Distributed Systems

Spring Semester 2020

Lecture 21: Big Data Systems (Spark)

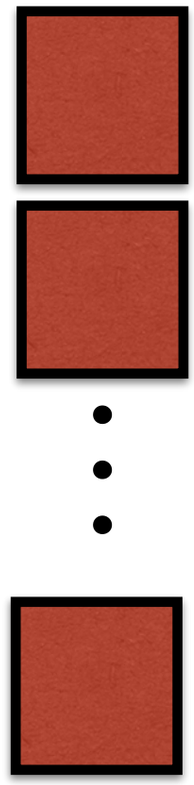
John Liagouris
liagos@bu.edu

Why SPARK?

- General-purpose cluster computing framework
- Better performance than Hadoop (open-source MapReduce)
- User-friendly fault-tolerant abstractions (RDDs)
- Rich dataflow API (incl. join(), flatMap(), groupByKey(), etc.)
- A big commercial success (Databricks' valuation: $6B)

MapReduce

— INPUT —
Massive amount
of data in files



eg. 1000's of files
containing text

map()

map()

map()

map()

reduce()

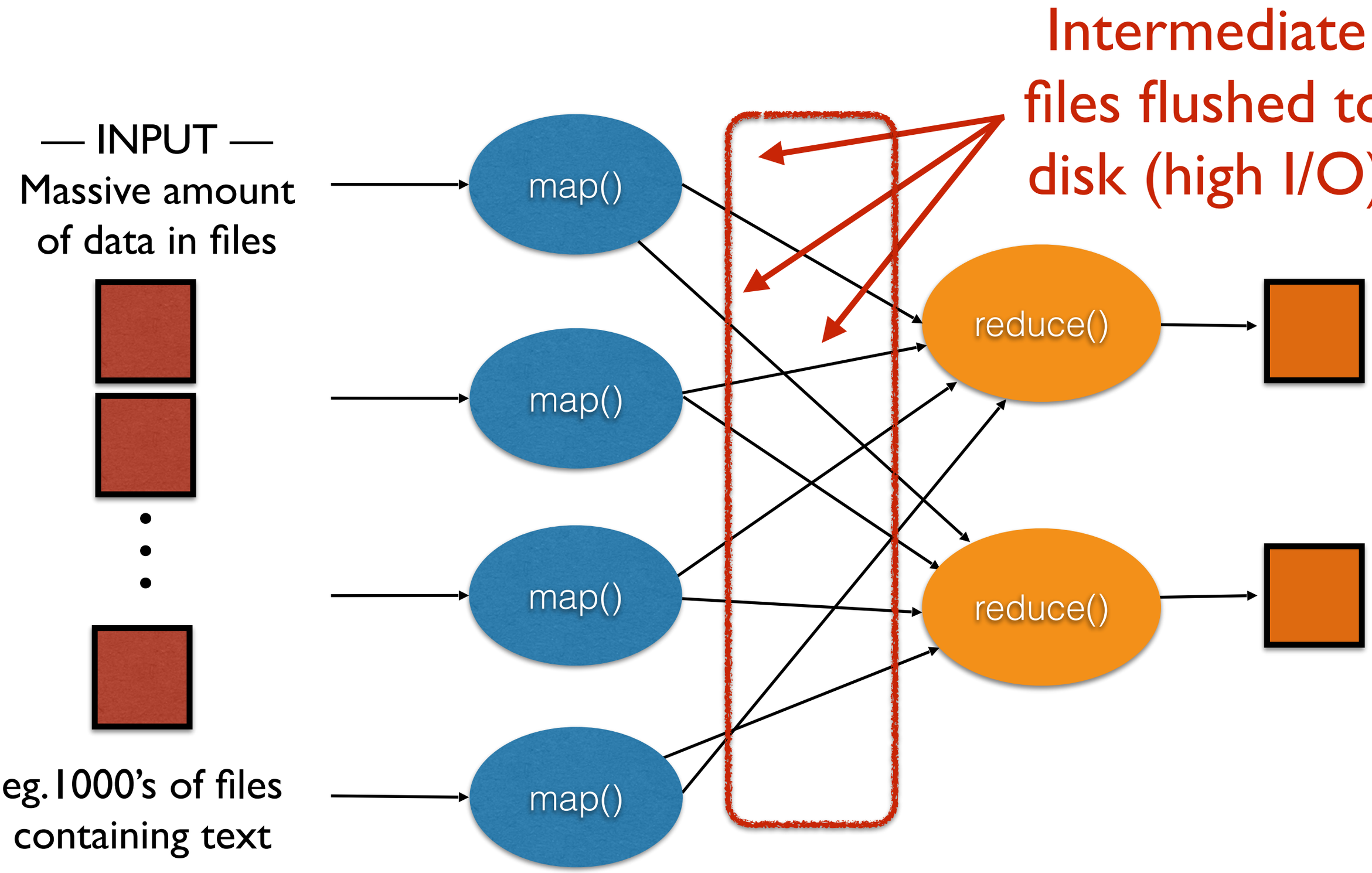
reduce()

map() and reduce()

```
map(k,v)
{
 split v into words
 for each word w
     emit(w,1)
}
```

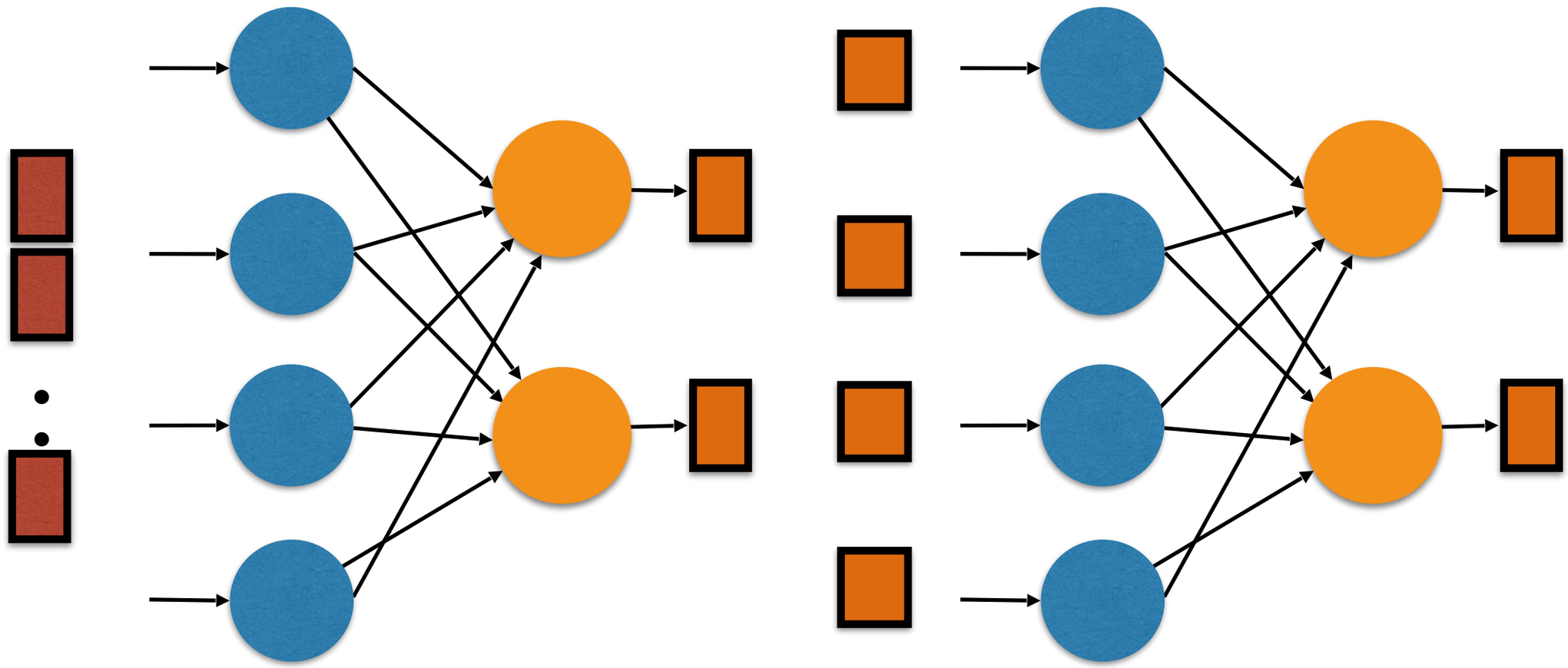
```
reduce(k,v)
{
   emit(len(v))
}
```

Limitations?



Limitations?

Iterations require manual work by users



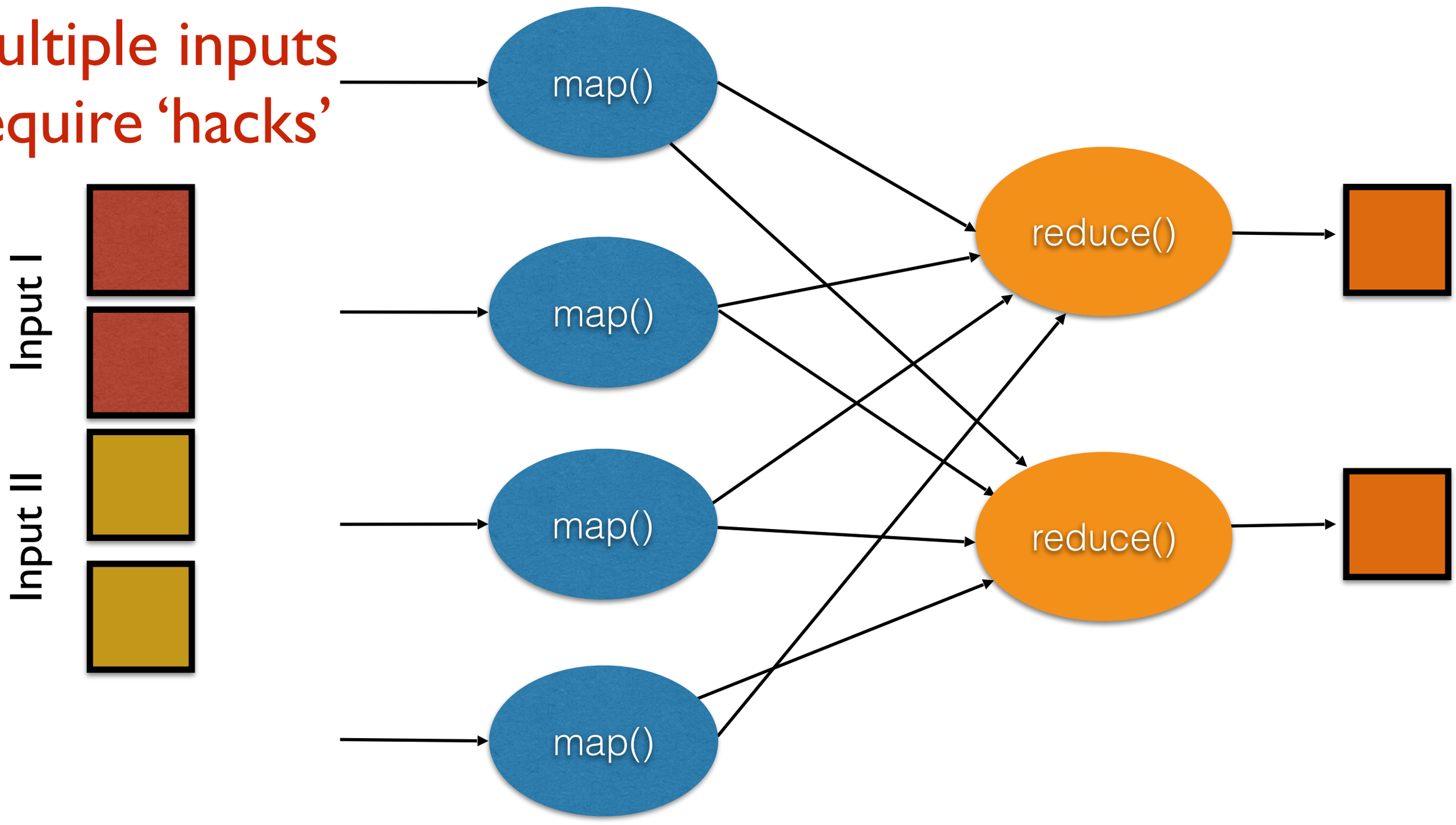
Iteration 1

Iteration 2

....

Limitations?

Multiple inputs require ‘hacks’



Limitations?

How would map() and reduce() distinguish between k-v pairs from different inputs?

```
map(k,v)
{
 split v into words
 for each word w
     emit(w,1)
}
```

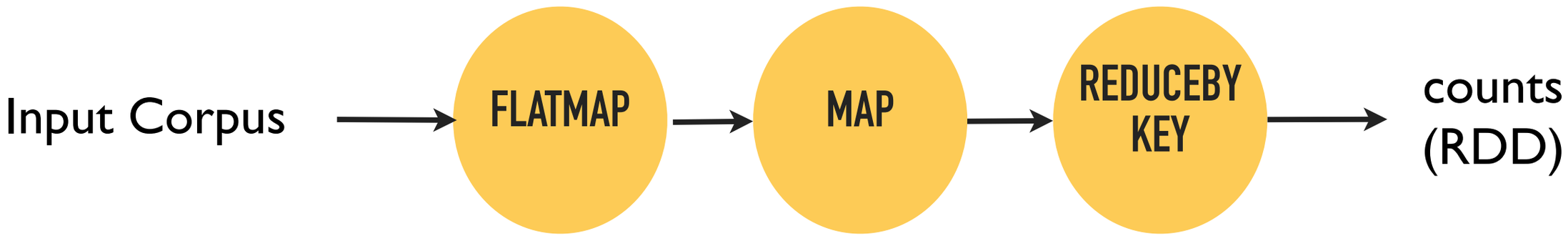
```
reduce(k,v)
{
   emit(len(v))
}
```

"Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse…They do not provide abstractions for more general reuse, e.g., to let a user load several datasets
into memory and run ad-hoc queries across them."

SPARK

"…a new abstraction called resilient distributed datasets (RDDs) that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators."

Dataflow Graphs



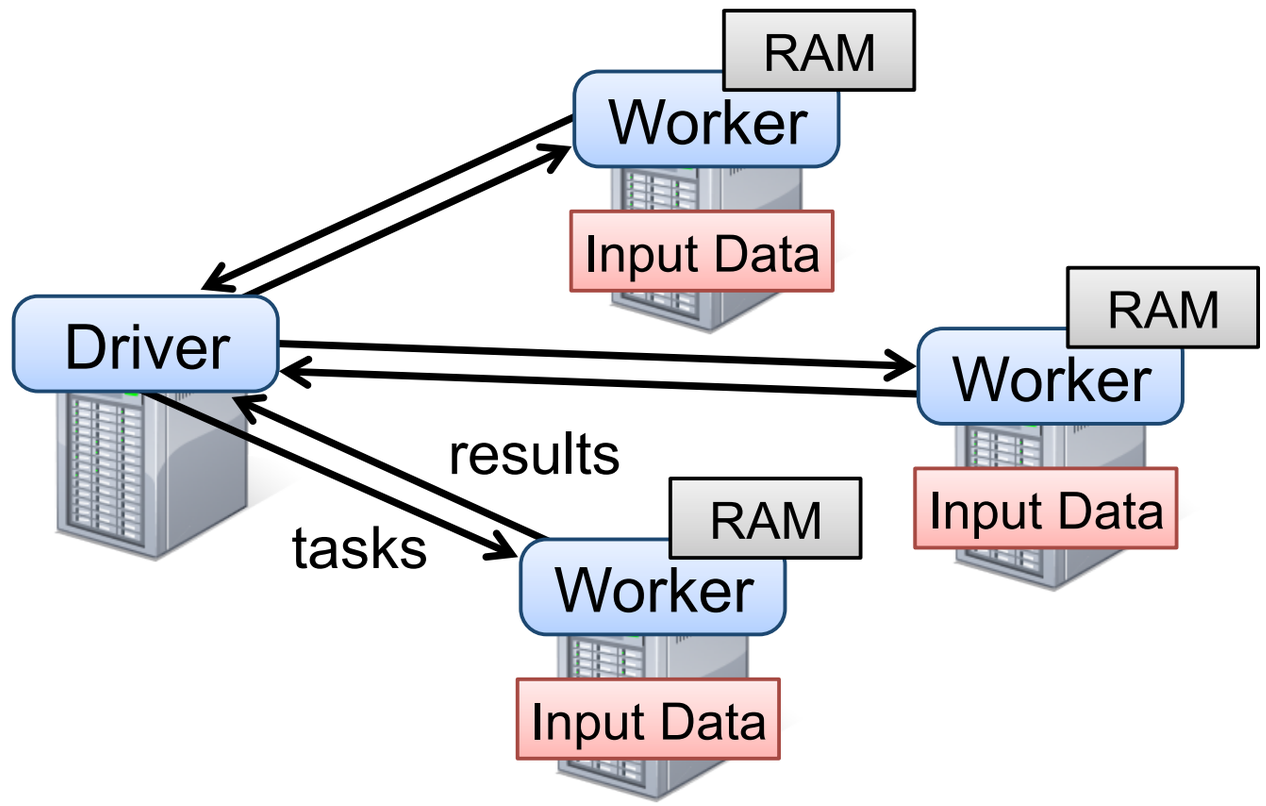
Data Operator

Flow of data

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

SPARK APPLICATION

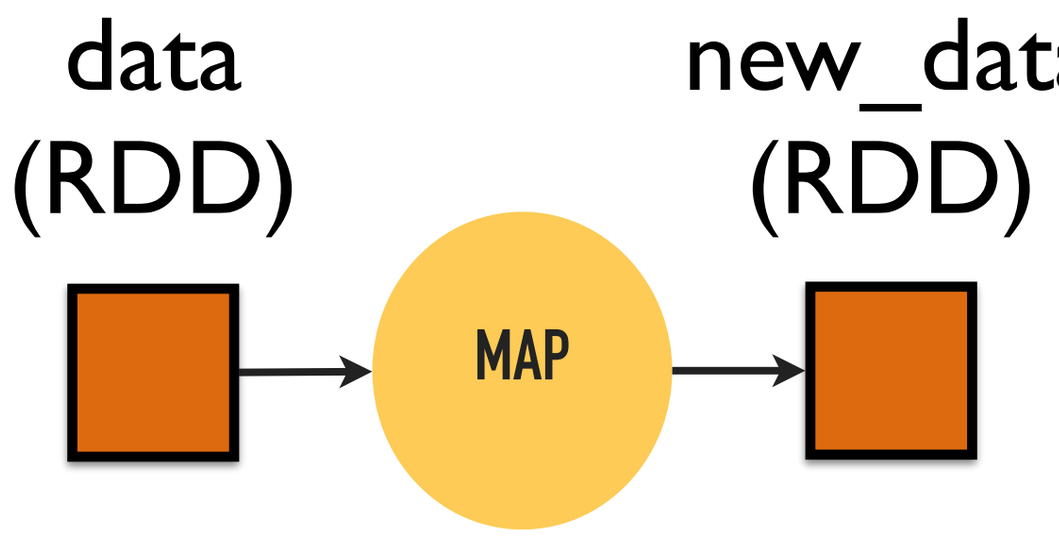
- User writes application that is managed by the 'driver'
- Driver defines one or more RDD and invokes actions on them
- Workers can store data in RAM across operations



More like writing a program than Map Reduce — explicitly naming data and operating on it

RDD

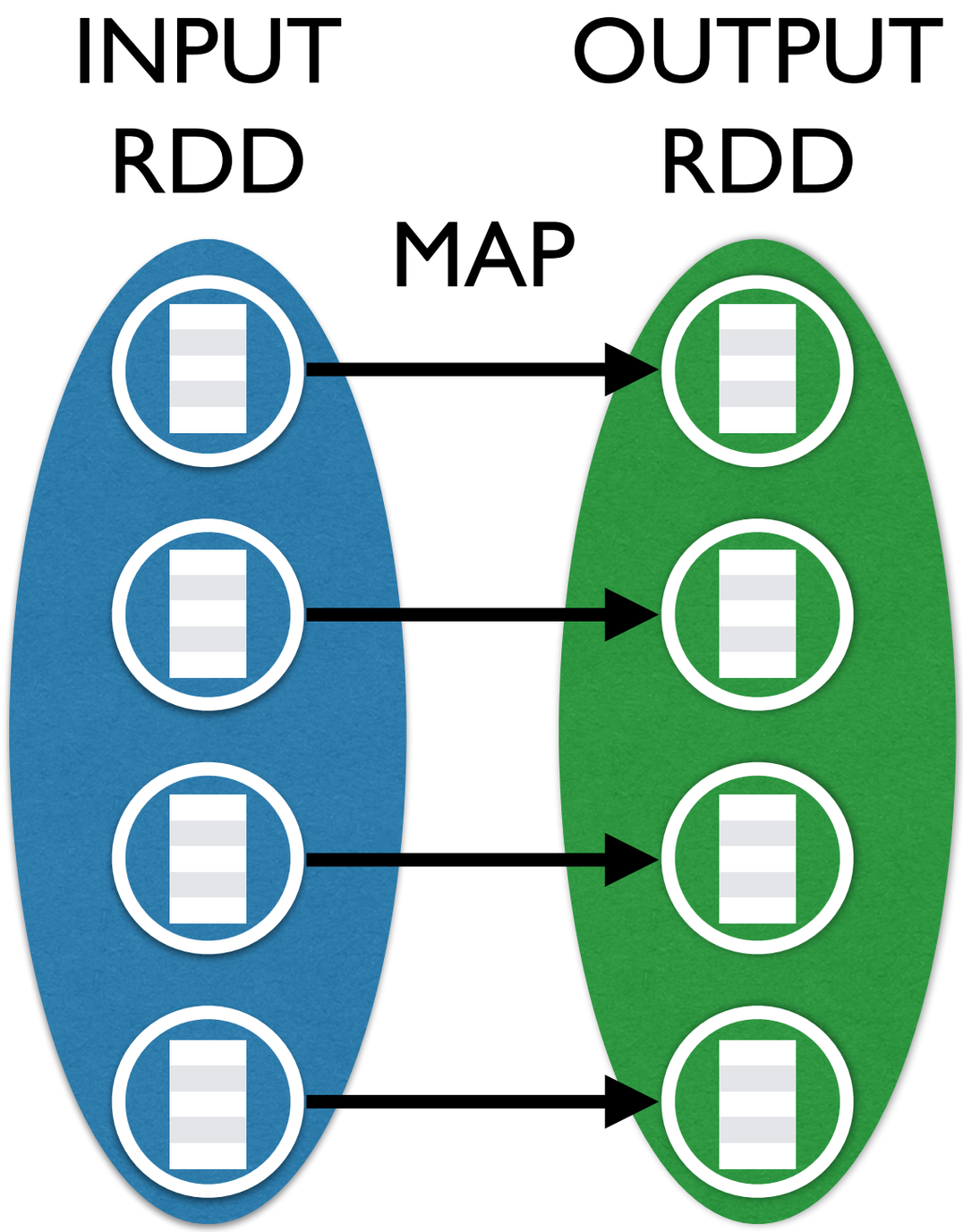
RDDs provide an interface based on coarse-grained transformations



val new_data = data.map(word => (word, 1))

- Coarse grain operations that hide fine-grain memory R/Ws
- Same operation across many items within the object
- Restricted interface provides basis for Fault-Tolerance — log of operators per RDD (lineage)

READ-ONLY & SIMD

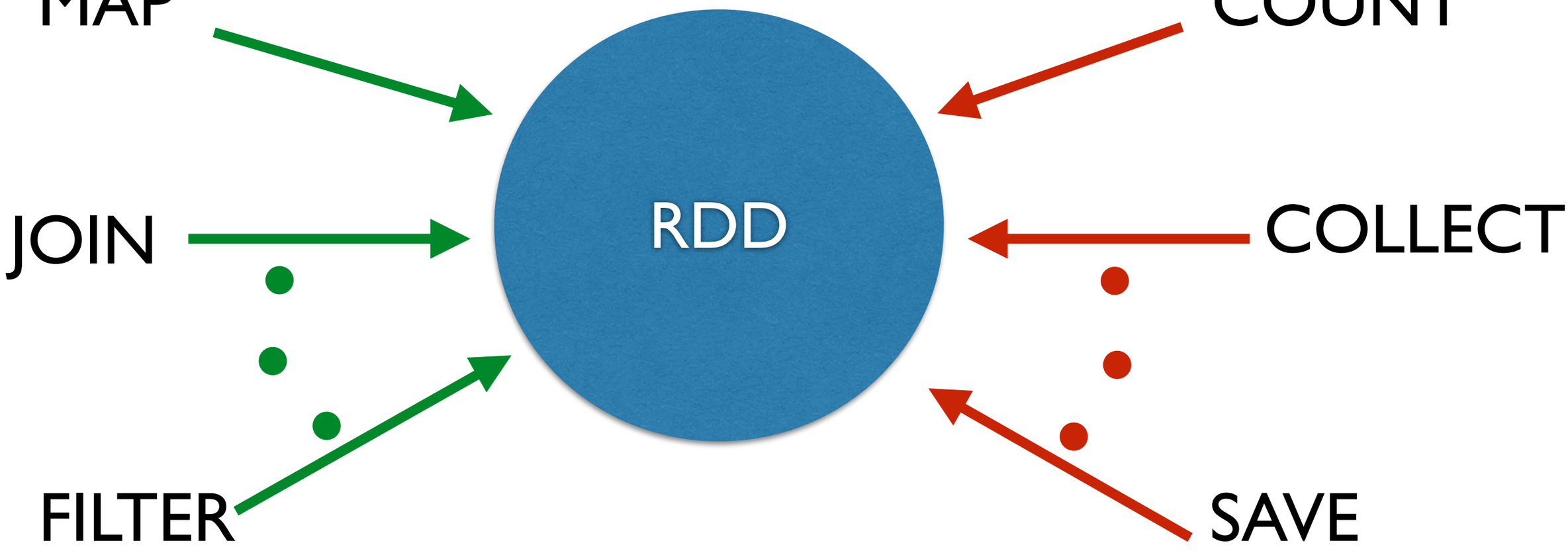


- created never modified
- computations is a series of transformations that produce create a new RDD
- Partitions of data items
- data parallel model of computing — single operation applied in parallel to many data items

Lazy Evaluation

TRANSFORMATIONS

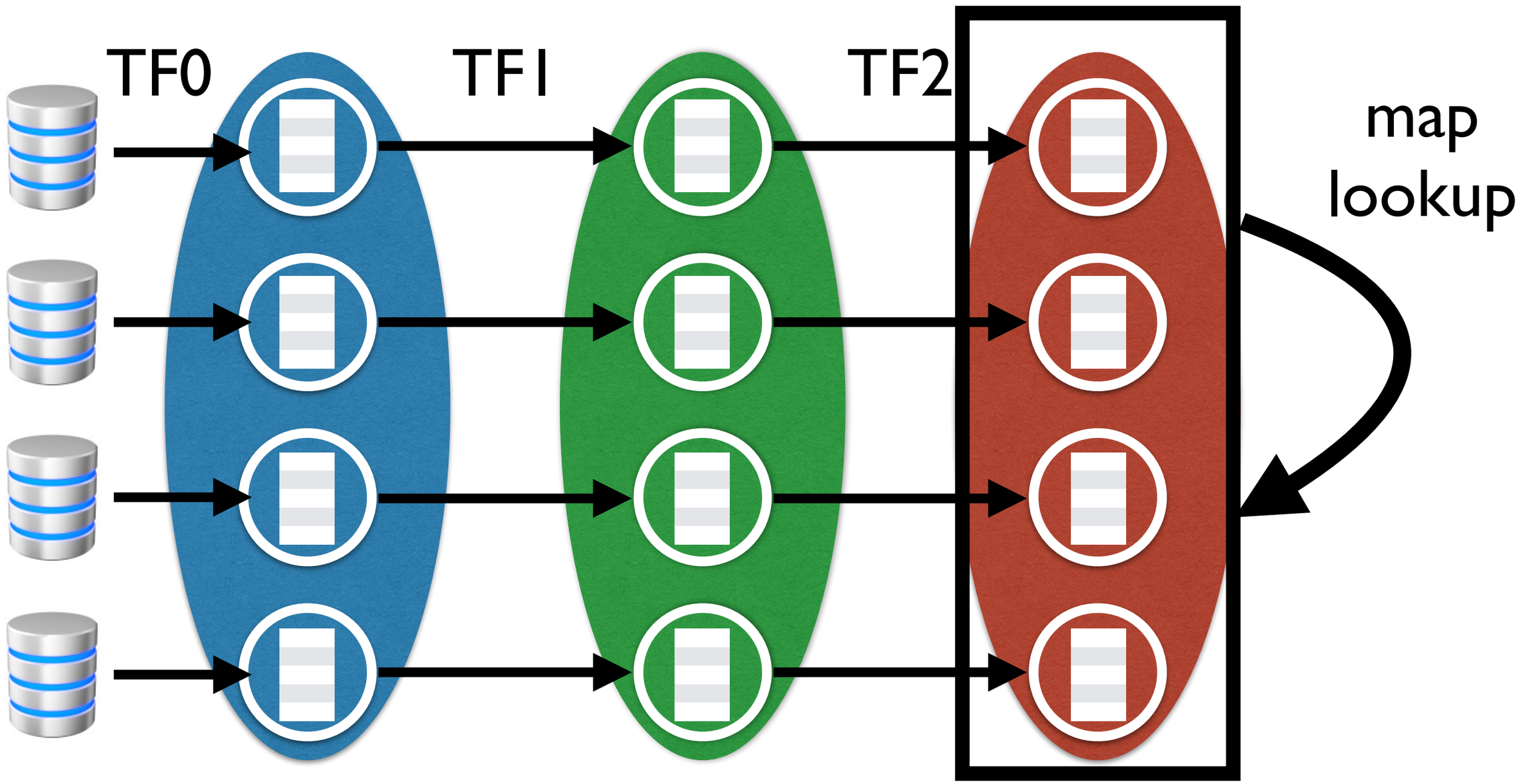
ACTIONS



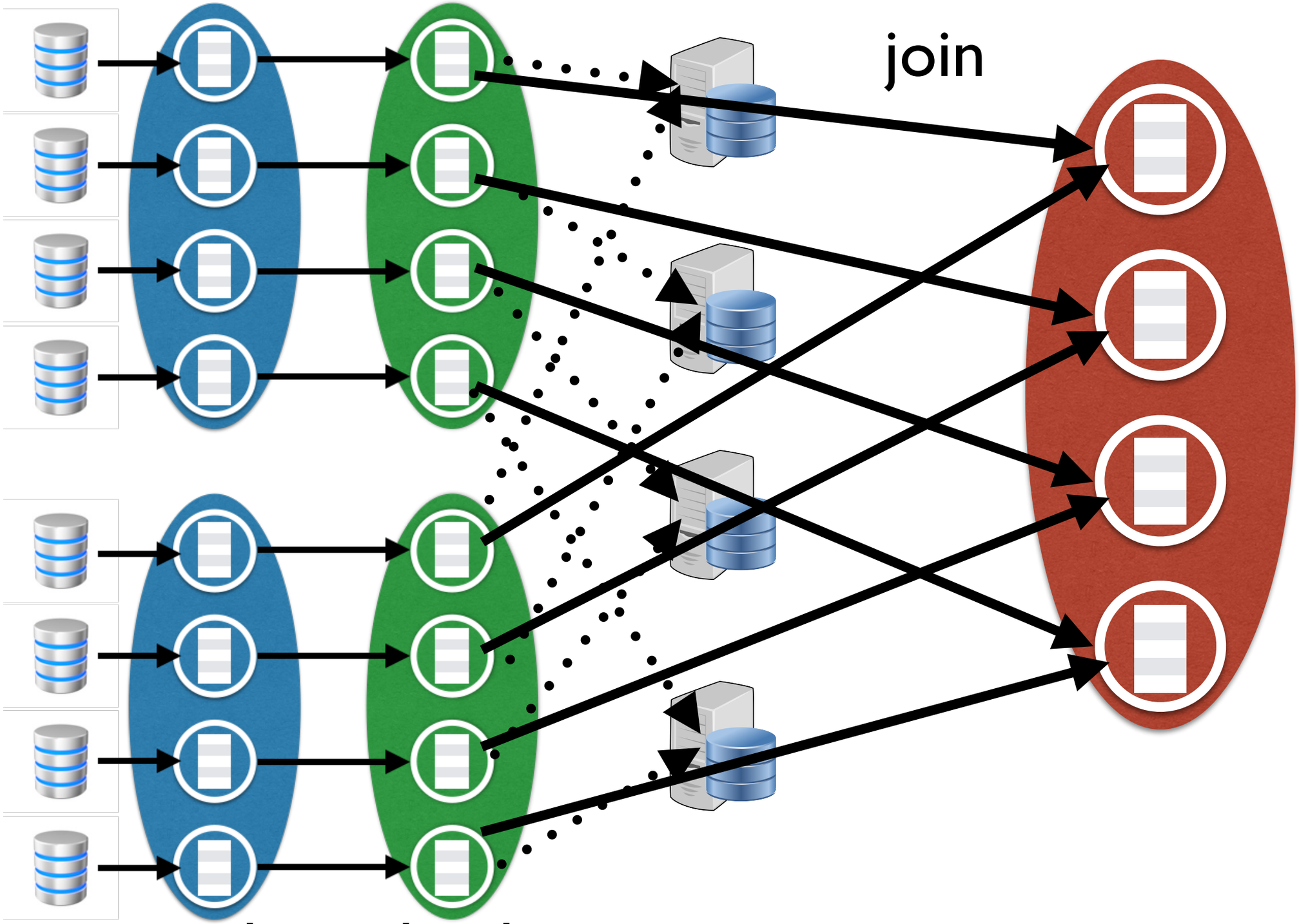
Compute a new RDD from existing RDDs this just specifies a plan. Runtime is lazy - doesn't have to materialize (compute), so it doesn't

Where some effect is requested: result to be stored, get specific value, etc. causes RDDs to materialize — return a value to app or to stable storage

PERSISTENCE: Users can indicate which RDD's they will reuse and choose a storage strategy for them (eg. in-memory)



PARTITIONING: Users can ask that an RDD's records be partition across machine base on a key in each record temp by city



Logistic Regression

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

- w is sent with the closure to the nodes

Logistic Regression

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

transformations

action (cf. Table 2)

- w is sent with the closure to the nodes

Logistic Regression

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

for each point p in points do:
p.x * f(w,p) * p.y

- w is sent with the closure to the nodes

Logistic Regression

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

sum all input vectors and materilize a new RDD 'gradient'

- w is sent with the closure to the nodes

PageRank

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
                  .mapValues(sum => a/N + (1-a)*sum)
}
```

URL of page

initial rank

neighbor

normalised contribution

sum all contributions to the current rank

a: the damping factor

N: total number of pages

PageRank

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
                  .mapValues(sum => a/N + (1-a)*sum)
}
```

neighbor

normalised contribution

sum all contributions to the current rank

a: the damping factor

N: total number of pages

PageRank

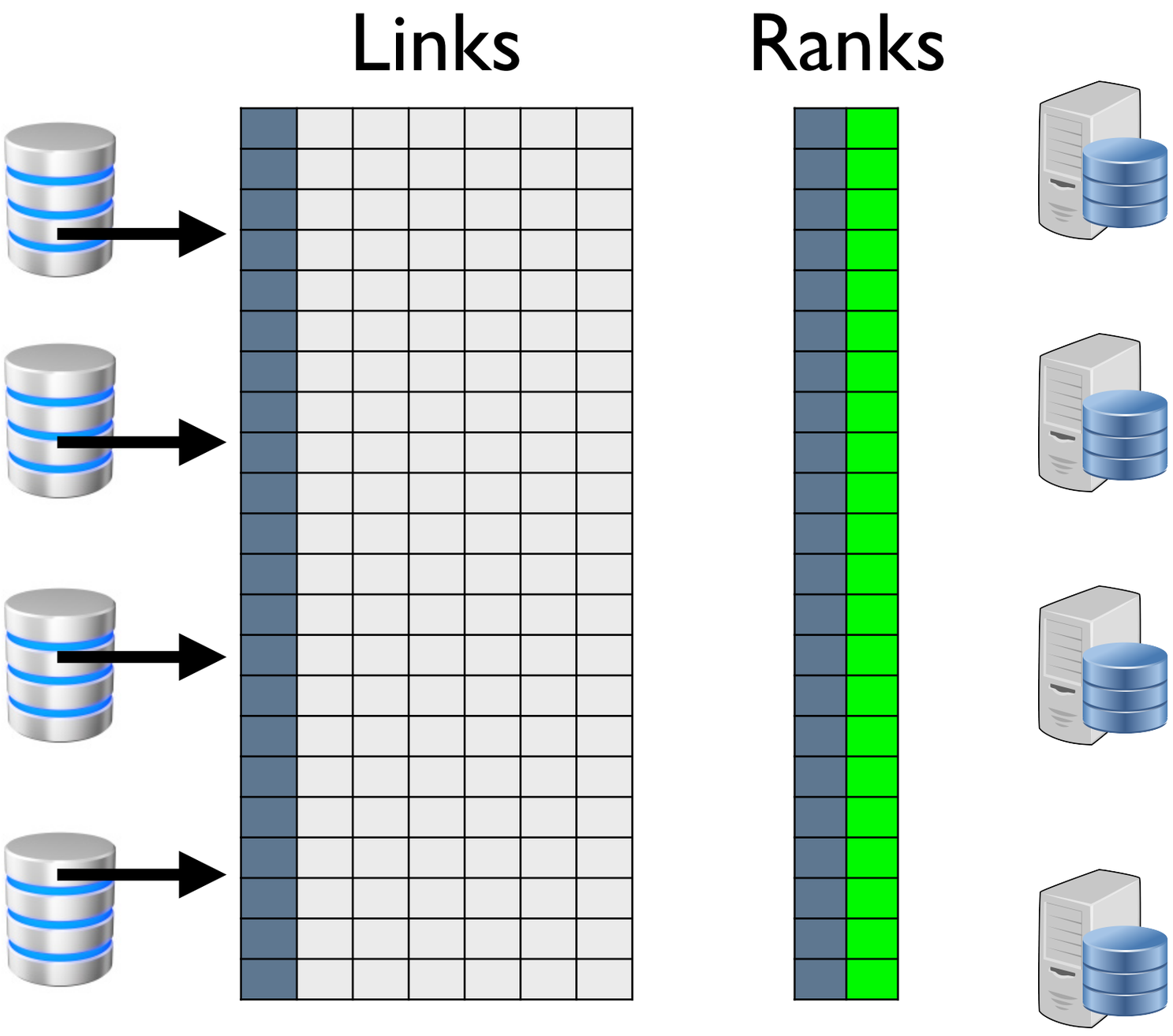
```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
                  .mapValues(sum => a/N + (1-a)*sum)
}
```

sum all contributions to the current rank

a: the damping factor

N: total number of pages

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
```



(URL of the page, list of pages pointed by the page)

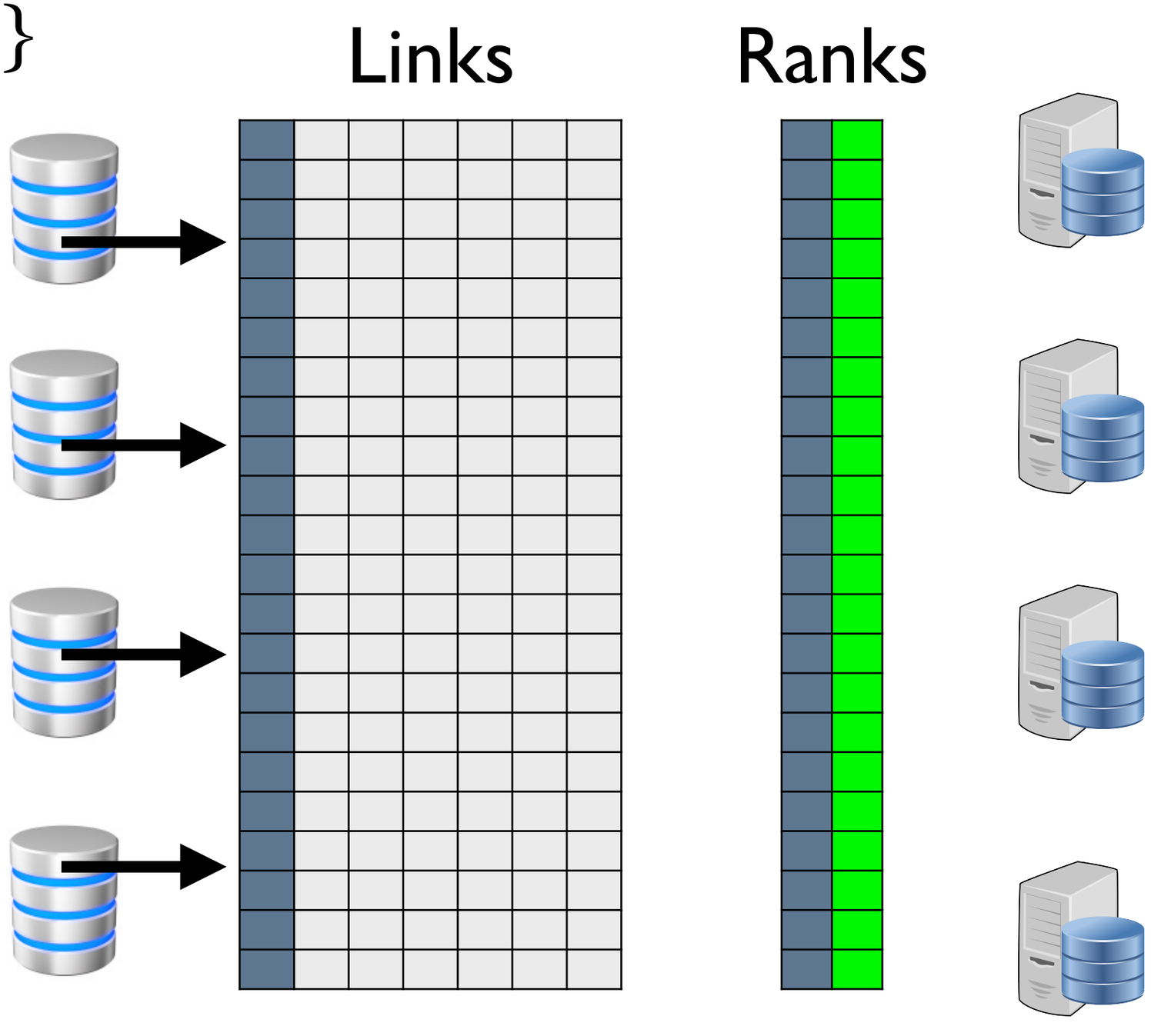
(URL, rank)

- At this point have we materialized / executed anything?

```
for (i<-1 to ITER) {
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
```

dest is a URL in the list of links

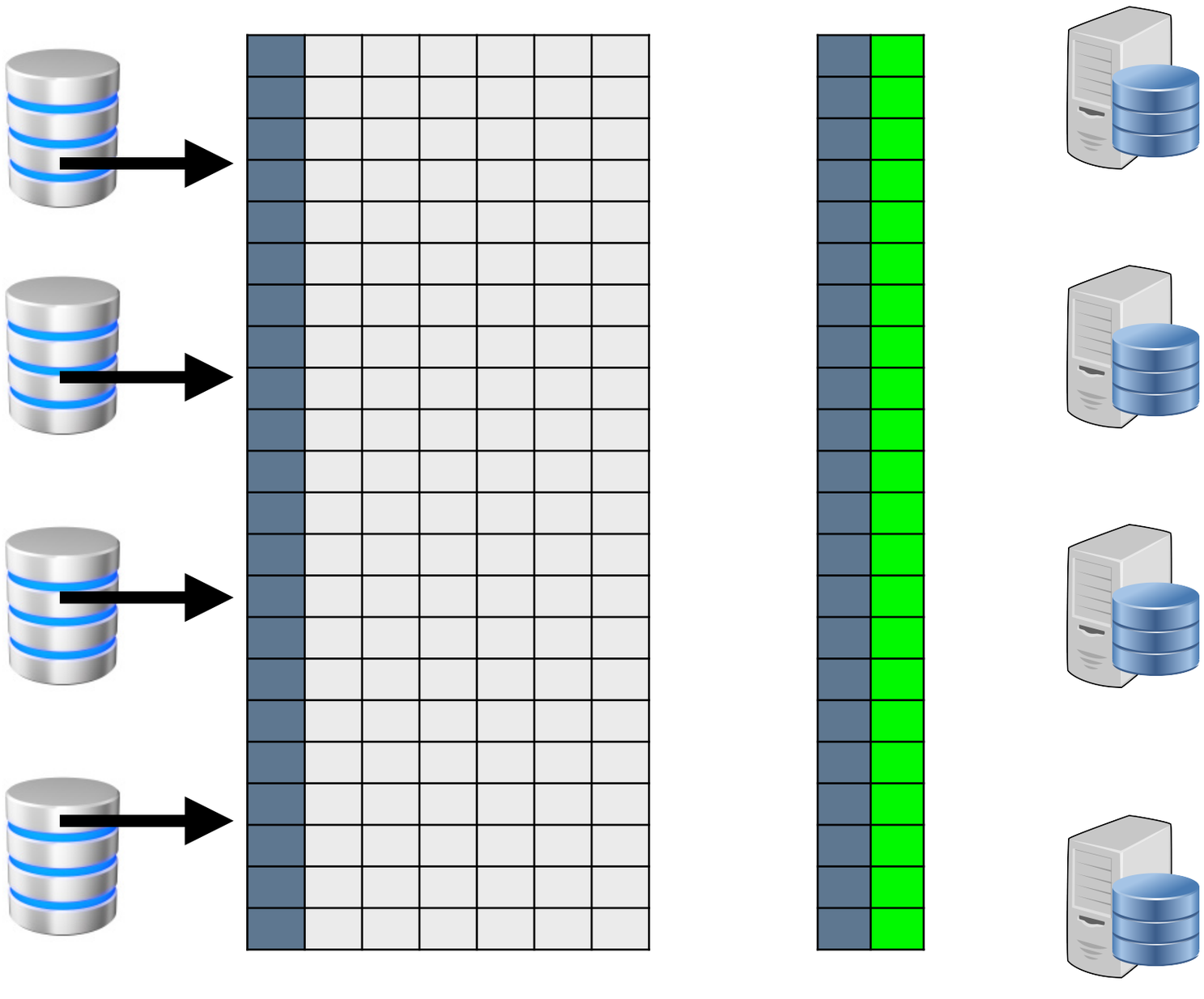
normalized contribution



- what does this say?
- have we materialized anything yet?

```
ranks = contribs.reduceByKey((x,y) => x+y)
                .mapValues(sum => a/N + (1-a)*sum)
}
```

sums all ‘rank/links.size()' for each distinct ‘dest’ (see previous slide)

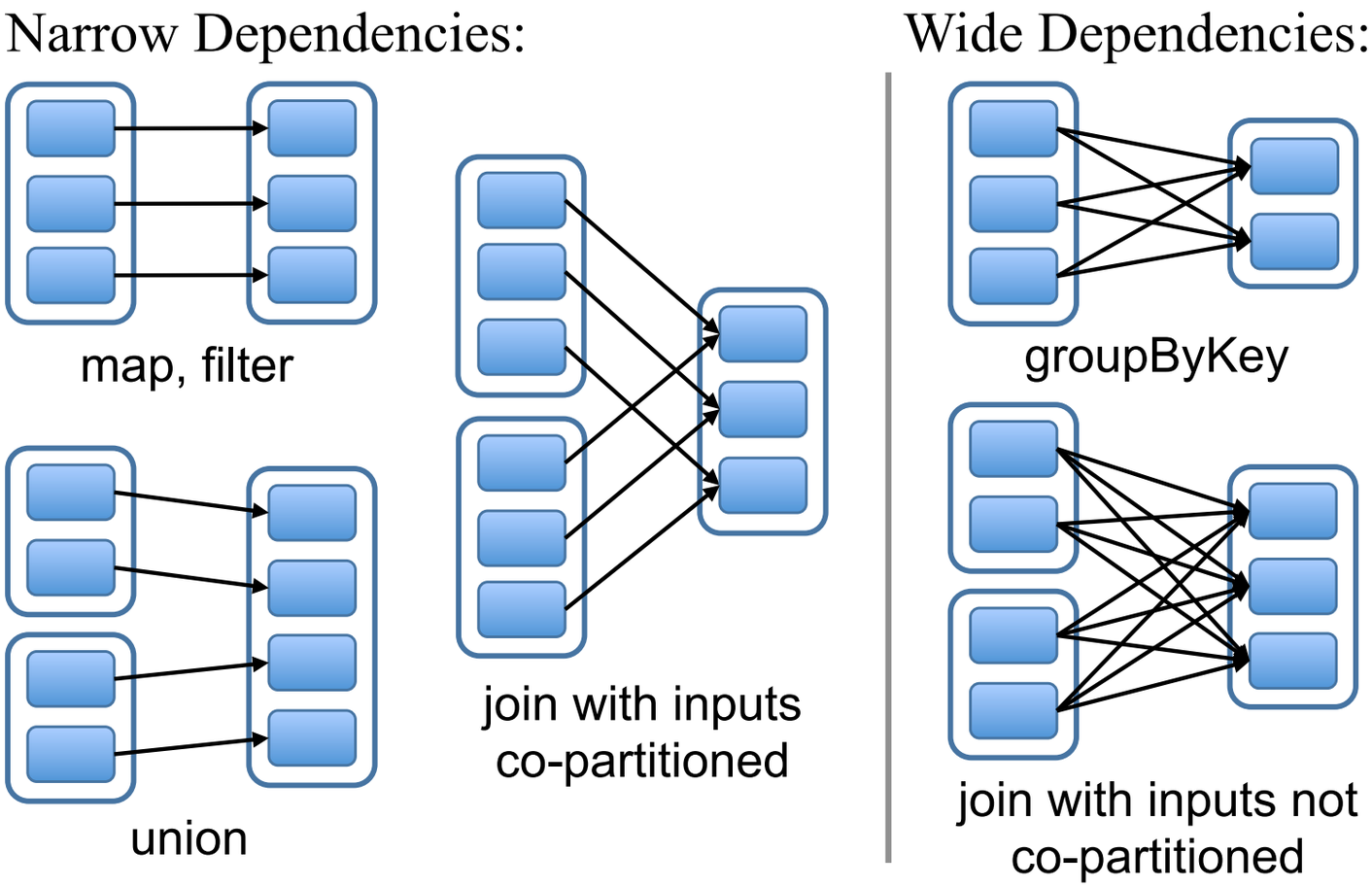


- describes the ith version of the ranks RDD
- have we materialized anything yet?

What an RDD is composed of

Operation	Meaning
partitions()	Return a 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

What an RDD is composed of



- Impact on communication/ scheduing
- Impact on recovery

Why does an RDD carry metadata on its partitioning?

Operation	Meaning
partitions()	Return a 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

Why does an RDD carry metadata on its partitioning?

Transformations that depend on multiple RDDs know whether they need to shuffle data (wide dependency) or not (narrow) and allows users control locality and reduce shuffles.

Handling Faults

- No replication by default — even when persisted
  - ram or disk node failure could mean permanent loss of data
- assume heartbeats used to detect lost worker
  - if worker lost need a way to recompute it's partitions — will need all dependent partitions
  - recompute if they can't be found in RAM or on disk
  - if wide dependency will need all partitions of dependent RDD if narrow then only one partition

So two mechanisms enable recovery from faults: lineage, and policy of what partitions to persist (either to one node or replicated)

The user can call persist() on an RDD.

With RELIABLE flag, will keep multiple copies (in RAM if possible, disk if RAM is full)

With REPLICATE flag, will write to stable storage (HDFS)

Without flags, will try to keep in RAM (will spill to disk when RAM is full)