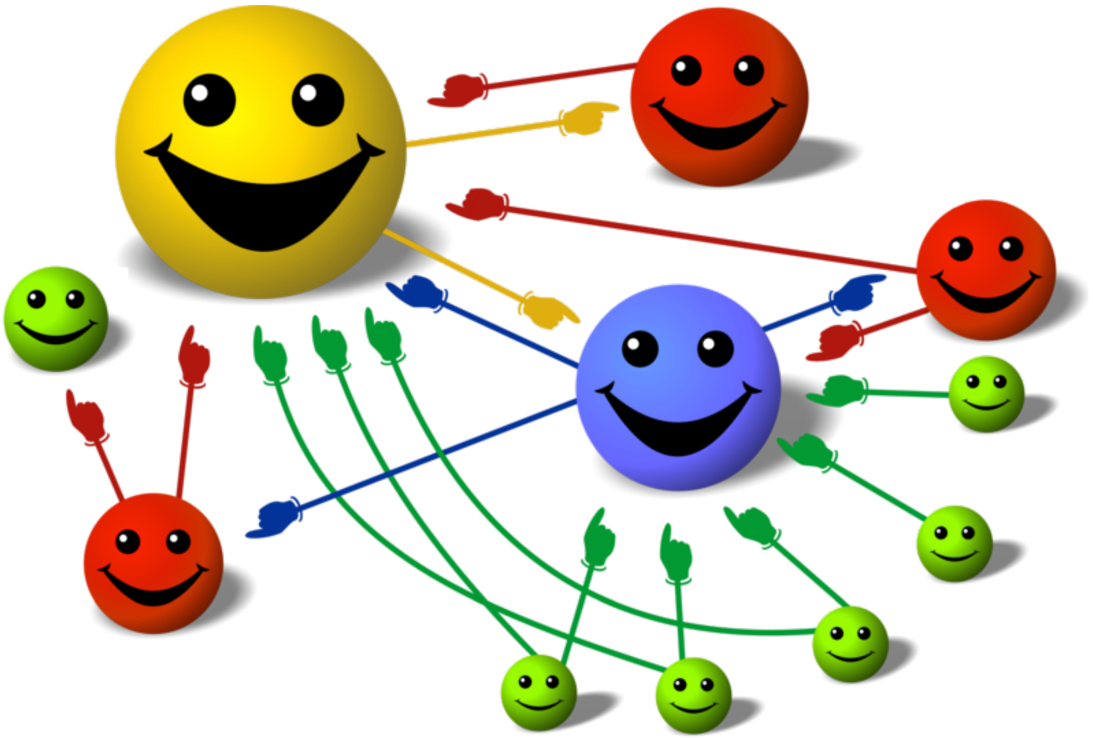
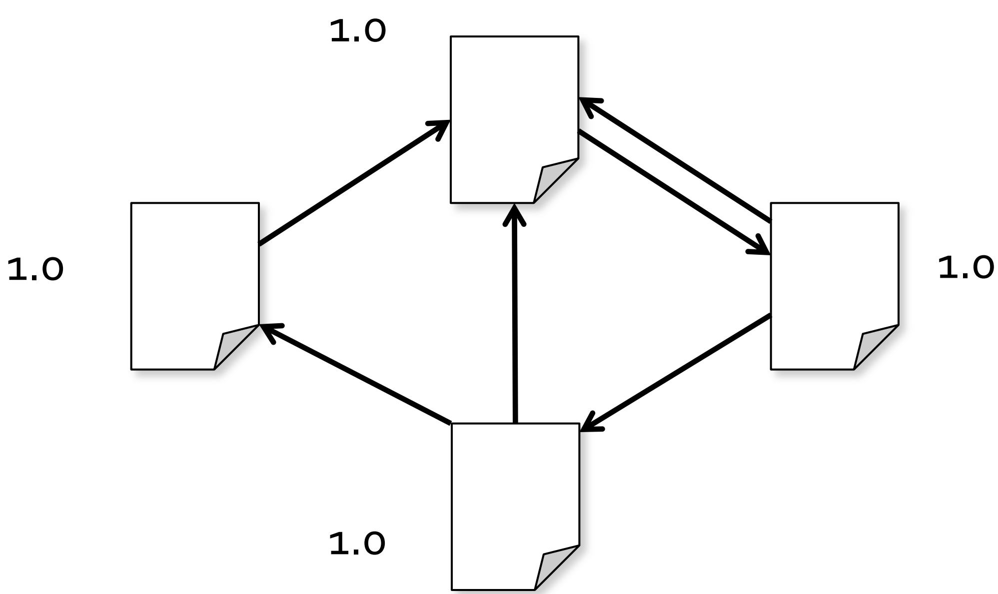


Image: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

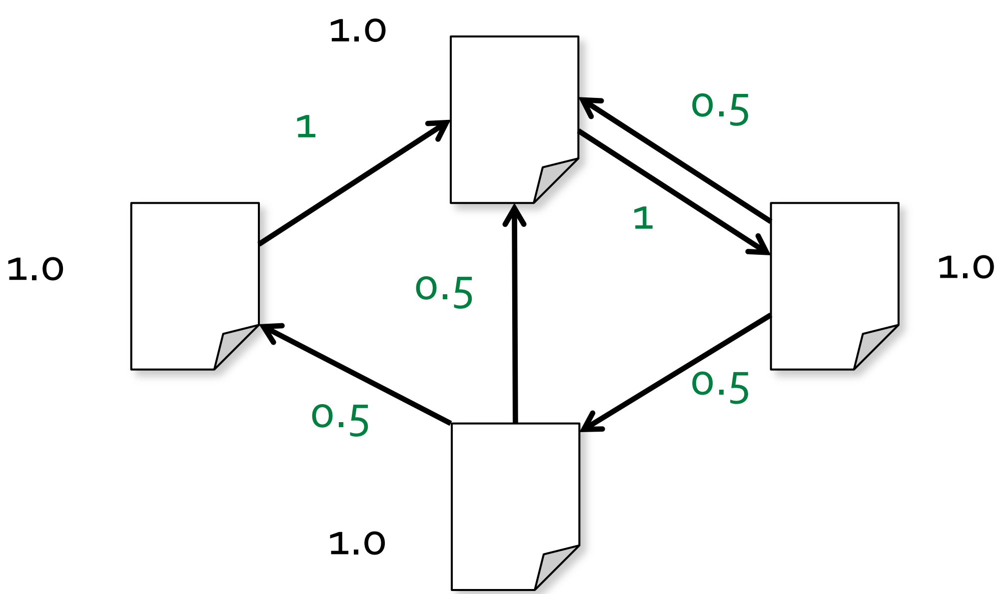
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs



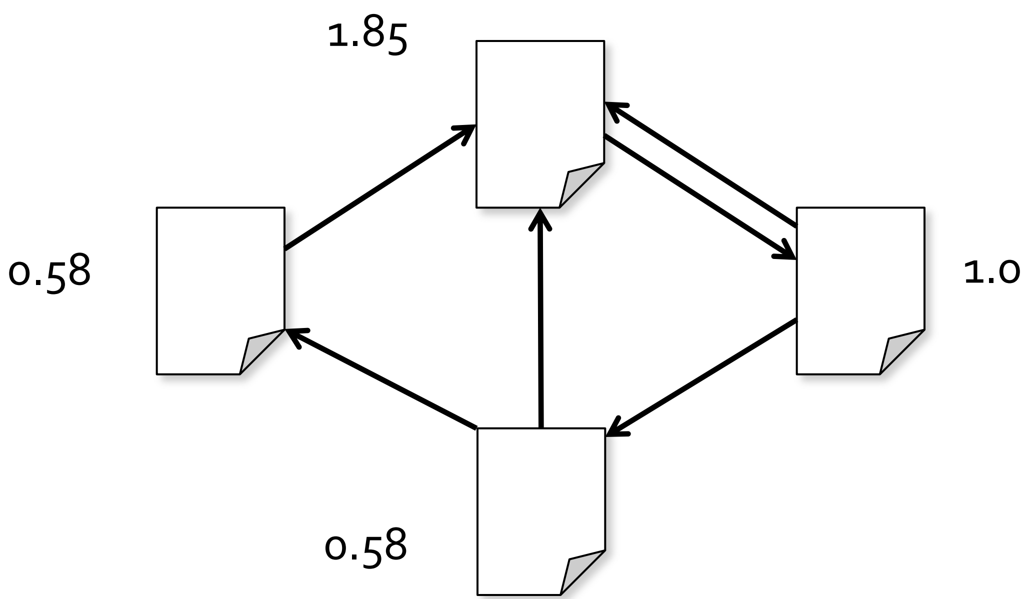
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs



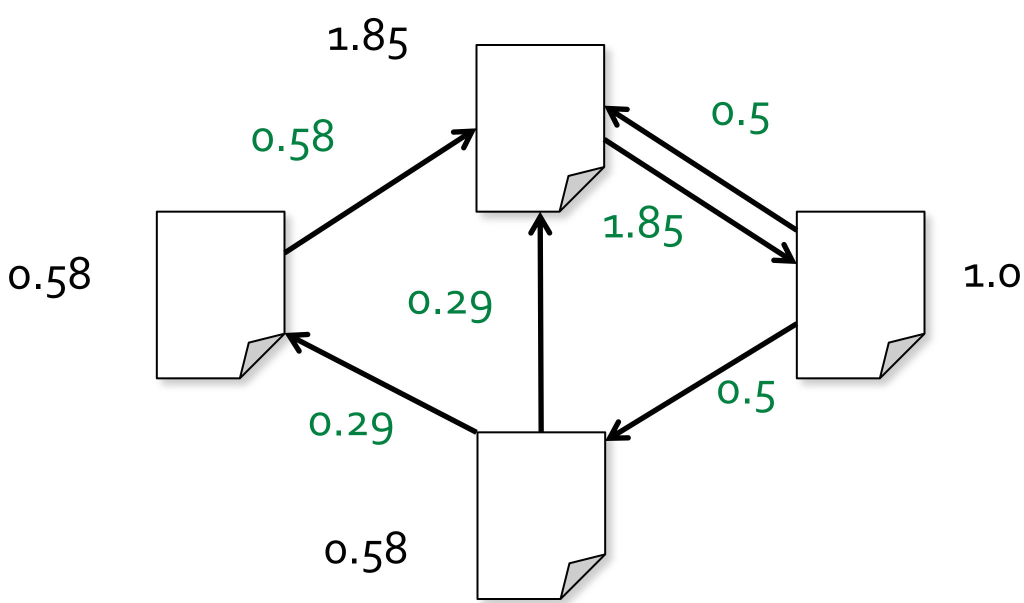
Algorithm

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



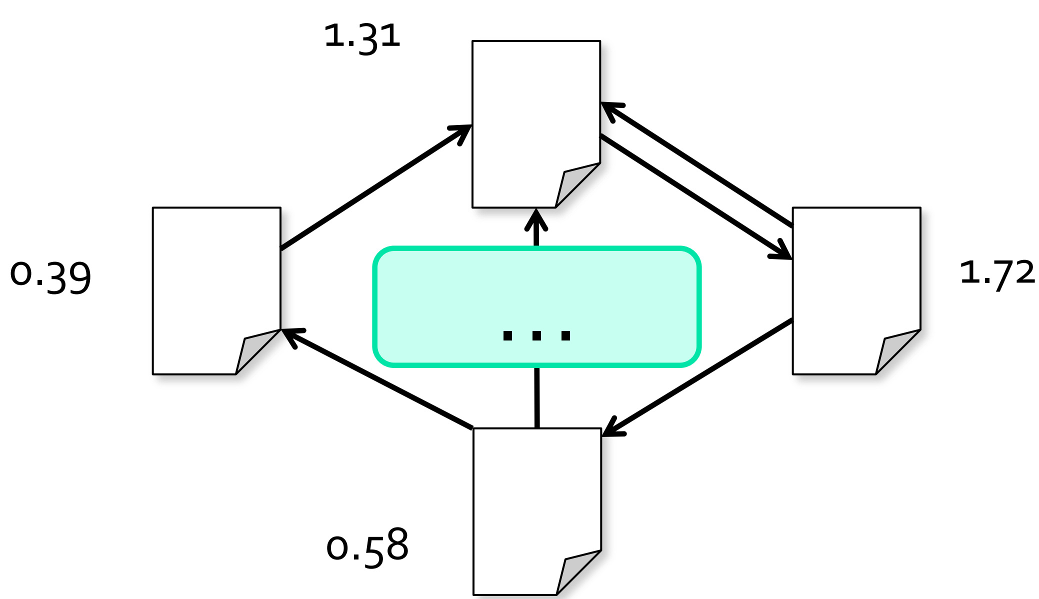
Algorithm

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



Algorithm

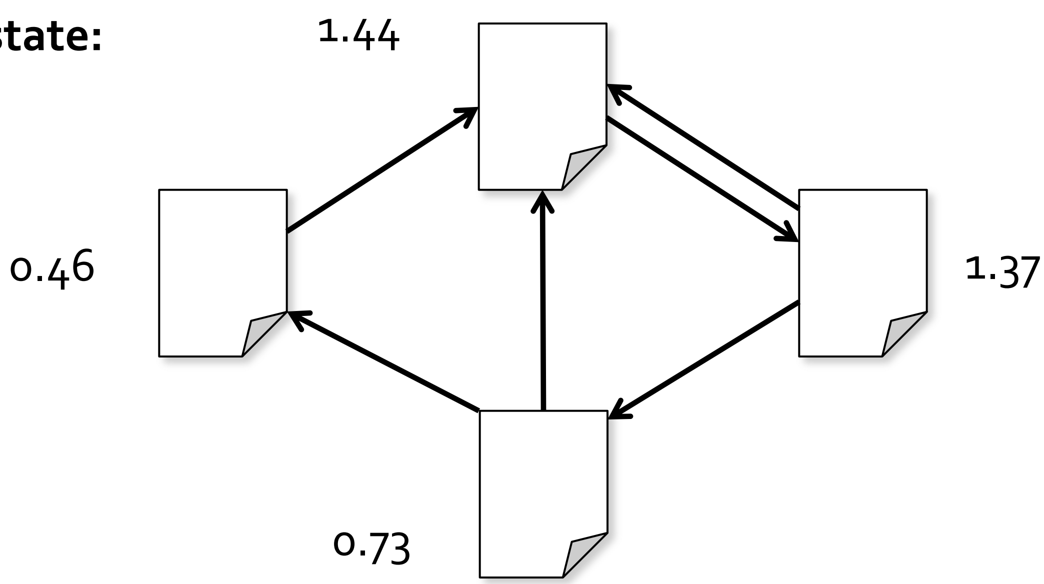
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



Algorithm

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

Final state:



Naïve Implementation of PageRank in Spark (in Scala)

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs

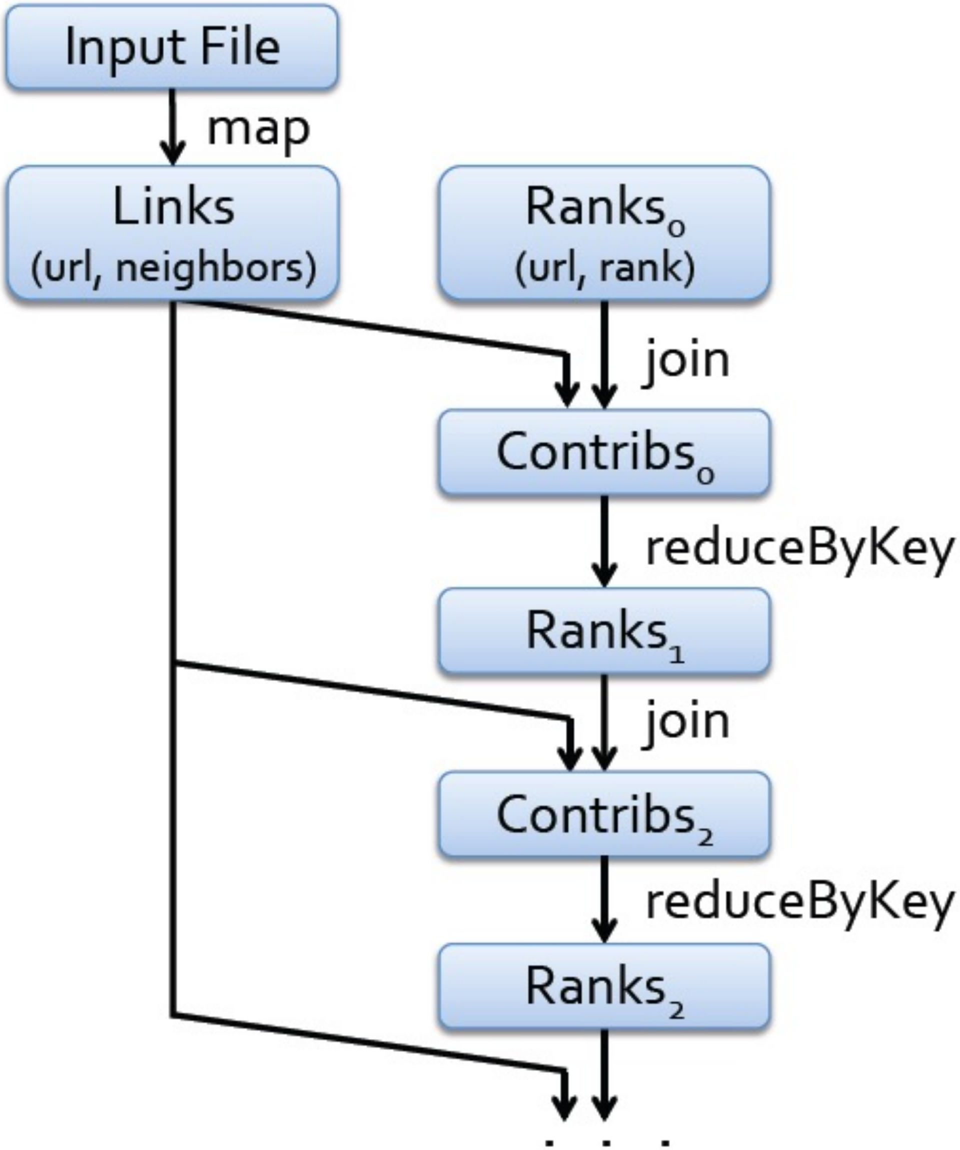
```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(.15 + .85*_)
}
```

Note: The need of the "case" primitive in scala:
http://danielwestheide.com/blog/2012/12/12/the-neophytes-guide-to-scala-part-4-pattern-matching-anonymous-functions.html

Naïve Implementation of PageRank in Spark (in Scala)

```
val sc = new SparkContext("local", "PageRank", sparkHome,
                          Seq("pagerank.jar"))

val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

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

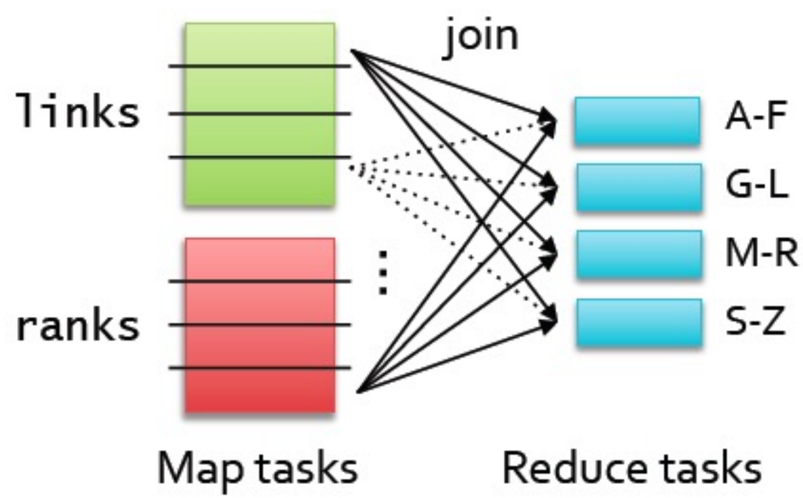
Execution of the Naïve Implementation of PageRank in Spark



`links` and `ranks` are repeatedly joined

Each join requires a full shuffle over the network

» Hash both onto same nodes

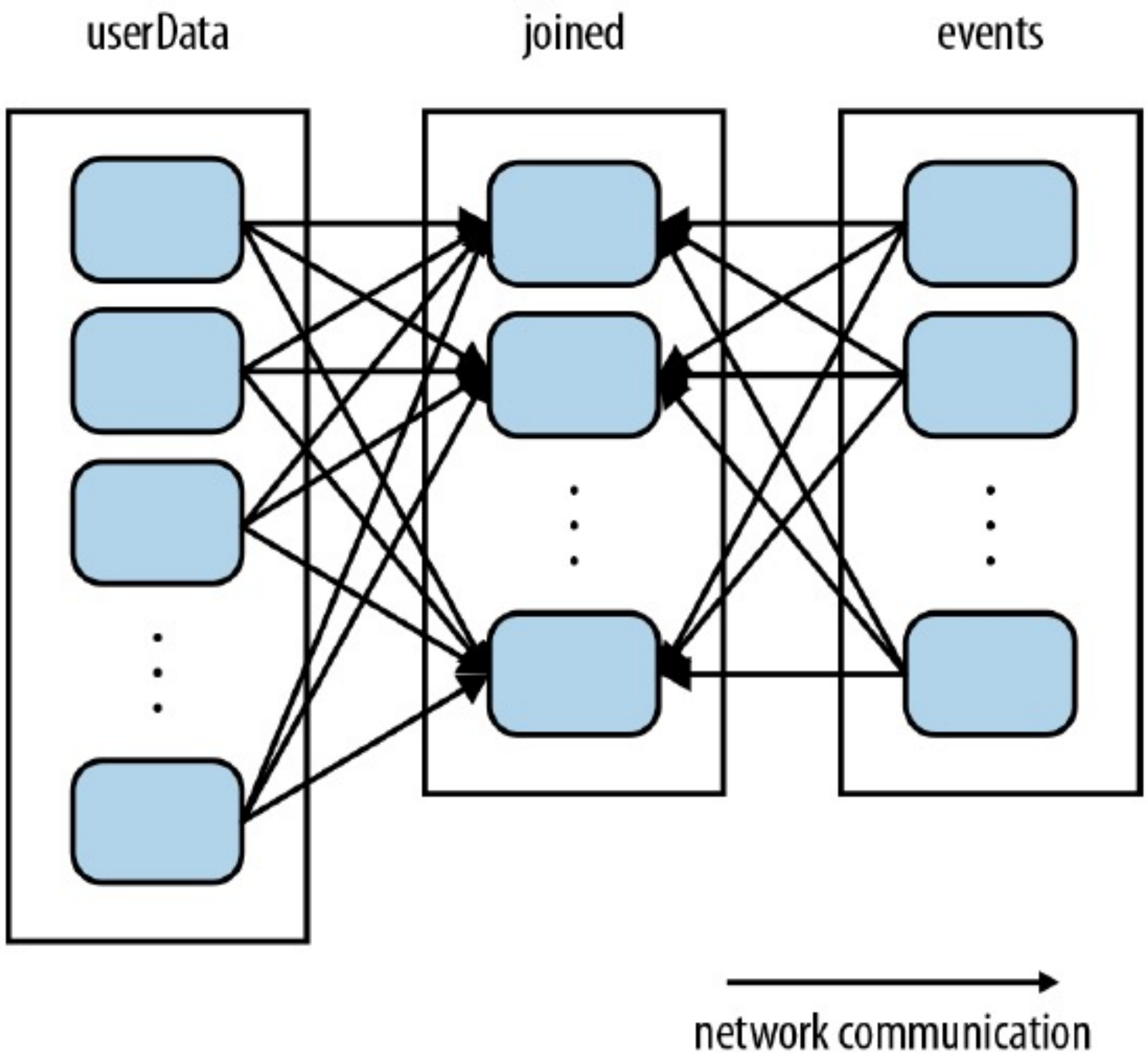


An Important (Optimization) Tool: Control the Partitioning of RDDs across different nodes

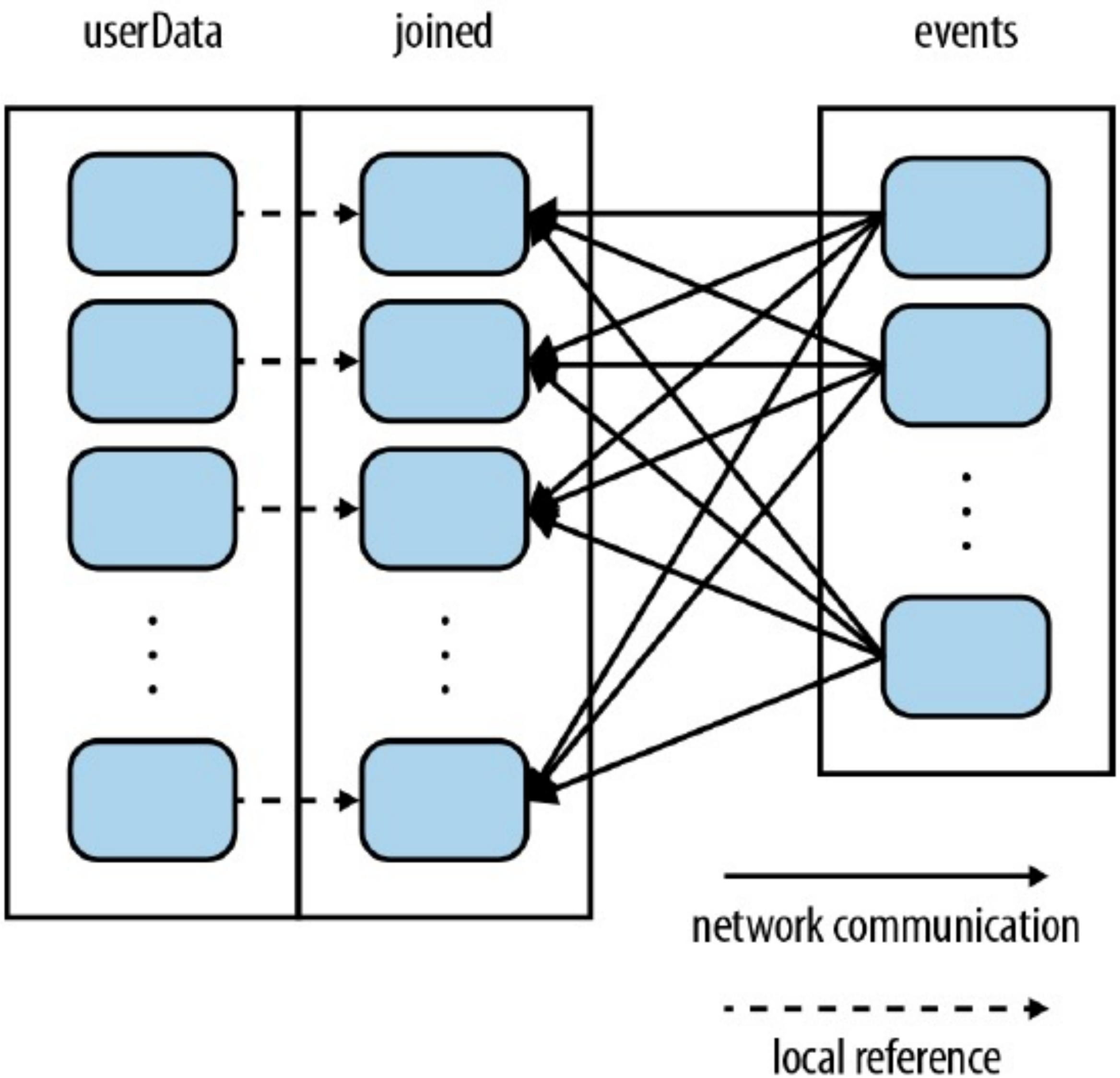
Pre-partition the `links` RDD so that links for URLs with the same hash code are on the same node

```
val ranks = // RDD of (url, rank) pairs
val links = sc.textFile(...).map(...)
               .partitionBy(new HashPartitioner(8))

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
   .mapValues(0.15 + 0.85 * _)
}
```

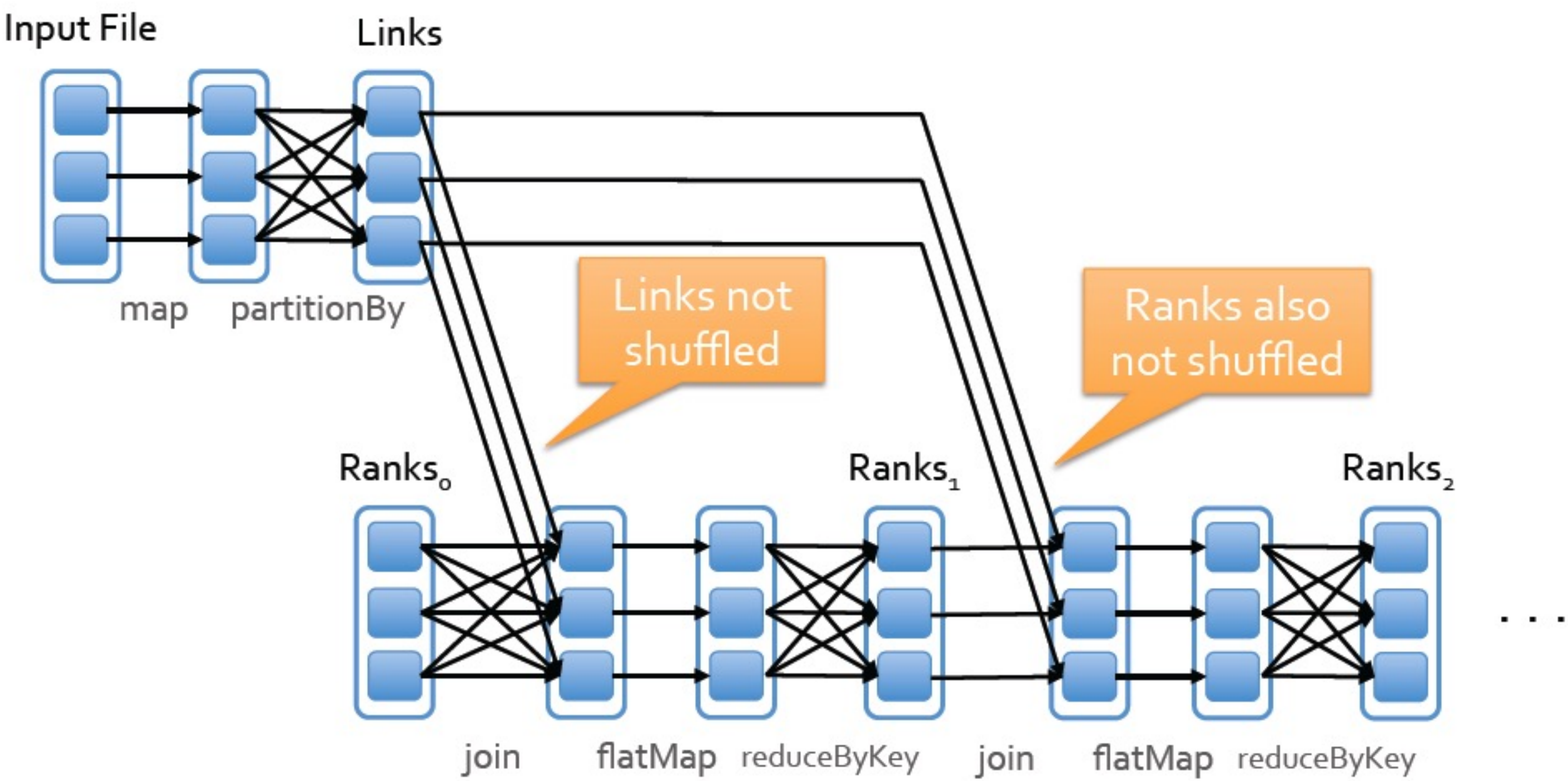
Join without using partitionBy



Join after using partitionBy



Execution Flow of the 2nd Implementation of PageRank in Spark



Yet Another Variation (Trick)

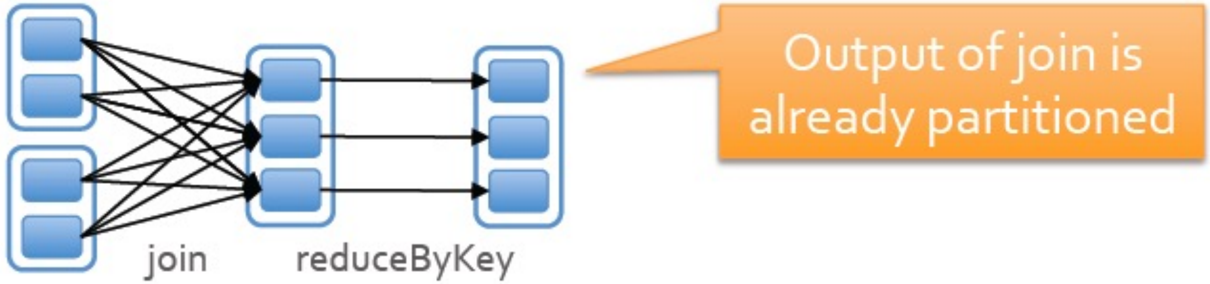
```
// Assume that our neighbor list was saved as a Spark objectFile
val links = sc.objectFile[(String, Seq[String])]("links")
               .partitionBy(new HashPartitioner(100))
               .persist()

// Initialize each page's rank to 1.0; since we use mapValues, the resulting RDD
// will have the same partitioner as links
var ranks = links.mapValues(v => 1.0)

// Run 10 iterations of PageRank
for (i <- 0 until 10) {
  val contributions = links.join(ranks).flatMap {
    case (pageId, (links, rank)) =>
      links.map(dest => (dest, rank / links.size))
  }
  ranks = contributions.reduceByKey((x, y) => x + y).mapValues(v => 0.15 + 0.85*v)
}

// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

How does it work ?

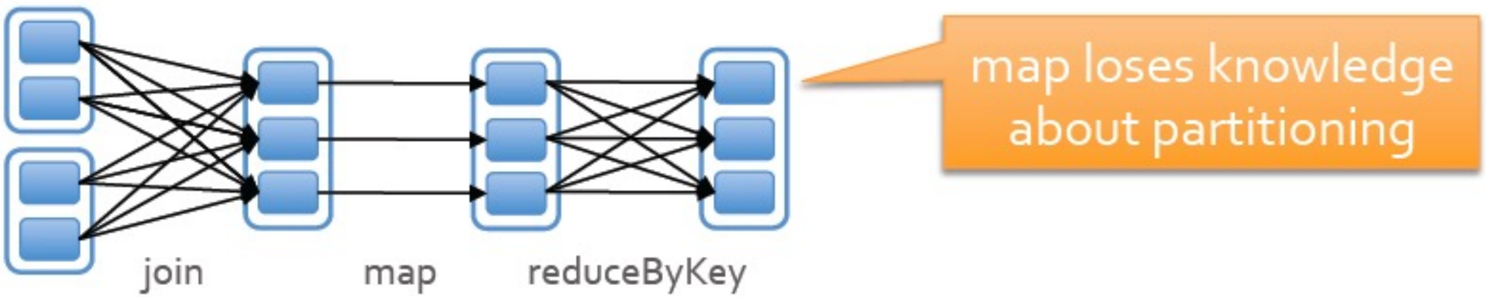
- Each RDD has an OPTIONAL `Partitioner` object
- Any `shuffle` operation on an RDD with a `Partitioner` will respect that `Partitioner`
- Any `shuffle` operation on two RDDs will take on the `Partitioner` of one of them, if one is set ;
  - Otherwise, will use the `HashPartitioner` by default

Examples of RDD Partitioning

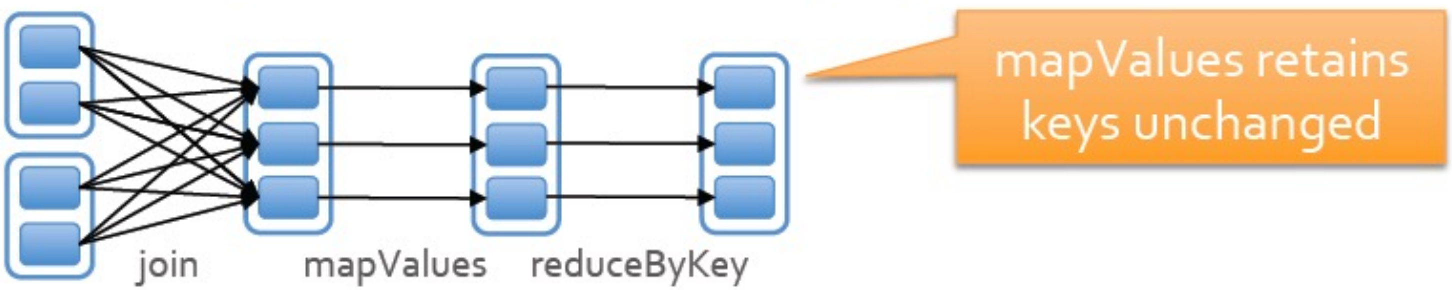
```
pages.join(visits).reduceByKey(...)
```



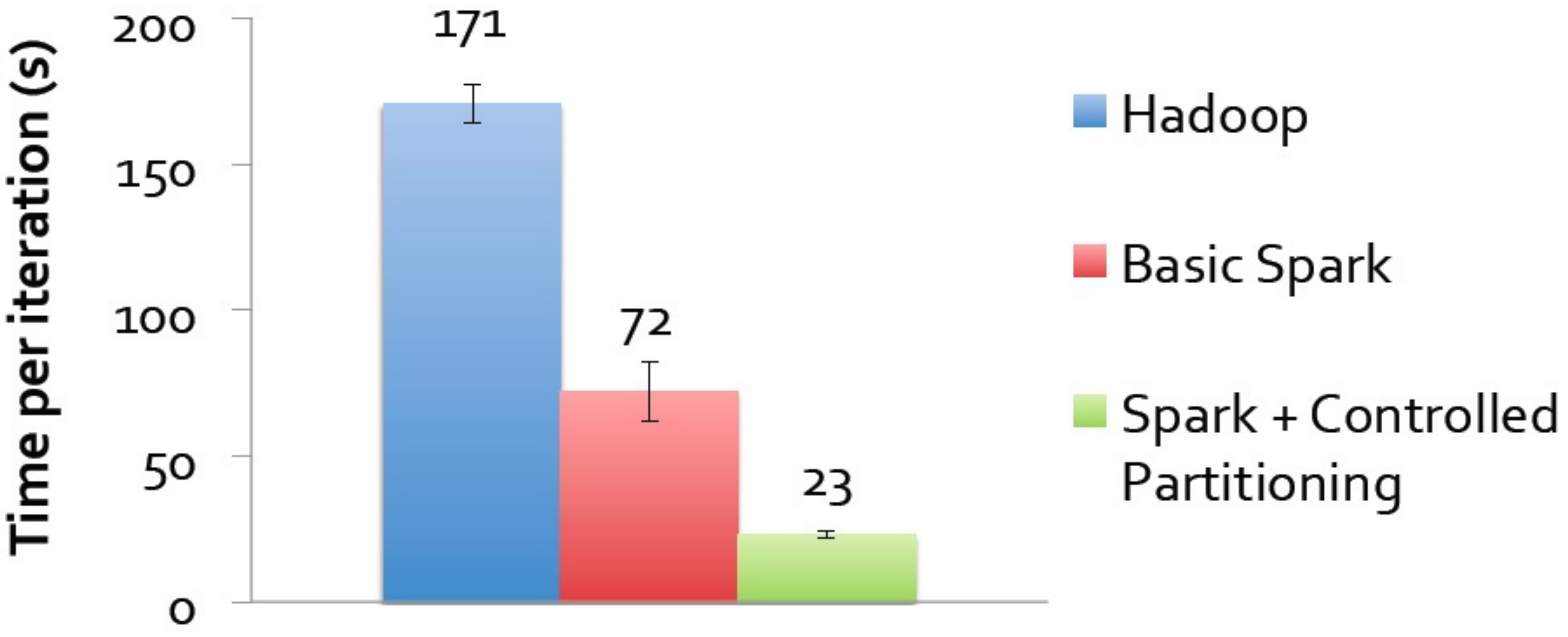

```
pages.join(visits).map(...).reduceByKey(...)
```




```
pages.join(visits).mapValues(...).reduceByKey(...)
```



PageRank Performance



Why it helps so much: `links` RDD is much bigger in bytes than `ranks`!

How to Customized RDD Partitioning

Can define your own subclass of `Partitioner` to leverage domain-specific knowledge

Example: in PageRank, hash URLs by domain name

```
class DomainPartitioner extends Partitioner {
  def numPartitions = 20

  def getPartition(key: Any): Int =
    parseDomain(key.toString).hashCode % numPartitions

  def equals(other: Any): Boolean =
    other.isInstanceOf[DomainPartitioner]
}
```

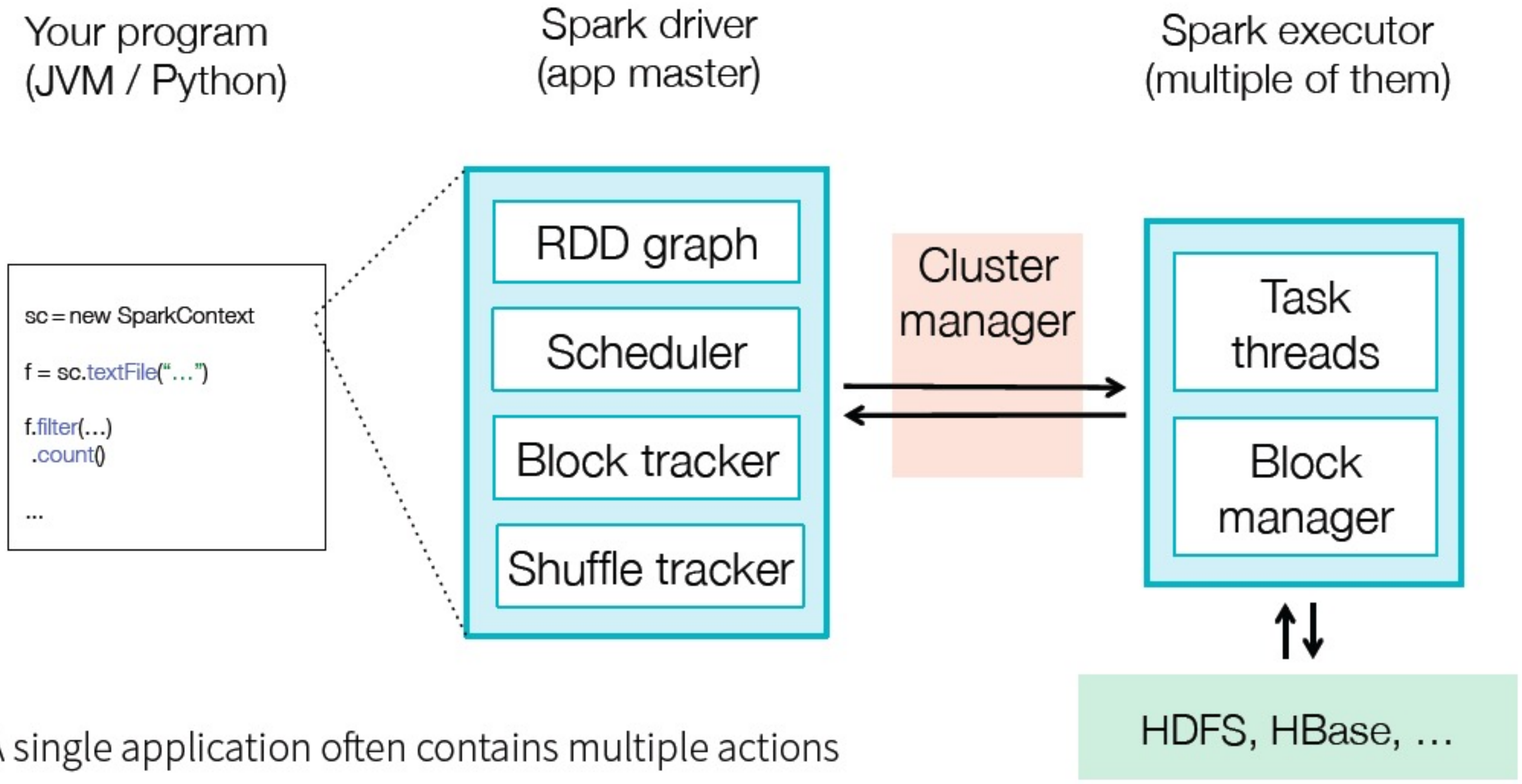
Needed for Spark to tell when two partitioners are equivalent

Way to find out how an RDD is Partitioned

Use the `.partitioner` method on RDD

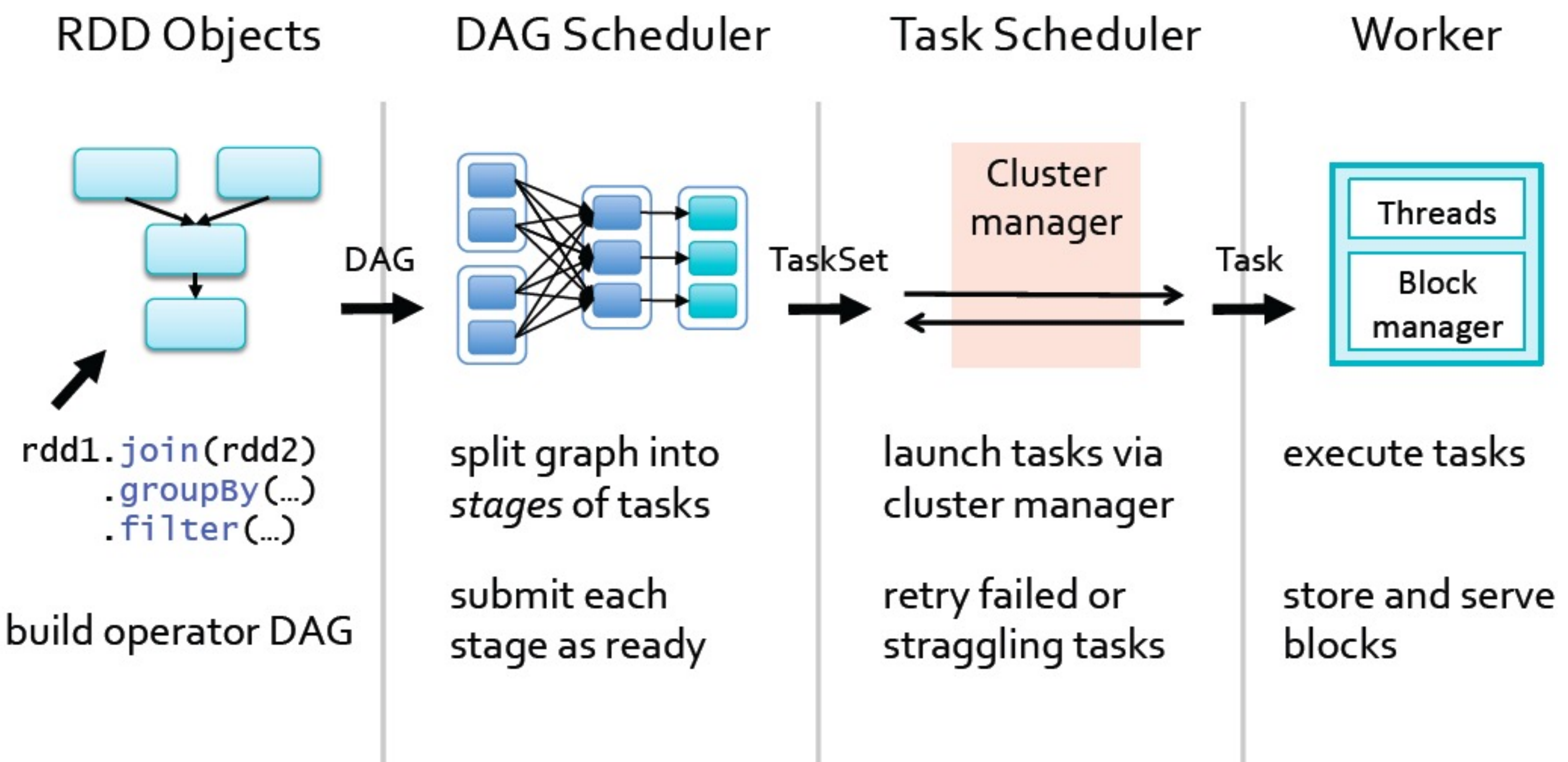
```
scala> val a = sc.parallelize(List((1, 1), (2, 2)))
scala> val b = sc.parallelize(List((1, 1), (2, 2)))
scala> val joined = a.join(b)

scala> a.partitioner
res0: Option[Partitioner] = None

scala> joined.partitioner
res1: Option[Partitioner] = Some(HashPartitioner@286d41c0)
```

A Spark Application



A single application often contains multiple actions

Execution Process of Spark

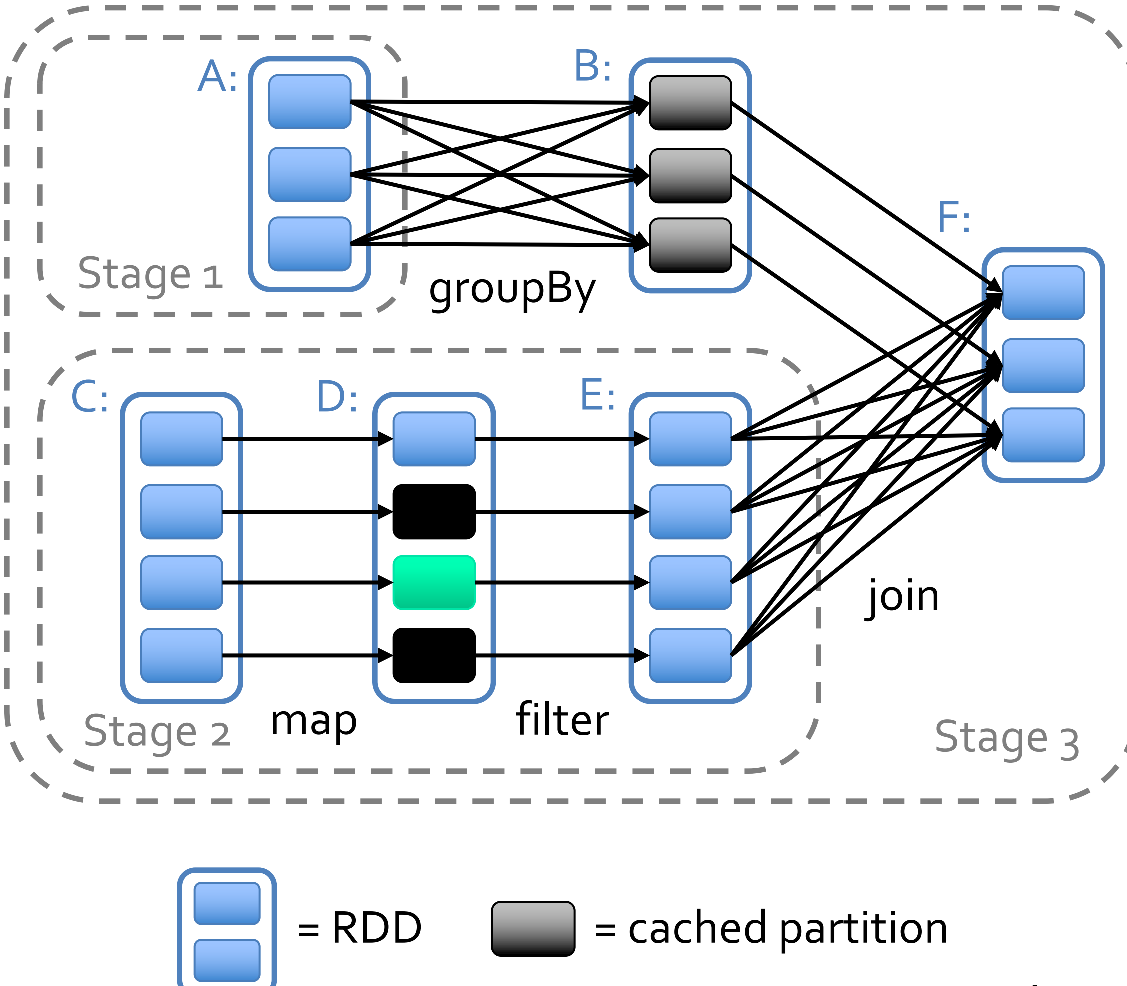


DAG Scheduler of Spark

- Input: RDD and Partitions to compute
- Output: Output from Actions of those Partitions
- Roles:
  - Build stages of tasks
  - Submit them to lower level scheduler, (e.g. YARN or Mesos, Standalone) as ready
  - Lower level scheduler will schedule data based on locality
  - Resubmit failed stages if outputs are lost

Job Scheduler of Spark

- Captures RDD dependency graph
- Pipelines functions into “stages”
- Cache-aware for data reuse & locality
- Partitioning-aware to avoid shuffles

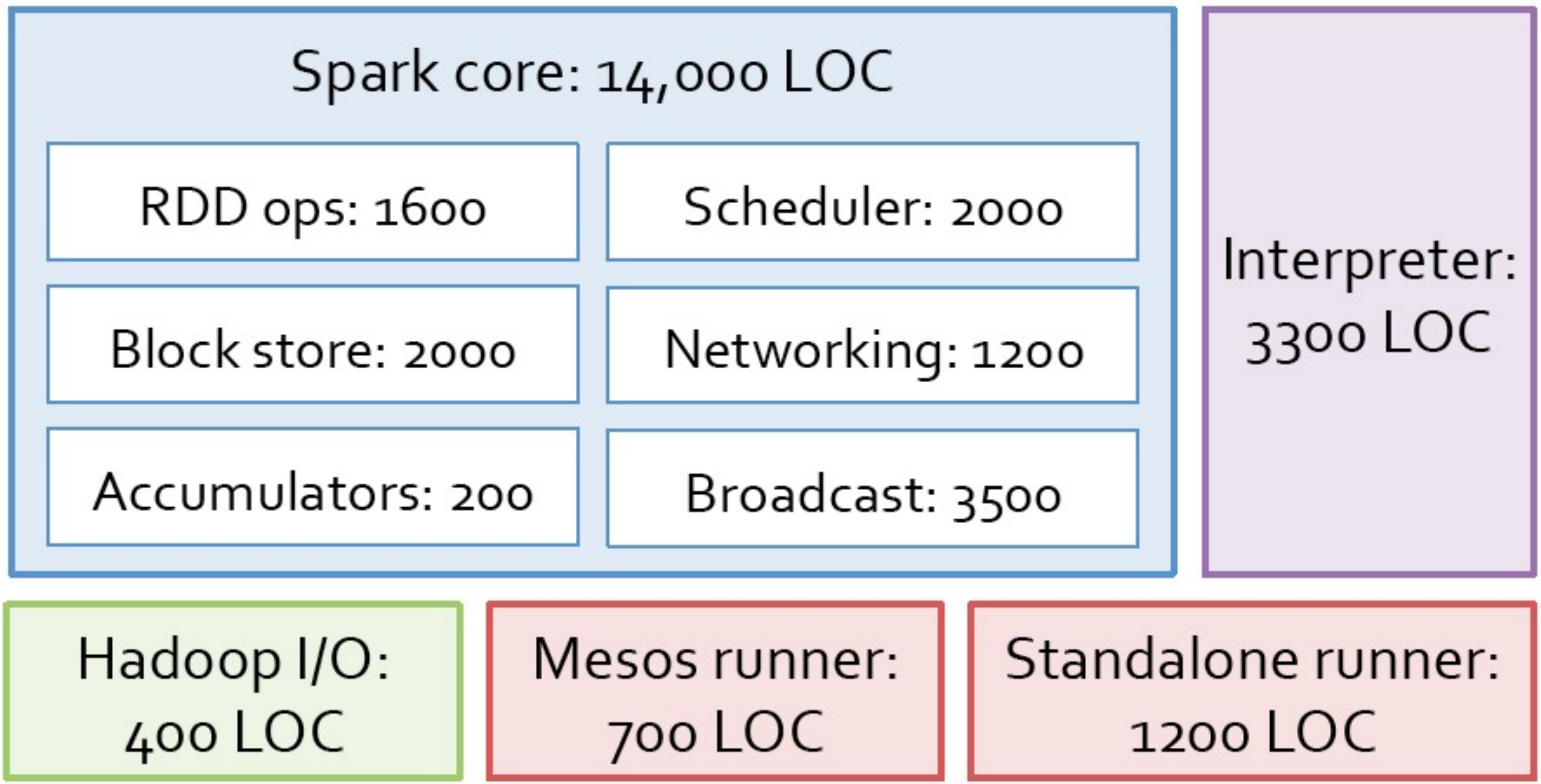


Outline

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

Codebase of Spark

Implement Spark Core in about 14,000 Lines of Scala:



Software Components: How to run Spark ?

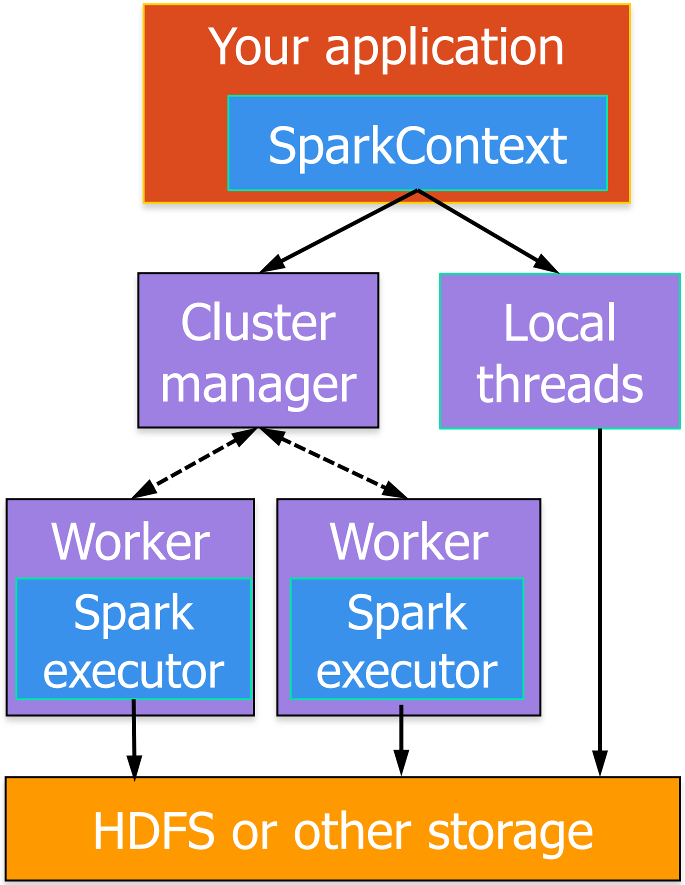
- Spark runs as a library in your program (1 instance per app)
- Runs tasks locally or on cluster
  - Mesos, YARN or standalone mode

>> **new SparkContext** (masterUrl, jobname, [sparkhome], [jars])

>> MASTER=local[n] ./spark-shell

>> MASTER=HOST:PORT ./spark-shell

- Access storage systems via Hadoop InputFormat API
  - Can use HBase, HDFS, Tachyon, S3, Cassandra, …



Add Spark to Your Project

- Scala / Java: add a Maven dependency on
  - groupId: `org.spark-project`
    artifactId: `spark-core_2.9.3`
    version: `0.7.3`

- Python: run program with our `pyspark` script

Create a SparkContext

(Generalized to SparkSession since Spark ver2.0)

https://stackoverflow.com/questions/49574511/what-is-difference-between-sparksession-and-sparkcontext

Scala

```
▪ import spark.SparkContext
▪ import spark.SparkContext._

▪ val sc = new SparkContext(“url”, “name”, “sparkHome”, Seq(“app.jar”))
```

- Cluster URL, or local / local[N]
- App name
- Spark install path on cluster
- List of JARs with app code (to ship)

Java

```
import spark.api.j… nt

JavaSparkContext sc = new JavaSparkContext(
    “masterUrl”, “name”, “sparkHome”, new String[] {“app.jar”}));
```

Python

```
from pyspark import SparkContext

sc = SparkContext(“masterUrl”, “name”, “sparkHome”, [“library.py”]))
```

Getting Started

- Download Spark:

  www.spark.apache.org/downloads.html

- Documentation and video tutorials:

  www.spark.apache.org/docs/latest

- Other Resources:

  www.Databricks.com

Local Execution

- Just pass `local` or `local[k]` as master URL
- Debug using local debuggers
  - For Java / Scala, just run your program in a debugger
  - For Python, use an attachable debugger (e.g. PyDev)
- Great for development & unit tests

Cluster Execution

- Easiest way to launch is EC2:

```
./spark-ec2 -k keypair –i id_rsa.pem –s slaves \
  [launch|stop|start|destroy] clusterName
```

- Several options for private clusters:
  - Standalone mode (similar to Hadoop's deploy scripts)
  - Mesos
  - Hadoop YARN
- Amazon EMR: tinyurl.com/spark-emr

Key Distinctions for Spark vs. MapReduce

- generalized patterns
  ⇒ unified engine for many use cases
- lazy evaluation of the lineage graph
  ⇒ reduces wait states, better pipelining
- generational differences in hardware
  ⇒ off-heap use of large memory spaces
- functional programming / ease of use
  ⇒ reduction in cost to maintain large apps
- lower overhead for starting jobs
- less expensive shuffles

Conclusion for Part I

- Scala : OOP + FP
- RDDs: fault tolerance, data locality/ partitioning-control, scalability
- RDD implemented in Spark using Scala
- Spark offers a rich API to make data analytics *fast*: both fast to write and fast to run
  - Achieves 50 or even 100+ speedups in real applications
- Rapidly growing community