Resilient Distributed Datasets: Spark

CS 675: Distributed Systems (Spring 2020)

Lecture 6

Yue Cheng

Some material taken/derived from:
- Matei Zarahia's NSDI'12 talk slides.
- Utah CS6450 by Ryan Stutsman.

Announcement

- Deadline of the project report gets extended to 11:59pm next Friday, 04/03

- Doodle poll for Lab 2 demo and proposal discussion meetings (Thursday and Friday)
  - https://doodle.com/poll/tbskp9hqaqsn7ysi

What's good with MapReduce

- Scaled analytics to thousands of machines
- Eliminated fault-tolerance as a concern

Problems with MapReduce

- Scaled analytics to thousands of machines
- Eliminated fault-tolerance as a concern

- Not very expressive
  - Iterative algorithms
    (PageRank, Logistic Regression, Transitive Closure)
  - Interactive and ad-hoc queries
    (Interactive Log Debugging)

- Lots of specialized frameworks
  - Pregel, GraphLab, PowerGraph, DryadLINQ, HaLoop…

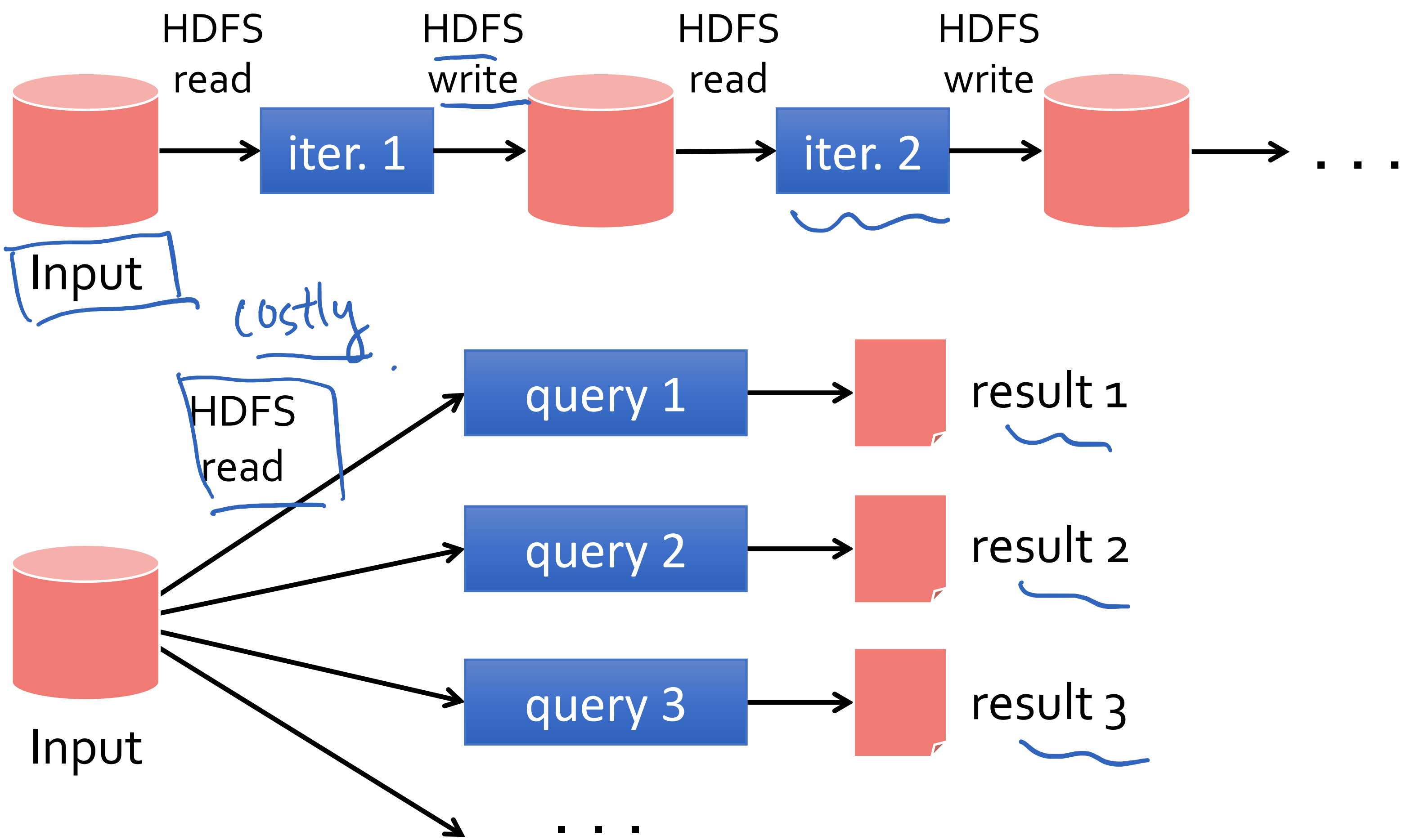
GFS / HDFS

Sharing data between iterations/ops

iter1 → output → iter 2.

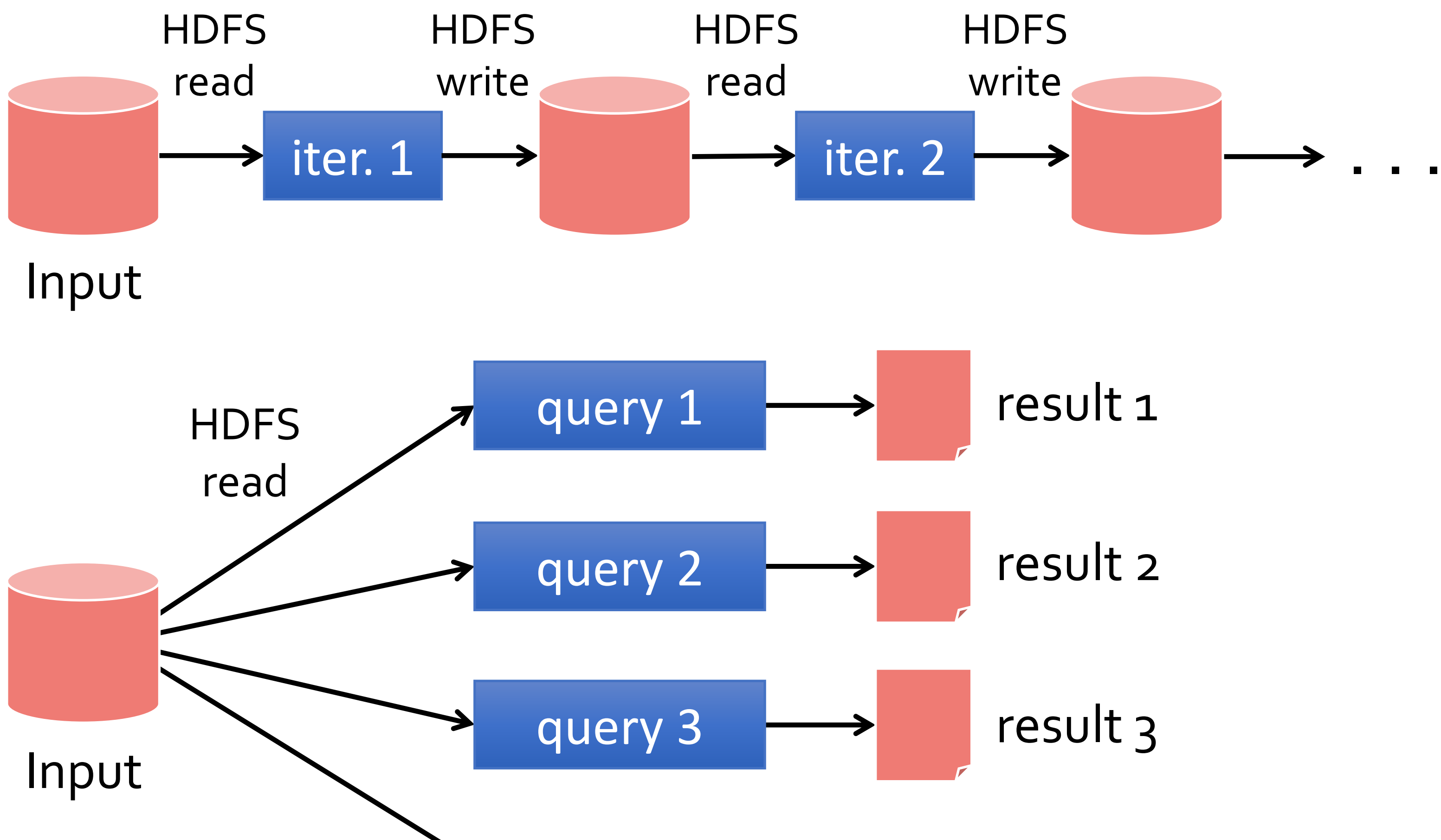
- Only way to share data between iterations / phases is through shared storage
  - Slow!
- Allow operations to feed data to one another
  - Ideally, through memory instead of disk-based storage
- Need the "chain" of operations to be exposed to make this work
- Also, does this break the MR fault-tolerance scheme?
  - Retry and Map or Reduce task since idempotent

Examples



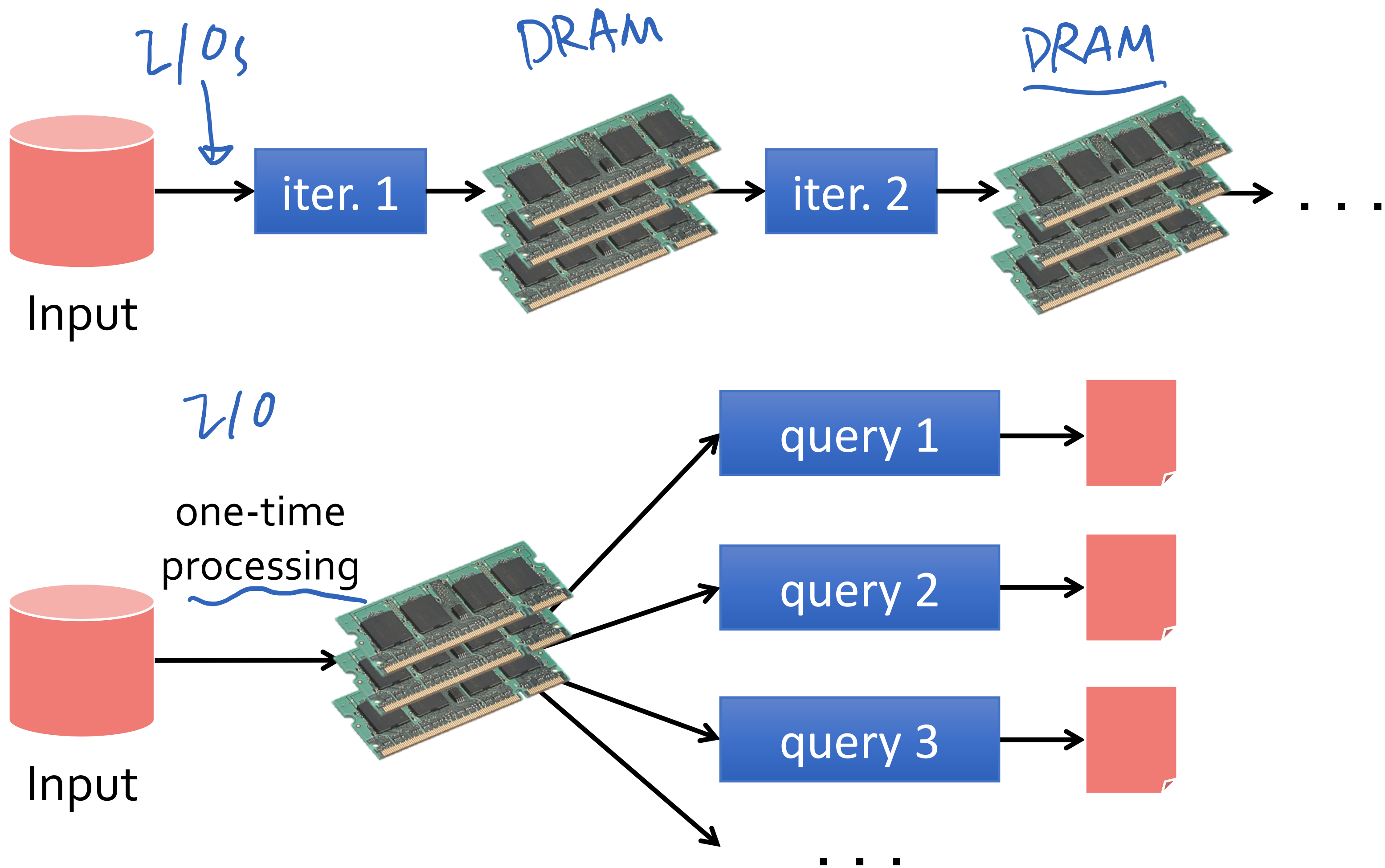
HDFS read
HDFS write
HDFS read
HDFS write
iter. 1
iter. 2
. . .
Input
costly
HDFS read
query 1
query 2
query 3
result 1
result 2
result 3
Input
. . .

Examples

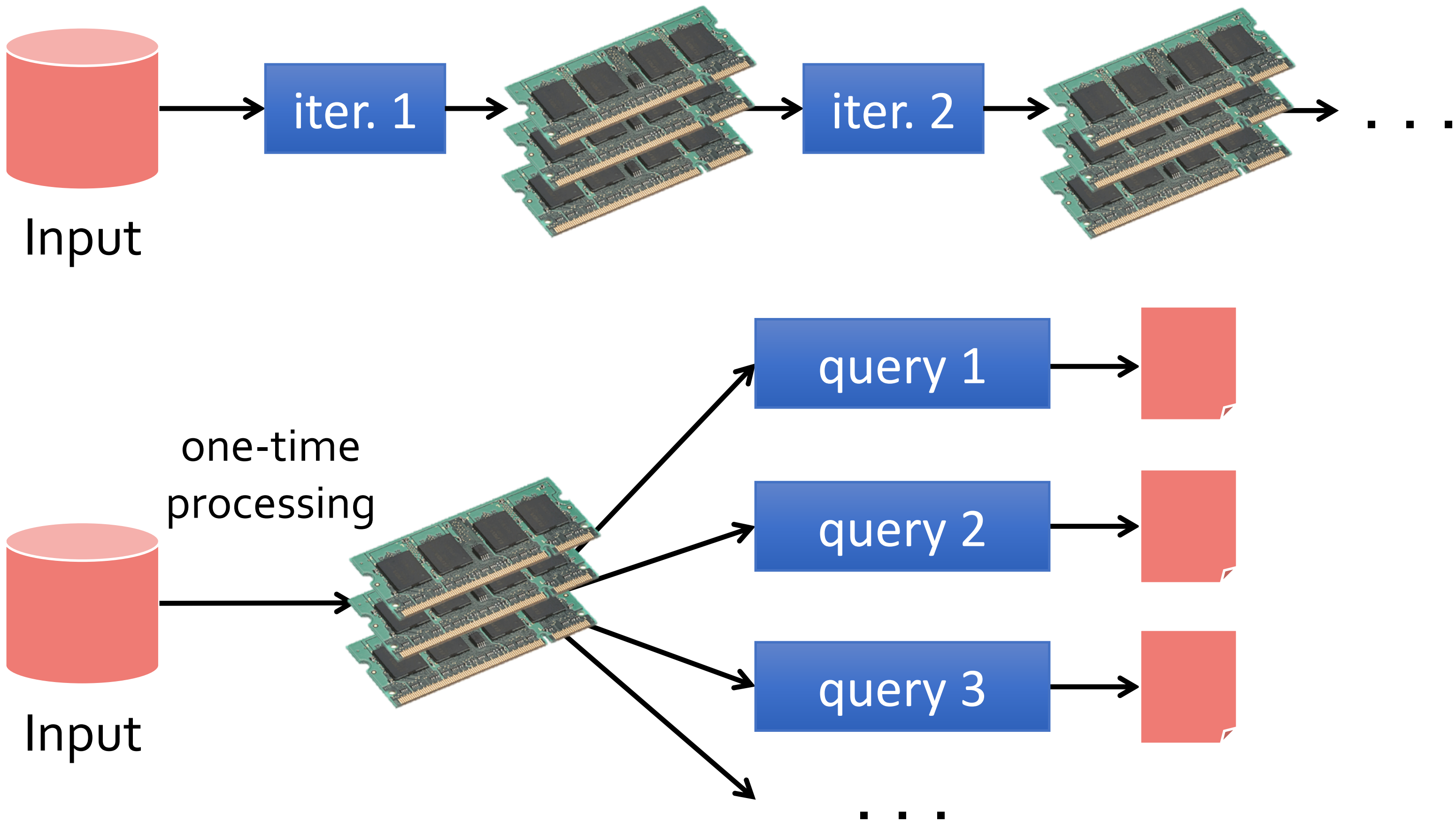


Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-memory data sharing



Goal: In-memory data sharing



10-100× faster than network/disk, but how to get FT?

Challenges

shared

- How to design a distributed memory abstraction that is both fault-tolerant and efficient?

Challenges

- How to design a distributed memory abstraction that is both fault-tolerant and efficient?

in-memory

- Existing storage systems allow fine-grained mutation to state
  - In-memory key-value stores → IMKV
  - Requires replicating data or logs across nodes for fault tolerance
    - Costly for data-intensive apps write-intensive
    - 10-100x slower than memory write
  - They also require costly on-the-fly replication for mutations

Challenges

MR → batch processing

IMKV → fine-grained

- How to design a distributed memory abstraction that is both fault-tolerant and efficient?

Spark: → batch-processing + Iterative + Interactive.

- Existing storage systems allow fine-grained mutation to state

Insight: leverage similar coarse-grained approach that transforms whole dataset per operation, like MapReduce (batch processing)

  - 10-100x slower than memory write
- They also require costly on-the-fly replication for mutations

Solution: Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be 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

Spark programming interface

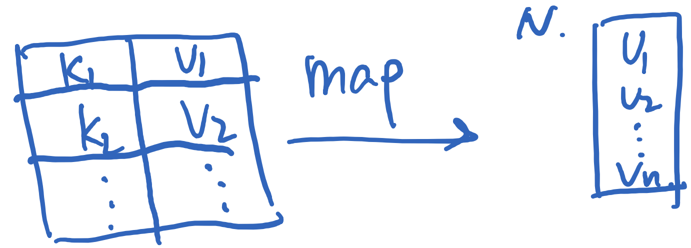
- Scala API, exposed within interpreter as well
- RDDs
- Transformations on RDDs ($RDD_1 \rightarrow RDD_2$)
- Actions on RDDs (RDD → output)
- Control over RDD partitioning (how items are split over nodes)

partitioner func.

- Control over RDD persistence (in memory, on disk, or recompute on loss)

flag

Transformations



Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues

RDDs in terms of Scala types → Scala semantics at workers

Transformations are lazy "thunks"; cause no cluster action

Actions

Directed Acyclic Graph. - DAG.

Actions (return a result to driver program)	collect reduce count save lookupKey

Consumes an RDD to produce output

either to storage (save), or

to interpreter/Scala (count, collect, reduce)

Causes RDD lineage chain to get executed on the cluster to produce the output

(for any missing pieces of the computation)

Interactive debugging

RDD1

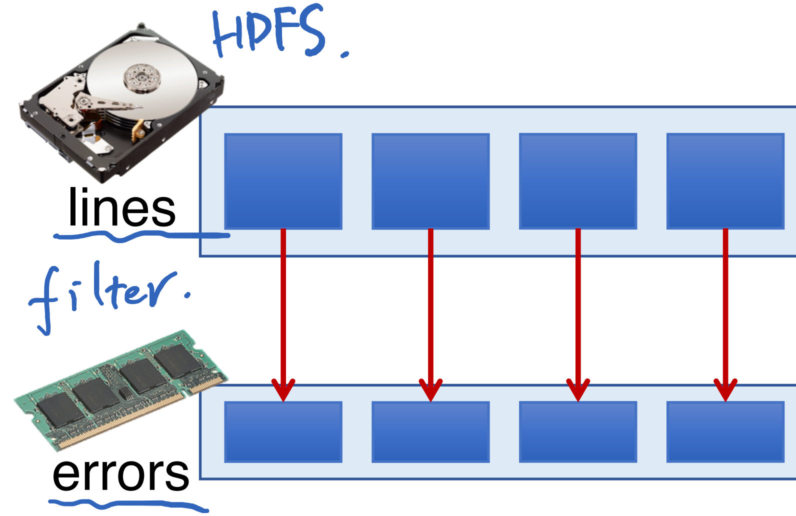
RDD2

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

Interactive debugging

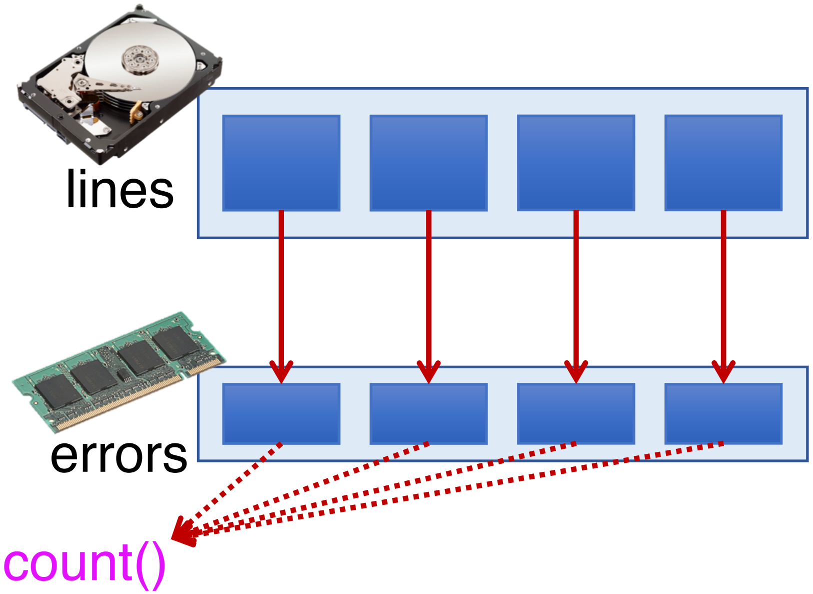
```
lines = textFile("hdfs://foo.log")
errors = lines.filter(
          _.startsWith("ERROR"))
errors.persist()

errors.count()
```

← Action.



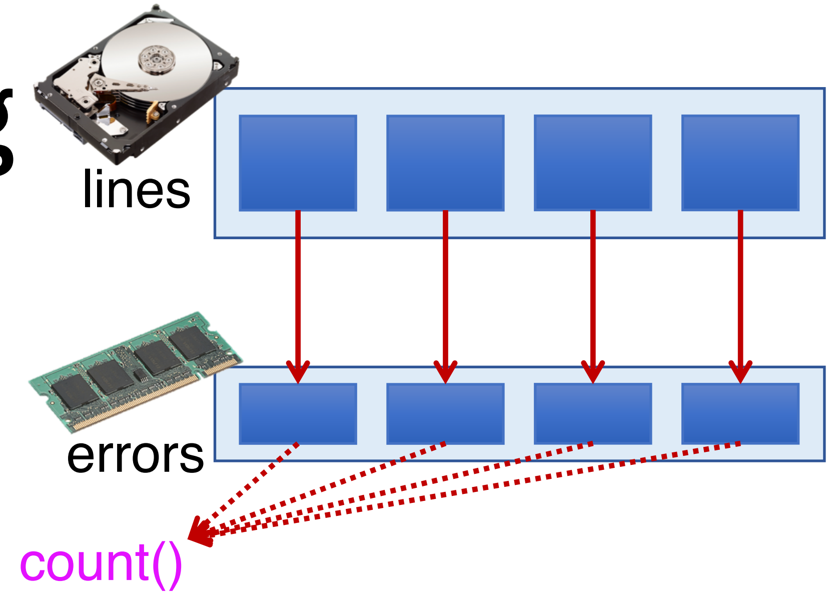
Interactive debugging

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

errors.count()
```

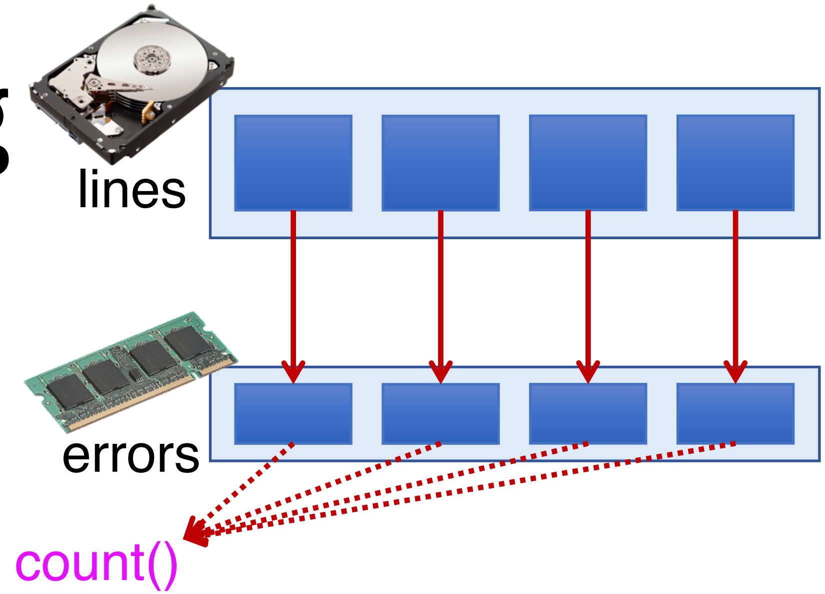


Interactive debugging

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

errors.count()

errors.filter(
       _.contains("MySQL"))
```



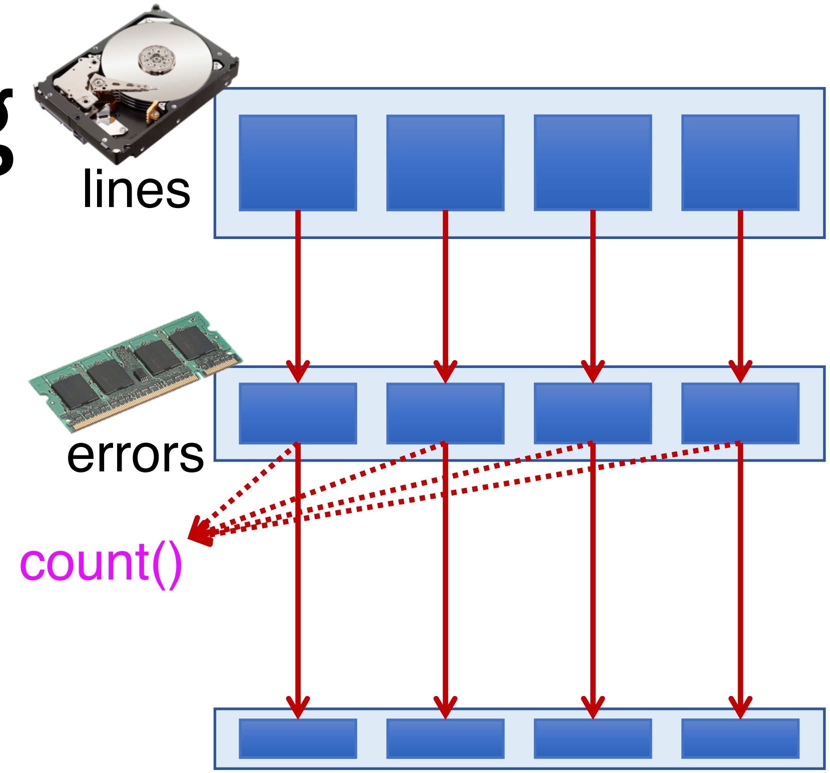
Interactive debugging




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

errors.count()

errors.filter(
     _.contains("MySQL")).count()
```

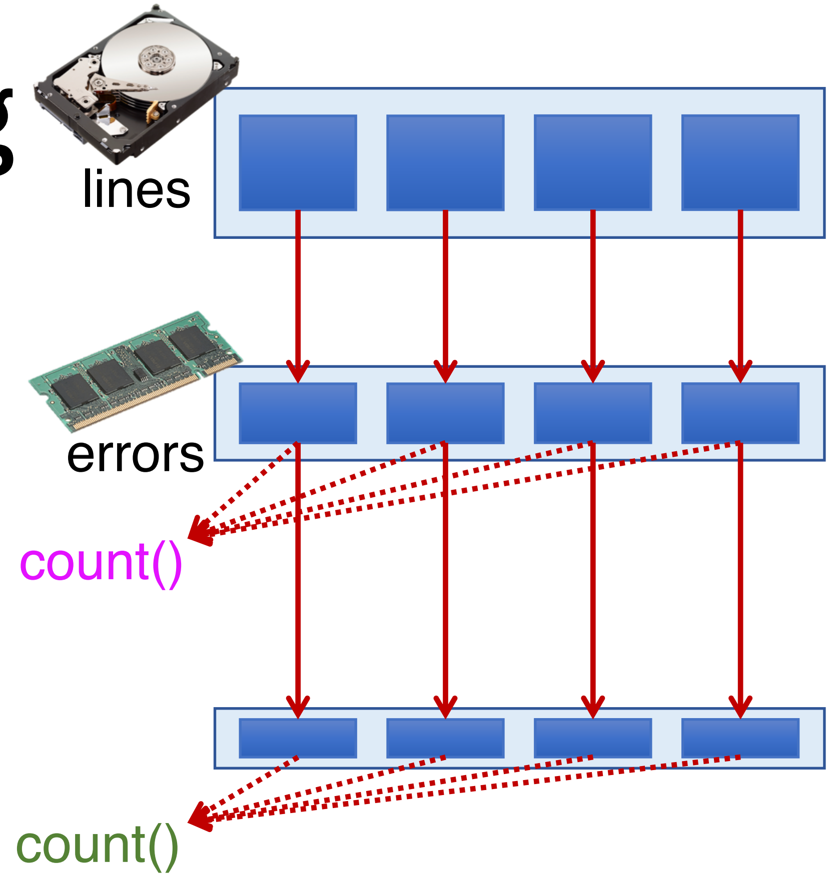
Interactive debugging

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

errors.count()

errors.filter(
       _.contains("MySQL")).count()
```



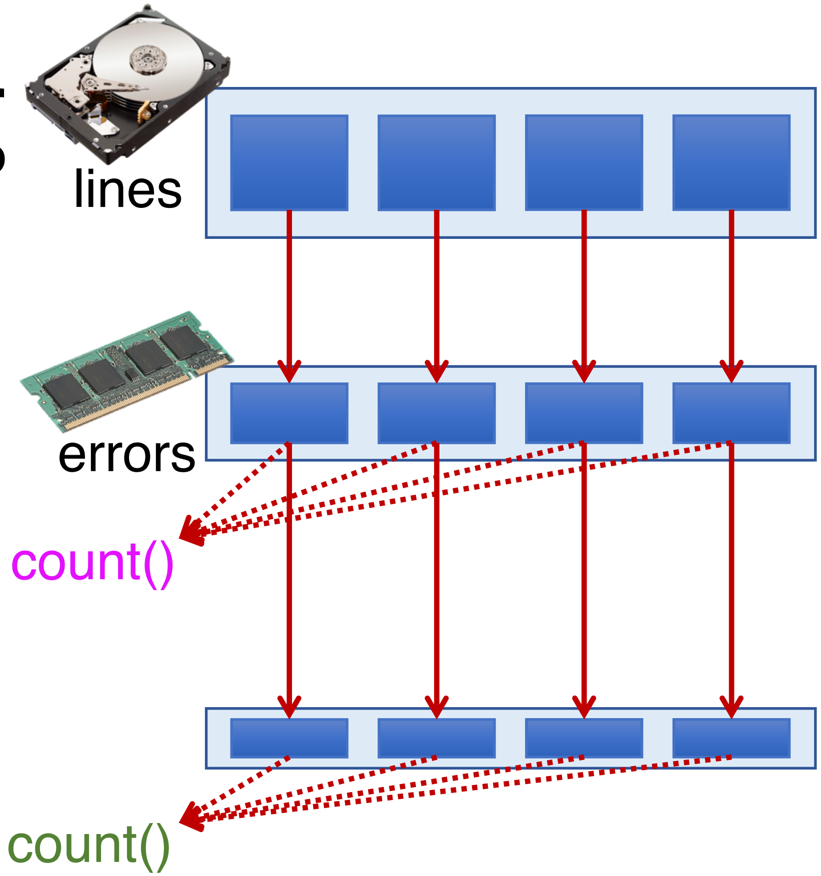
Interactive debugging

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

errors.count()

errors.filter(
     _.contains("MySQL")).count()
```

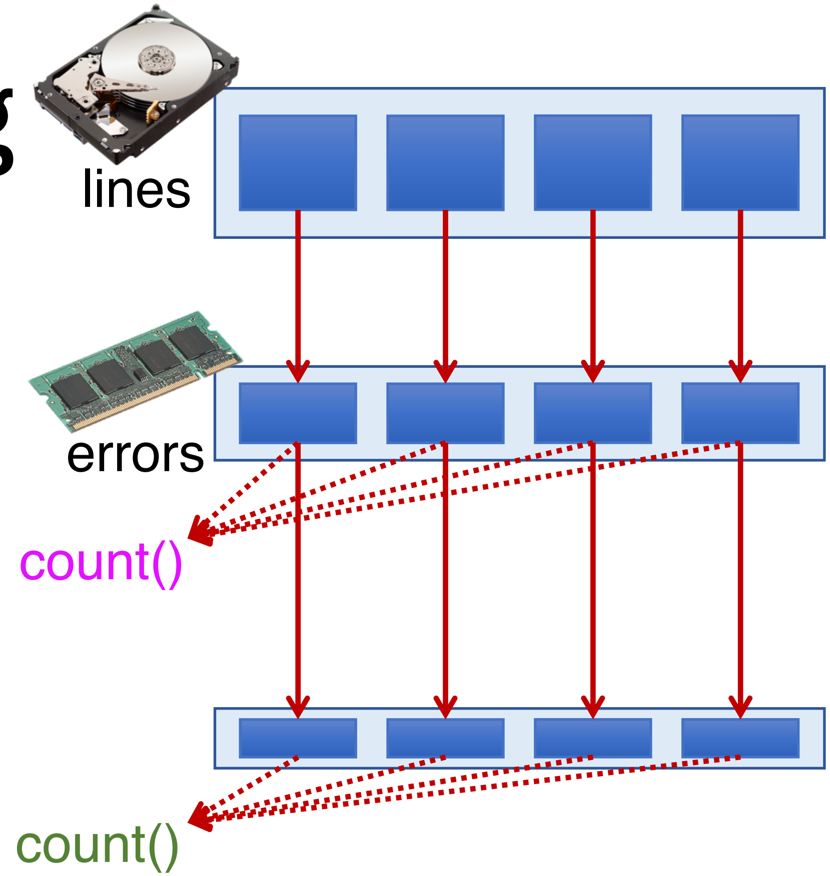


Interactive debugging

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

errors.count()

errors.filter(
     _.contains("MySQL")).count()
errors.filter(
     _.contains("HDFS"))
```



Interactive debugging

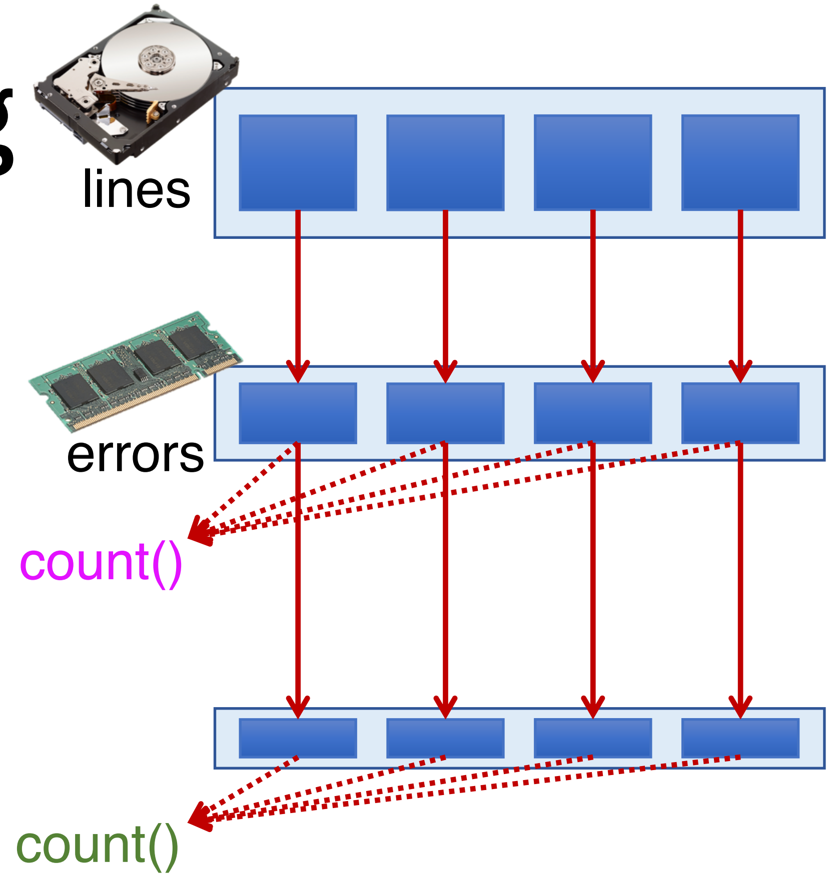



```
lines = textFile("hdfs://foo.log")
errors = lines.filter(
            _.startsWith("ERROR"))
errors.persist()

errors.count()

errors.filter(
      _.contains("MySQL")).count()
errors.filter(
      _.contains("HDFS"))
      _.map(_.split("\t")(3))
```

Interactive debugging

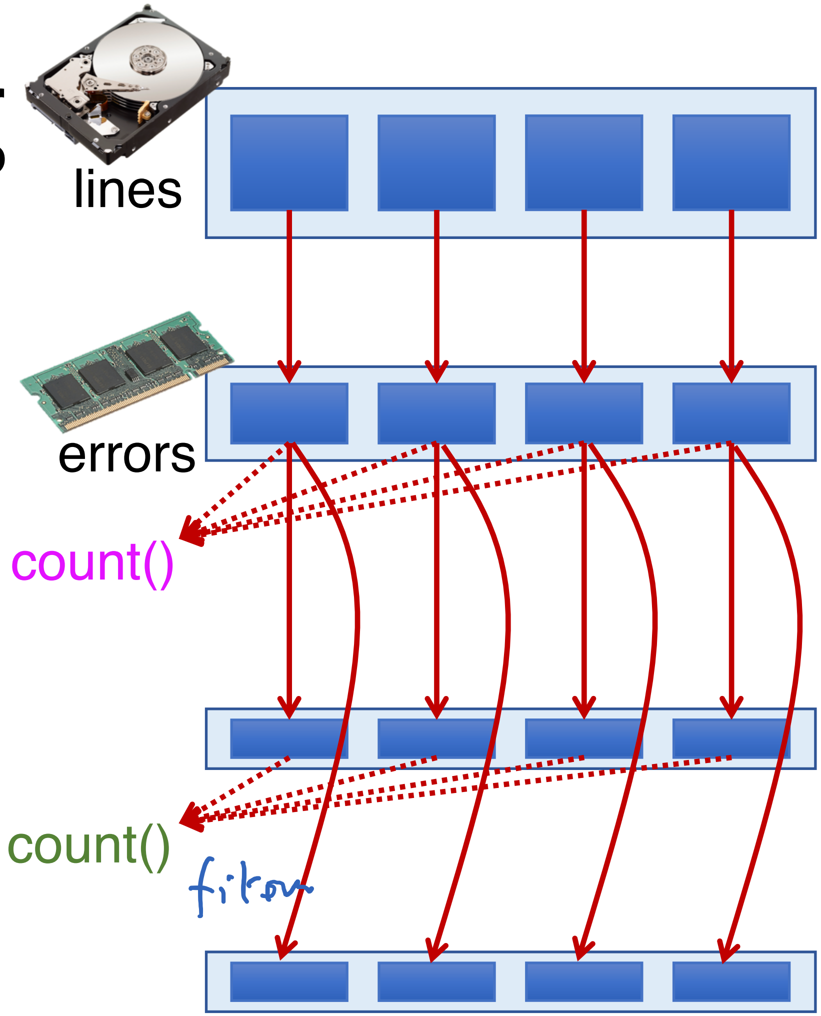
```
lines = textFile("hdfs://foo.log")
errors = lines.filter(
               _.startsWith("ERROR"))
errors.persist()

errors.count()

errors.filter(
       _.contains("MySQL")).count()
errors.filter(
       _.contains("HDFS"))
       _.map(_.split("\t")(3))
       .collect()
```

Action



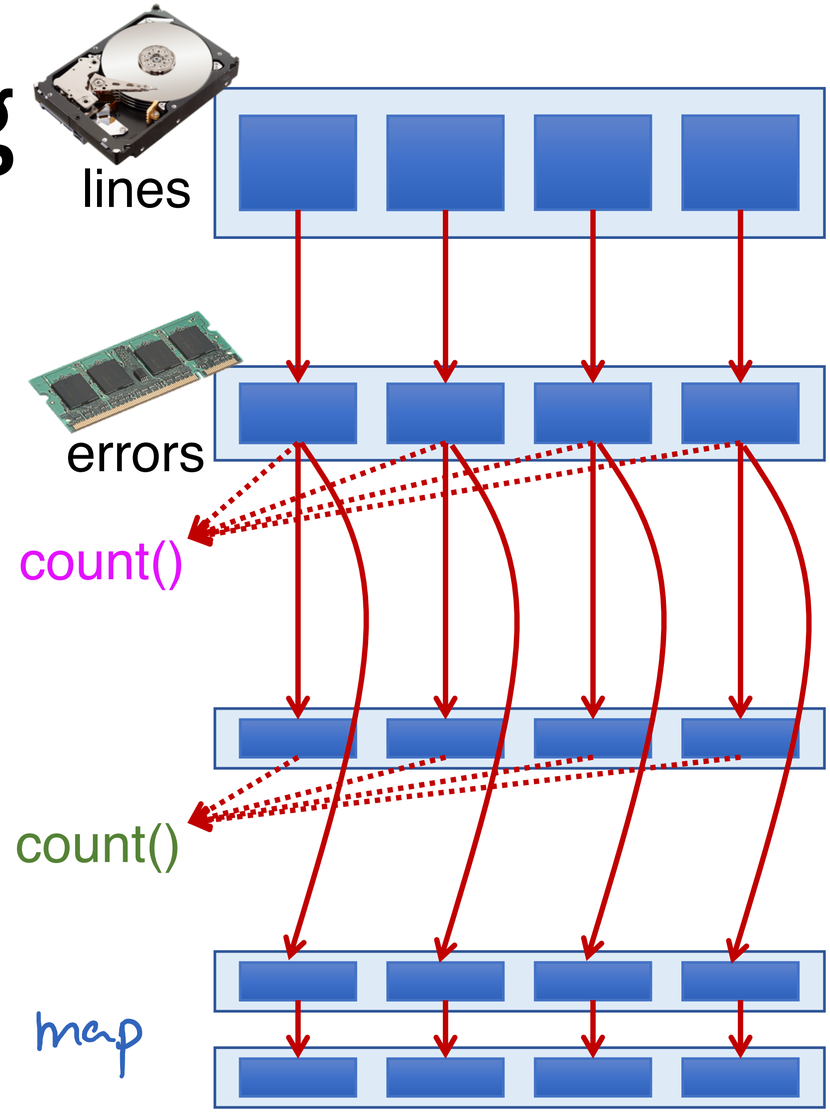
Interactive debugging

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

errors.count()

errors.filter(
      _.contains("MySQL")).count()
errors.filter(
      _.contains("HDFS"))
      _.map(_.split("\t")(3))
      .collect()
```



Interactive debugging

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

errors.count()

errors.filter(
     _.contains("MySQL")).count()
errors.filter(
     _.contains("HDFS"))
     _.map(_.split("\t")(3))
     .collect()
```

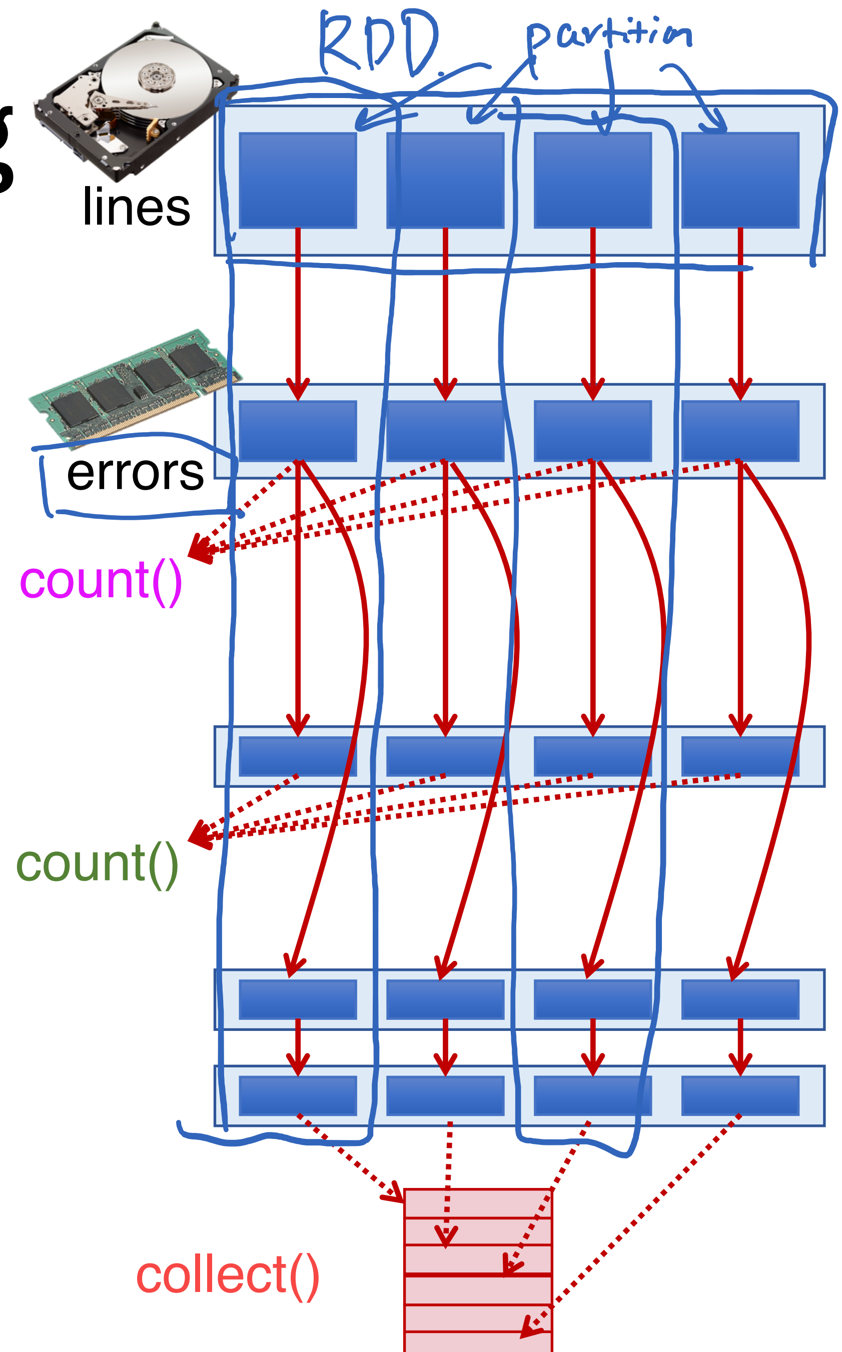


Interactive debugging

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

→ hint → sched.

```
errors.count()

errors.filter(
      _.contains("MySQL")).count()
errors.filter(
      _.contains("HDFS"))
      _.map(_.split("\t")(3))
      .collect()
```



persist()

- Not an action and not a transformation
- A scheduler hint
- Tells which RDDs the Spark schedule should materialize and whether in memory or storage
- Gives the user control over reuse/recompute/recovery tradeoffs

persist()

- Not an action and not a transformation
- A scheduler hint
- Tells which RDDs the Spark schedule should materialize and whether in memory or storage
- Gives the user control over reuse/recompute/recovery tradeoffs

- Q: If persist() asks for the materialization of an RDD why isn't it an action?

Lineage graph of RDDs

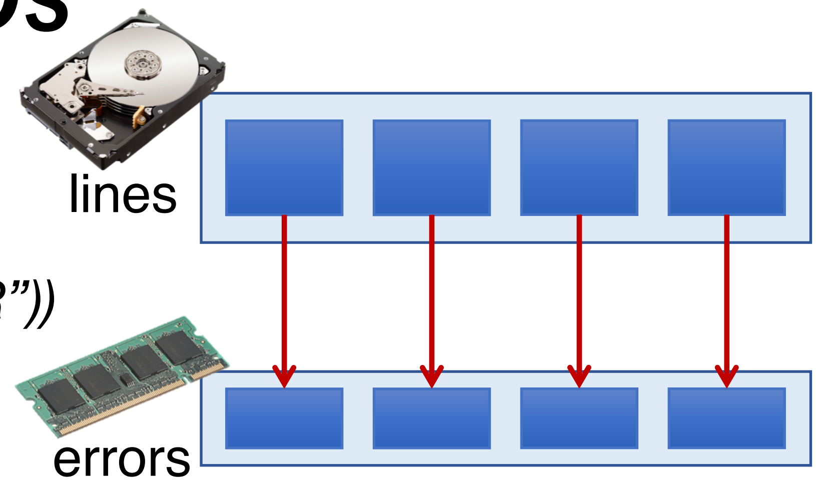
`lines`

Lineage graph of RDDs

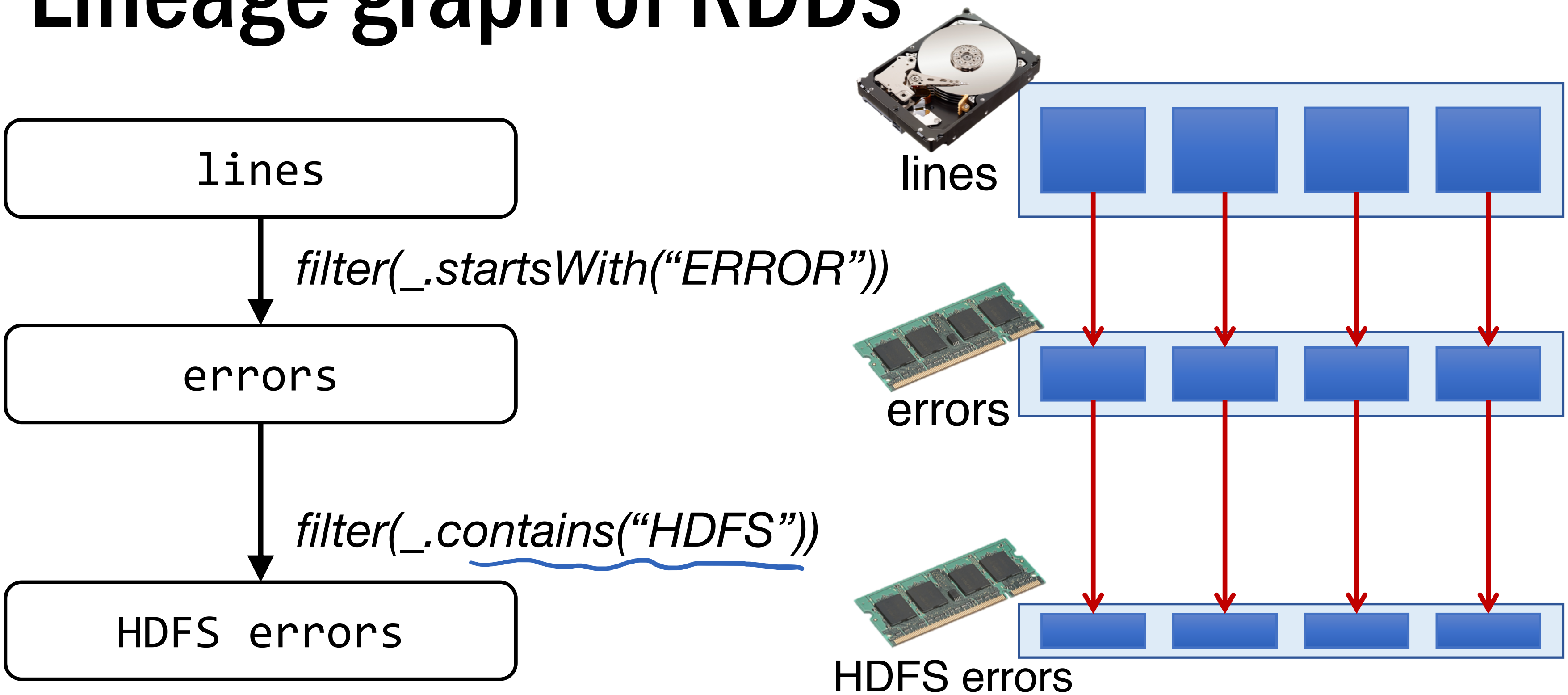
`lines`

↓ *filter(_.startsWith("ERROR"))*

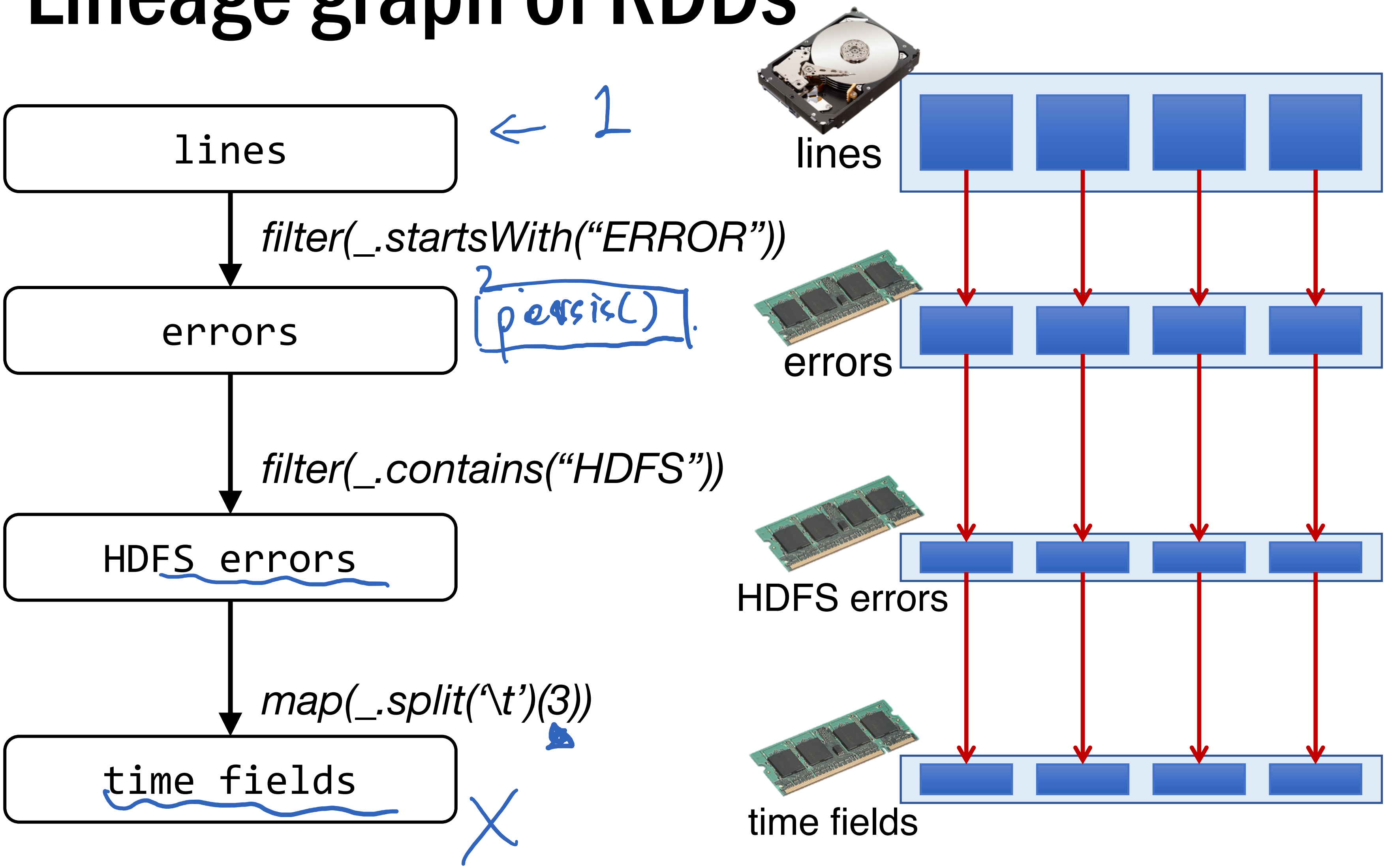
`errors`



Lineage graph of RDDs

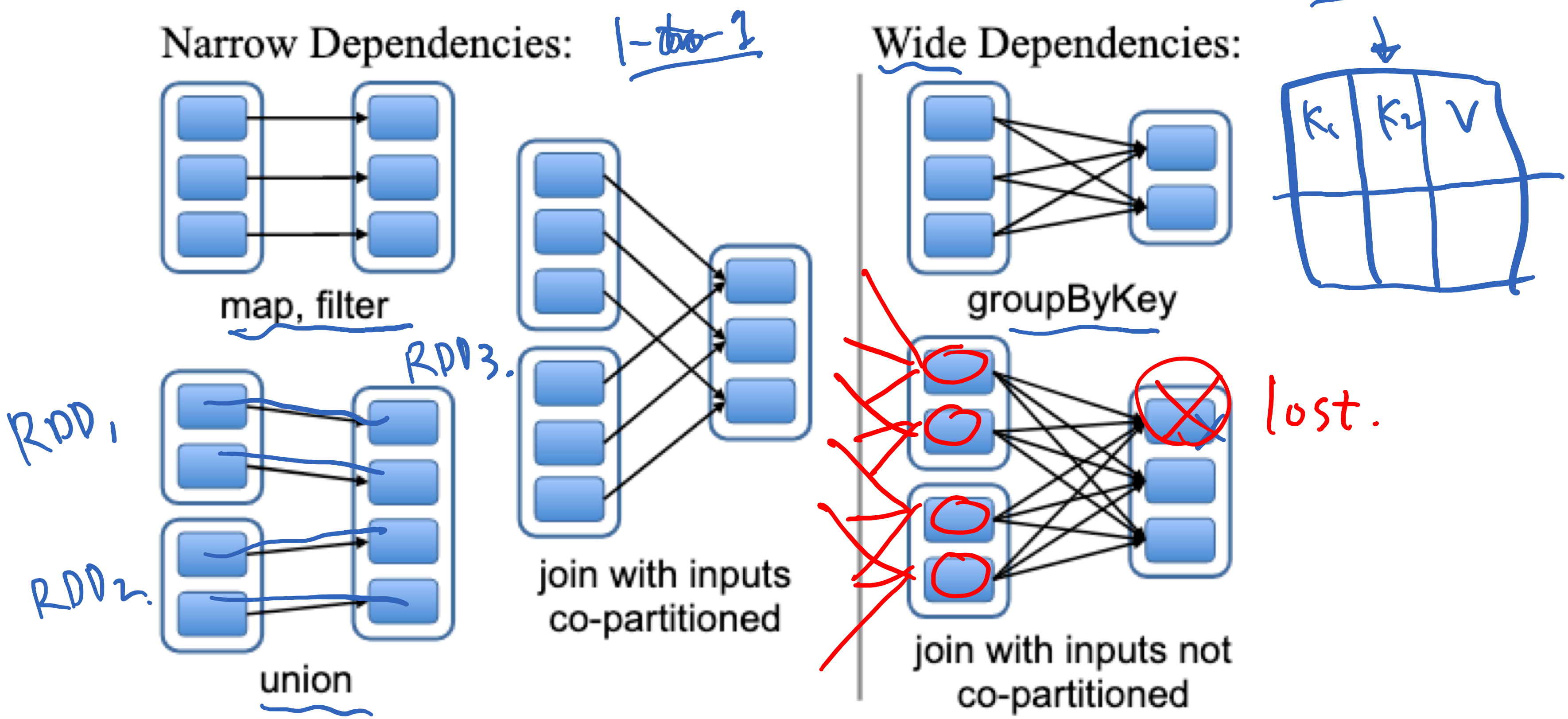


Lineage graph of RDDs



Narrow & wide dependencies

M-N



Narrow: each parent partition used by at most one child partition (can partition on one machine)

Wide: multiple child partitions depend on one parent partition

Must stall for all parent data, loss of child requires whole parent RDD (not just a small # of partitions)

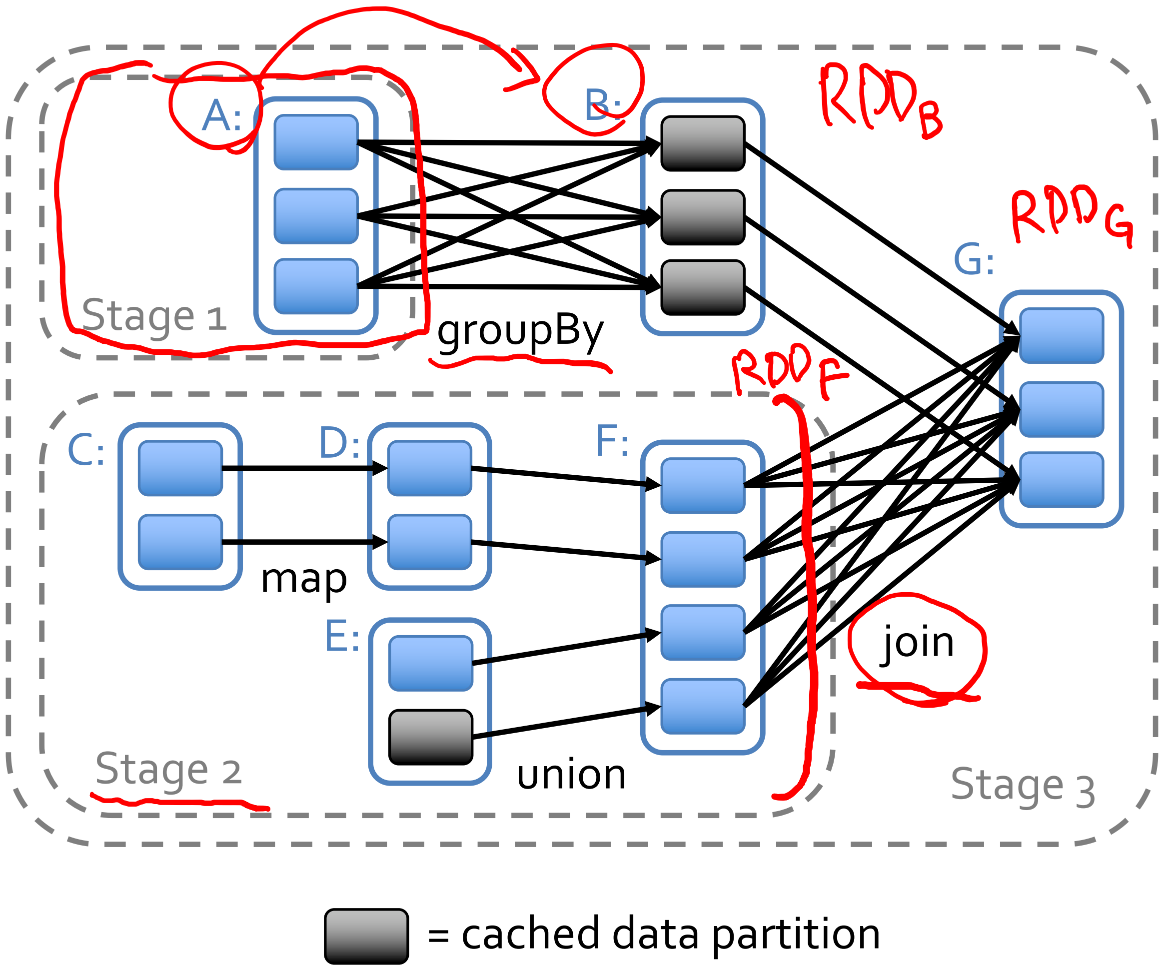
Task scheduler

Dryad-like DAGs

Pipelines functions within a stage

Locality & data reuse aware

Partitioning-aware to avoid shuffles



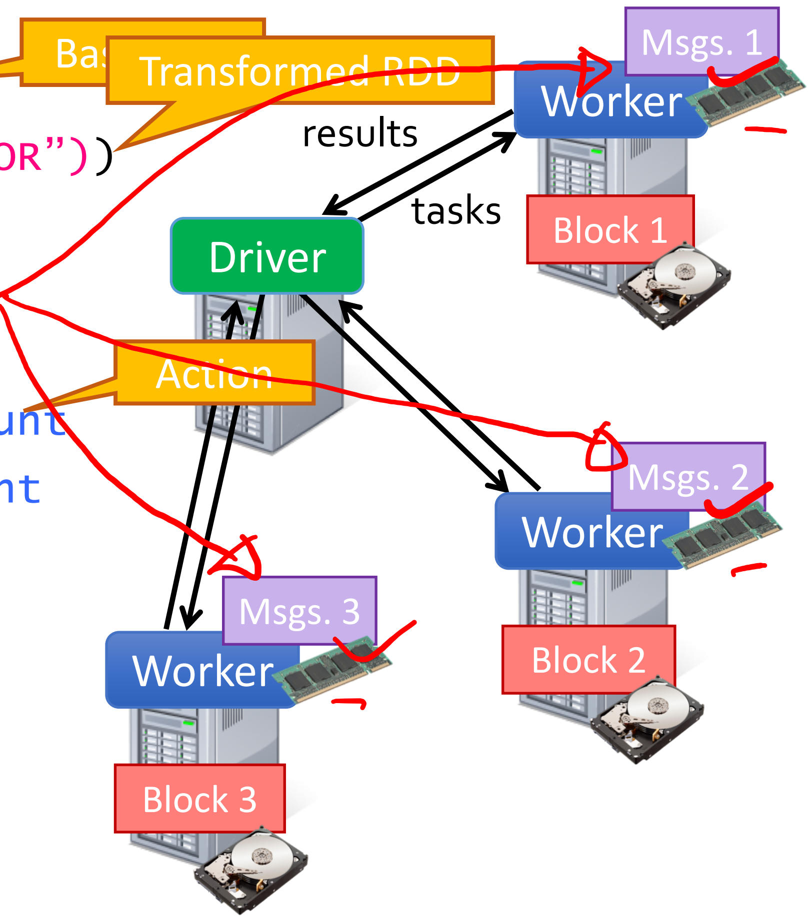
Interactive debugging (control and data flow)

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))
messages.persist()

messages.filter(_.contains("MySQL")).count
messages.filter(_.contains("HDFS")).count
```

transf

Action



Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Fault recovery

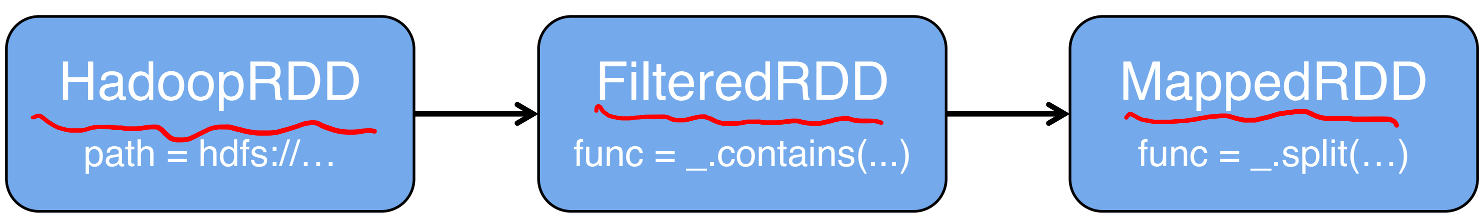
- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

pattern

E.g.:

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

2nd col

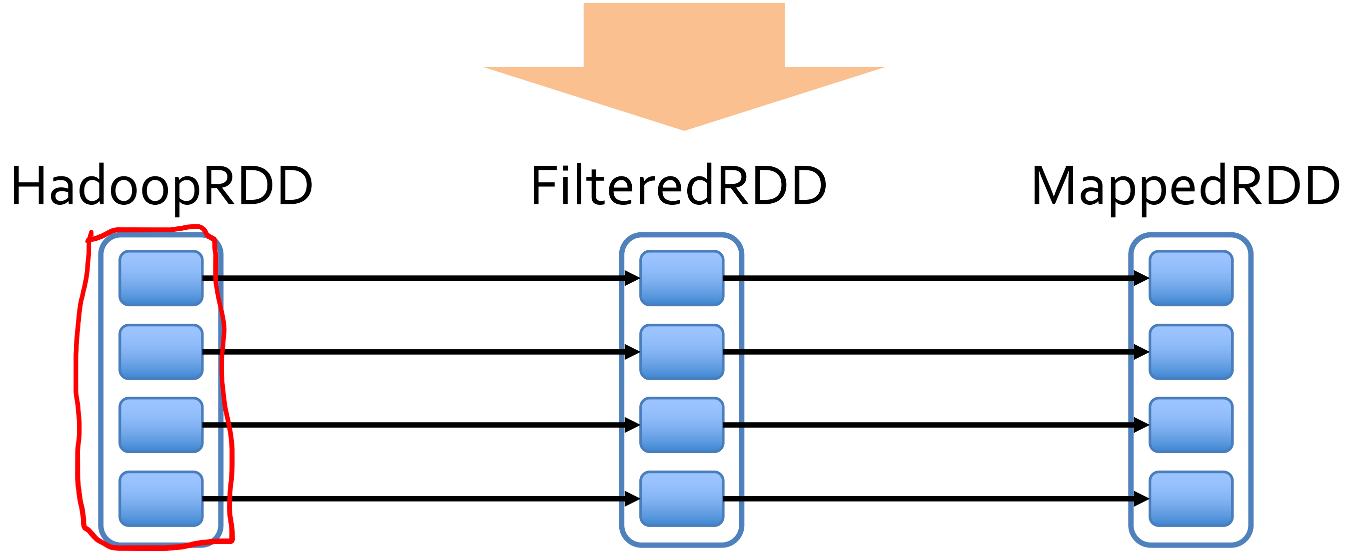


Fault recovery

- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



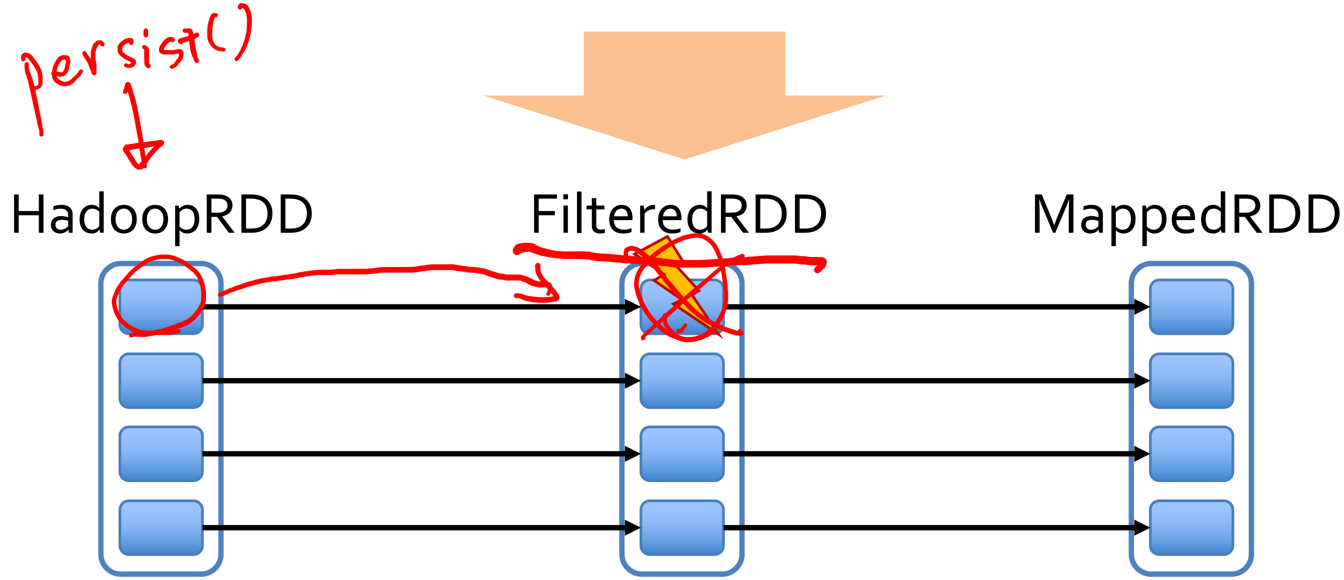
Fault recovery

- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

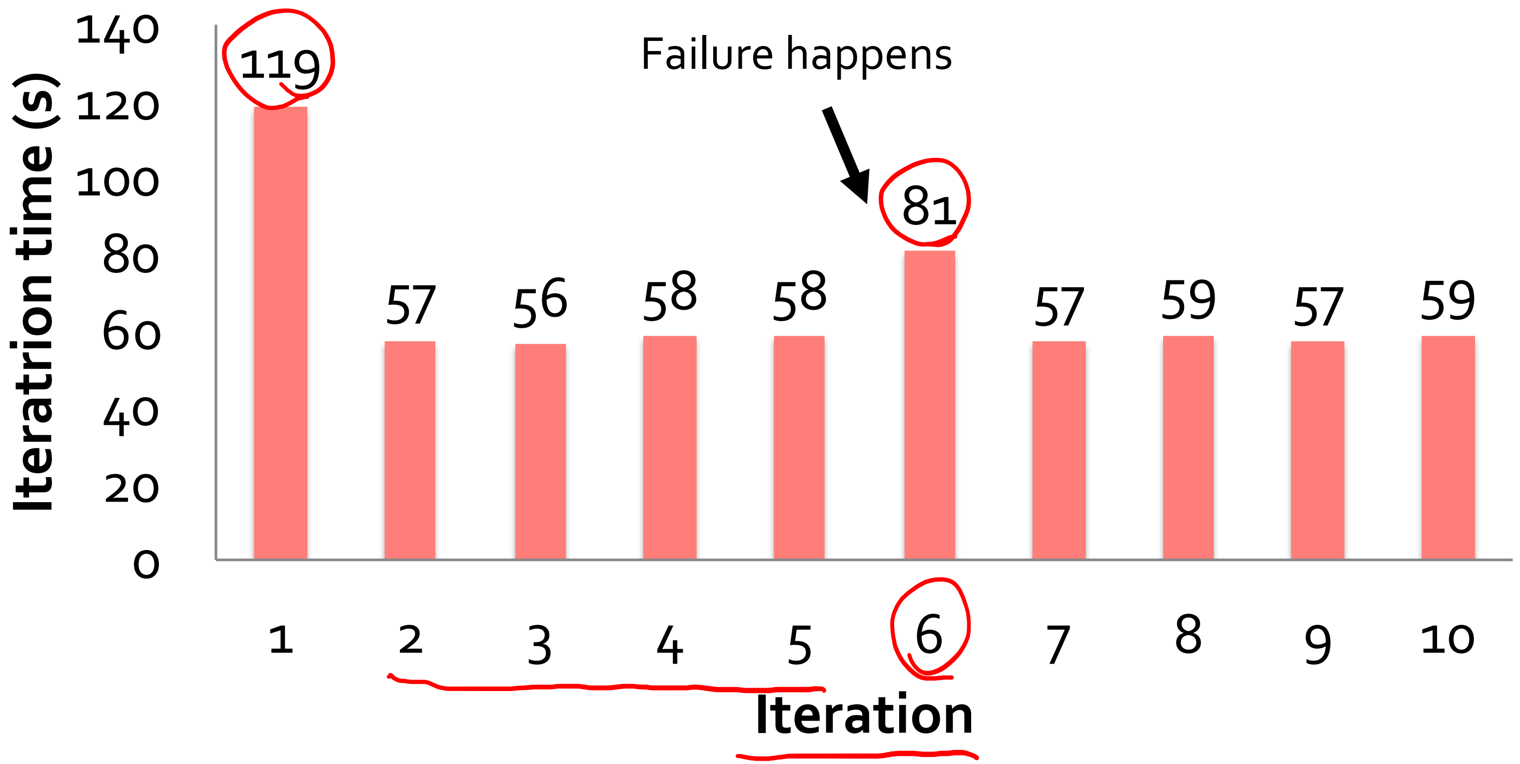
E.g.:

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

persist()



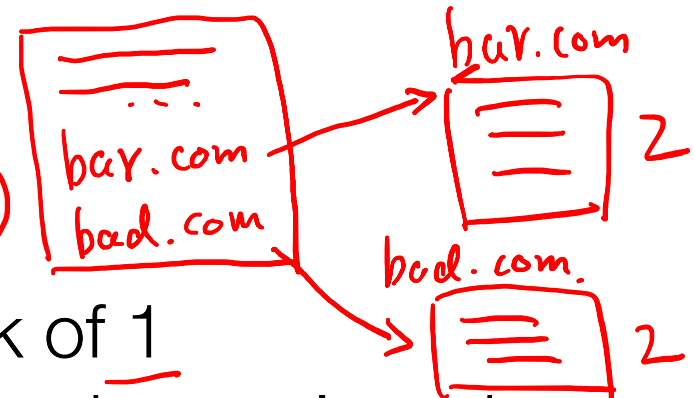
Fault recovery results



foo.com

Example: PageRank

4



1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\Sigma_{i \in neighbors} \text{rank}_i \,/\, |\text{neighbors}_i|$$

$\frac{4}{|\text{neighbors}|_{foo}} = \frac{4}{2} = 2$

```
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(_ + _)
}
```

Contrib factor

URL

Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\Sigma_{i \in neighbors} \text{rank}_i / |\text{neighbors}_i|$$

RDD[(URL, Seq[URL])]

```
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(_ + _)
}
```

RDD[(URL, Rank)]

key

RDD[(URL, (Seq[URL],Rank))]

For each neighbor in links emits (URL, RankContrib)

Reduce to RDD[(URL, Rank)]

Join (⋈)

	Age
Alice	5
Bob	6
Claire	4

⋈

	Gender.
Alice	F
Bob	M
Claire	F

=

	Age.	Gender.
Alice	5	F
Bob	6	M
Claire	4	F

T_1

S_1	A	5
	A	2
	A	3
S_2	B	4
	B	1
S_3	C	6
	C	8

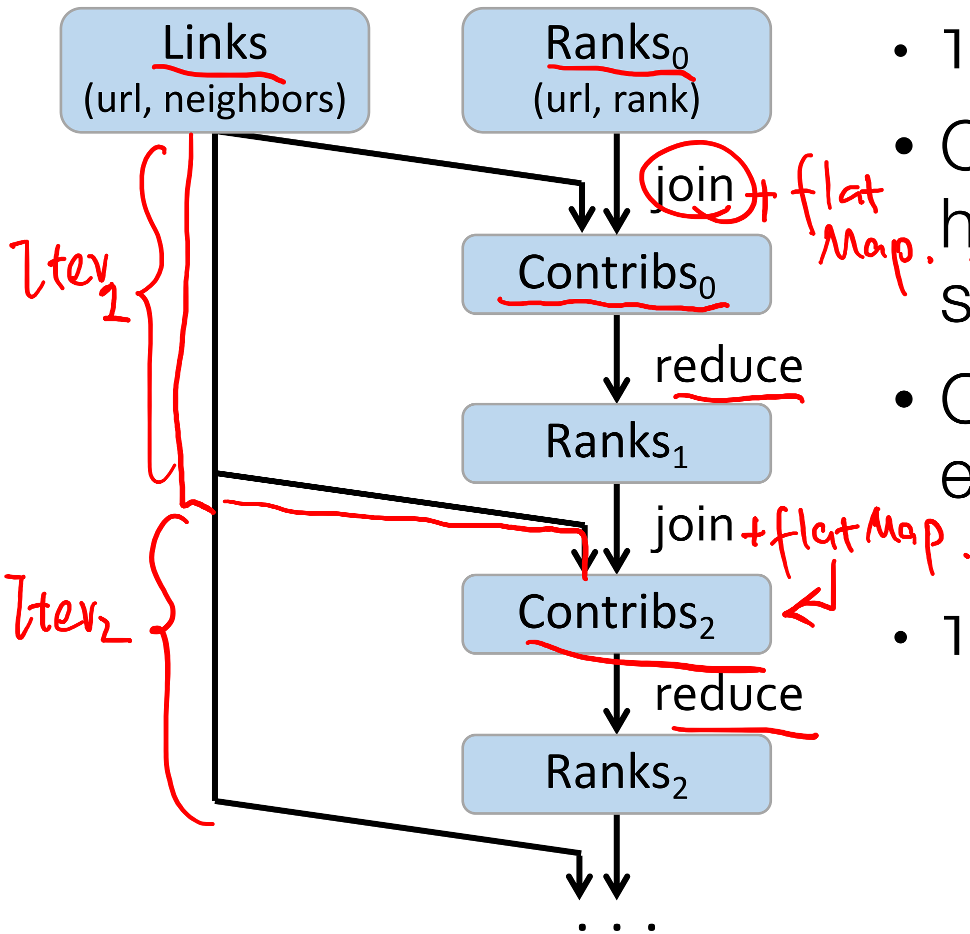
⋈

T_2

C	5	S_1
B	2	
A	3	
B	4	S_2
A	1	
B	6	
C	8	

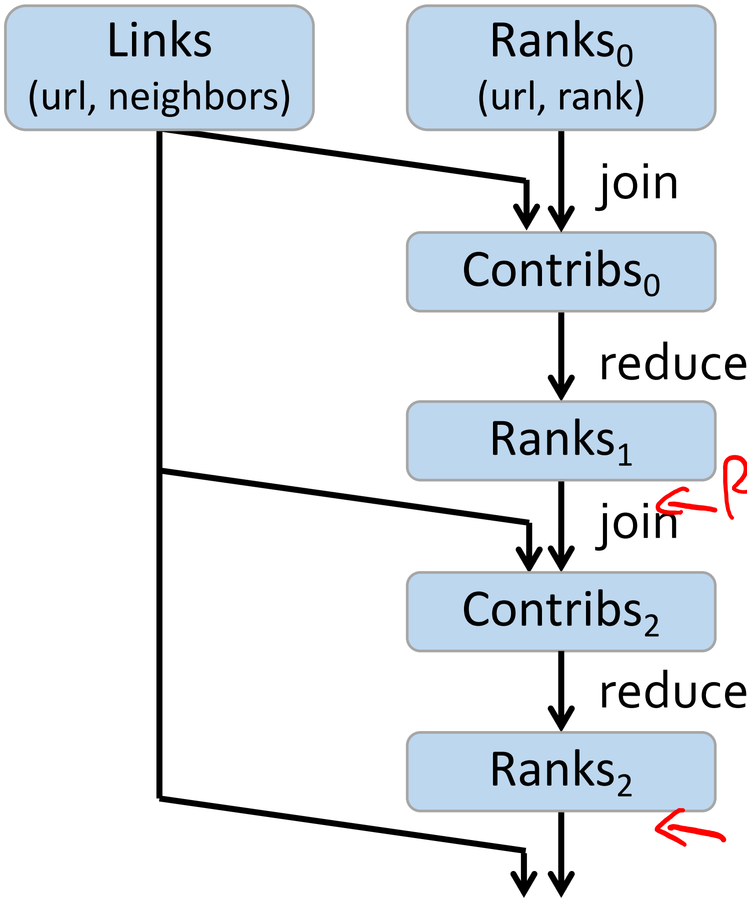
If partitioning doesn't match, then need to reshuffle to match pairs. Same problem in `reduce()` for MapReduce.

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())`

Optimizing placement



Persist

?? Persist.

- `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()`**?

flatMap.

reduceByKey

Co-partitioning example

ranks

foo	5
foo	4
widget.	3
bar	2
foo	2

N/W cost

bar.	2
bad.	0
foo.	5+4+2=11 → 11
widget	3

$\frac{5}{1}$ $\frac{4}{1}$

links:

S_1	bar	foo
	bad	foo
S_2	foo	widget
	widget	bar, foo

⋈

ranks,

bar	5	S_1
bad	4	
foo	3	S_2
widget	4	

=

flatten

bar	foo	5
bad	foo	4
foo	widget	3
widget	bar, foo	4

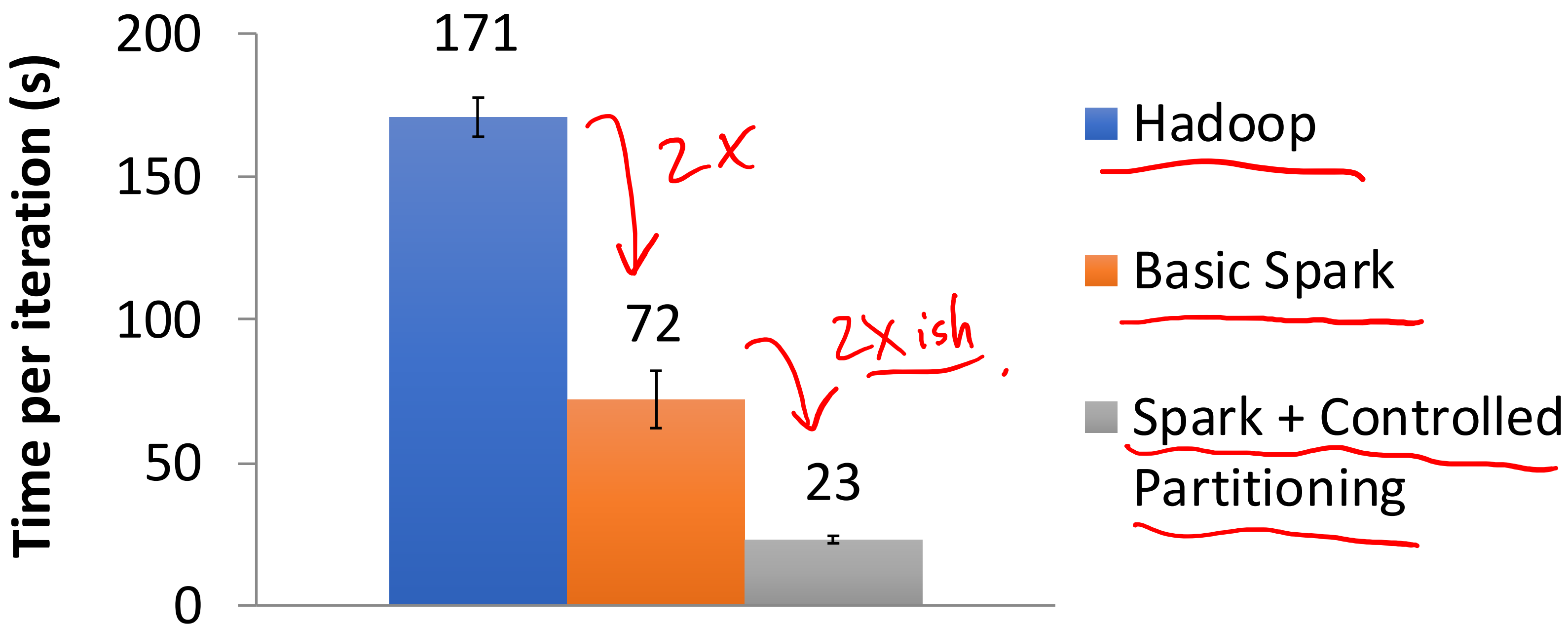
rank
neighbor key

$3/1$ $\frac{4}{2}$ $\frac{4}{2}$

- Co-partitioning can avoid shuffle on join
- But, fundamentally a shuffle on `reduceByKey`
- Optimization: custom partitioner on domain

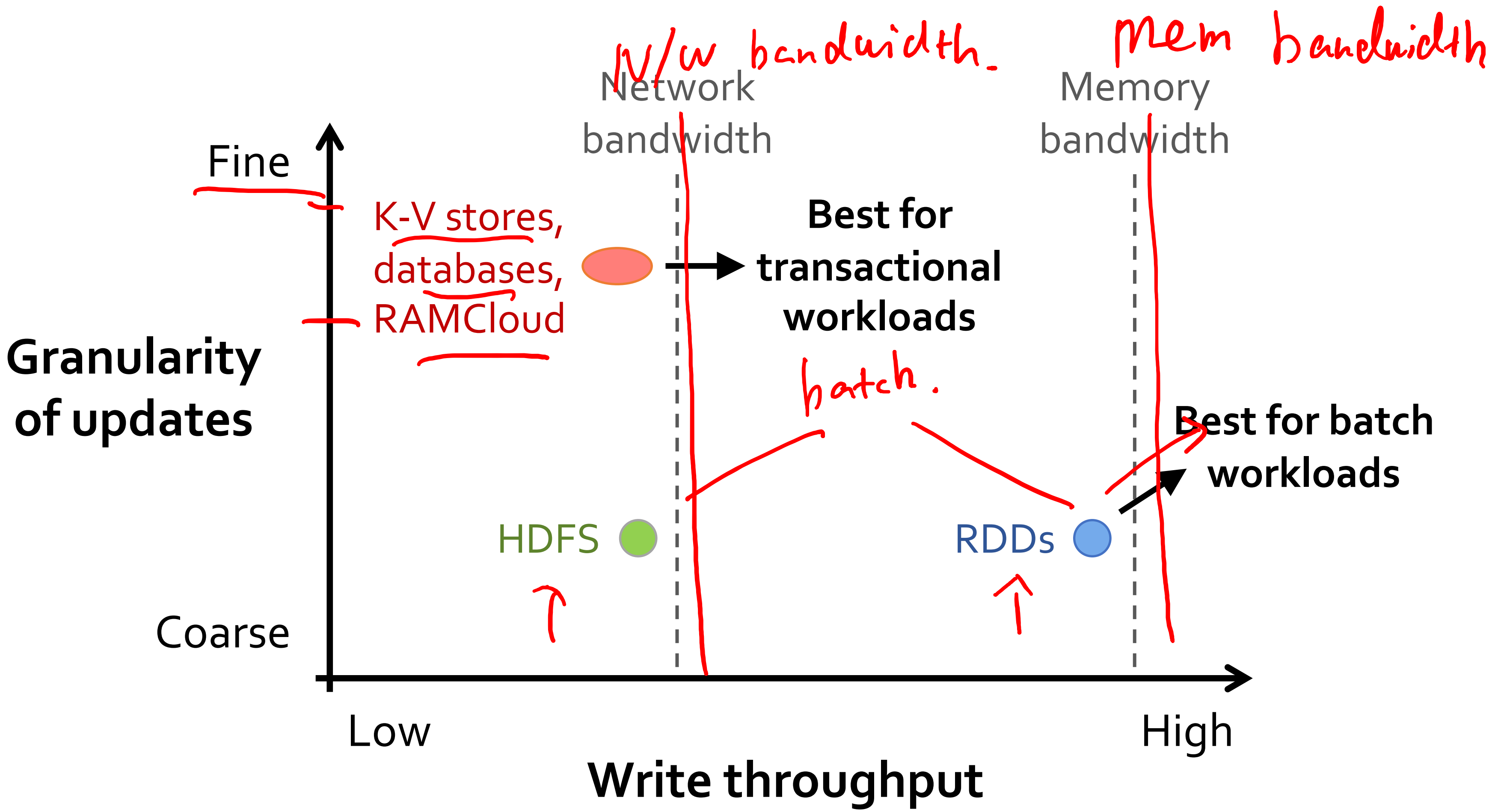
.com .edu

PageRank performance



* Figure 10a: 30 machines on 54 GB of Wikipedia data computing PageRank

Tradeoff space



Discussion & wrap-up

MapReduce. expressivity poor. → bad performance.

MR

Spark. expressivity
- iterative
- ad-hoc interactive.

tradeoff.

users. → reason about FT

(persist())