Spark: Resilient Distributed Datasets as Workflow System

H. Andrew Schwartz

CSE545

Spring 2020

Big Data Analytics, The Class

Goal: Generalizations
A *model* or *summarization* of the data.

Data Frameworks

- Hadoop File System
- Spark
- Streaming
- MapReduce
- Tensorflow

Algorithms and Analyses

- Similarity Search
- Hypothesis Testing
- Graph Analysis
- Recommendation Systems
- Deep Learning

Where is MapReduce Inefficient?

DFS ⇨ Map ⇨ LocalFS ⇨ Network ⇨ Reduce ⇨ DFS ⇨ Map ⇨ ...

Where is MapReduce Inefficient?

- Long pipelines sharing data
- Interactive applications
- Streaming applications
- Iterative algorithms (optimization problems)

DFS ⇨ Map ⇨ LocalFS ⇨ Network ⇨ Reduce ⇨ DFS ⇨ Map ⇨ ...

(Anytime where MapReduce would need to write and read from disk a lot).

Where is MapReduce Inefficient?

- Long pipelines sharing data
- Interactive applications
- Streaming applications
- Iterative algorithms (optimization problems)

DFS ⇨ Map ⇨ LocalFS ⇨ Network ⇨ Reduce ⇨ DFS ⇨ Map ⇨ ...

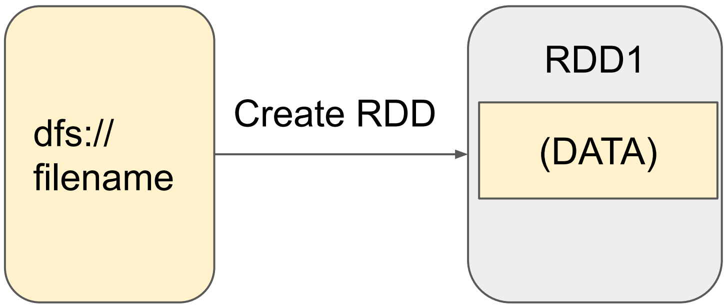
(Anytime where MapReduce would need to write and read from disk a lot).

Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

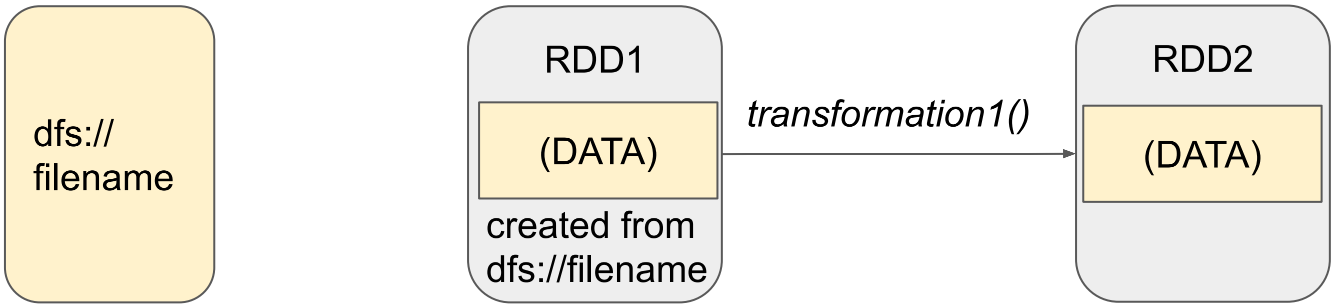
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

Spark's Big Idea

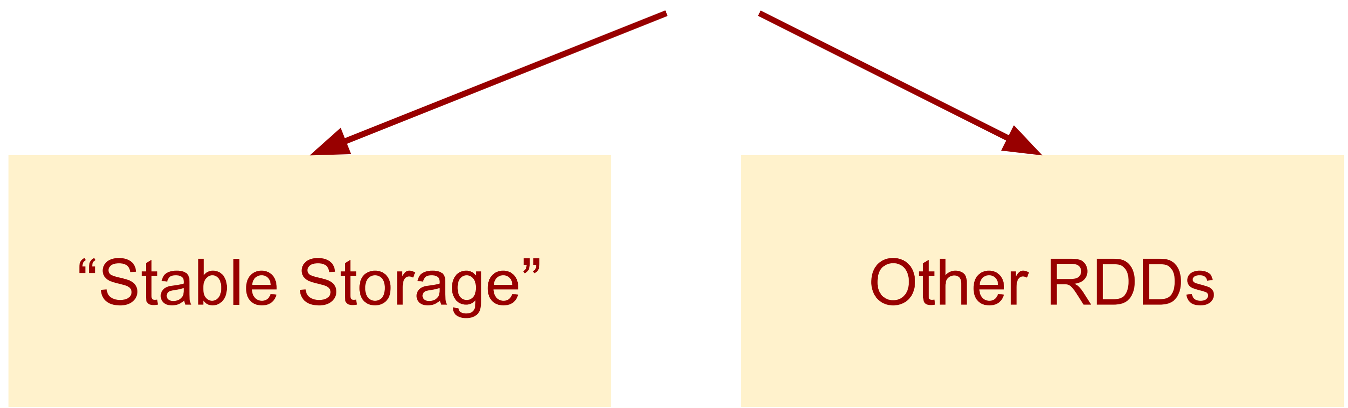
Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

- Enables rebuilding datasets on the fly.
- Intermediate datasets not stored on disk
  (and only in memory if needed and enough space)

⇨ Faster communication and I O

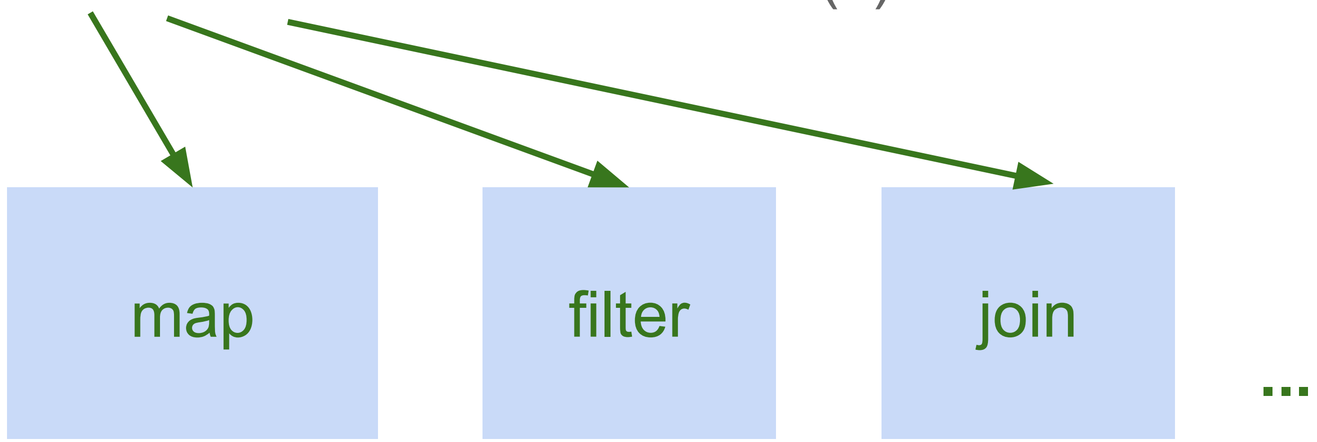
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

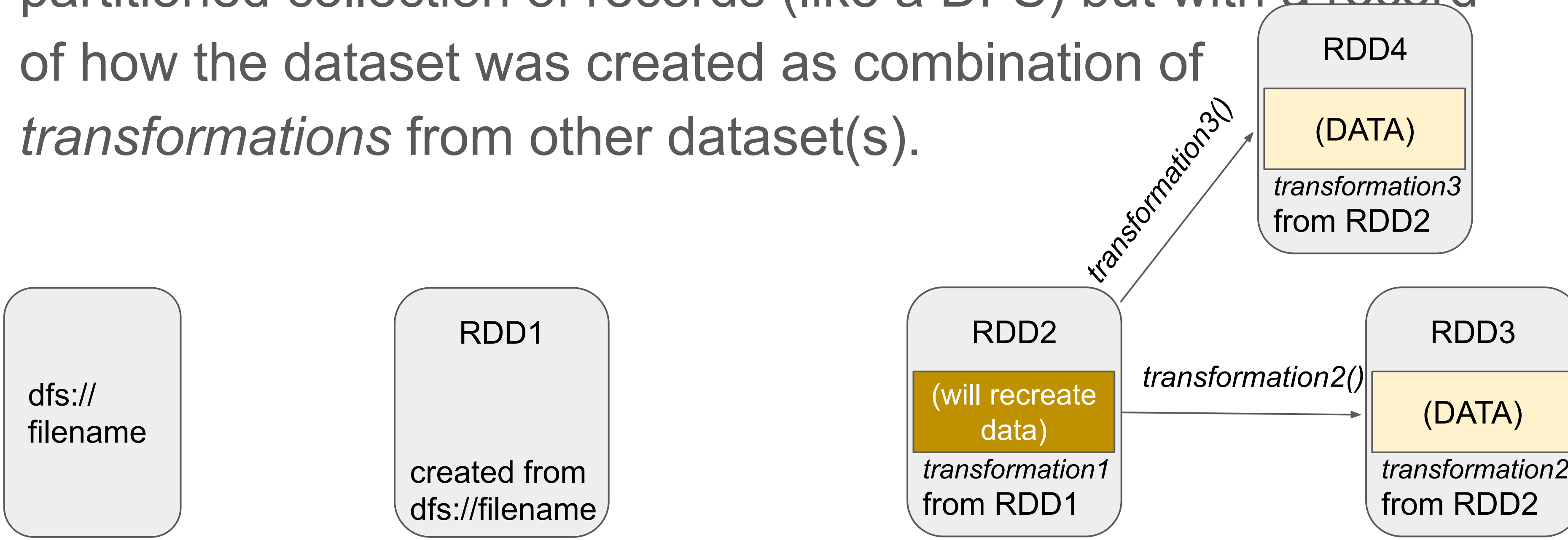
Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

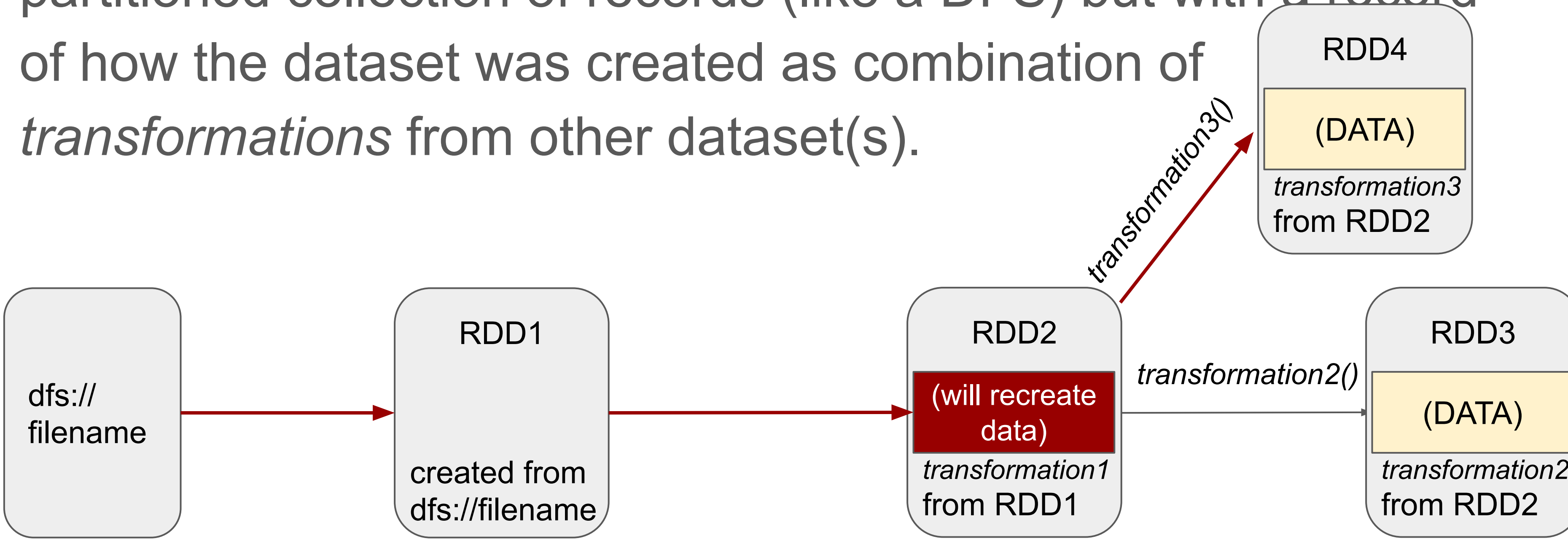
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



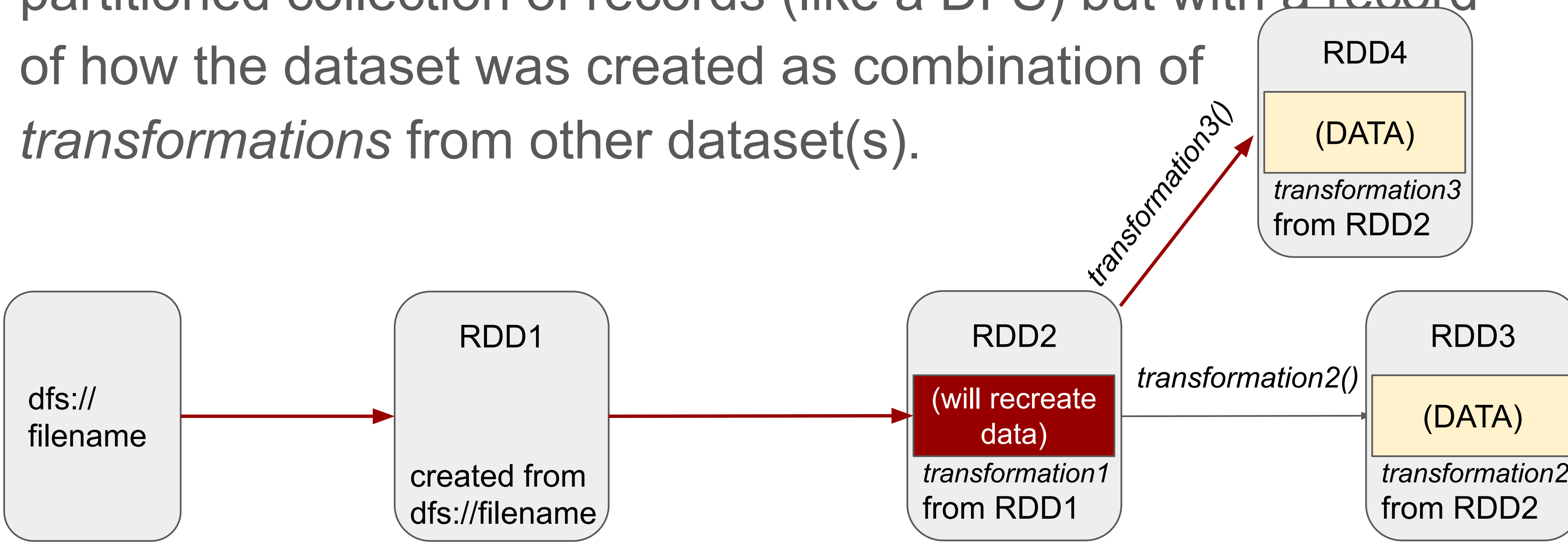
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



(original) *Transformations*: RDD to RDD

Transformations	$map(f: \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f: \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f: \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction: \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f: (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f: \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c: \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p: \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

(original) *Transformations*: RDD to RDD

Transformations	$map(f : \mathrm{T} \Rightarrow \underline{\mathrm{U}})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \underline{\mathrm{Seq[U]}})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	Multiple Records $join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.”. *NSDI 2012*. April 2012.

(original) *Transformations*: RDD to RDD

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.”. *NSDI 2012*. April 2012.

(original) *Transformations*: RDD to RDD

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]

(orig.) *Actions*: RDD to Value Object, or Storage

Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.”. *NSDI 2012*. April 2012.

Current Transformations and Actions

http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

common transformations: *filter, map, flatMap, reduceByKey, groupByKey*

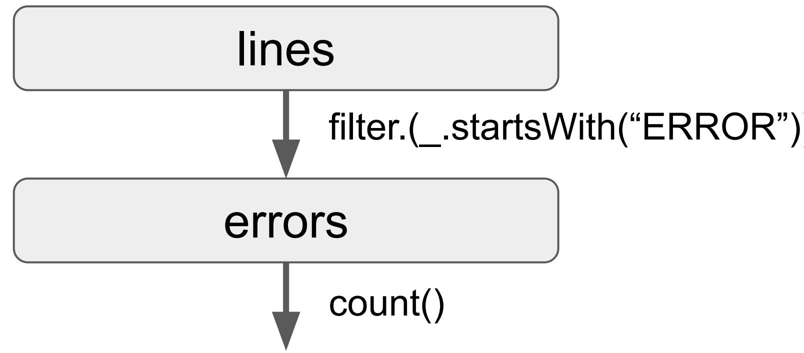
http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

common actions: *collect, count, take*

Example

Count errors in a log file:

TYPE MESSAGE TIME



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

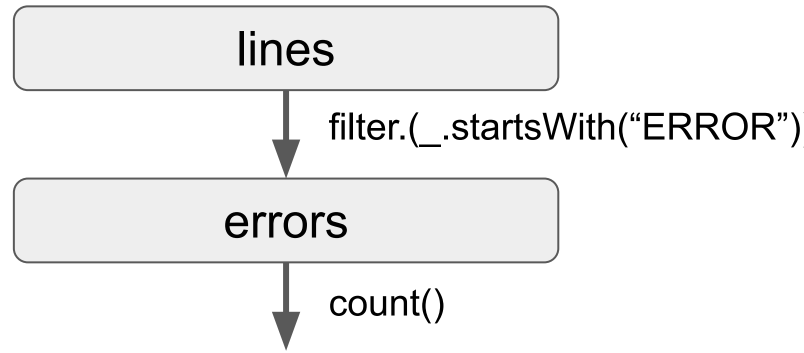
Example

Count errors in a log file:

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")
errors =
    lines.filter(_.startswith("ERROR"))
errors.count
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

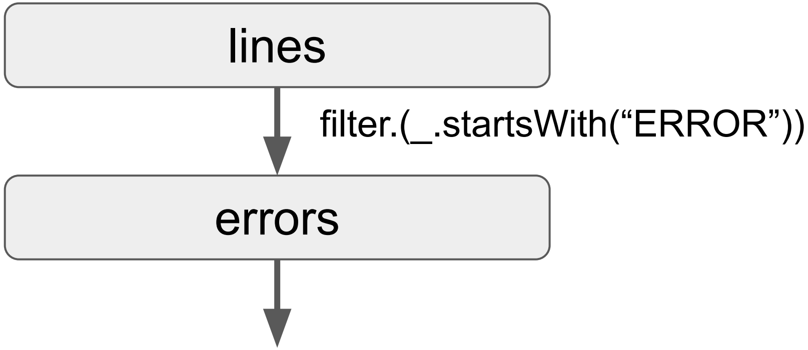
Example 2

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")
errors =
    lines.filter(_.startswith("ERROR"))
errors.persist
~~errors.count~~
...
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

Example 2

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")
errors =
    lines.filter(_.startswith("ERROR"))
errors.persist
~~errors.count~~
...
```

Persistance

Can specify that an RDD "persists" in memory so other queries can use it.
Can specify a priority for persistance; lower priority => moves to disk, if needed, earlier

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

Example 2

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")
errors =
    lines.filter(_.startswith("ERROR"))
errors.persist
~~errors.count~~
...
```

Persistance

Can specify that an RDD "persists" in memory so other queries can use it.

Can specify a priority for persistance; lower priority => moves to disk, if needed, earlier

[parameters for persist]

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

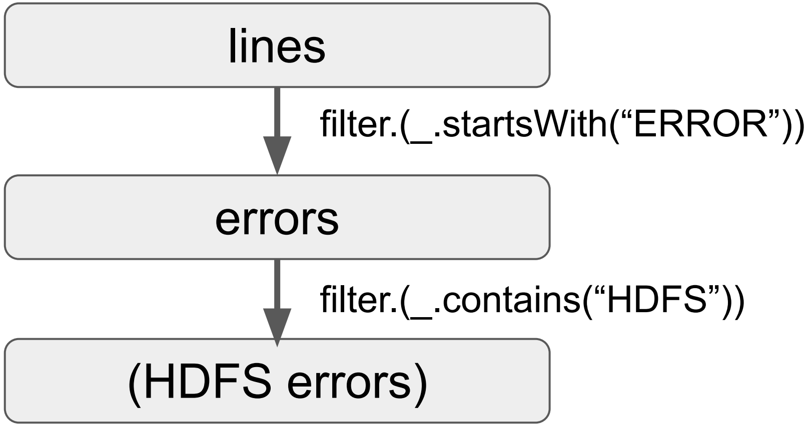
Example

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")
errors =
    lines.filter(_.startswith("ERROR"))
errors.persist
errors.count
errors.filter(_.contains("HDFS"))
      ...
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

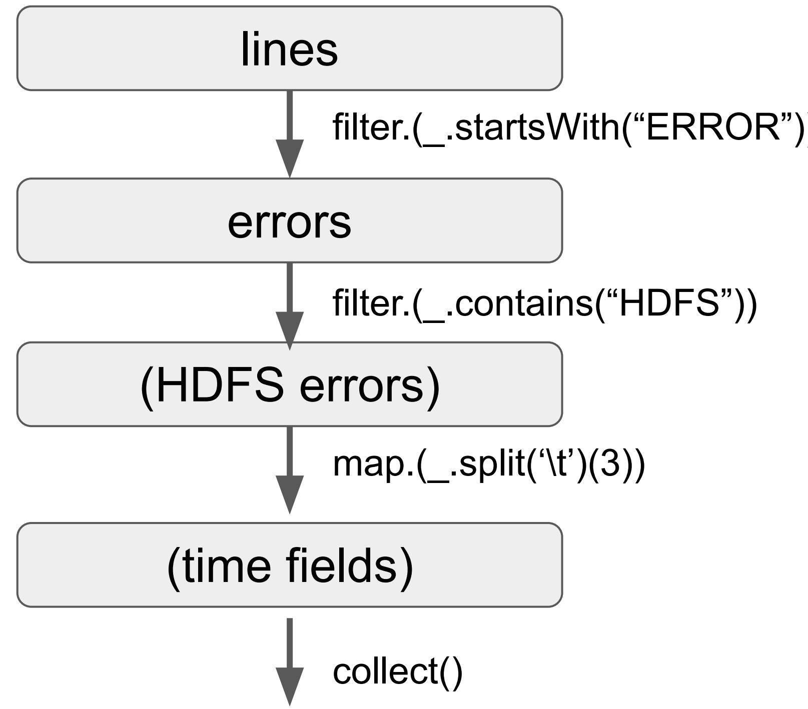
Example

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile(“dfs:...”)
errors =
    lines.filter(_.startswith(“ERROR”))
errors.persist
errors.count
errors.filter(_.contains(“HDFS”))
     .map(_split(‘\t’)(3))
     .collect()
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.”. *NSDI 2012*. April 2012.

Example

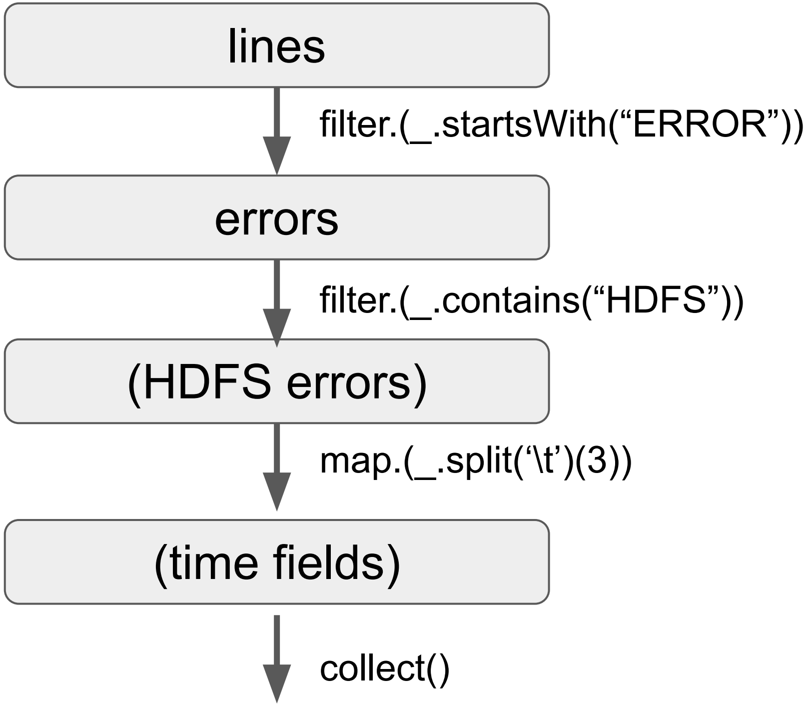
Collect times of hdfs-related errors

TYPE   MESSAGE   TIME

Pseudocode:

```
lines = sc.textFile(“dfs:...”)
errors =
    lines.filter(_.startswith(“ERROR”))
errors.persist
errors.count
errors.filter(_.contains(“HDFS”))
     .map(_split(‘\t’)(3))
     .collect()
```

Functional Programming



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.”. *NSDI 2012*. April 2012.

Example

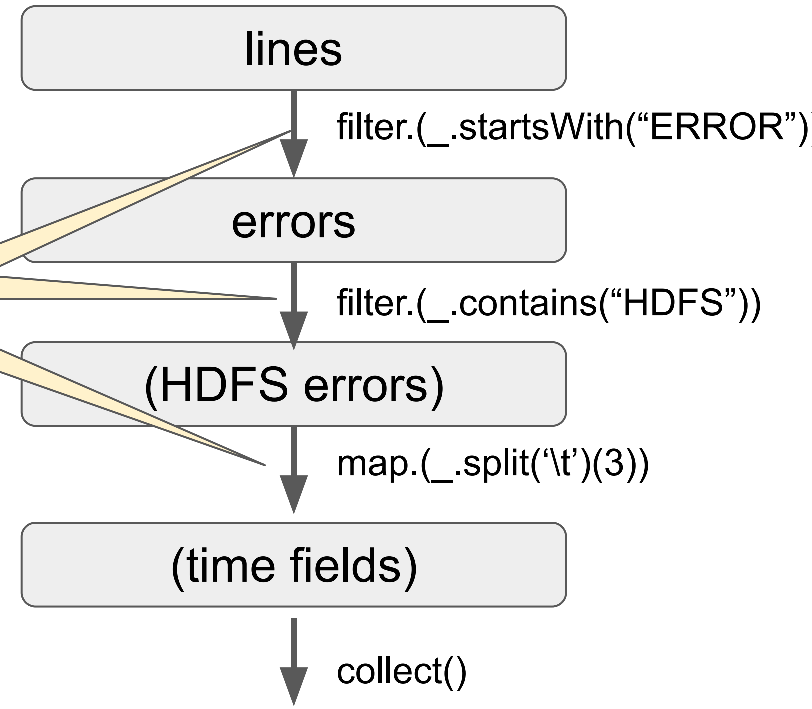
Collect times of hdfs-related errors

TYPE   MESSAGE   TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")
errors =
    lines.filter(_.startswith("ERROR"))
errors.persist
errors.count
errors.filter(_.contains("HDFS"))
     .map(_split('\t')(3))
     .collect()
```

Functional Programming



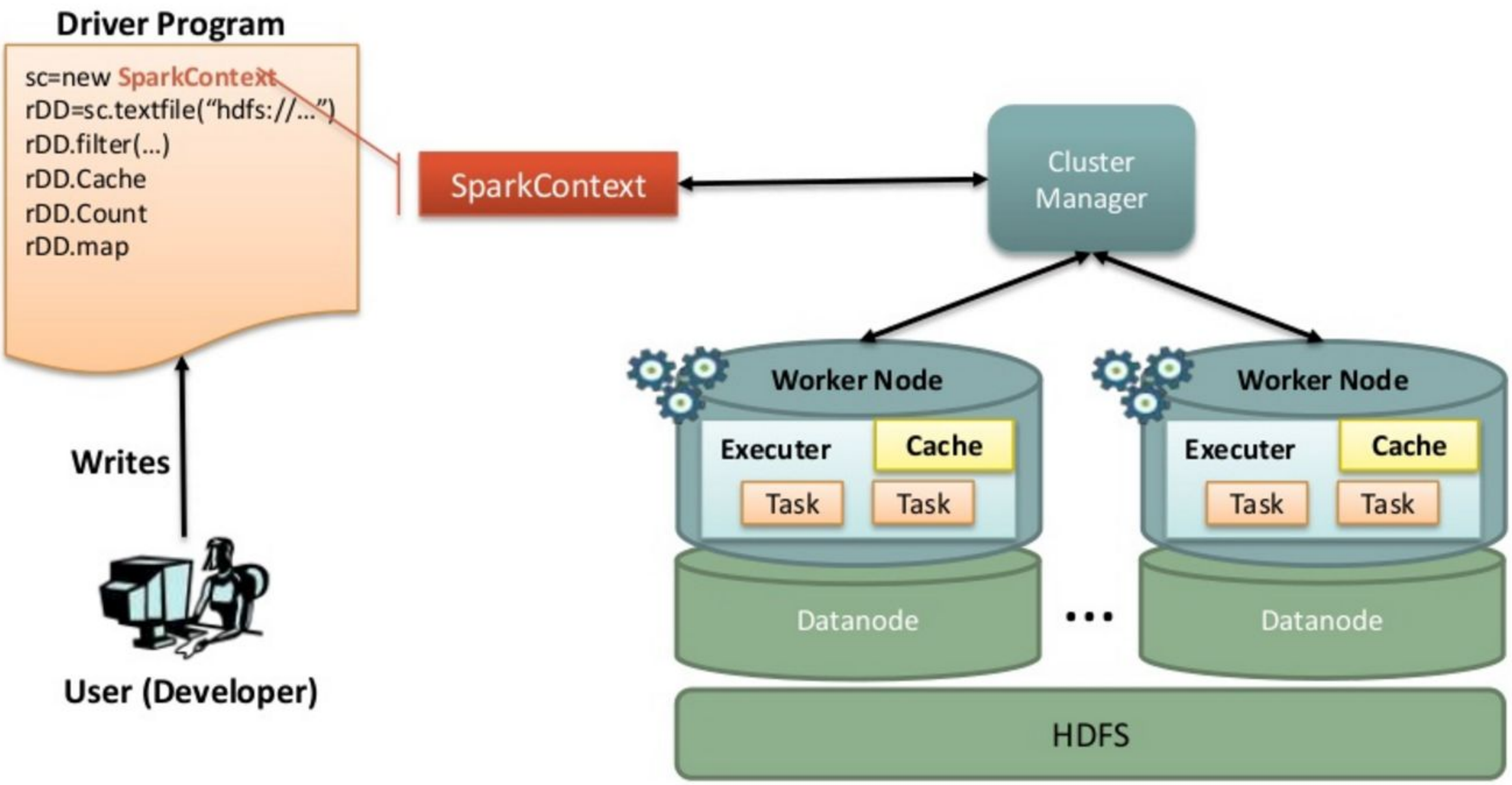
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". *NSDI 2012*. April 2012.

Advantages as Workflow System

- More efficient failure recovery
- More efficient grouping of tasks and scheduling
- Integration of programming language features:
  - loops (not a “cyclic” workflow system).
  - function libraries

(MMDSv3)

The Spark Programming Model



Gupta, Manish. Lightening Fast Big Data Analytics using Apache Spark. *UniCom 2014.*

Example

Word Count

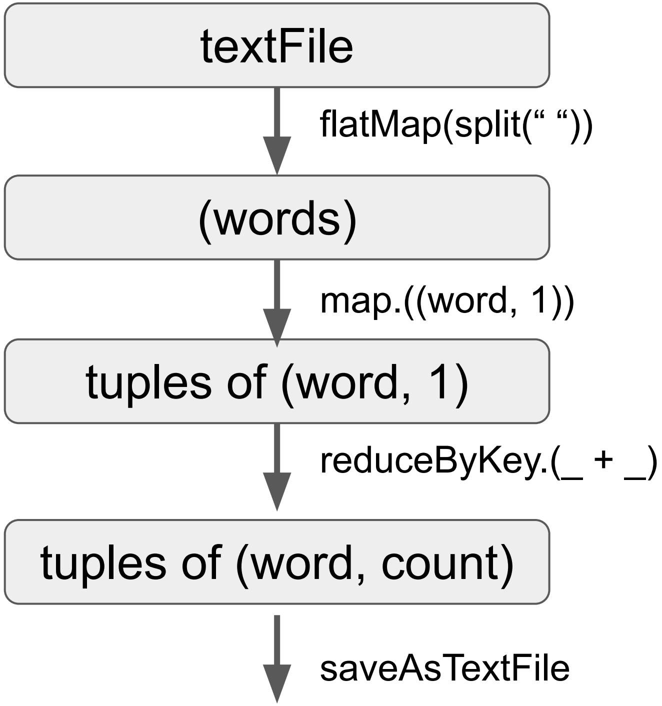
textFile

Example

Word Count

Scala:

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



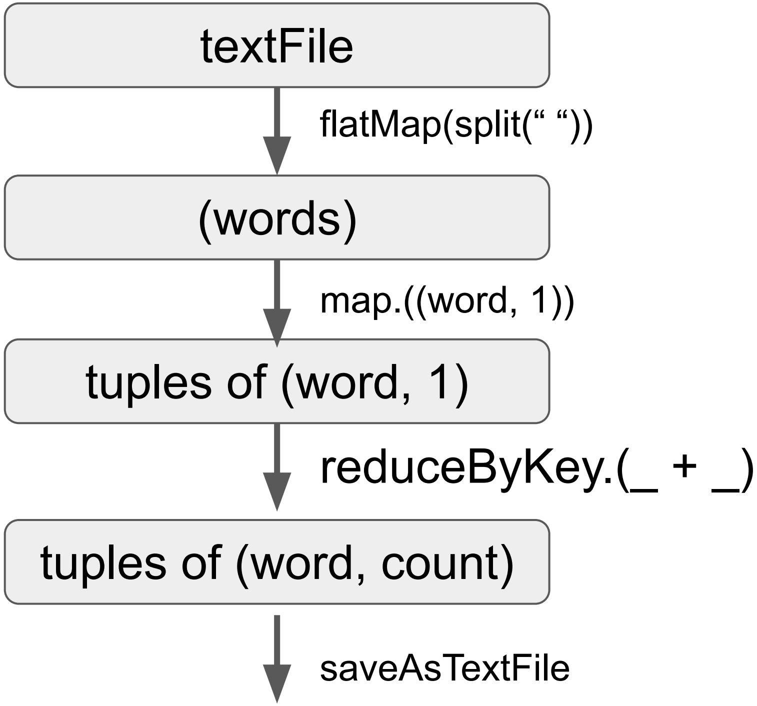
Apache Spark Examples
http://spark.apache.org/examples.html

Example

Word Count

Python:

```
textFile = sc.textFile("hdfs://...")
counts = textFile
    .flatMap(lambda line: line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```



Apache Spark Examples
http://spark.apache.org/examples.html

PySpark Demo

https://data.worldbank.org/data-catalog/poverty-and-equity-database

Lazy Evaluation

Spark waits to **load data** and **execute transformations** until necessary -- ***lazy***
Spark tries to complete **actions** as immediately as possible -- ***eager***

Why?

- Only executes what is necessary to achieve action.
- Can optimize the complete *chain of operations* to reduce communication

Lazy Evaluation

Spark waits to *load data* and *execute transformations* until necessary -- ***lazy***
Spark tries to complete actions as quickly as possible -- ***eager***

Why?

- Only executes what is necessary to achieve action.
- Can optimize the complete *chain of operations* to reduce communication

e.g.

```
rdd.map(lambda r: r[1]*r[3]).take(5)  #only executes map for five records

rdd.filter(lambda r: "ERROR" in r[0]).map(lambda r: r[1]*r[3])
                        #only passes through the data once
```

Broadcast Variables

Read-only objects can be shared across all nodes.

Broadcast variable is a wrapper: access object with .value

Python:

```
filterWords = [‘one’, ‘two’, ‘three’, ‘four’, …]
fwBC = sc.broadcast(set(filterWords))
```

Broadcast Variables

Read-only objects can be shared across all nodes.

Broadcast variable is a wrapper: access object with .value

Python:

```
filterWords = [‘one’, ‘two’, ‘three’, ‘four’, …]
fwBC = sc.broadcast(set(filterWords))

textFile = sc.textFile("hdfs:...")
counts = textFile
    .map(lambda line: line.split(" "))
    .filter(lambda words: len(set(words) and word in fwBC.value) > 0)
    .flatMap(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs:...")
```

Accumulators

Write-only objects that keep a running aggregation

- Default Accumulator assumes sum function

```
initialValue = 0
sumAcc = sc.accumulator(initialValue)
rdd.foreach(lambda i: sumAcc.add(i))
print(sumAcc.value)
```

Accumulators

Write-only objects that keep a running aggregation

Default Accumulator assumes sum function

Custom Accumulator: Inherit (AccumulatorParam) as class and override methods

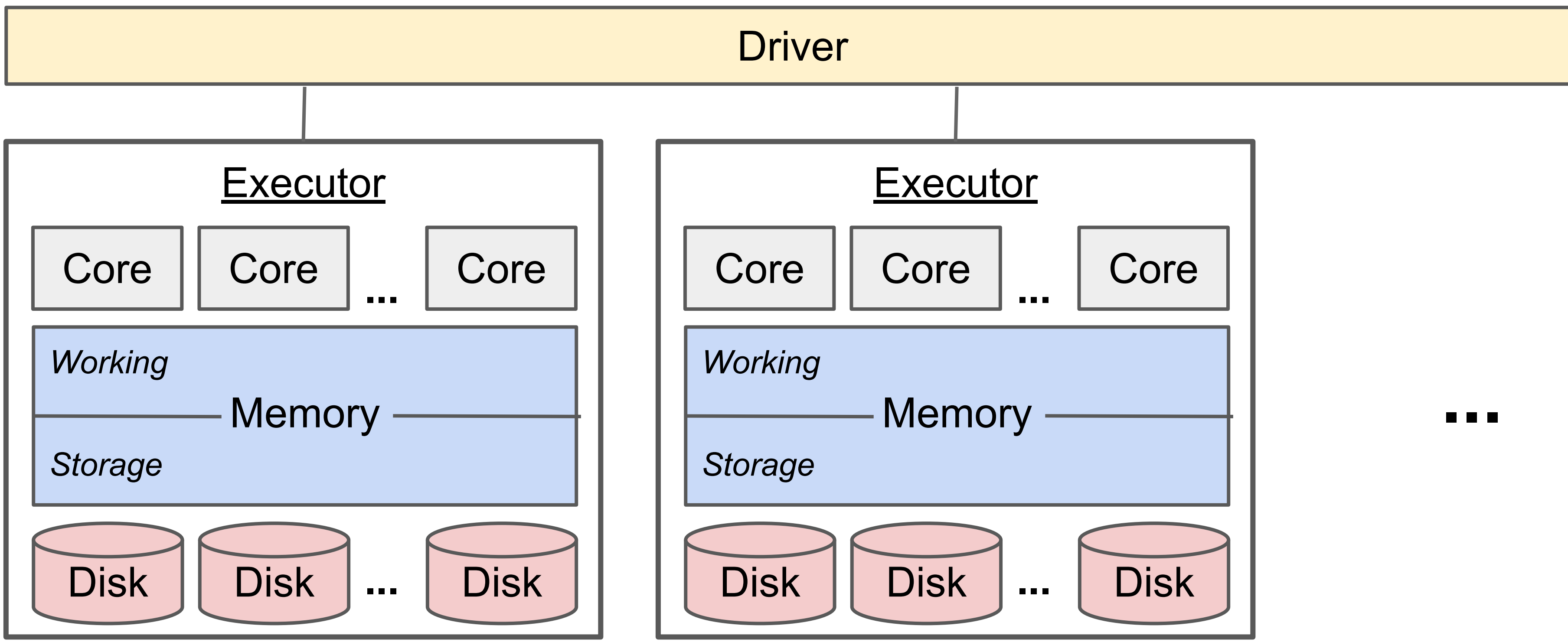
```
initialValue = 0
sumAcc = sc.accumulator(initialValue)
rdd.foreeach(lambda i: sumAcc.add(i))
print(minAcc.value)

class MinAccum(AccumulatorParam):
	def zero(self, zeroValue = np.inf):#overwrite this
		return zeroValue
	def addInPlace(self, v1, v2):#overwrite this
		return min(v1, v2)
minAcc = sc.accumulator(np.inf, minAccum())
rdd.foreeach(lambda i: minAcc.add(i))
print(minAcc.value)
```

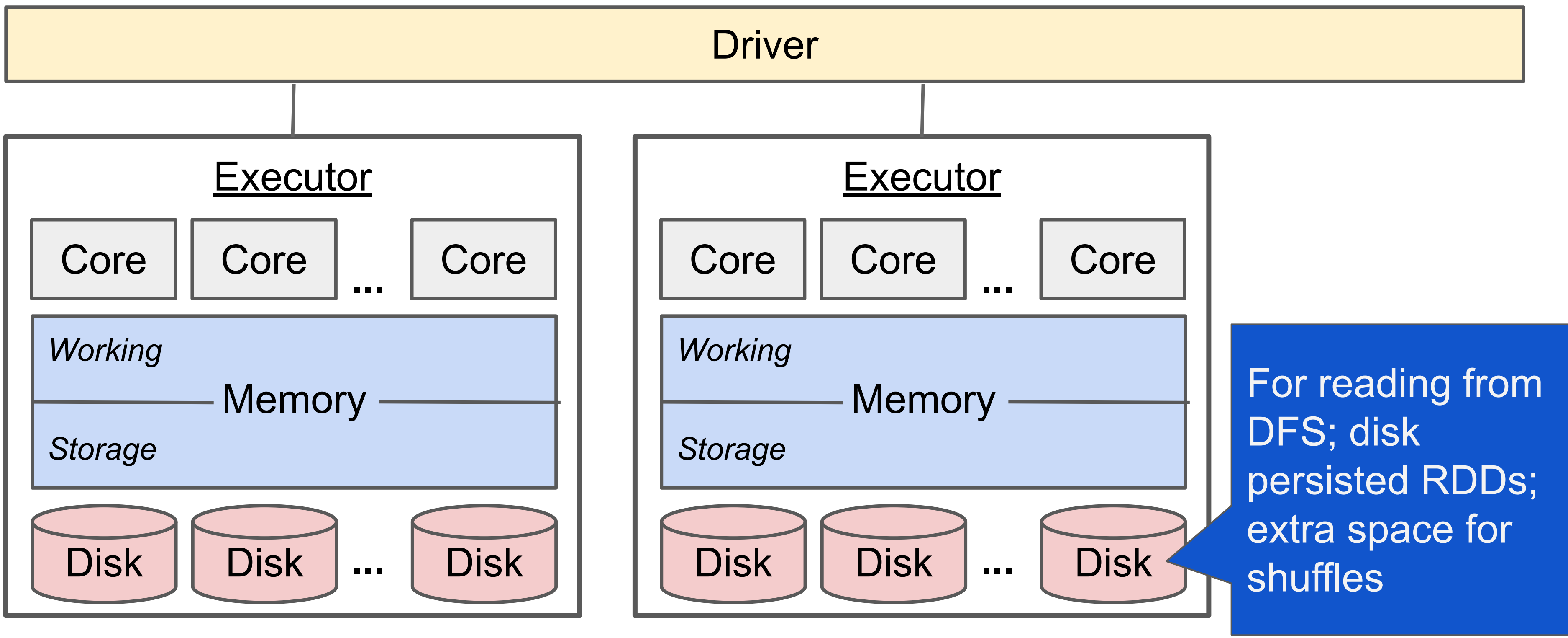
Spark System: Review

- RDD provides full recovery by backing up transformations from stable storage rather than backing up the data itself.
- RDDs, which are immutable, can be stored in memory and thus are often much faster.
- Functional programming is used to define transformation and actions on RDDs.

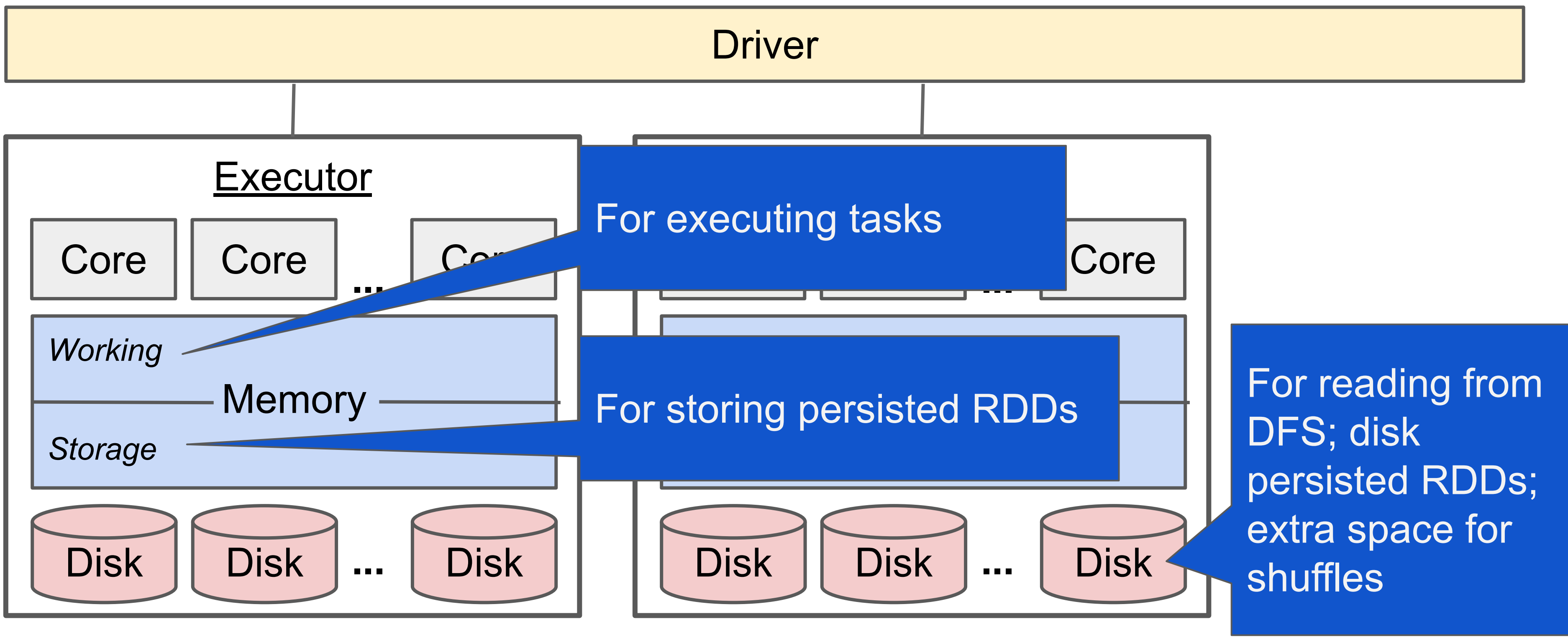
Spark System: Hierarchy



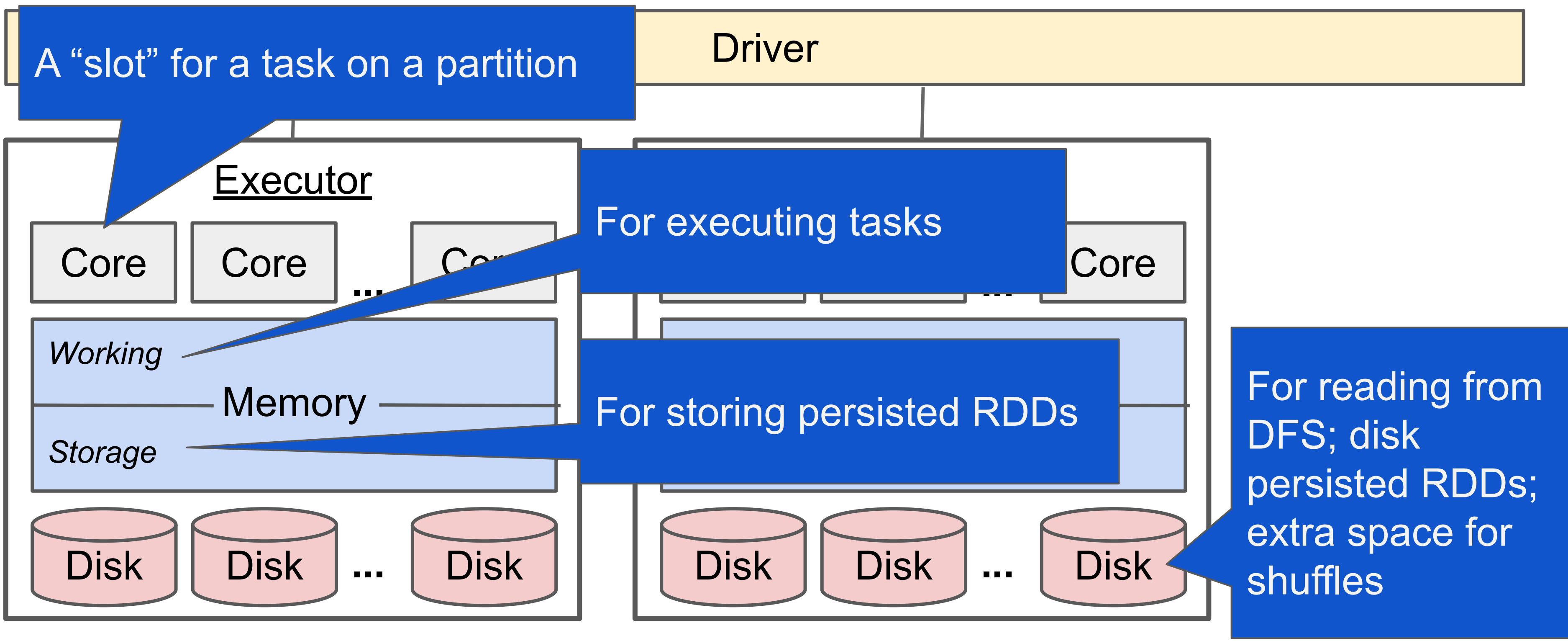
Spark System: Hierarchy



Spark System: Hierarchy

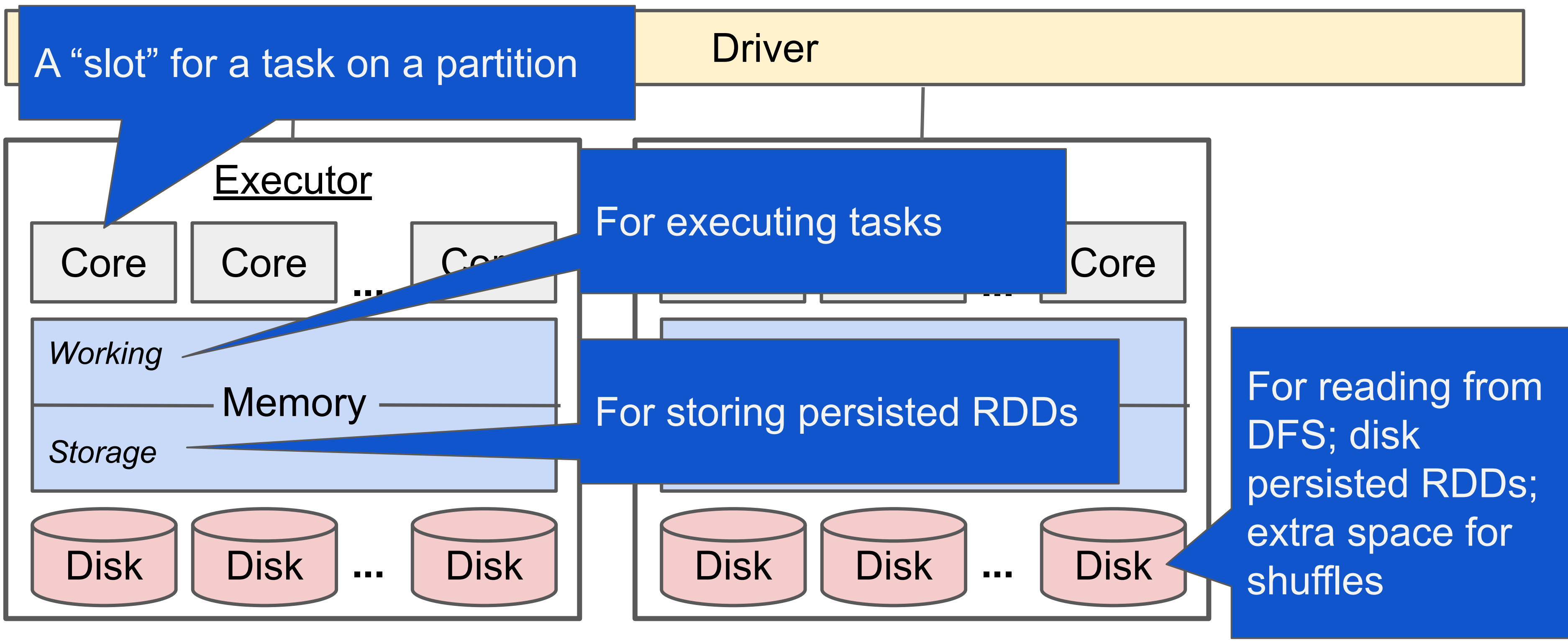


Spark System: Hierarchy



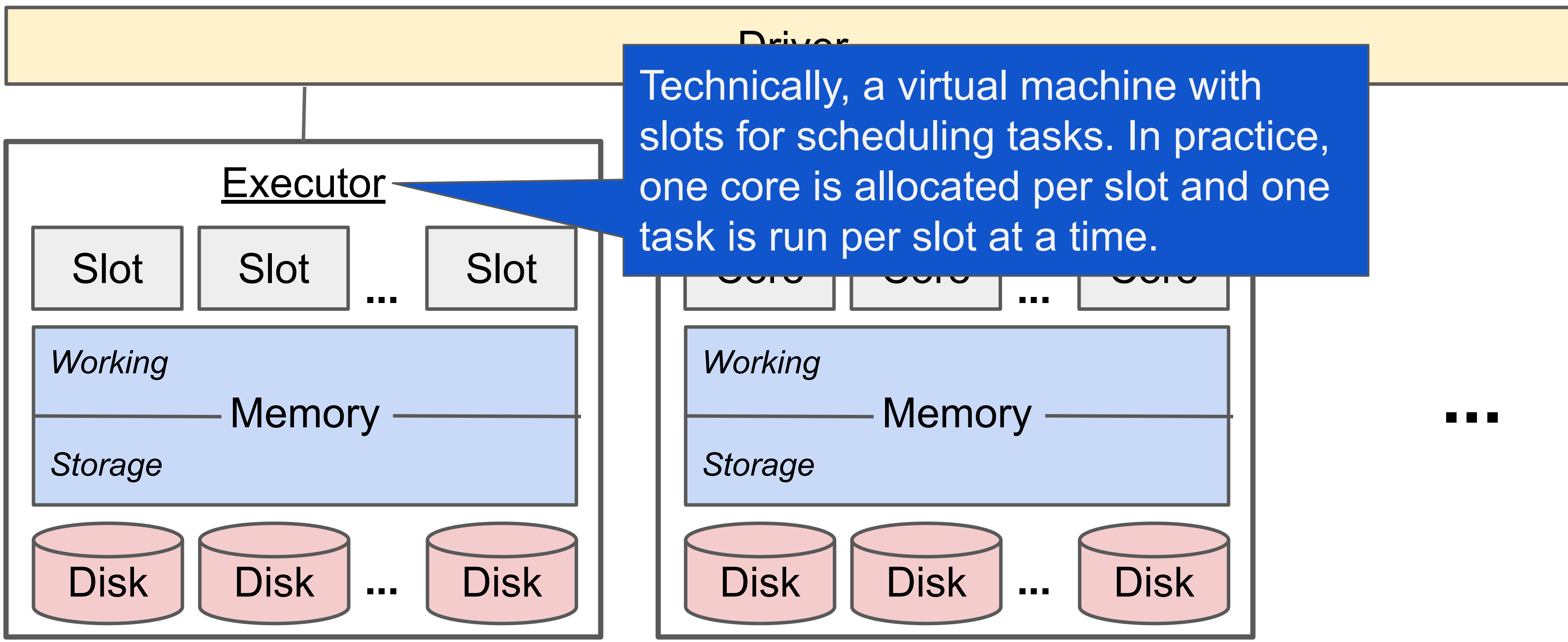
Spark System: Hierarchy

Eager *action* -> sets off (lazy) chain of *transformations*
-> launches *jobs* -> broken into *stages* -> broken into ***tasks***



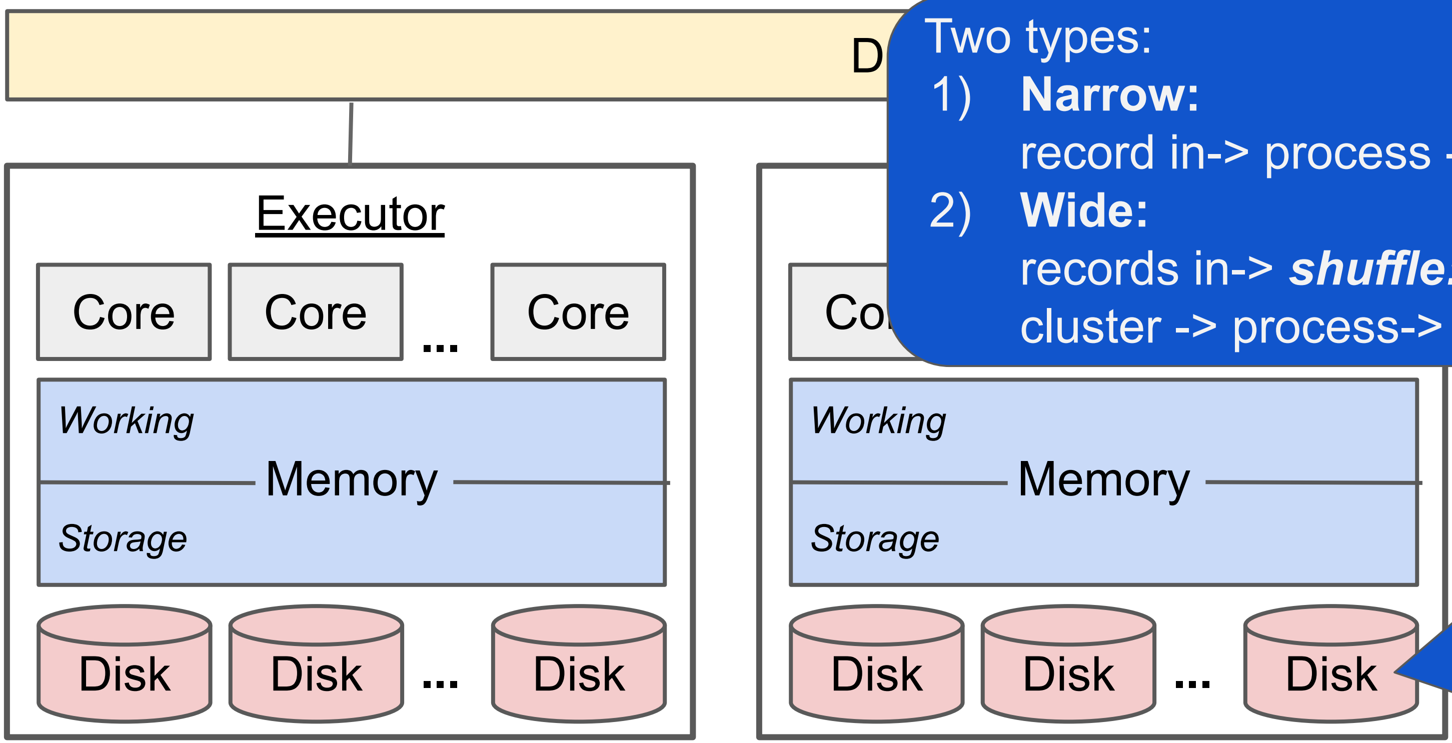
Spark System: Hierarchy

Eager *action* -> sets off (lazy) chain of *transformations*
-> launches *jobs* -> broken into *stages* -> broken into ***tasks***



Spark System: Hierarchy

Eager *action* -> sets off (lazy) chain of ***transformations***
-> launches *jobs* -> broken into *stages* -> broken into *tasks*



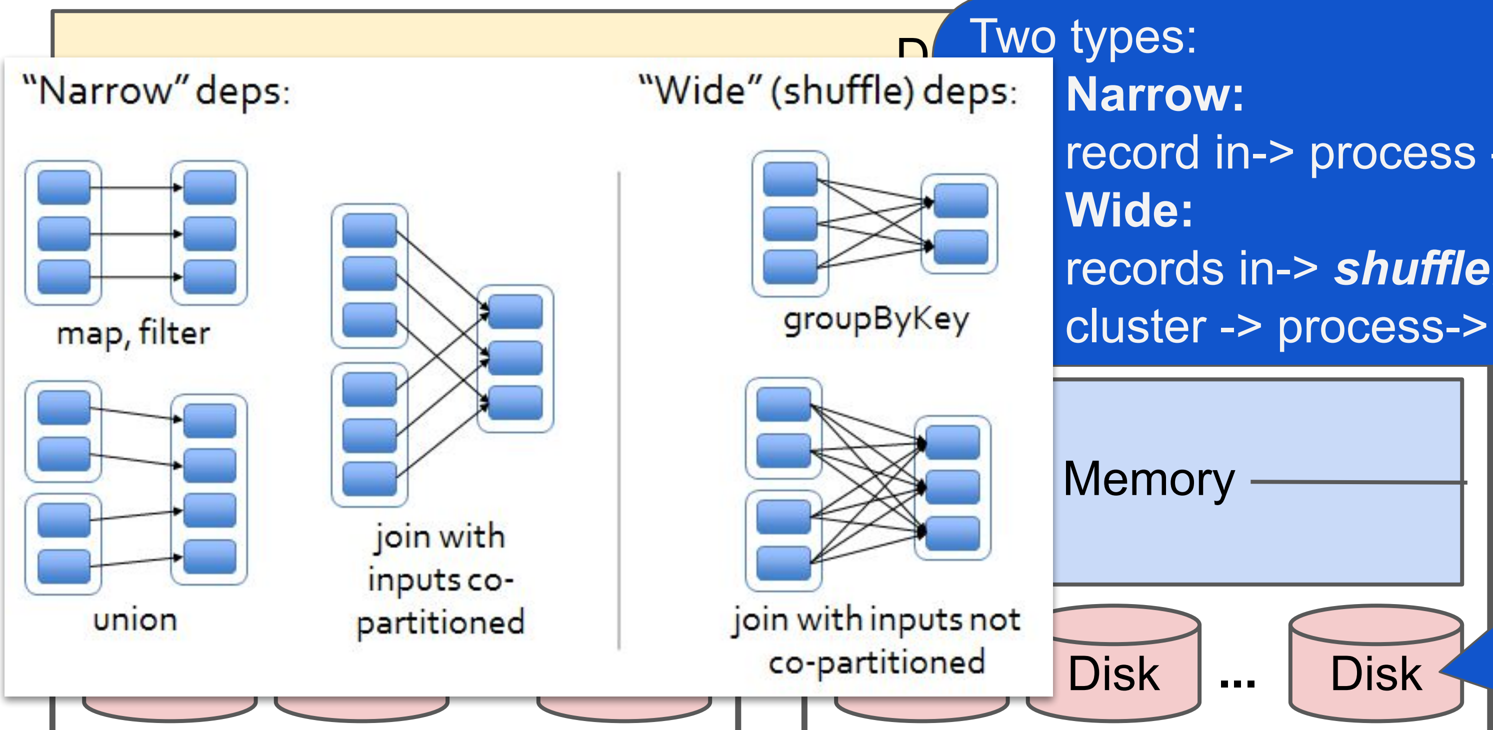
Two types:

1) **Narrow:**
record in-> process -> record[s] out
2) **Wide:**
records in-> ***shuffle:*** regroup across cluster -> process-> record[s] out

For reading from DFS; disk persisted RDDs; extra space for ***shuffles***

Spark System: Hierarchy

Eager *action* -> sets off (lazy) chain of ***transformations***
-> launches *jobs* -> broken into *stages* -> broken into *tasks*



Two types:
Narrow:
record in-> process -> record[s] out
Wide:
records in-> ***shuffle:*** regroup across cluster -> process-> record[s] out

For reading from DFS; disk persisted RDDs; extra space for ***shuffles***

Image from Nguyen: https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/

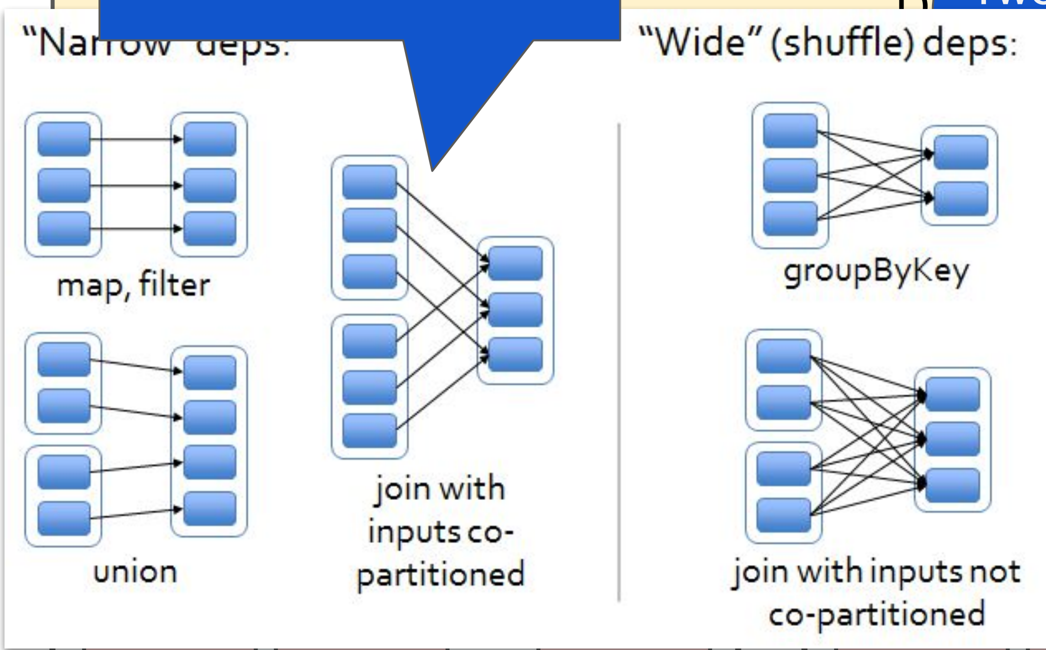
Spark System: Hierarchy

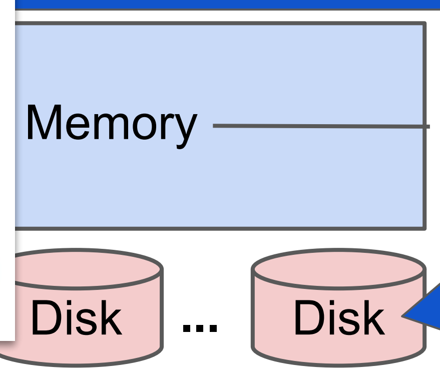
Co-partitions:
If the partitions for two RDDs are based on the same hash function and key.

Ea... y) chain of ***transformations***

...es *jobs* -> broken int... *stages* -> broken into *tasks*

Two types:
Narrow:
record in-> process -> record[s] out
Wide:
records in-> ***shuffle:*** regroup across cluster -> process-> record[s] out





For reading from DFS; disk persisted RDDs; extra space for ***shuffles***

Image from Nguyen: https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/

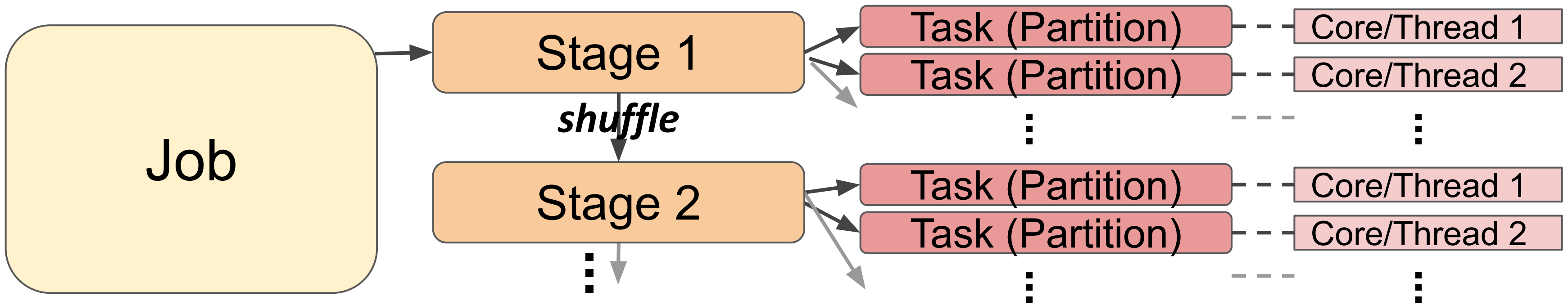
Spark System: Scheduling

Eager *action* -> sets off (lazy) chain of *transformations*
-> launches *jobs* -> broken into *stages* -> broken into *tasks*

Jobs: A series of transformations (in a DAG) needed for the action

Stages: 1 or more per job -- 1 per set of operations separated by shuffle

Tasks: many per stage -- repeats exact same operation per partition



Spark System: Scheduling

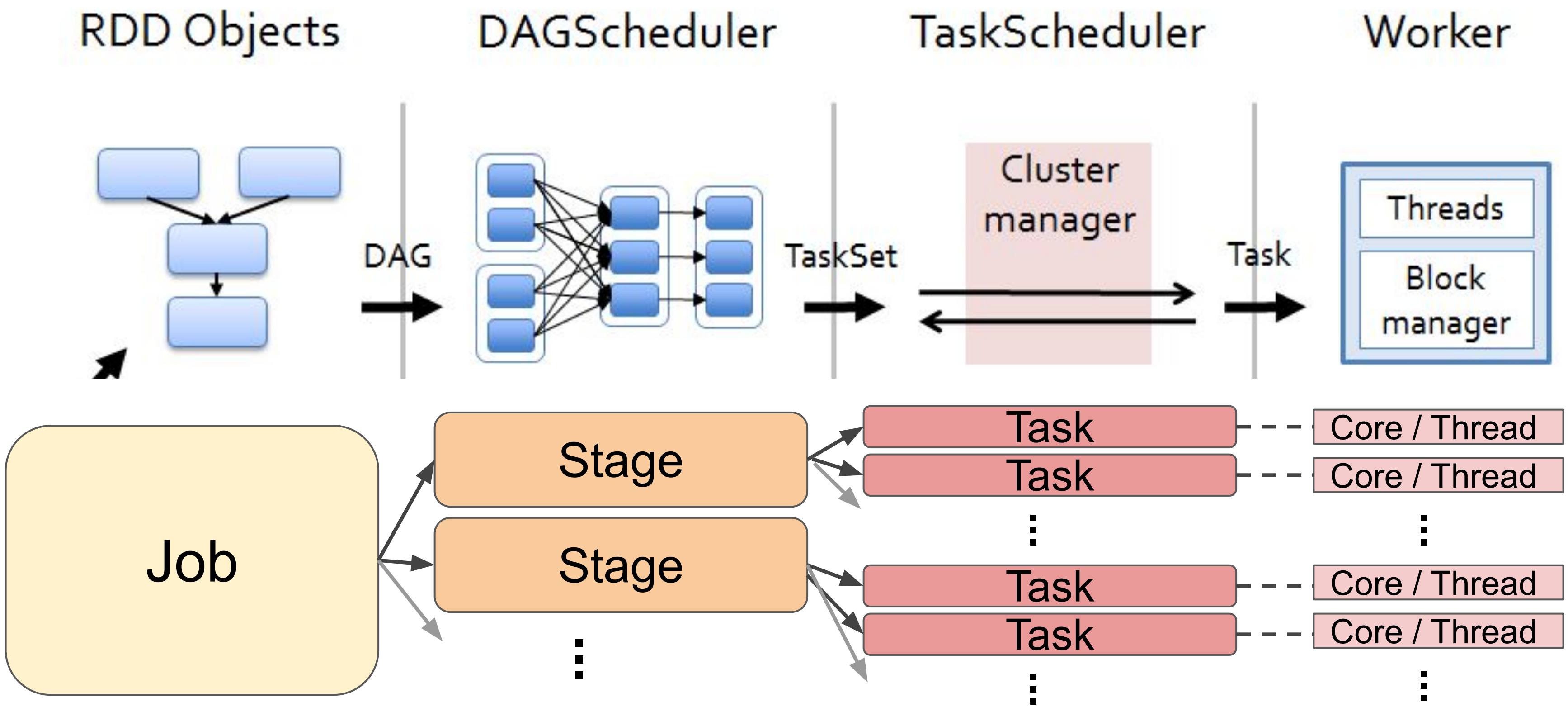


Image from Nguyen: https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/

Spark System: Scheduling

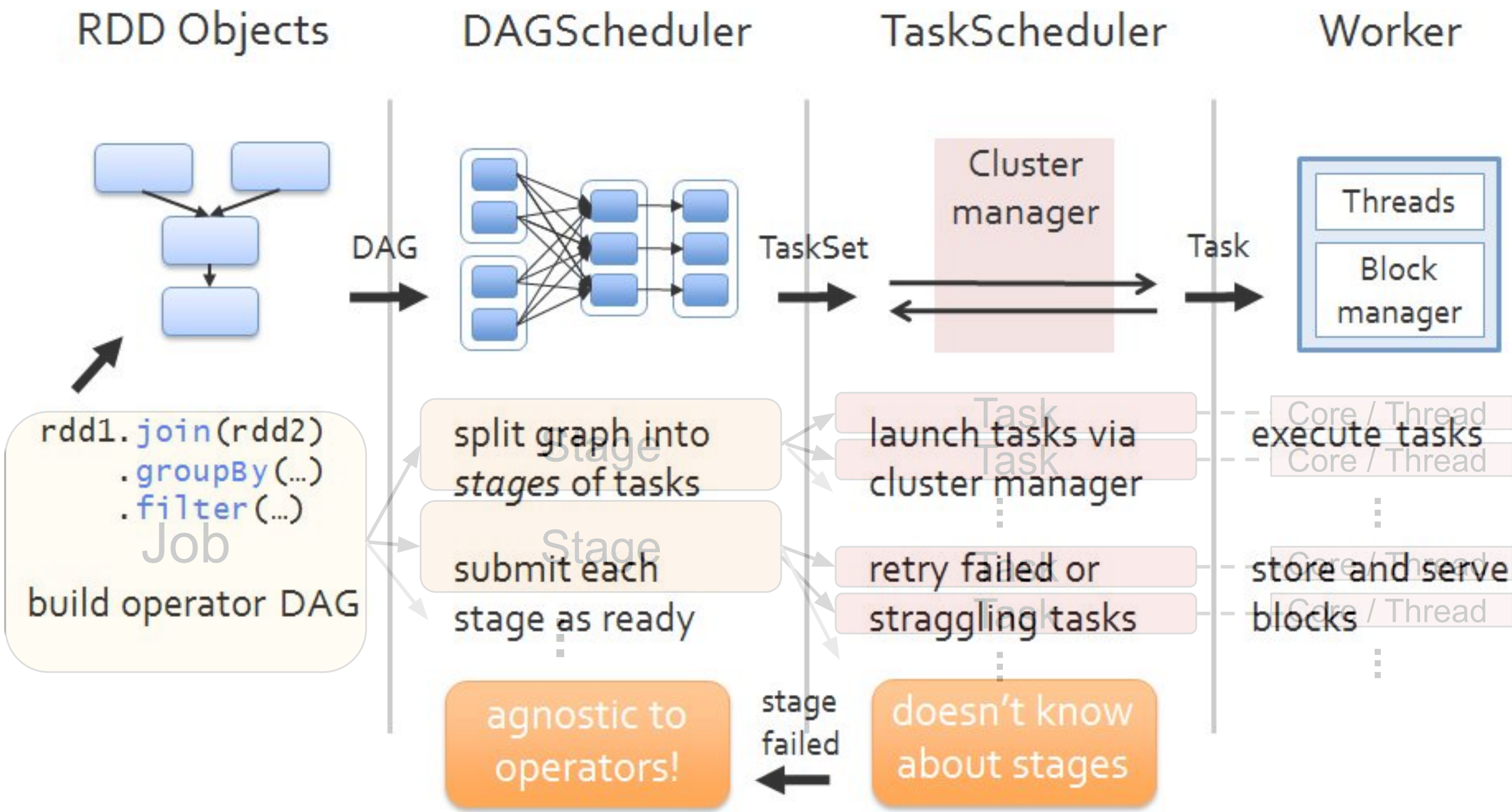


Image from Nguyen: https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/

MapReduce or Spark?

- Spark is typically faster
  - RDDs in memory
  - Lazy evaluation enables optimizing chain of operations.
- Spark is typically more flexible (custom chains of transformations)

MapReduce or Spark?

- Spark is typically faster
  - RDDs in memory
  - Lazy evaluation enables optimizing chain of operations.
- Spark is typically more flexible (custom chains of transformations)

However:

- Still need Hadoop (or some DFS) to hold original or resulting data efficiently and reliably.
- Memory across Spark cluster should be large enough to hold entire dataset to fully leverage speed.

Thus, MapReduce may sometimes be more cost-effective for very large data that does not fit in memory.