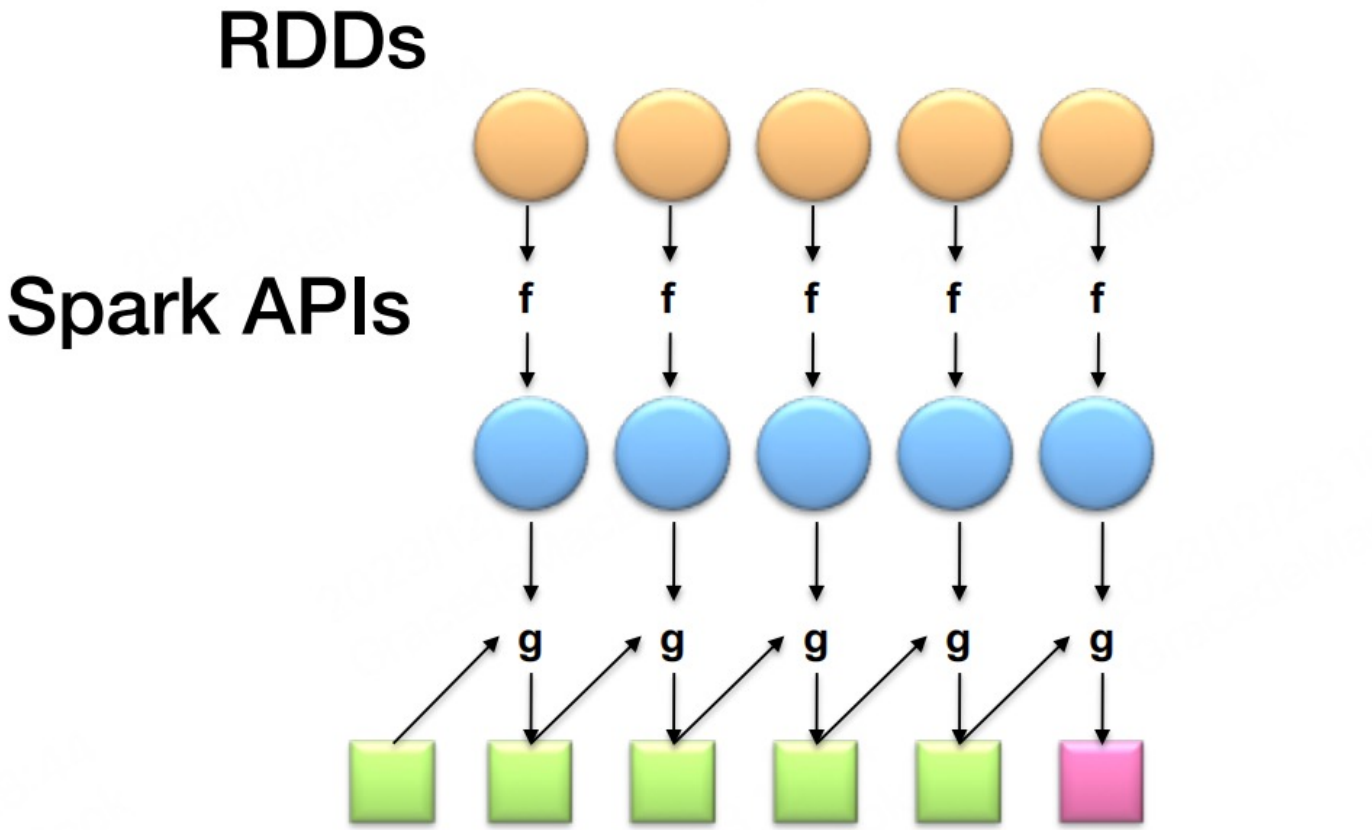
Cloud Computing
RDD and Spark APIs

Minchen Yu
SDS@CUHK-SZ
Fall 2024

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF
DATA SCIENCE
數據科學學院

Roots in functional programming



What's an RDD?

Resilient Distributed Dataset

= immutable = partitioned

Developers define *transformations* on RDDs

Framework keeps track of *lineage*

RDDs

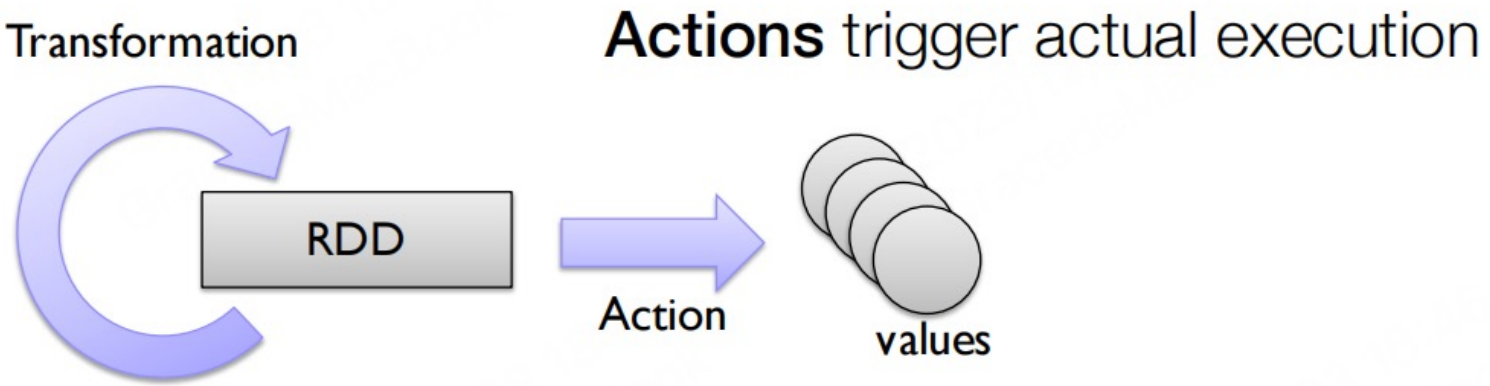
Restricted form of distributed shared memory

- Immutable, partitioned collections of records
- Built through coarse-grained, deterministic transformations (map, filter, join, …)

Efficient fault recovery using lineage

- Log one operation to apply to many elements
- Recompute lost partitions on failure
- No cost if nothing fails

RDD lifecycle



Transformations are *lazy*:
Framework keeps track of *lineage*

Working with RDDs

Create an RDD from a data source

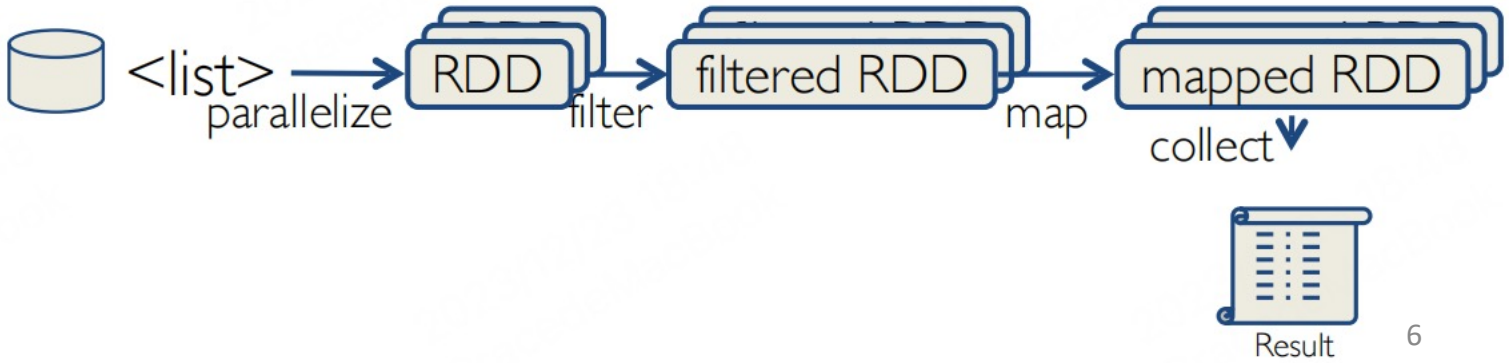
- HDFS, S3, Python collections, etc.

Apply transformations to an RDD

- map, flatMap, filter, etc.

Apply actions to an RDD

- collect, count, etc.



Creating RDDs

Create RDDs from Python collections

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> rDD = sc.parallelize(data, 4)
>>> rDD

ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

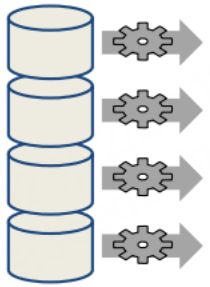
# of partitions

No computation so far! Spark only records how to create the RDD with four partitions

Creating RDDs

From HDFS, text files, S3, HBase, any Hadoop **InputFormat**, and directory or glob wildcard: **/data/201703***

```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
MappedRDD[2] at textFile at
   NativeMethodAccessorImpl.java:-2
```



RDD distributed in 4 partitions

Elements are lines of input

No RDD materialization due to *lazy evaluation*

Transformations

Create new datasets from an existing one

Use *lazy evaluation*

- results not computed right away
- provide optimization opportunities for required calculations
- recovers from failures and slow workers

Think of this as a recipe for creating results!

Some 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
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`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)

Transformation examples

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]


>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]


>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

For illustrative purpose only
Nothing executed right away

distinct() can be expensive, why?

map vs. **flatMap**

map: *one-to-one* mapping

flatMap: *one-to-many* mapping

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Function literals are closures

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]


>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]


>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

Function literals (green) are closures

- automatically shipped to workers!
- more to come…

Spark Key-Value RDDs

Similar to MapReduce, Spark supports Key-Value pairs

Each element of a Pair RDD is a pair tuple

```
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```

Some Key-Value Transformations

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

Why is **reduceByKey** a transformation but

reduce an action?

reduceByKey as a transformation

Operates on RDDs of key-value pairs

- a member function of class **PairRDDFunctions[K, V]**
- output: **Map<Key, Value>**

The output could potentially be a large distributed data collection subject to further operations

reduce

- clear semantics of aggregating/combining all elements

Key-Value transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]
```

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()

RDD: [(1,'a'), (2,'c'), (1,'b')] →
                [(1,'a'), (1,'b'), (2,'c')]
```

Key-Value transformations

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
                [(1,['a','b']), (2,['c'])]
```

Be careful using **groupByKey()** as it shuffles data across network and creates large iterables at workers!

Actions

Cause Spark to execute recipe to transform source

- mechanism for getting results out of Spark

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

Getting data out of RDDs

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```

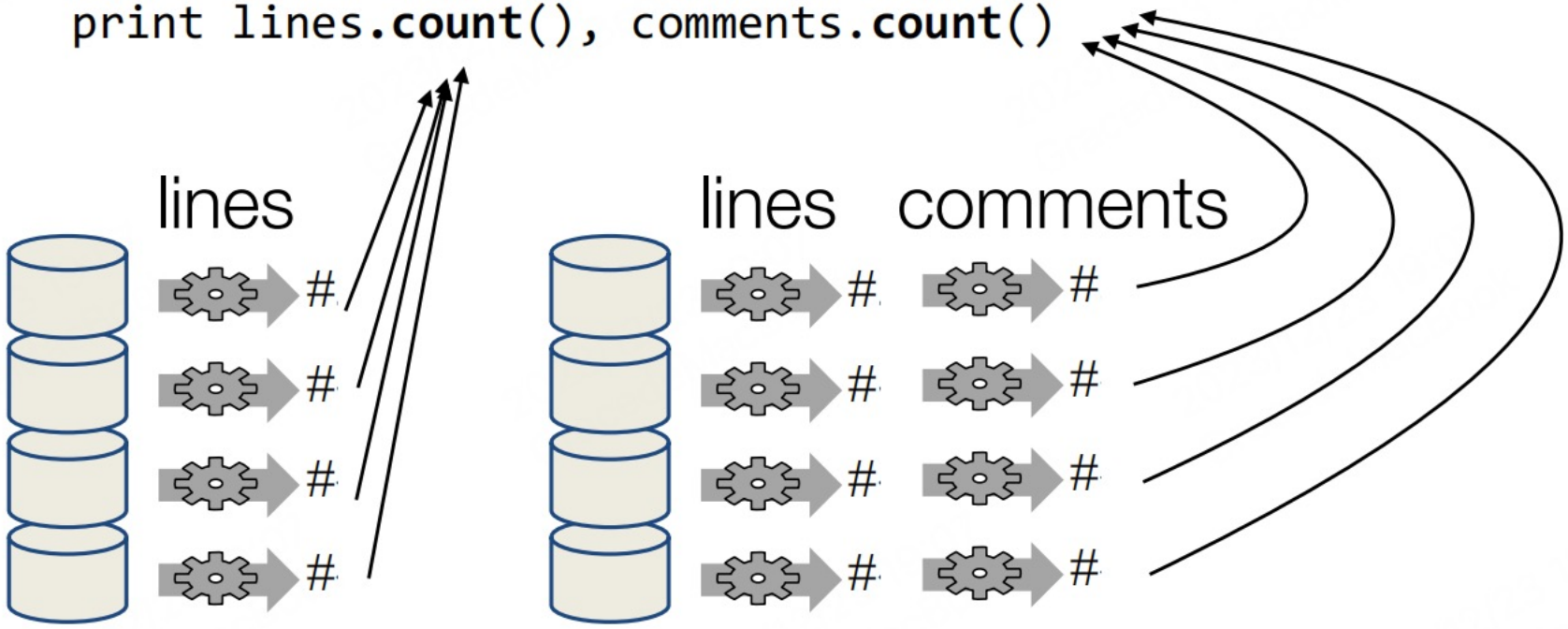
Getting data out of RDDs

```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list
```

`takeOrdered(n, key=func)`	return n elements ordered in ascending order or as specified by the optional key function

Getting data out of RDDs

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

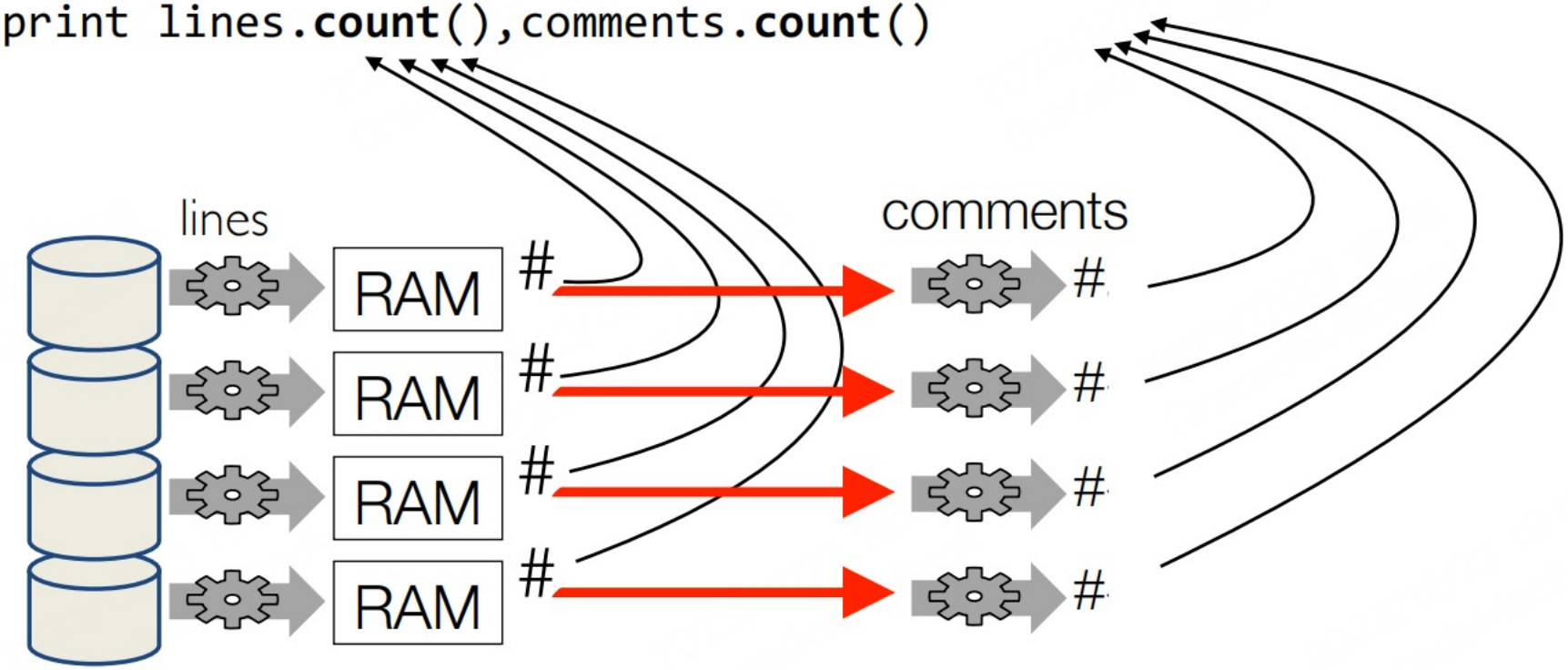


Spark recomputes lines from scratch!
How to fix it?

Cache RDDs in memory to avoid recomputation!

Let’s do caching

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(),comments.count()
```



Spark program lifecycle w/ RDDs

1. Create RDDs from external data or parallelize() a collection in your driver

2. Lazily transform them into new RDDs

3. **cache()** some RDDs for reuse

4. Perform actions to execute parallel computation and produce results

Let’s revisit reduce-like operators

reduce(), reduceByKey(), aggregateByKey(), …

Revisiting **reduce()**

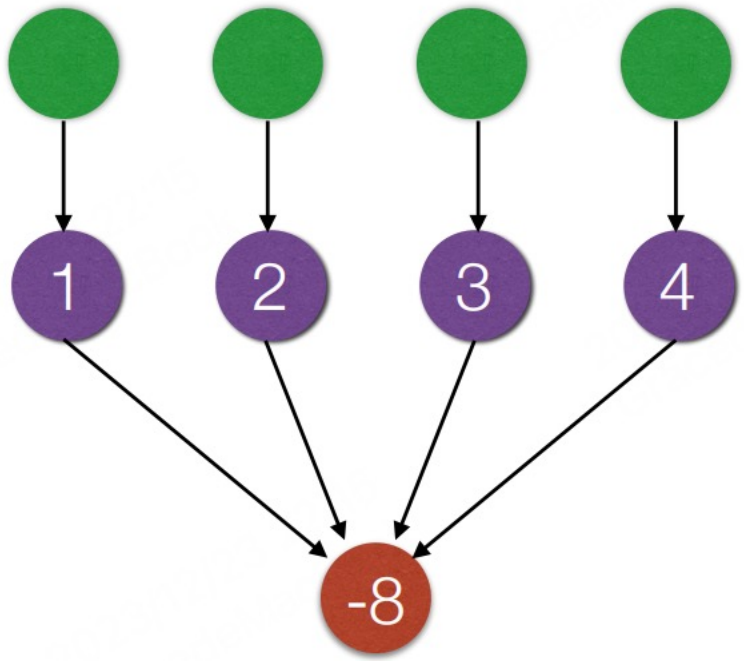
```
>>> data = [1, 2, 3, 4, 5]
>>> rDD = sc.parallelize(data, 4)
```

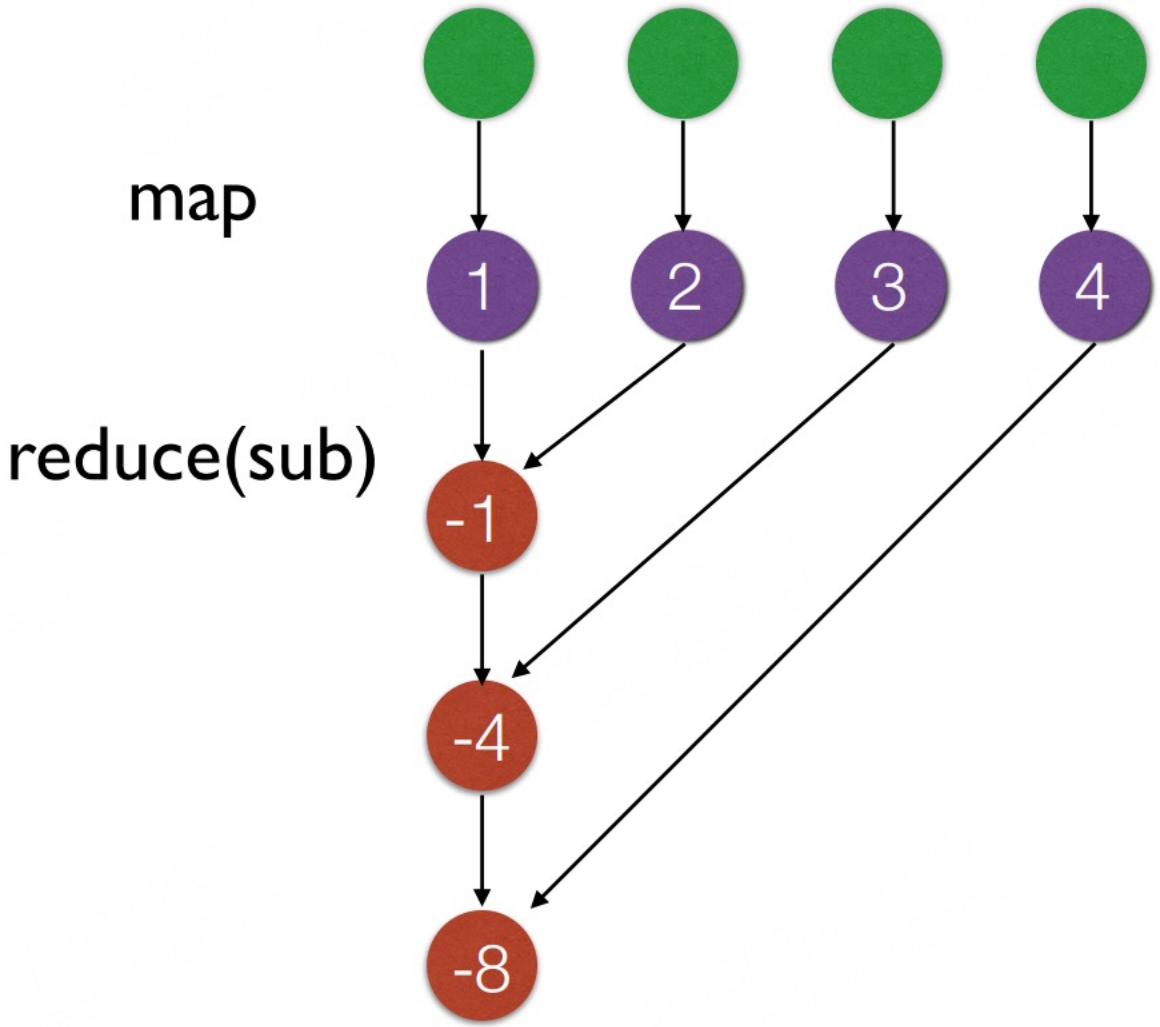
What's the output of the following reduce()?

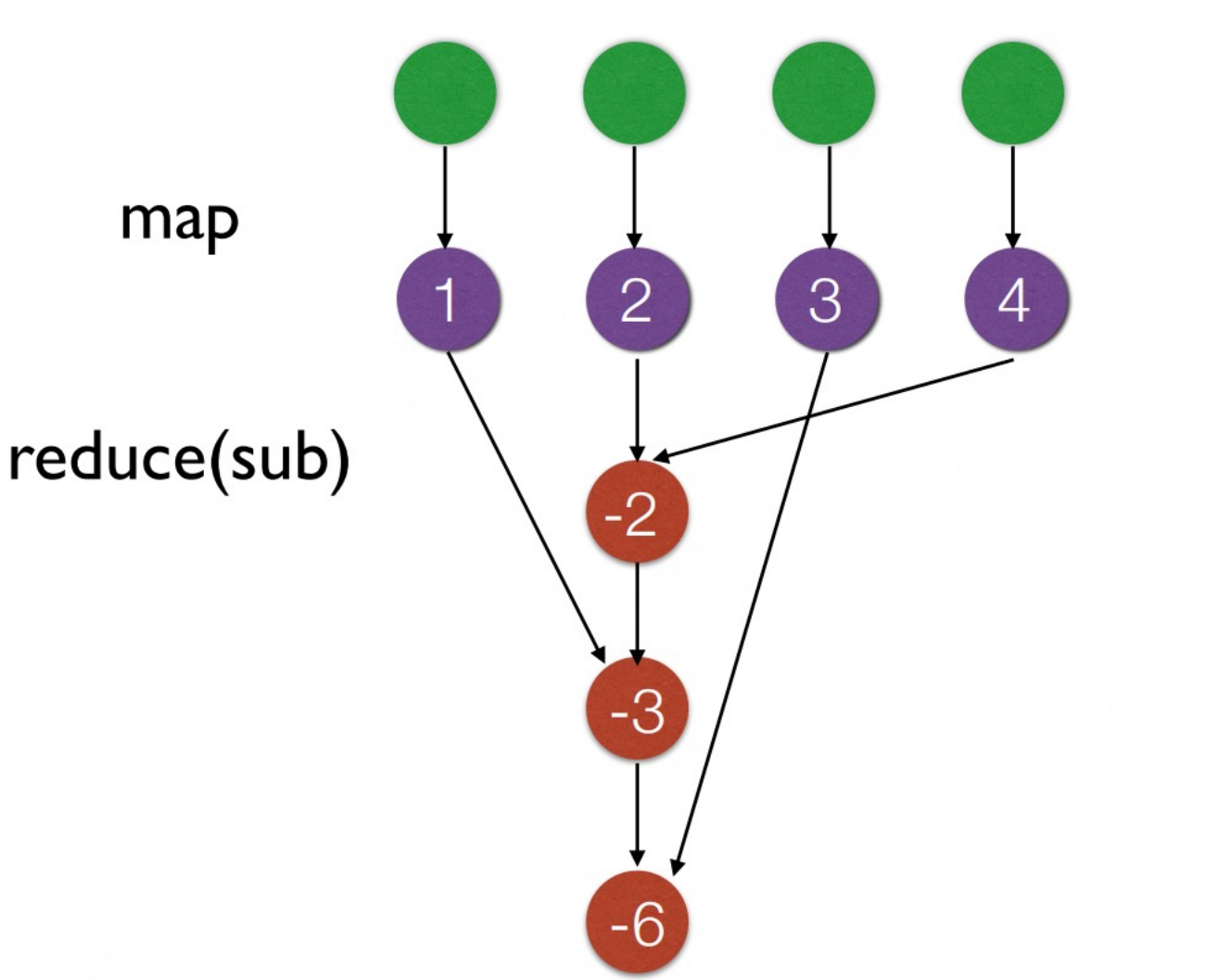
- rDD.reduce(lambda a, b: a + b)
- rDD.reduce(lambda a, b: a * b)
- rDD.reduce(lambda a, b: a - b)
- rDD.reduce(lambda a, b: a / b)

map

reduce(sub)




map
1
2
3
4
reduce(sub)
-1
-4
-8


map
reduce(sub)
1
2
3
4
-2
-3
-6

Implications for distributed processing?

You don’t know when the tasks begin

You don’t know when the tasks end

You don’t know when the tasks interrupt each other

You don’t know when intermediate data arrive

…

Two superpowers

Associativity

Commutativity

The Power of Associativity

You can put parenthesis wherever you want!

$$(v_1 \oplus v_2 \oplus v_3) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_8 \oplus v_9)$$

$$(v_1 \oplus v_2) \oplus (v_3 \oplus v_4 \oplus v_5) \oplus (v_6 \oplus v_7 \oplus v_8 \oplus v_9)$$

$$(v_1 \oplus v_2 \oplus (v_3 \oplus v_4 \oplus v_5)) \oplus (v_6 \oplus v_7 \oplus v_8 \oplus v_9)$$

The Power of Commutativity

You can swap order of operands however you want!

$$(v_1 \oplus v_2 \oplus v_3) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_8 \oplus v_9)$$

$$(v_1 \oplus v_2) \oplus (v_3 \oplus v_4 \oplus v_5) \oplus (v_6 \oplus v_7 \oplus v_8 \oplus v_9)$$

$$(v_1 \oplus v_2 \oplus (v_3 \oplus v_4 \oplus v_5)) \oplus (v_6 \oplus v_7 \oplus v_8 \oplus v_9)$$

Implications for distributed processing?

You don’t know when the tasks begin

You don’t know when the tasks end

You don’t know when the tasks interrupt each other

You don’t know when intermediate data arrive

…

It’s okay!

Revisiting **reduce()**

```
>>> data = [1, 2, 3, 4, 5]
>>> rDD = sc.parallelize(data, 4)
```

Associativity and Commutativity?

- rDD.reduce(lambda a, b: a + b) ✔
- rDD.reduce(lambda a, b: a * b) ✔
- rDD.reduce(lambda a, b: a - b) ✘
- rDD.reduce(lambda a, b: a / b) ✘

It is your responsibility to ensure that the reduce logic satisfies both commutativity and associativity!

PySpark Closures

Function literals are closures

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]


>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]


>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

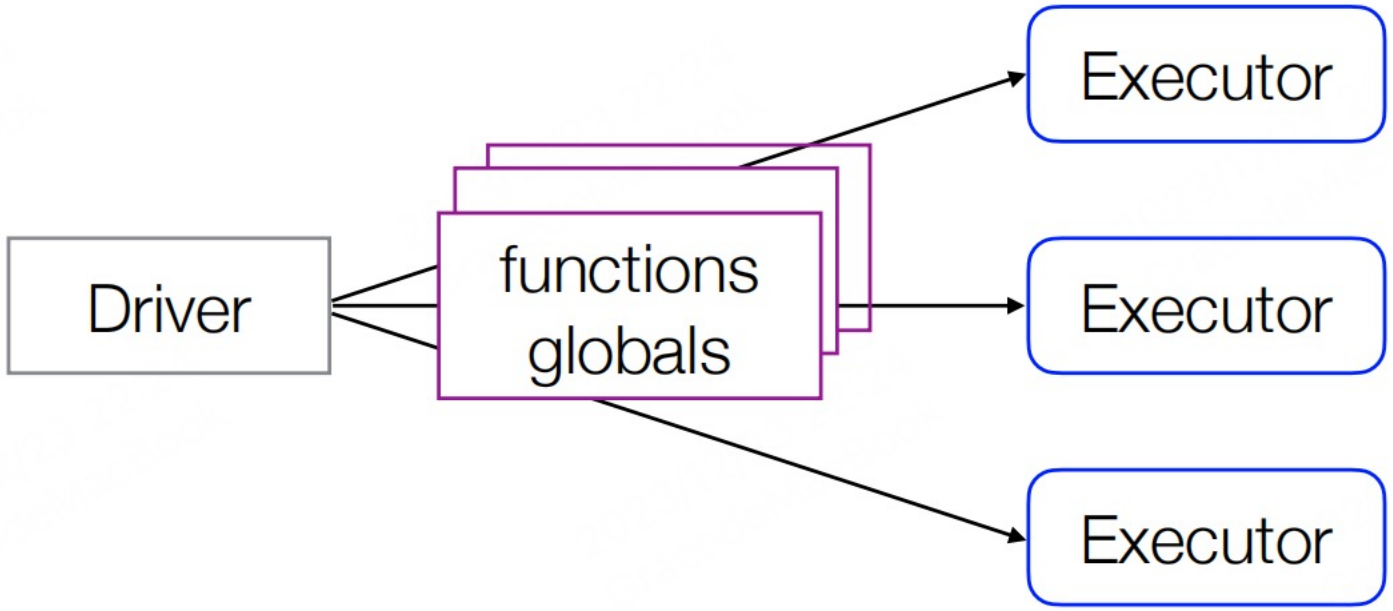
Function literals (green) are closures

- automatically shipped to workers!

PySpark Closures

Spark automatically creates closures for

- functions that run on RDDs at executors
- any global variables used by those executors



PySpark Closures

One closure per executor

- sent for every task
- no communication between executors
- changes to global variables at executors are not sent to driver

Consider these use cases

Counting events that occur during job execution

- how many input lines were blank?
- how many input records were corrupted?

Iterative or single jobs with large global variables

- sending large read-only lookup table to executors
- sending large feature vector in a ML algorithm to executors

What's wrong here?

Problems

Closures are (re-)sent with every task

Inefficient to send large data to each worker

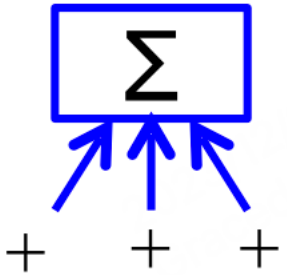
Closures are one way

- driver -> worker

How to share variables efficiently?

PySpark Shared Variables

Broadcast variables
Accumulators

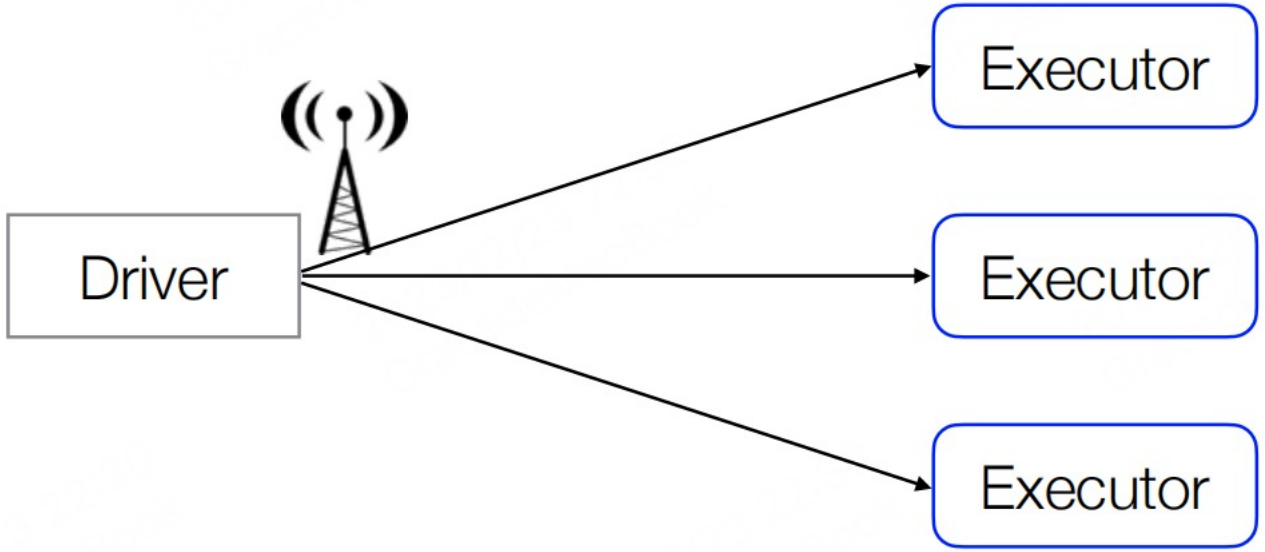


Broadcast variables

Efficiently send large, read-only value to all executors

Saved workers for use in one or more Spark operations

Like sending a large, read-only lookup table to all nodes



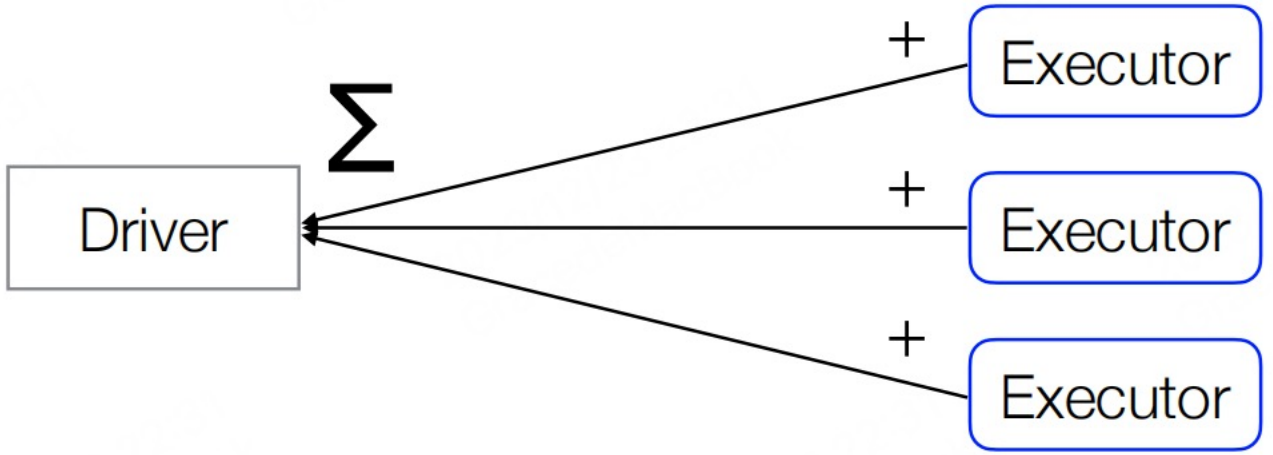
Accumulators

Aggregate values from executors back to driver

Only driver can access value of accumulator

For tasks, accumulators are **write-only**

Use to count errors seen in RDD across executors



Broadcast variables

Keep read-only variable cached on executors

- ship to each worker only once instead of with each task

Example

- efficiently give every executor a large dataset

Usually distributed using efficient broadcast algorithms

Broadcast variables syntax

```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At an executor (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]
```

Broadcast variables example

Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

(Re-)sent for every mapper!

H. Karau et al. “Learning Spark,” O’Reilly, 2015

Broadcast variables example

Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Efficiently sent once to executors

H. Karau et al. “Learning Spark,” O’Reilly, 2015

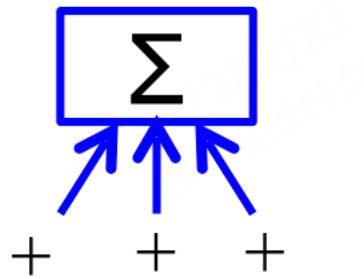
Accumulators

Variables that can only be “added” to

- the “add” logic must be associative and commutative

Used to efficiently implement parallel counters and sums

Only drivers can read an accumulator’s value, not tasks



Accumulators

```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>   global accum
>>>   accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```

f() is commutative and associative

Each task updates accumulator only once

Accumulators example

Counting empty lines

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```

used in transformations

Accumulators

Accumulators can be used in actions or transformations

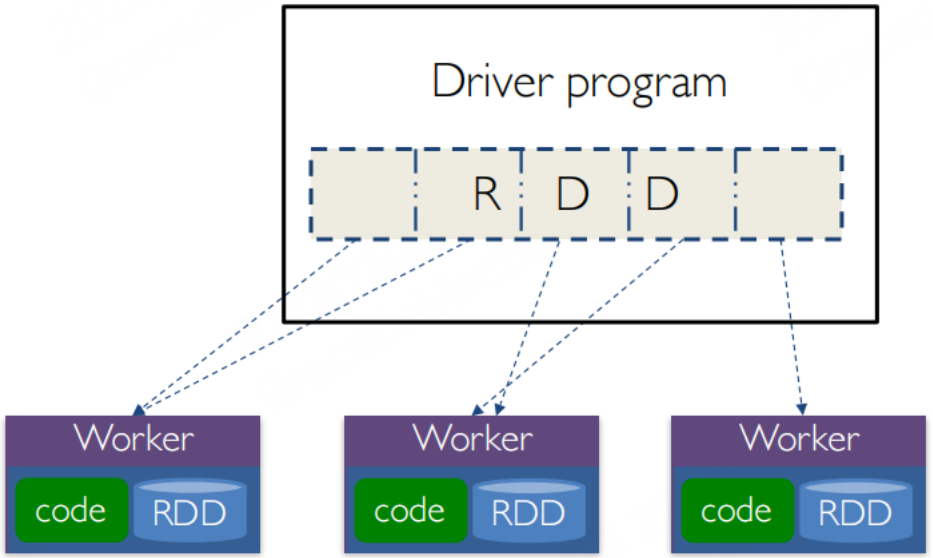
- **Actions**: each task's update to accumulator is applied only once
- **Transformations**: no guarantees
  - failed/slow tasks may get rescheduled
  - use only for debugging

Types

- integers, double, long, float, and customizable as well

PySpark summary

Spark automatically pushes closures to Spark executors at workers

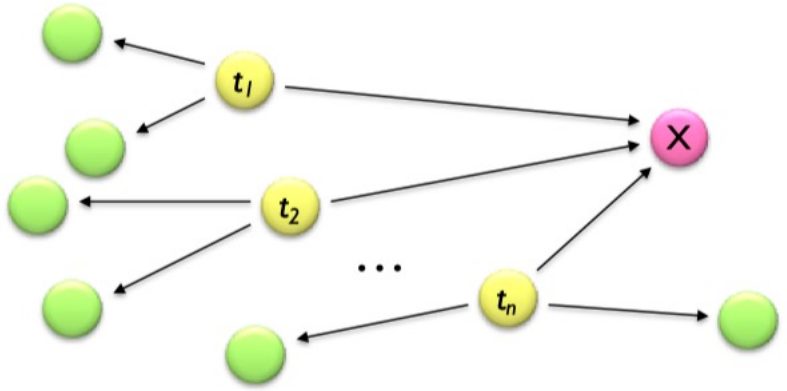


Spark Example

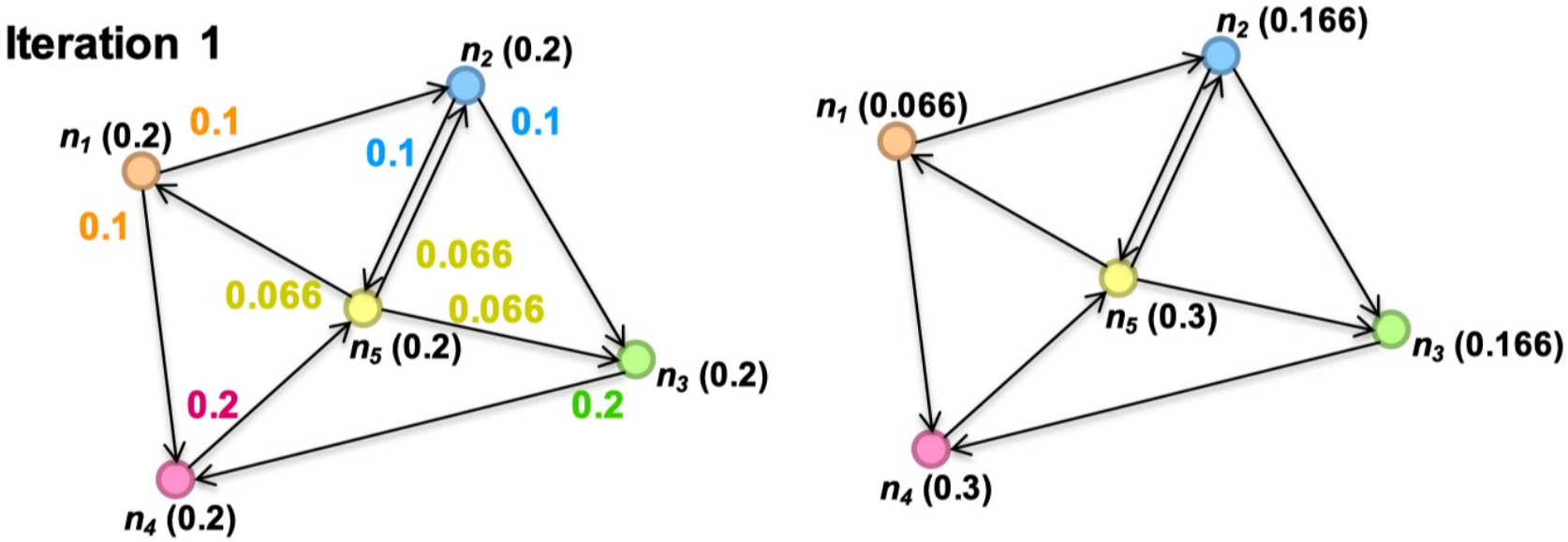
PageRank

Evaluate the importance of web pages, rank them in search engines

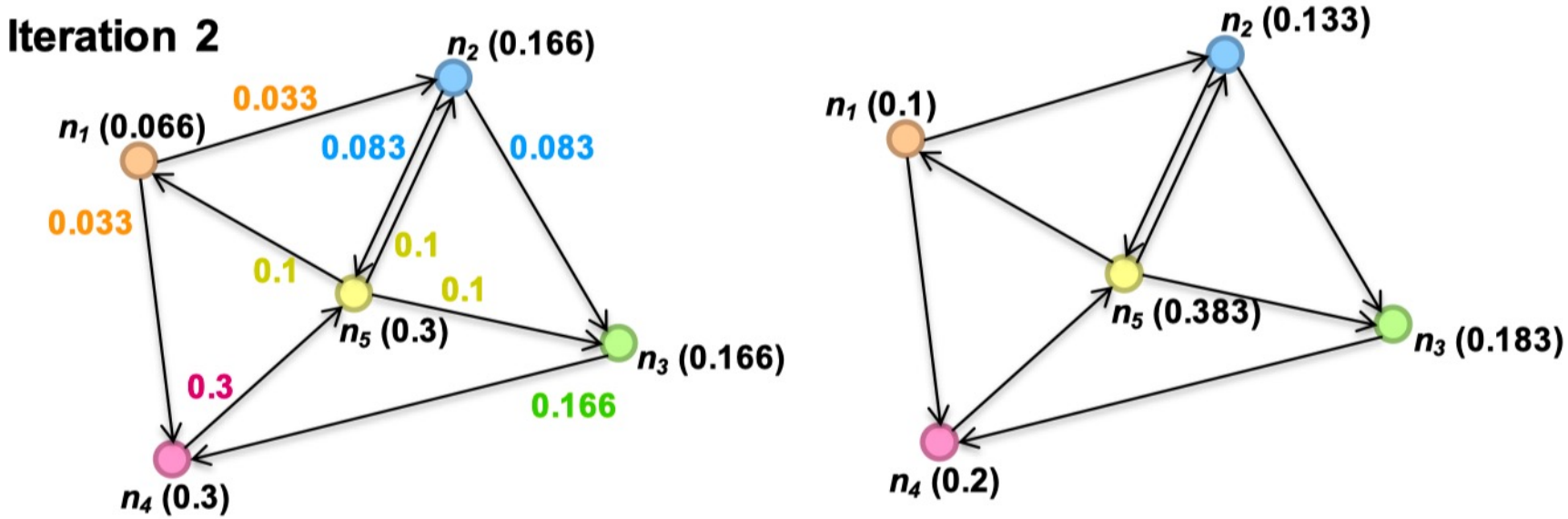
- Start each page with a rank
- On each iteration, update each page's rank to $\Sigma_{i \in neighbors} rank_i / |neighbors_i|$



PageRank Example



PageRank Example



PageRank

Evaluate the importance of web pages, rank them in search engines

- Start each page with a rank
- On each iteration, update each page's rank to $\Sigma_{i \in neighbors} rank_i / |neighbors_i|$

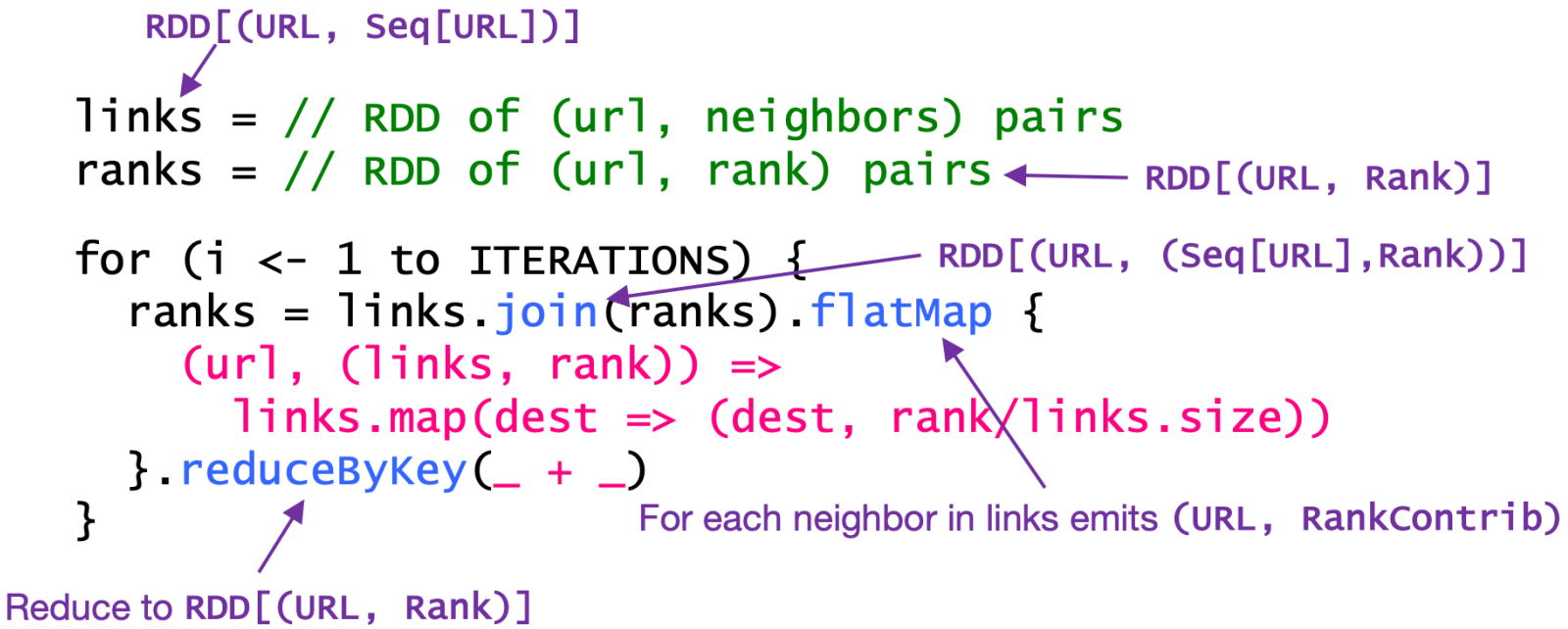
```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

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

PageRank

Evaluate the importance of web pages, rank them in search engines

- Start each page with a rank
- On each iteration, update each page's rank to $\Sigma_{i \in neighbors} rank_i / |neighbors_i|$



Join(⋈)

Alice	5
Bob	6
Claire	4

⋈

Alice	F
Bob	M
Claire	F

=

Alice	5	F
Bob	6	M
Claire	4	F

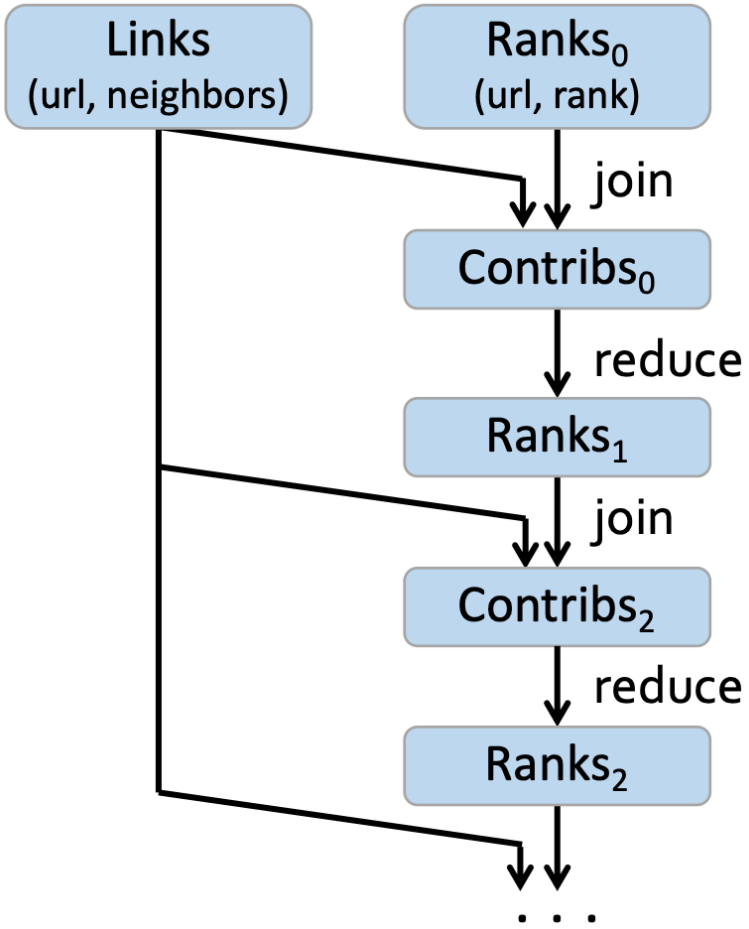
A	5
A	2
A	3
B	4
B	1
C	6
C	8

⋈

C	5
B	2
A	3
B	4
A	1
B	6
C	8

If partitioning doesn't match, then need to reshuffle to match pairs.

Optimizing placement



- links & ranks repeatedly joined
- Can co-partition them (e.g. hash both on URL) to avoid shuffles
- Can also use app knowledge, e.g., hash on DNS name

```
links = links.partitionBy(
          new URLPartitioner())
```

Q: Where might we have placed persist()?

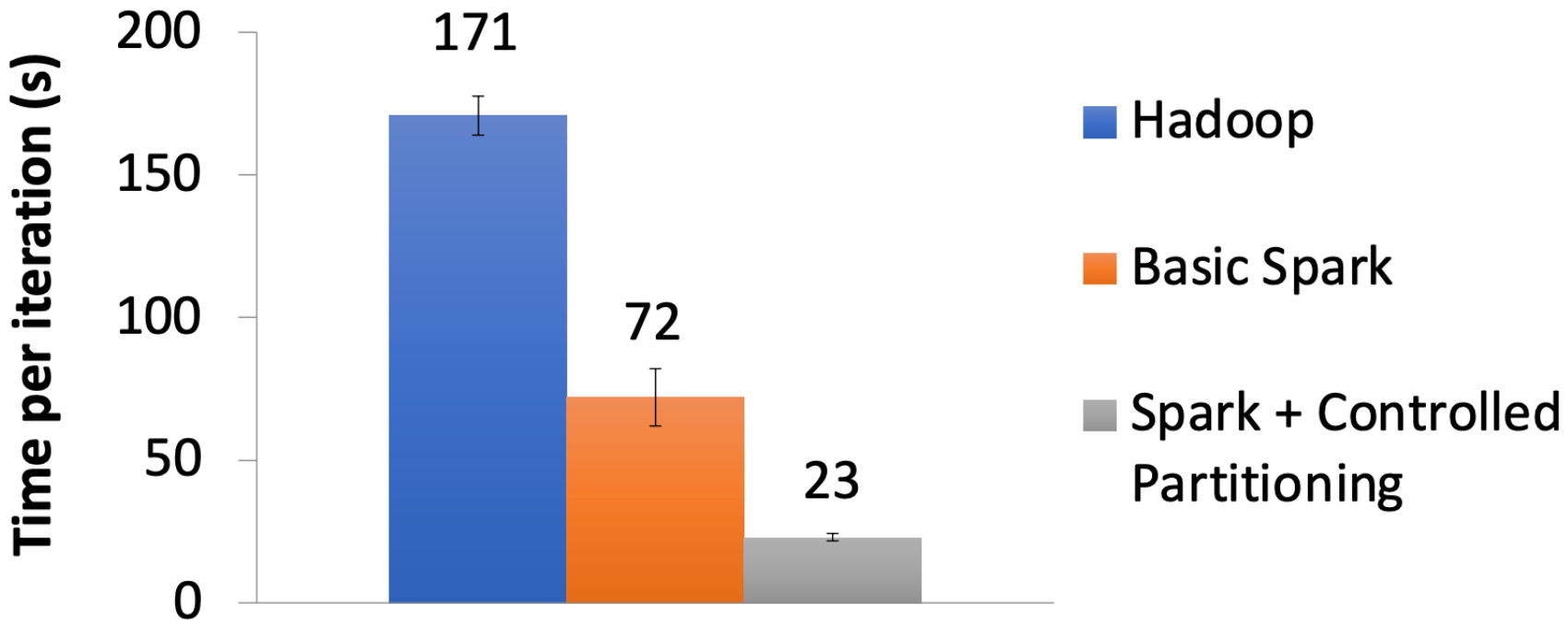
Co-partitioning

Co-partitioning can avoid shuffle on join

But fundamentally a shuffle on reduceByKey

Optimization: custom partitioner on domain

PageRank performance



Credits

- Some slides are adapted from course slides of COMP 4651 in HKUST
- Some slides adapted from course slides of DS5110 in UVA