Resilient Distributed Datasets: Spark

CS 475: Concurrent & Distributed Systems (Fall 2021)

Lecture 16

Yue Cheng

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

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…

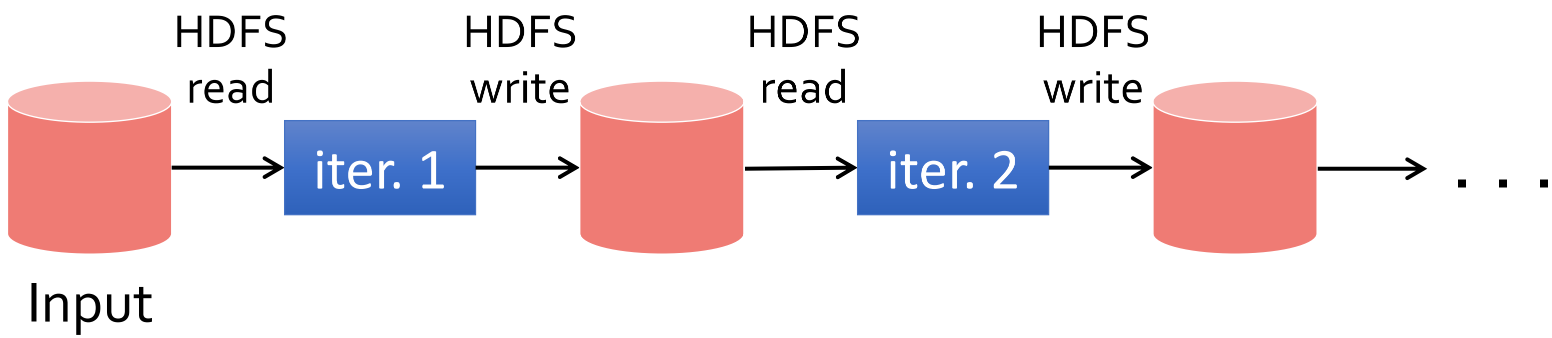
Sharing data between iterations/ops

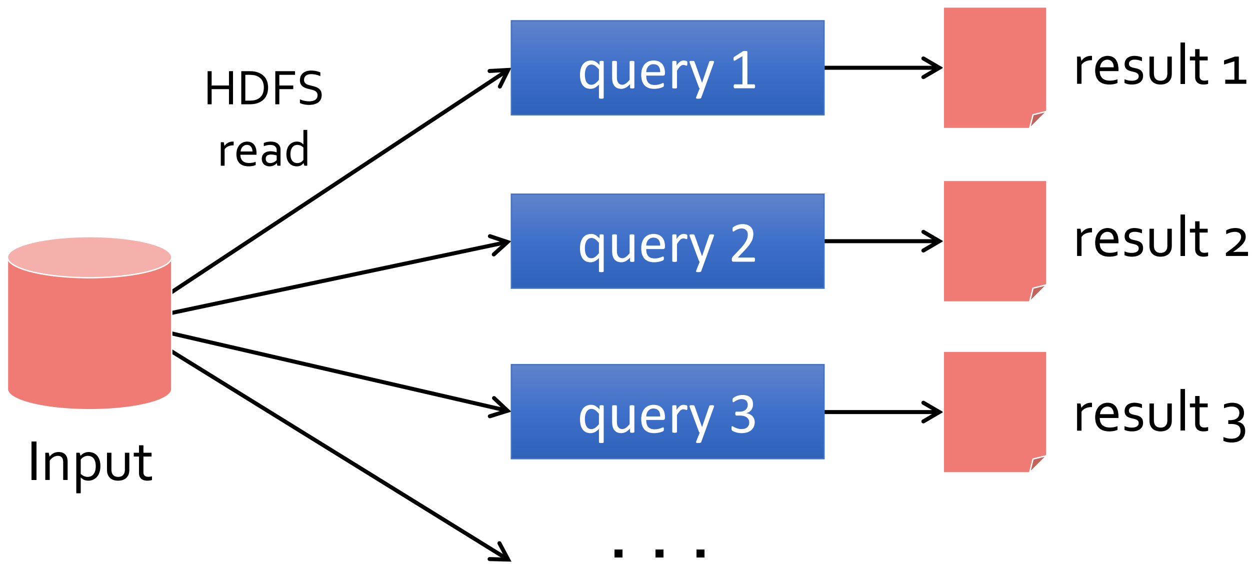
- Only way to share data between iterations / phases is through shared storage
  - Slow!
- Allow operations to feed data to one another
  - Ideally, through memory instead of disk-based storage

Sharing data between iterations/ops

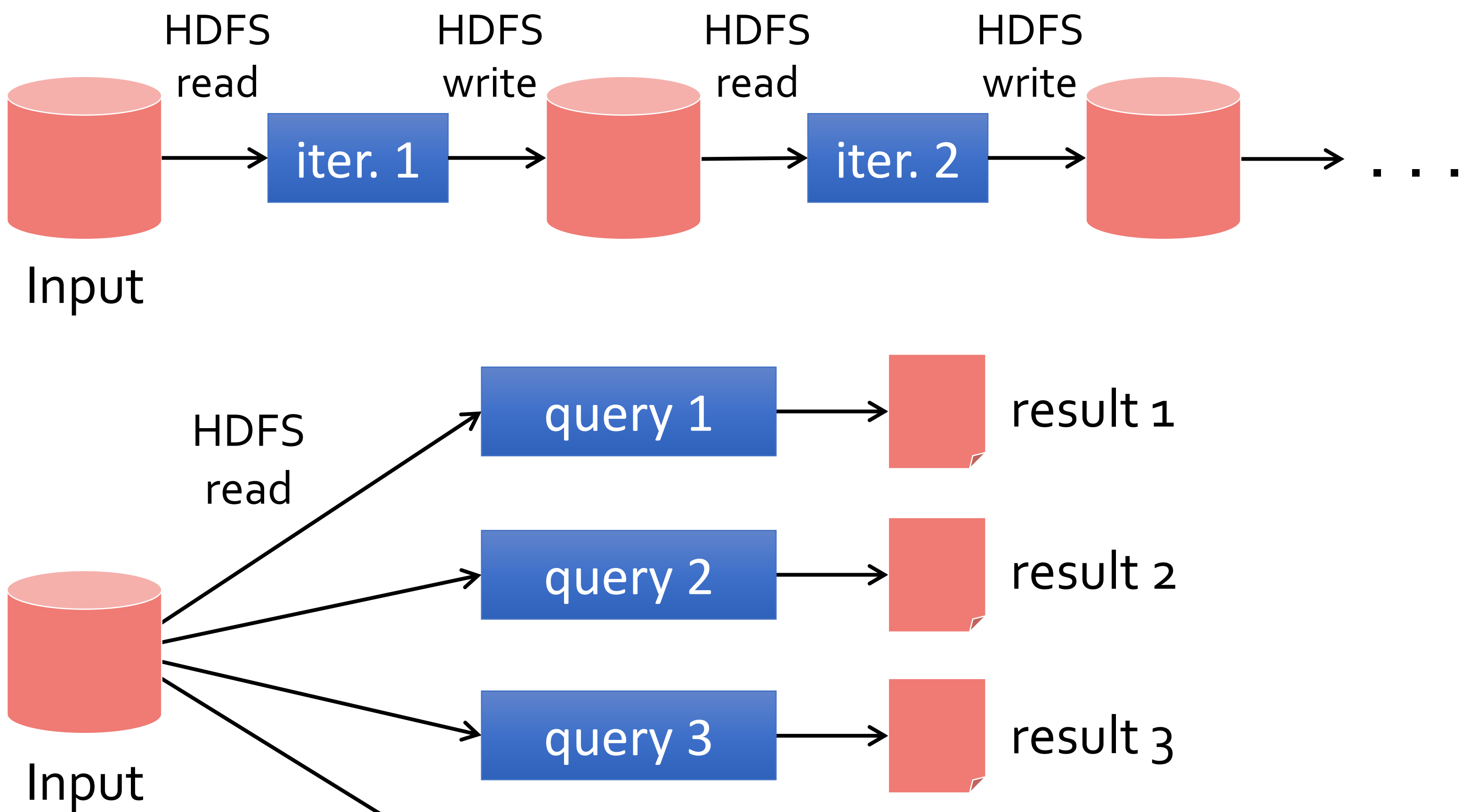
- Only way to share data between iterations / phases is through shared storage
  - Slow!
- Allow operations to feed data to one another
  - Ideally, through memory instead of disk-based storage
- Need the "chain" of operations to be exposed to make this work
- **Problem to solve:** Would this break the 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

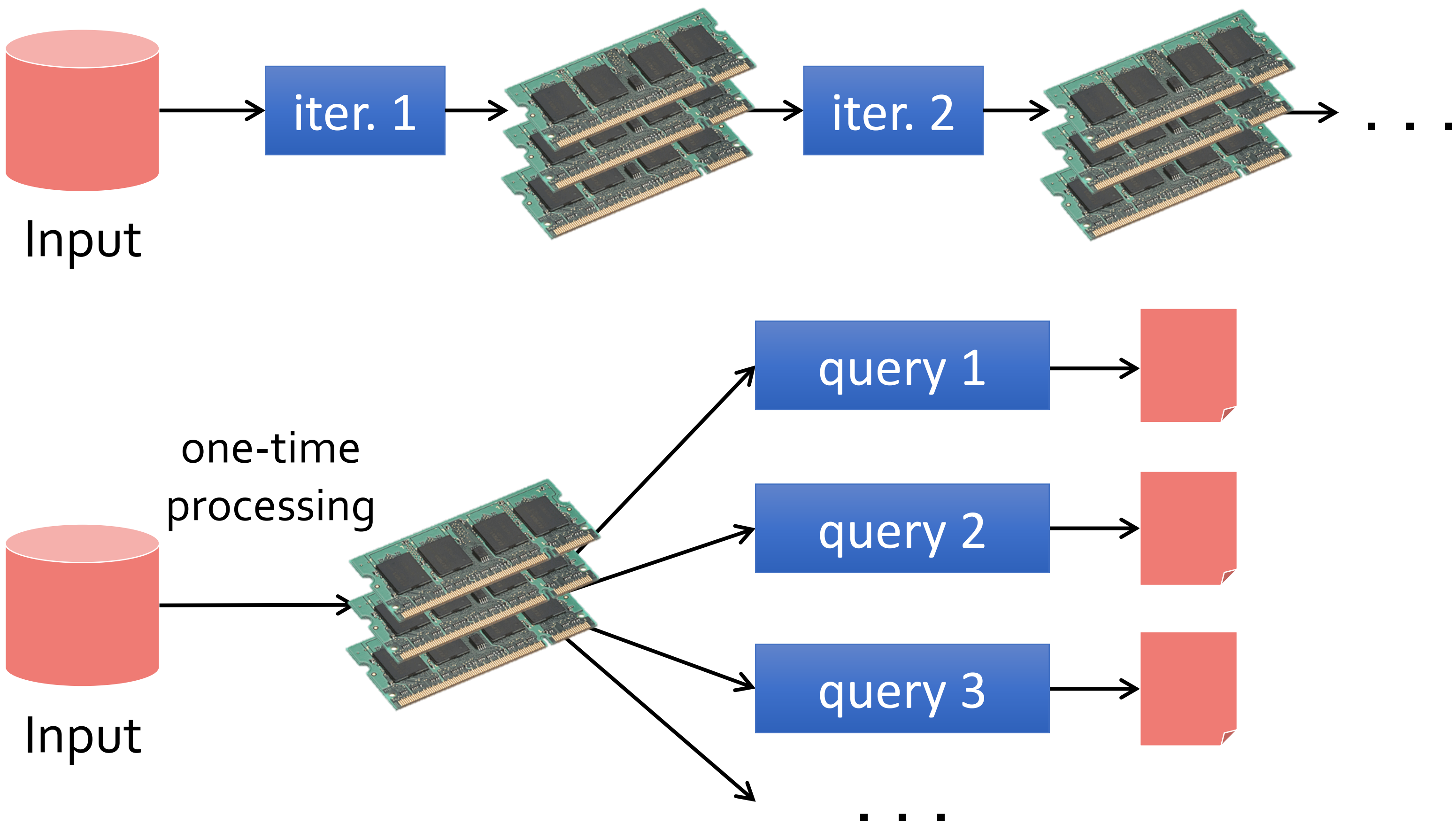

HDFS
read
query 1
query 2
query 3
result 1
result 2
result 3
Input
. . .

Examples

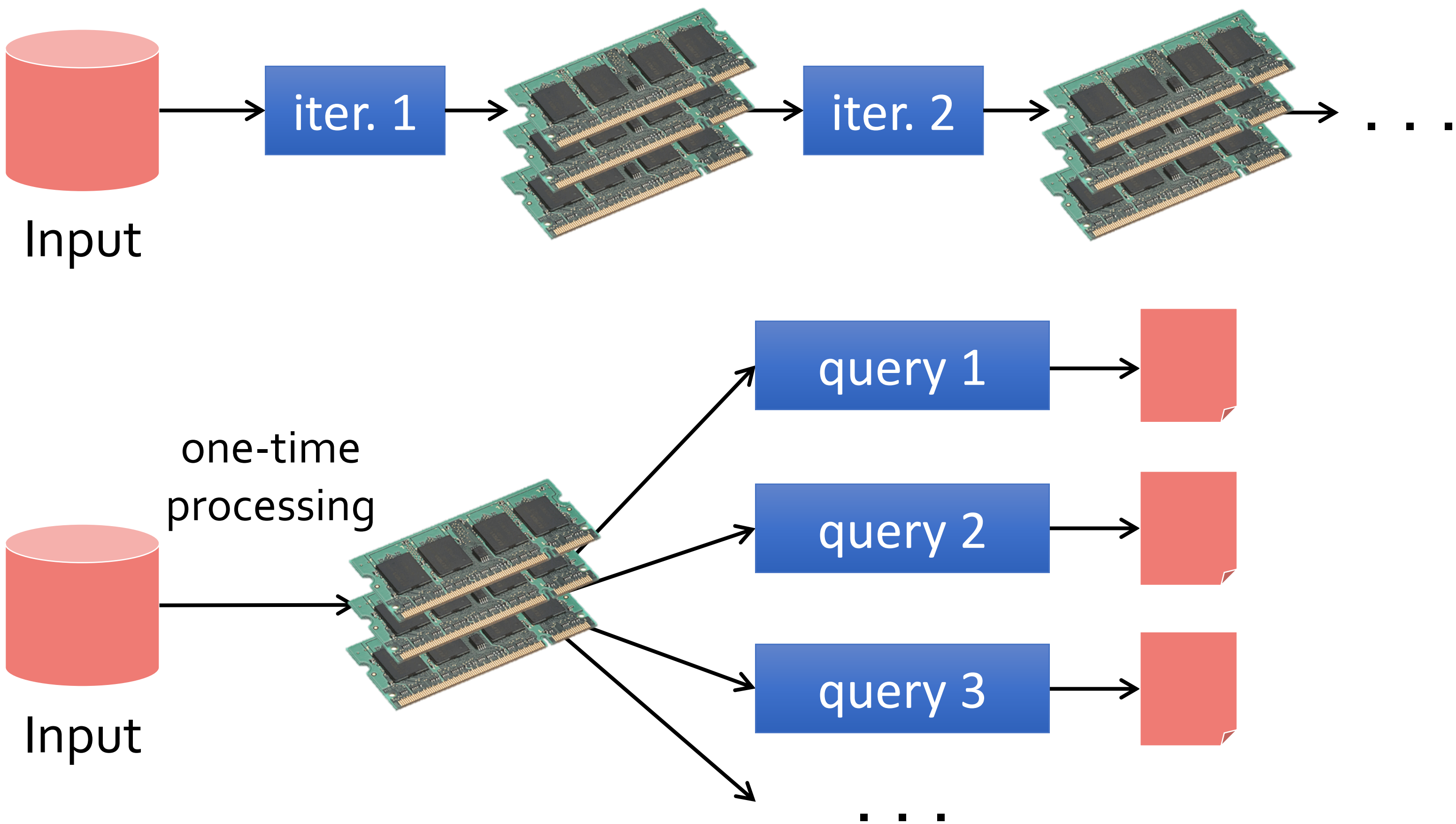


Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-memory data sharing



Goal: In-memory data sharing



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

Challenges

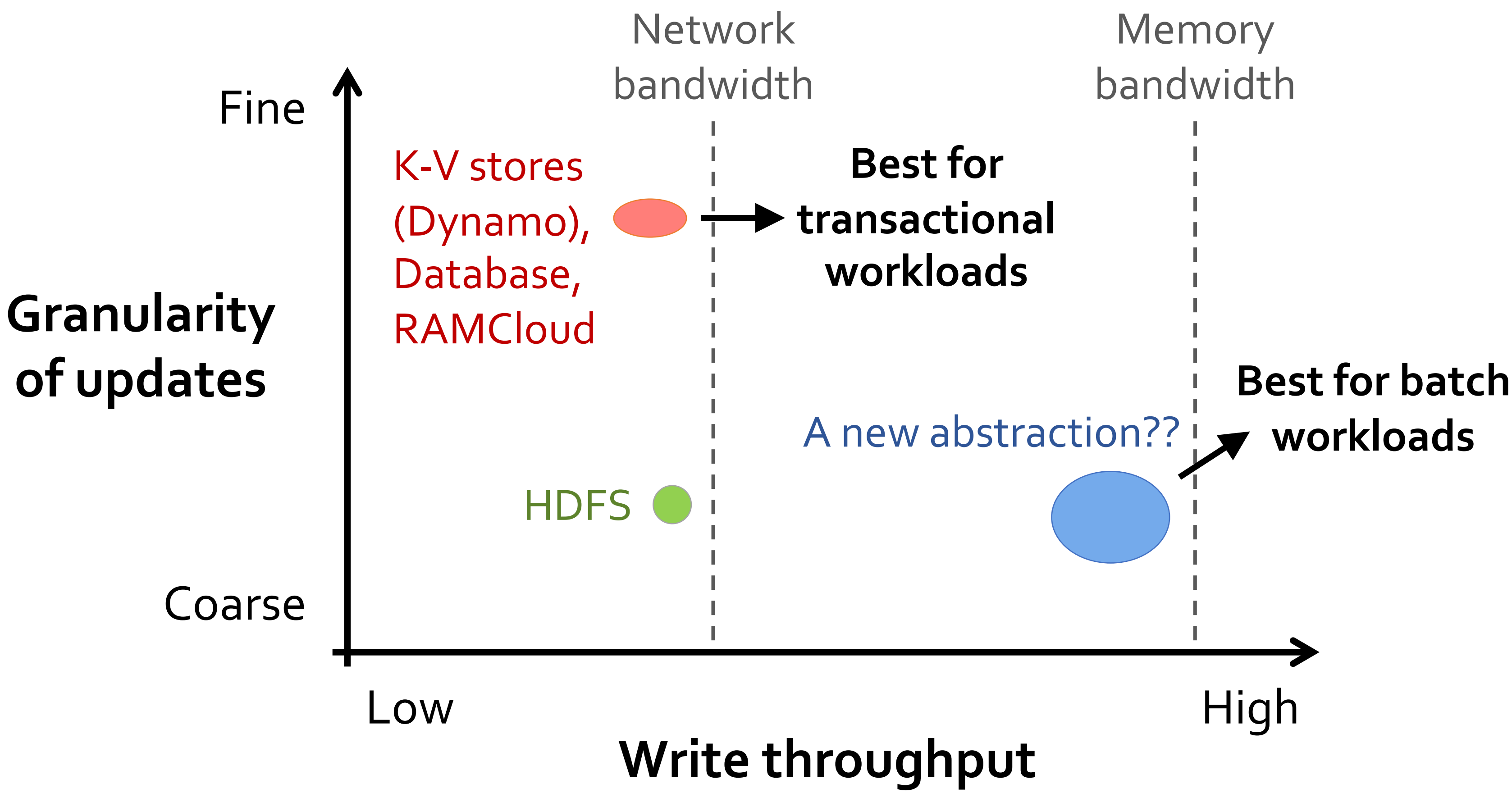
- How to design a distributed memory abstraction that is both fault-tolerant and efficient?

Challenges

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

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

Tradeoff space



Challenges

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

- 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

Managing RDDs

- Transformations on RDDs ($RDD_1 \rightarrow RDD_2$)
- Actions on RDDs (RDD → output)
- Control over RDD partitioning (how items are split over nodes)
- Control over RDD persistence (in memory, on disk, or recompute on loss)

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

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

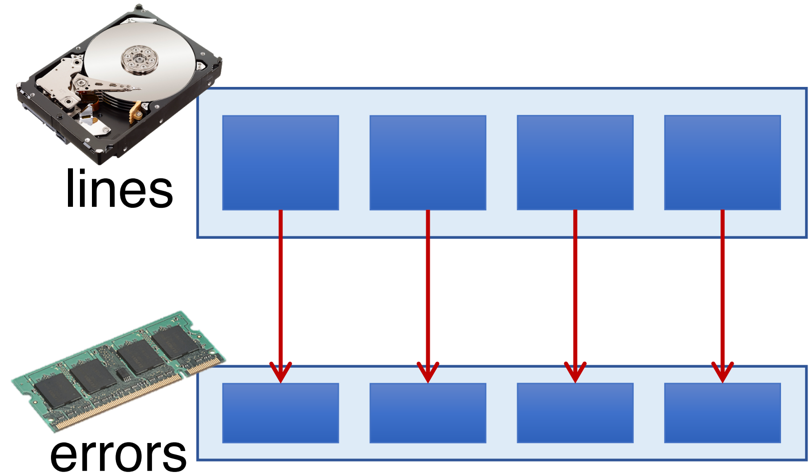
Consumes an RDD to produce output
either to storage (save), or
to interpreter/Scala (count, collect, reduce)

Causes RDD lineage chain to get executed on the cluster to produce the output
(for any missing pieces of the computation)

Interactive debugging

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

Interactive debugging



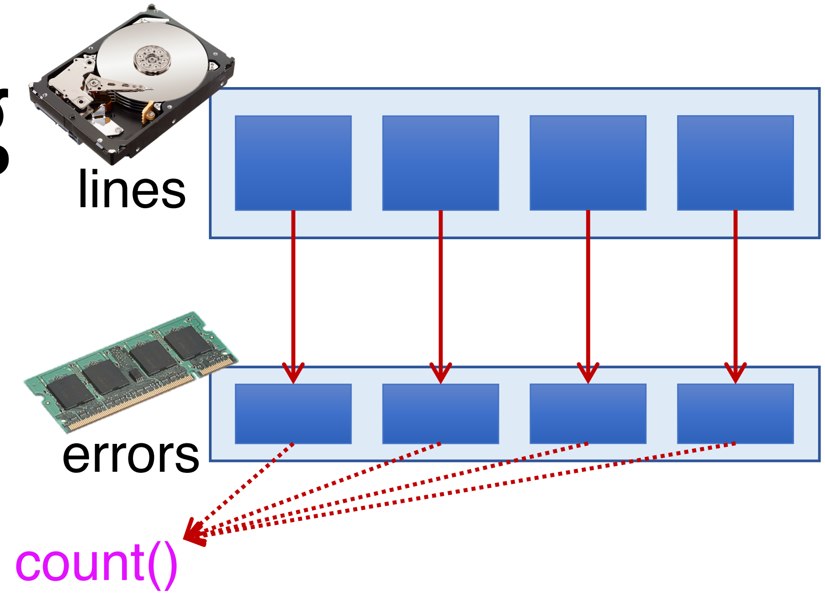

```
lines = textFile("hdfs://foo.log")
errors = lines.filter(
            _.startsWith("ERROR")
errors.persist()

errors.count()
```

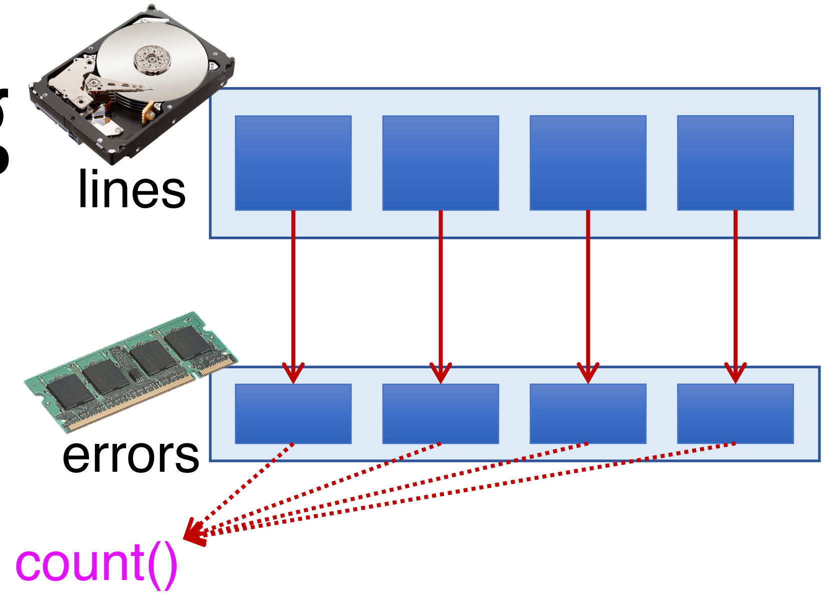
Interactive debugging

```
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()

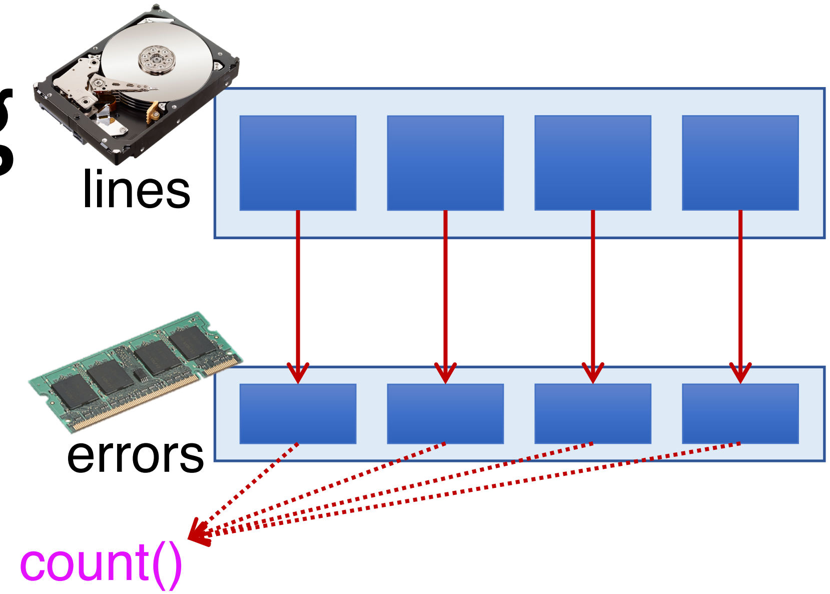
errors.filter(
     _.contains("MySQL"))
```

Interactive debugging

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

errors.count()

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

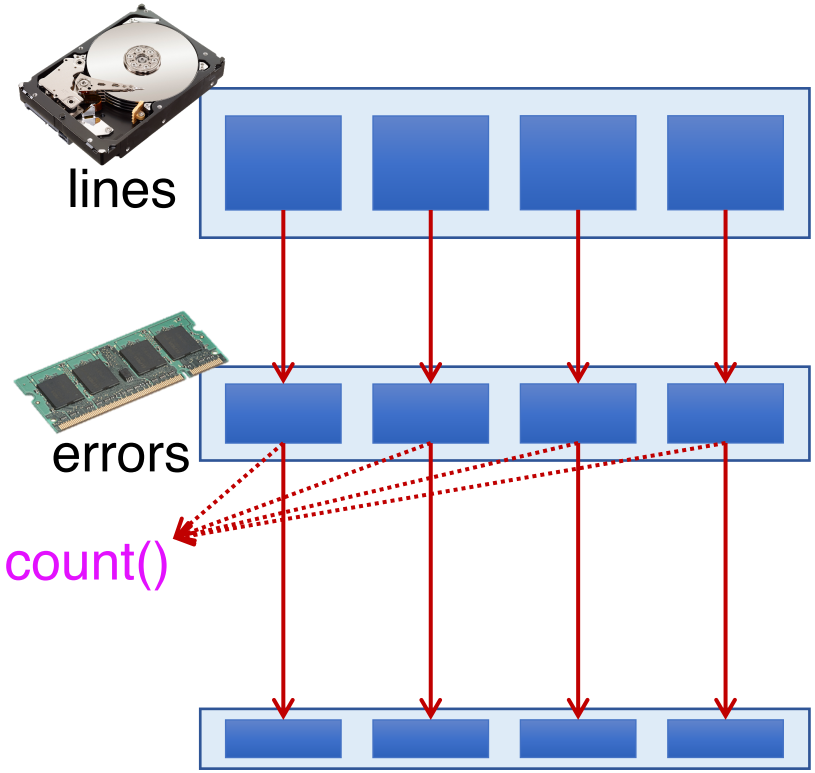


Interactive debugging

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

errors.count()

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

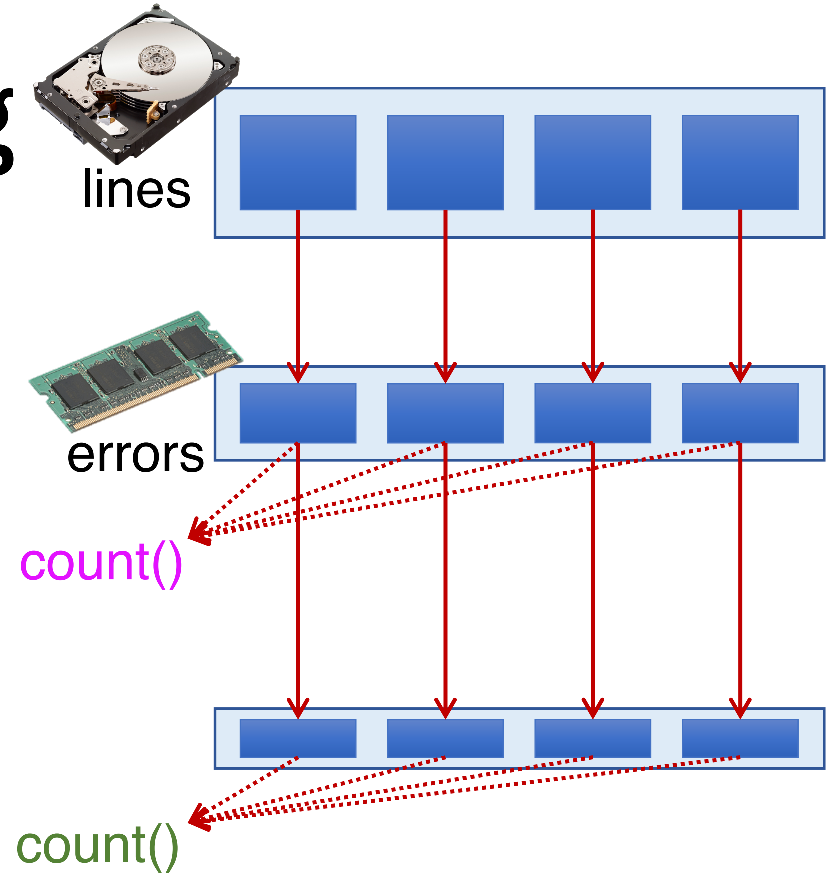


Interactive debugging

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

errors.count()

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

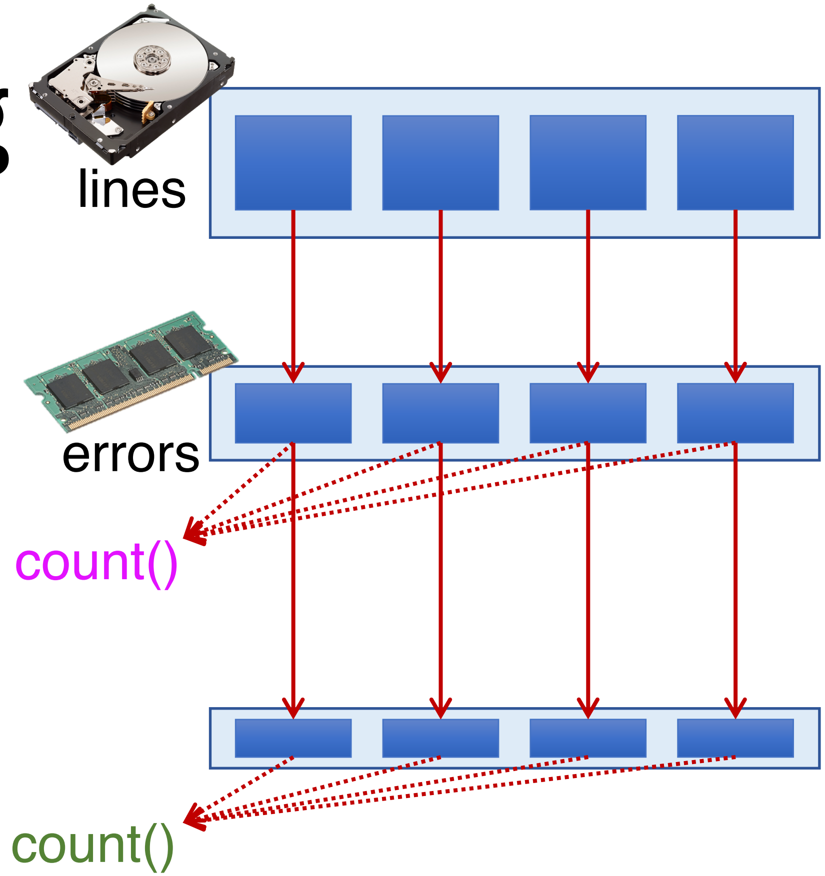


Interactive debugging

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

errors.count()

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

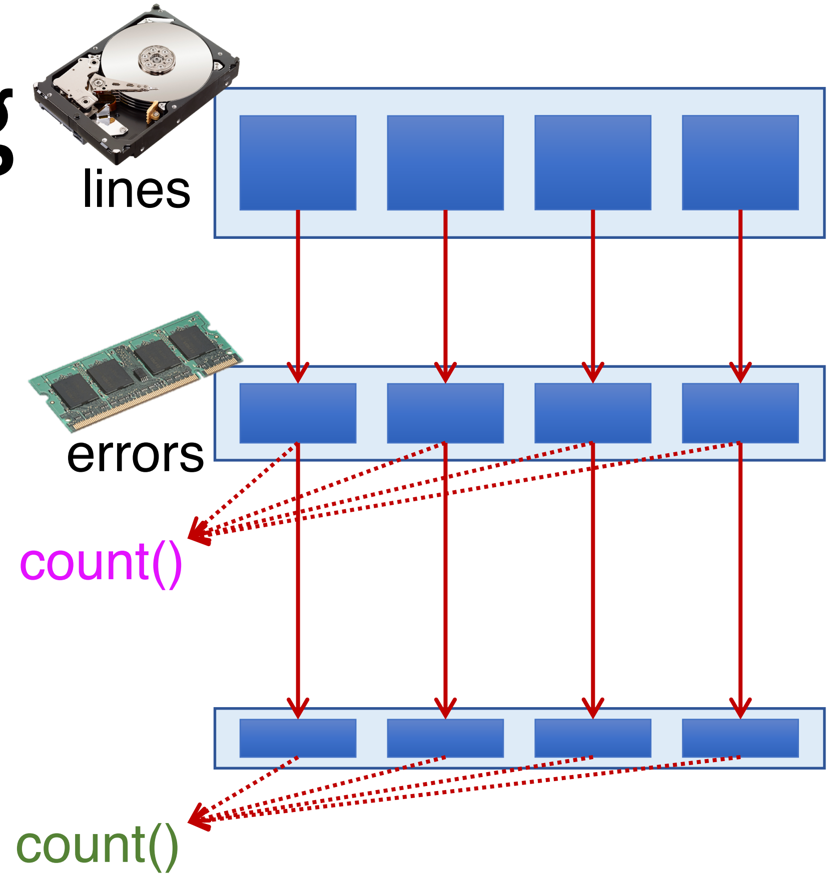


Interactive debugging

```
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))
```

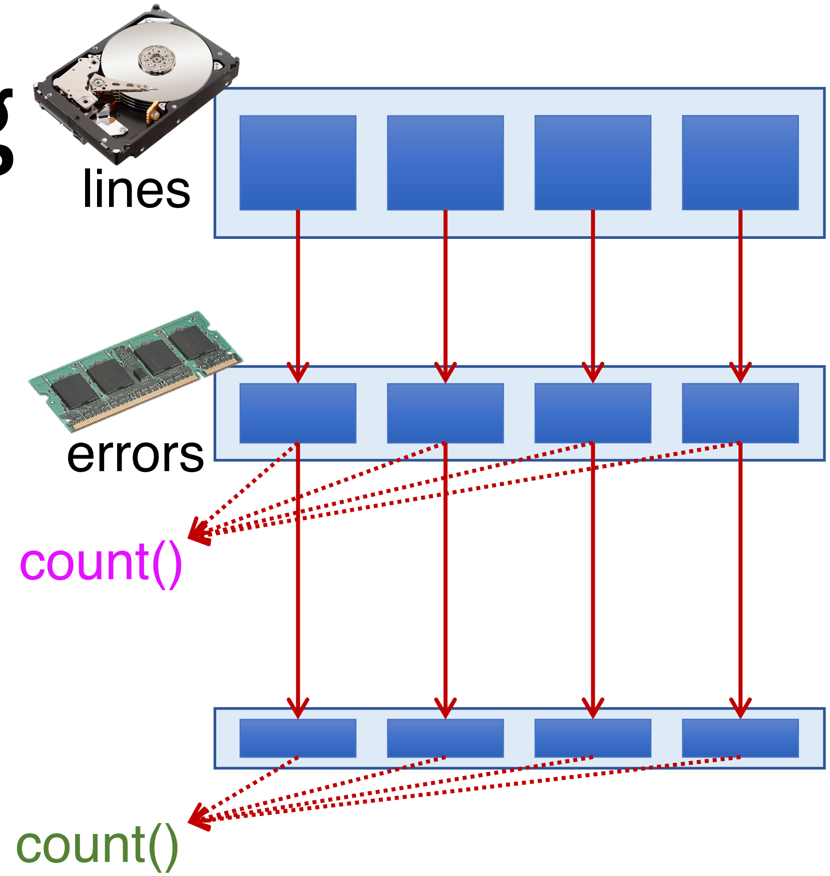


Interactive debugging

```
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()
```

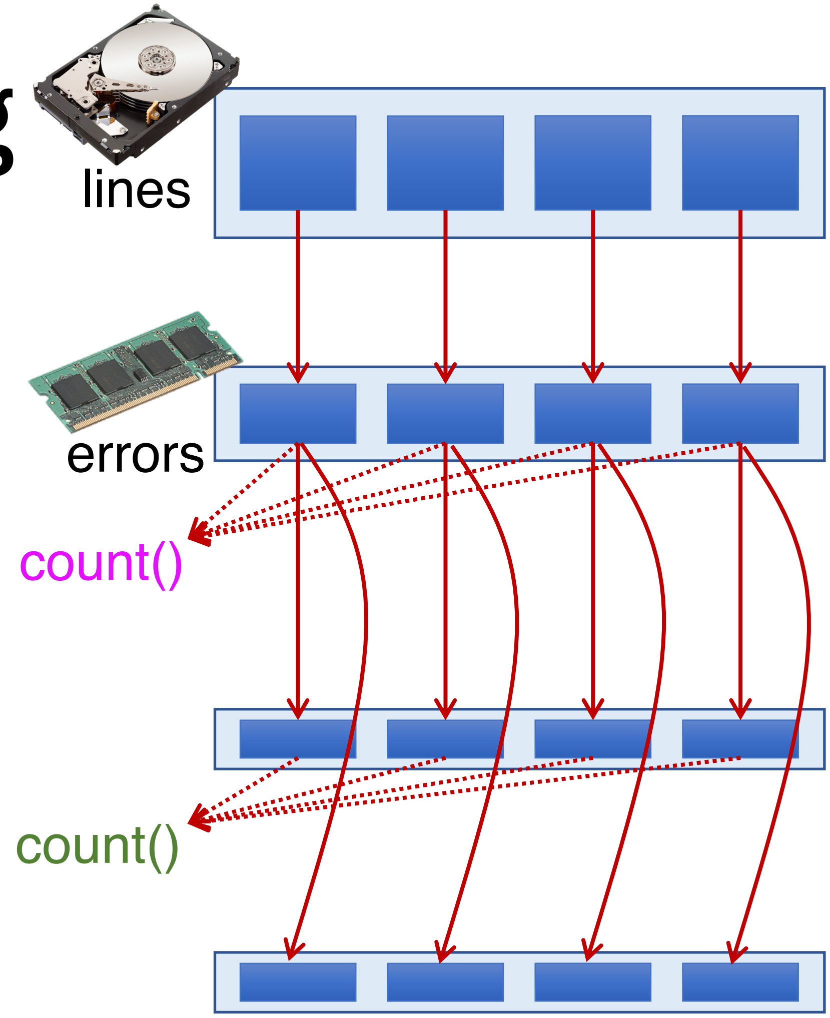


Interactive debugging

```
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()
```

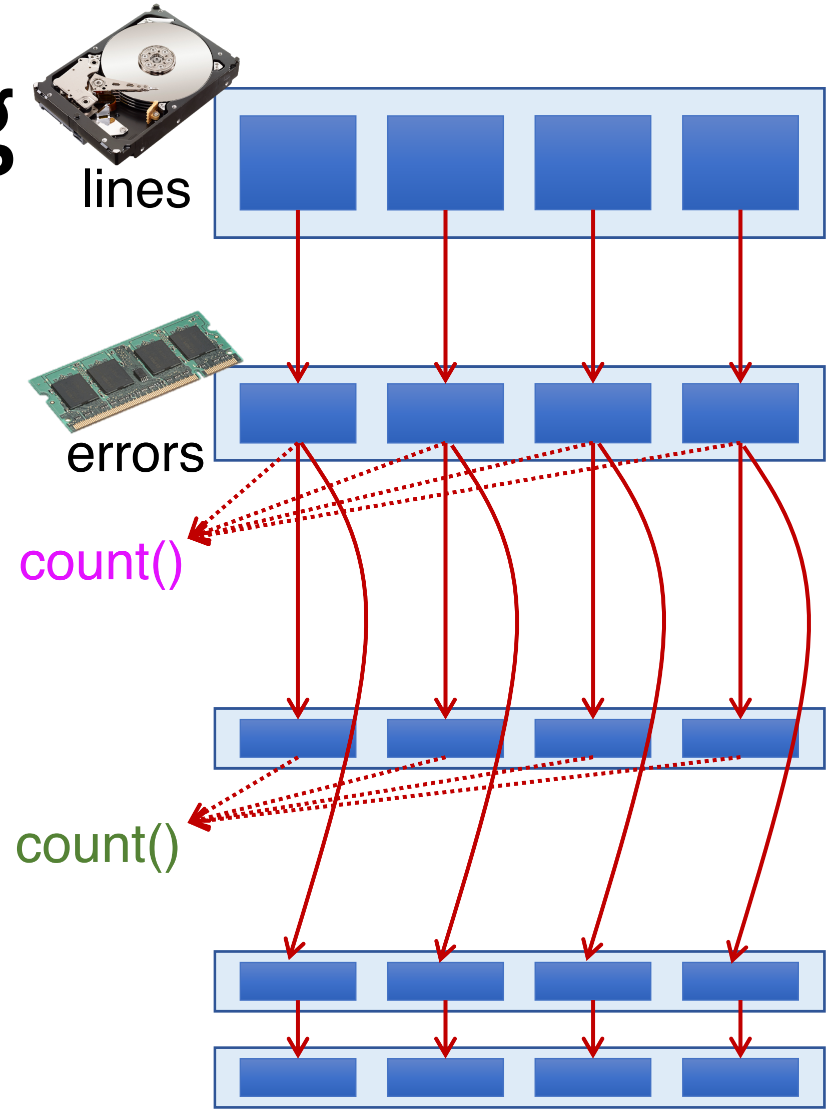


Interactive debugging

```
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()
```

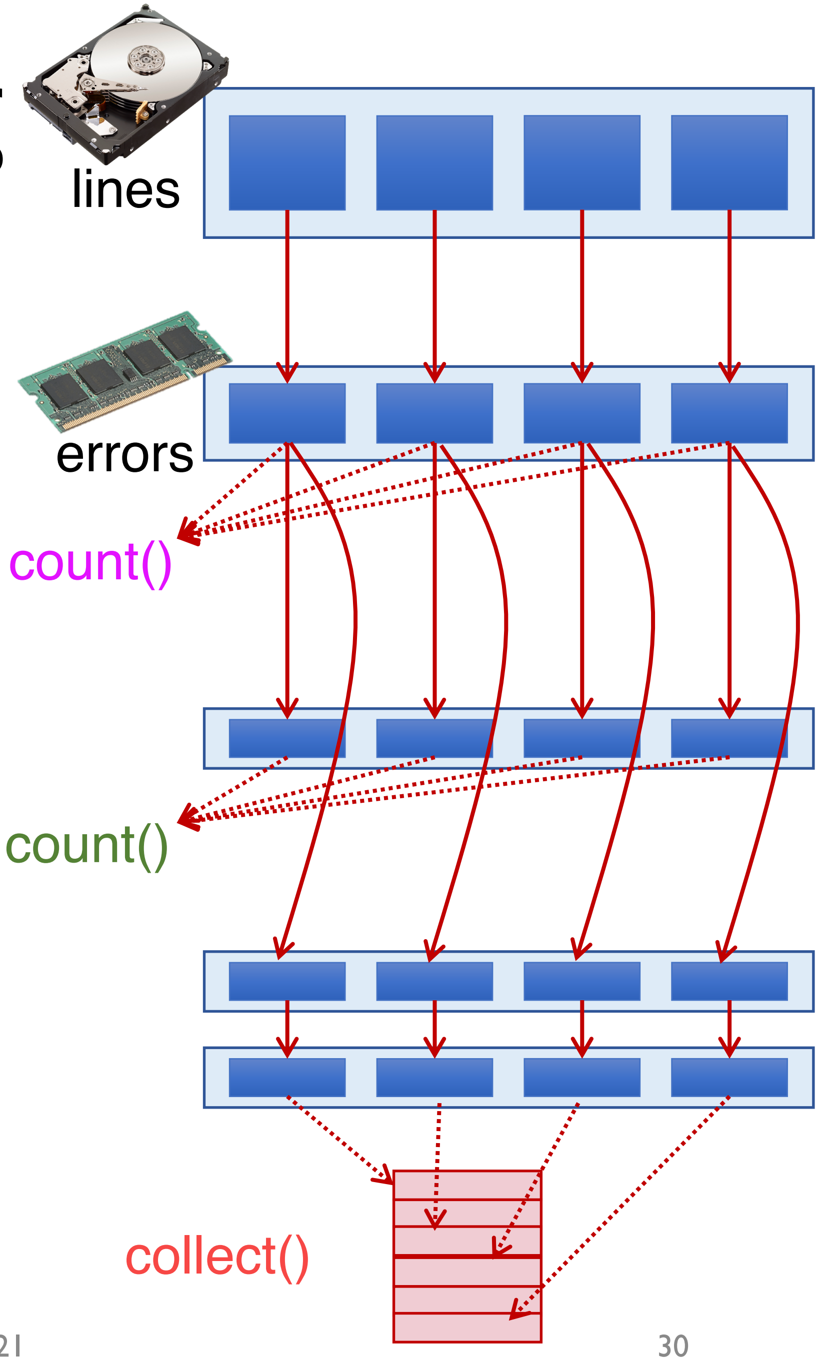


Interactive debugging

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

errors.count()

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



persist()

- Not an action nor 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

Lineage graph of RDDs

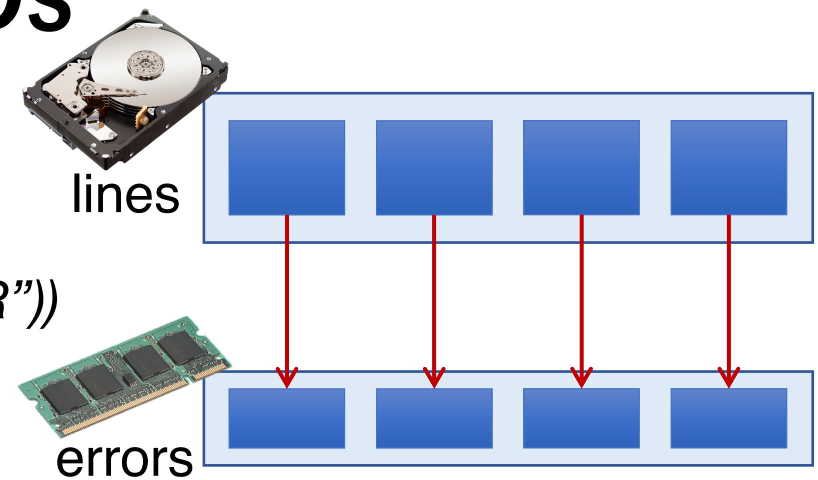
```
lines
```

Lineage graph of RDDs

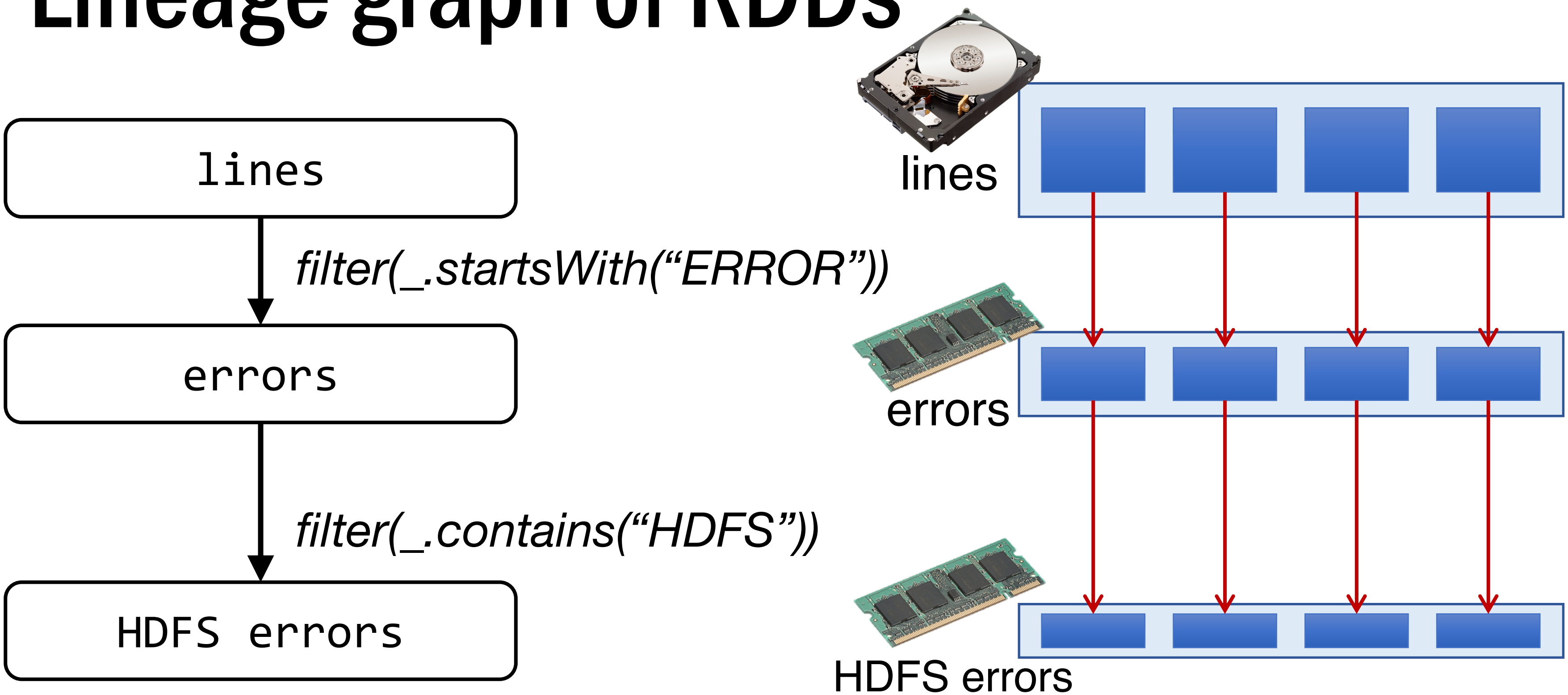
`lines`

↓ *filter(_.startsWith("ERROR"))*

`errors`



Lineage graph of RDDs



Lineage graph of RDDs

`lines`

↓ *filter(_.startsWith("ERROR"))*

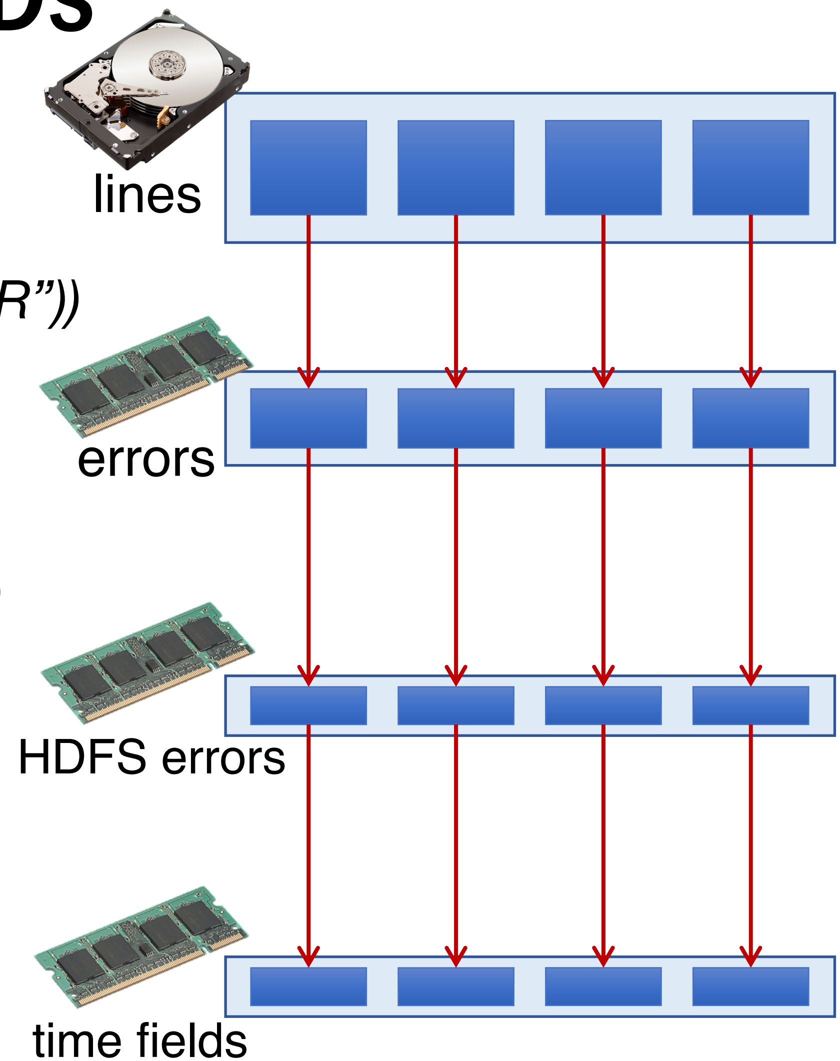
`errors`

↓ *filter(_.contains("HDFS"))*

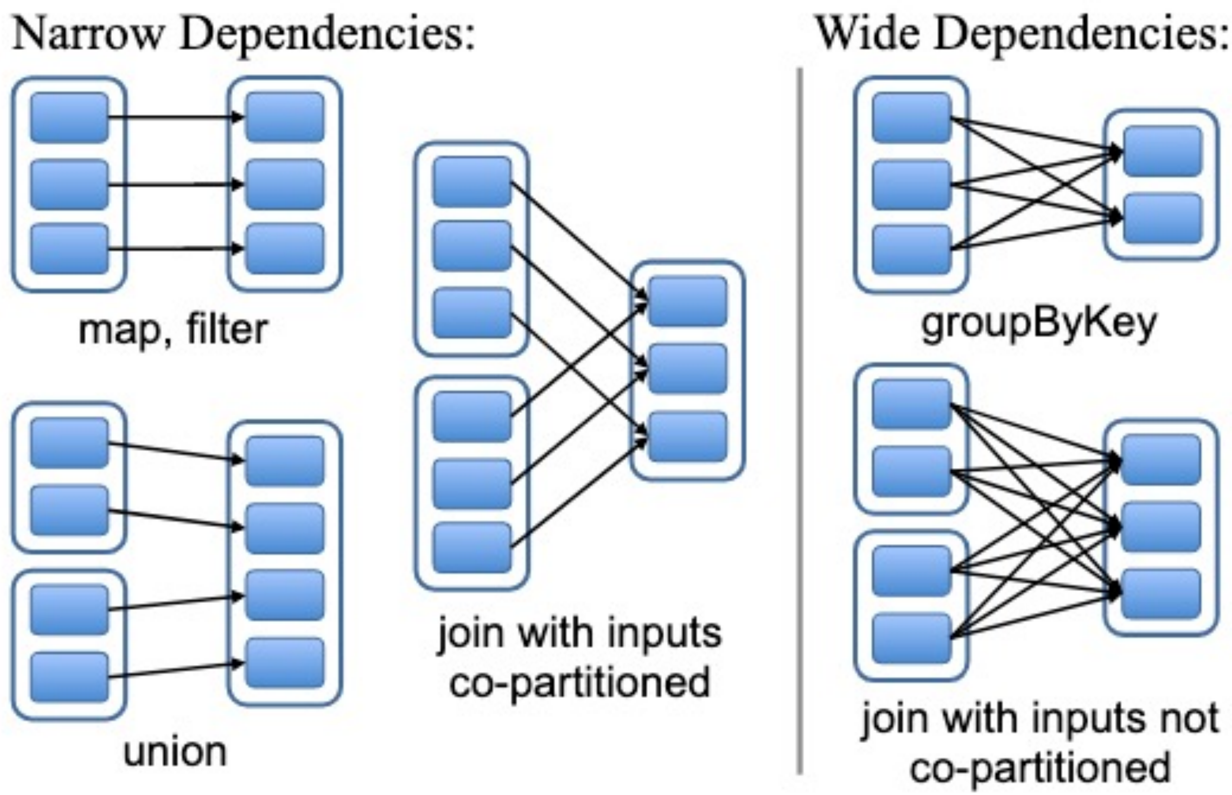
`HDFS errors`

↓ *map(_.split('\t')(3))*

`time fields`



Narrow & wide dependencies



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)

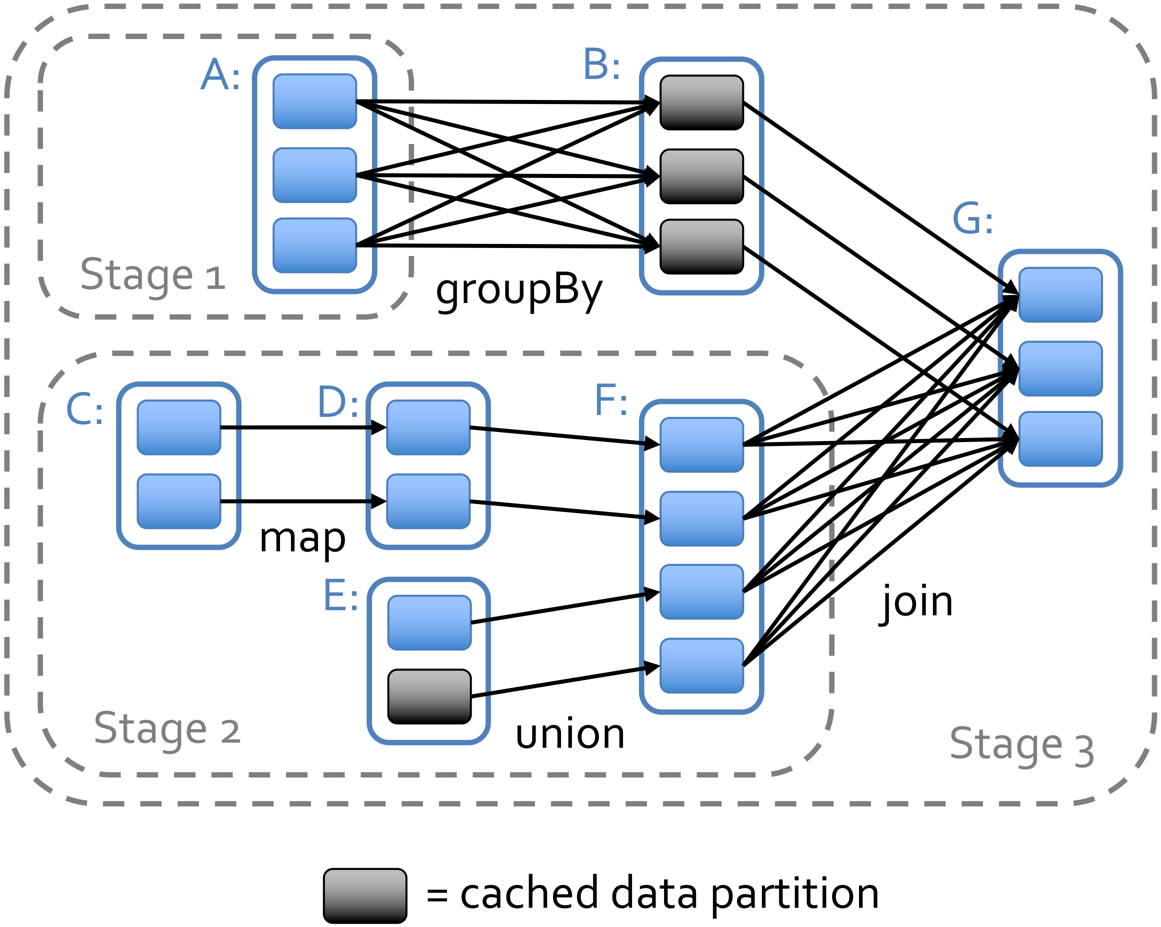
Task scheduler

Dryad-like DAGs

Pipelines functions within a stage

Locality & data reuse aware

Partitioning-aware to avoid shuffles

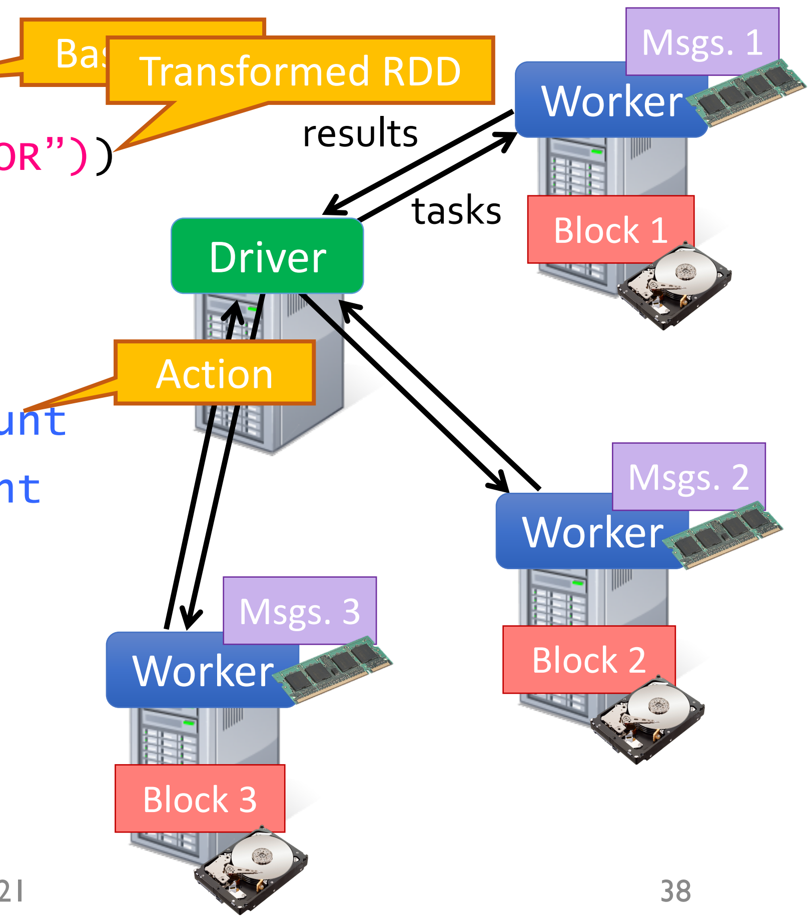


Interactive debugging (control and data flow)

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
```



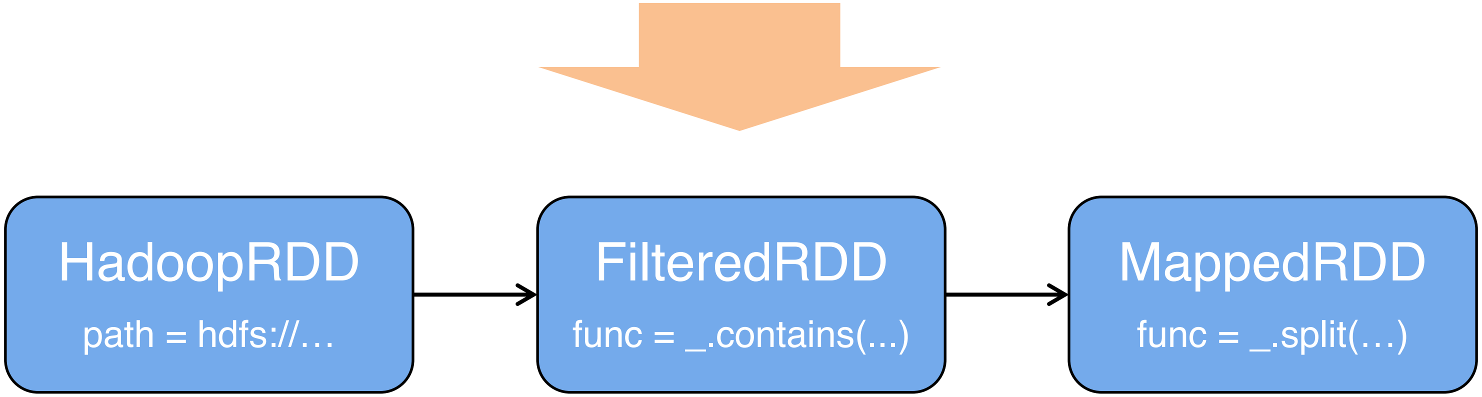
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Fault recovery

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

E.g.:

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

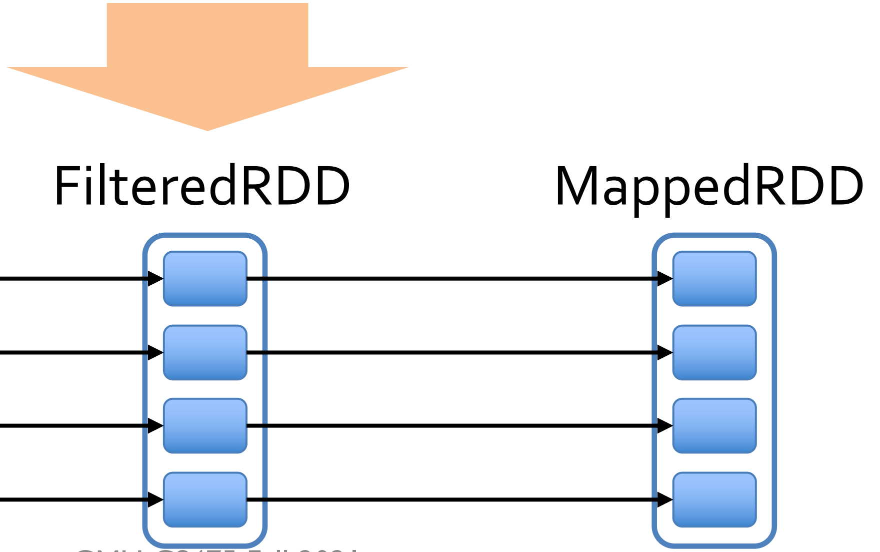


Fault recovery

- 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))
```

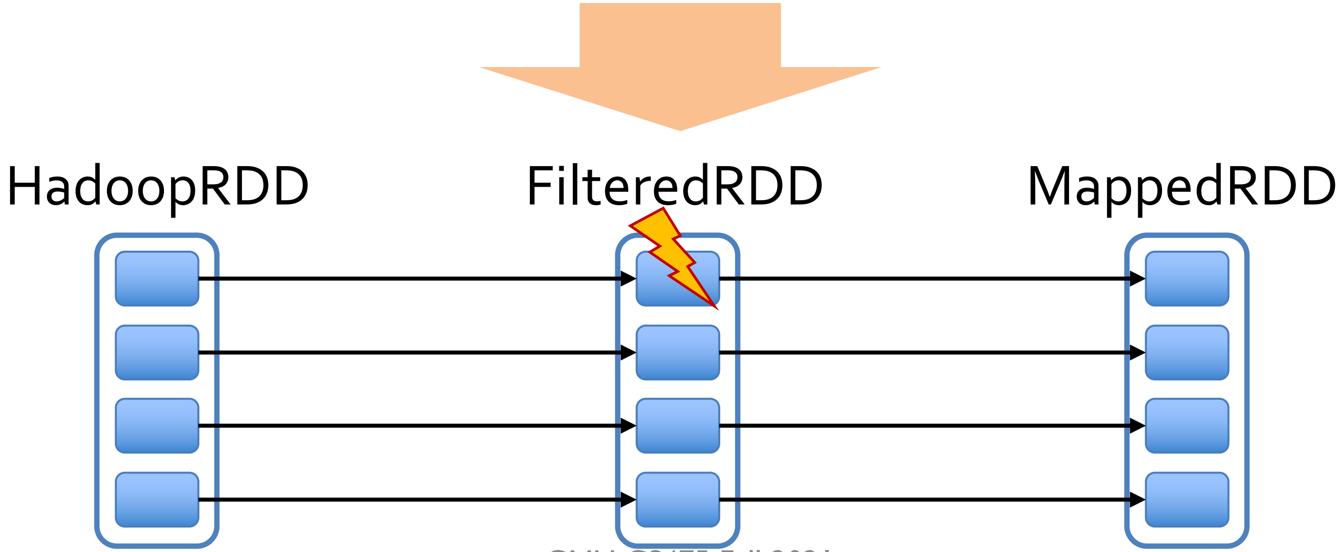


Fault recovery

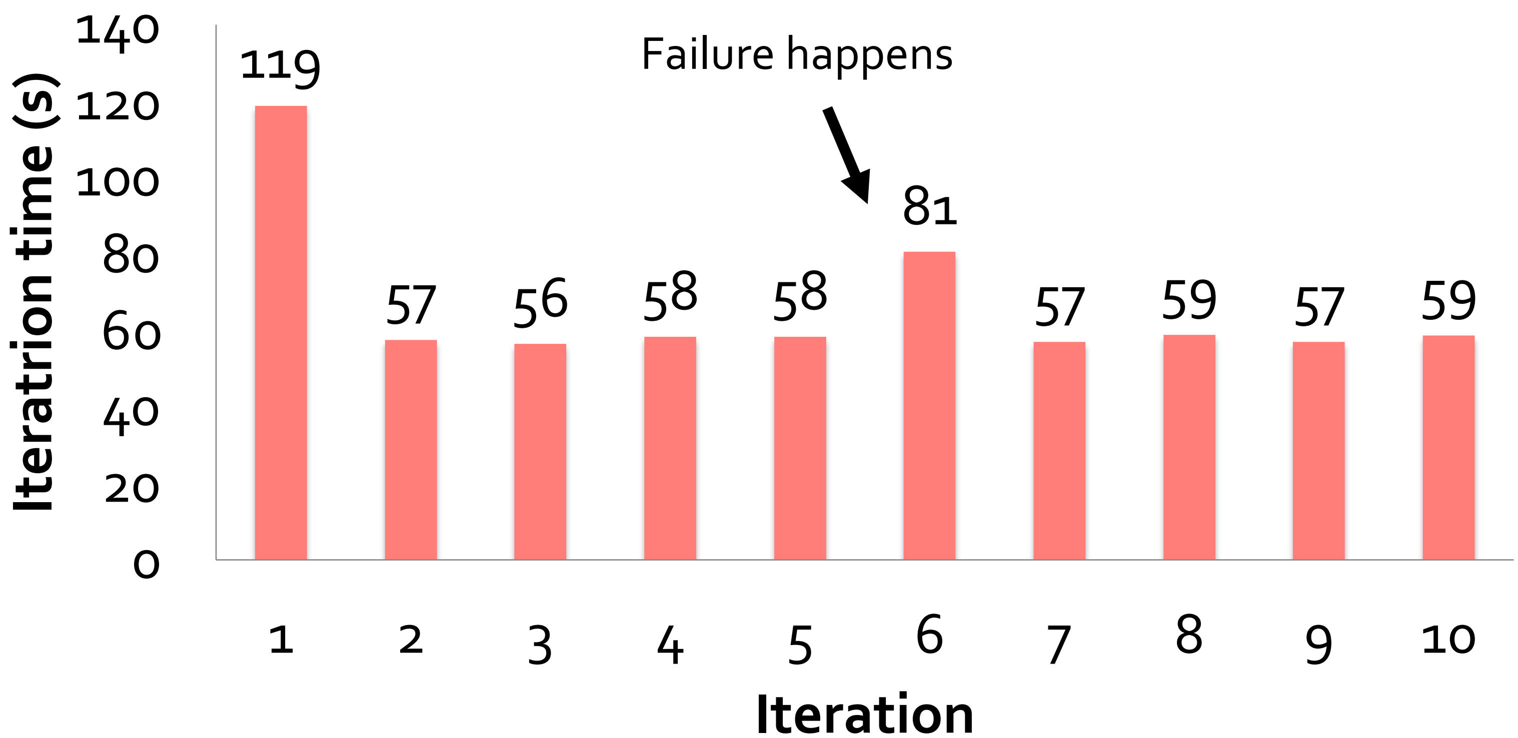
- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:

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



Fault recovery results



Example: PageRank

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

$$\Sigma_{i\in neighbors}\ rank_i\ /\ |neighbors_i|$$

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

Example: PageRank

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

$$\Sigma_{i \in neighbors}\ rank_i\ /\ |neighbors_i|$$

RDD[(URL, Seq[URL])]

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

RDD[(URL, Rank)]

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

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

For each neighbor in links emits (URL, RankContrib)

Reduce to RDD[(URL, Rank)]

Join (⋈)

Alice	5
Bob	6
Claire	4

⋈

Alice	F
Bob	M
Claire	F

=

Alice	5	F
Bob	6	M
Claire	4	F

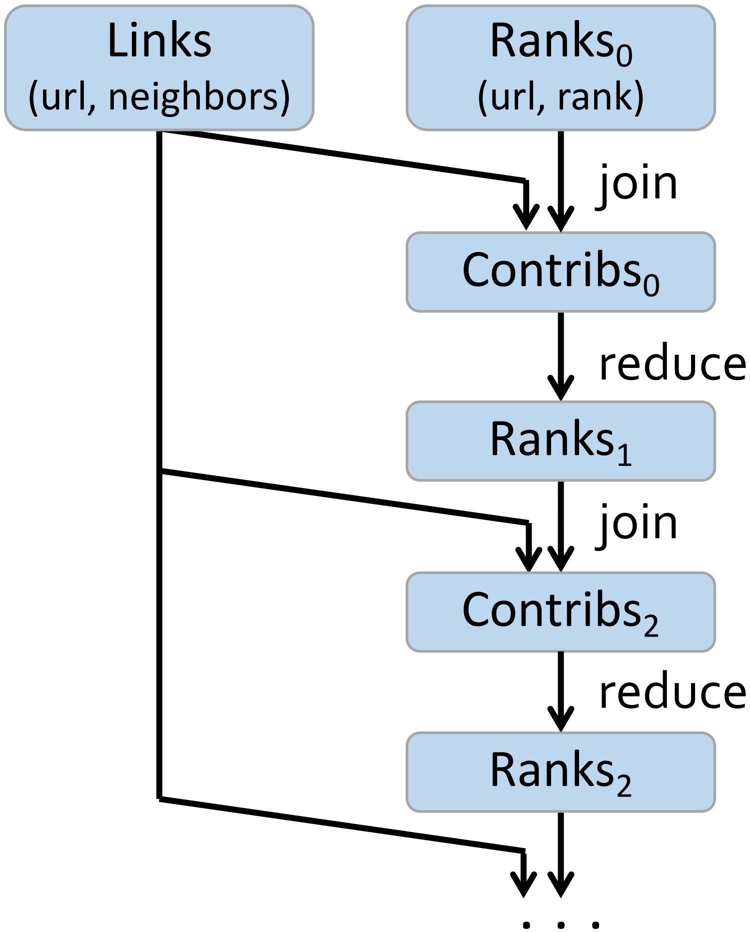
A	5
A	2
A	3
B	4
B	1
C	6
C	8

⋈

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. Same problem in `reduce()` for MapReduce.

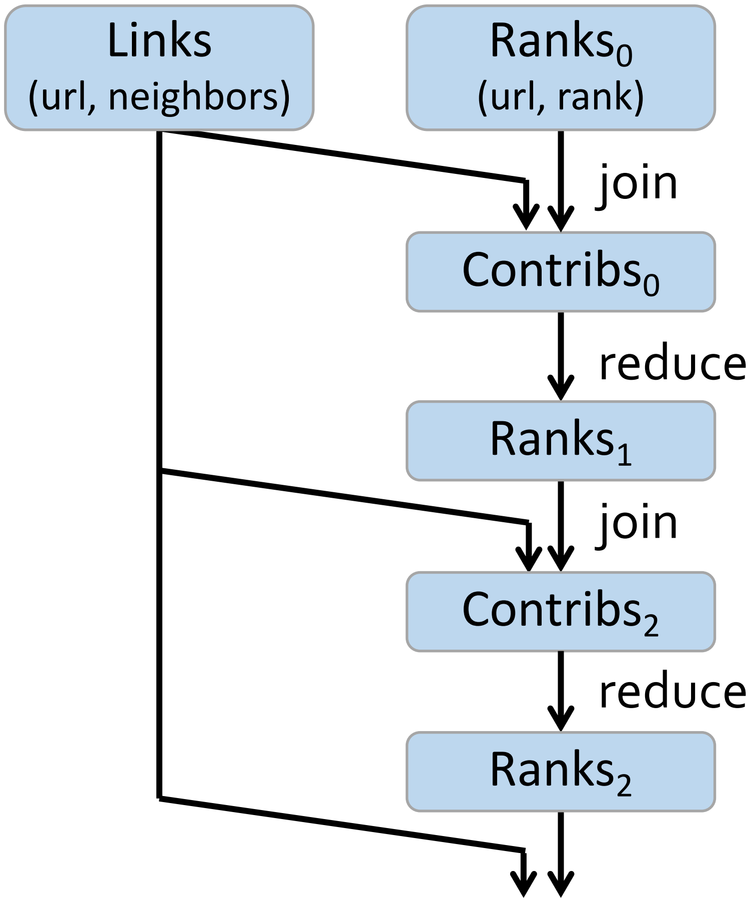
Optimizing placement



- `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



- `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()`**?
  ```

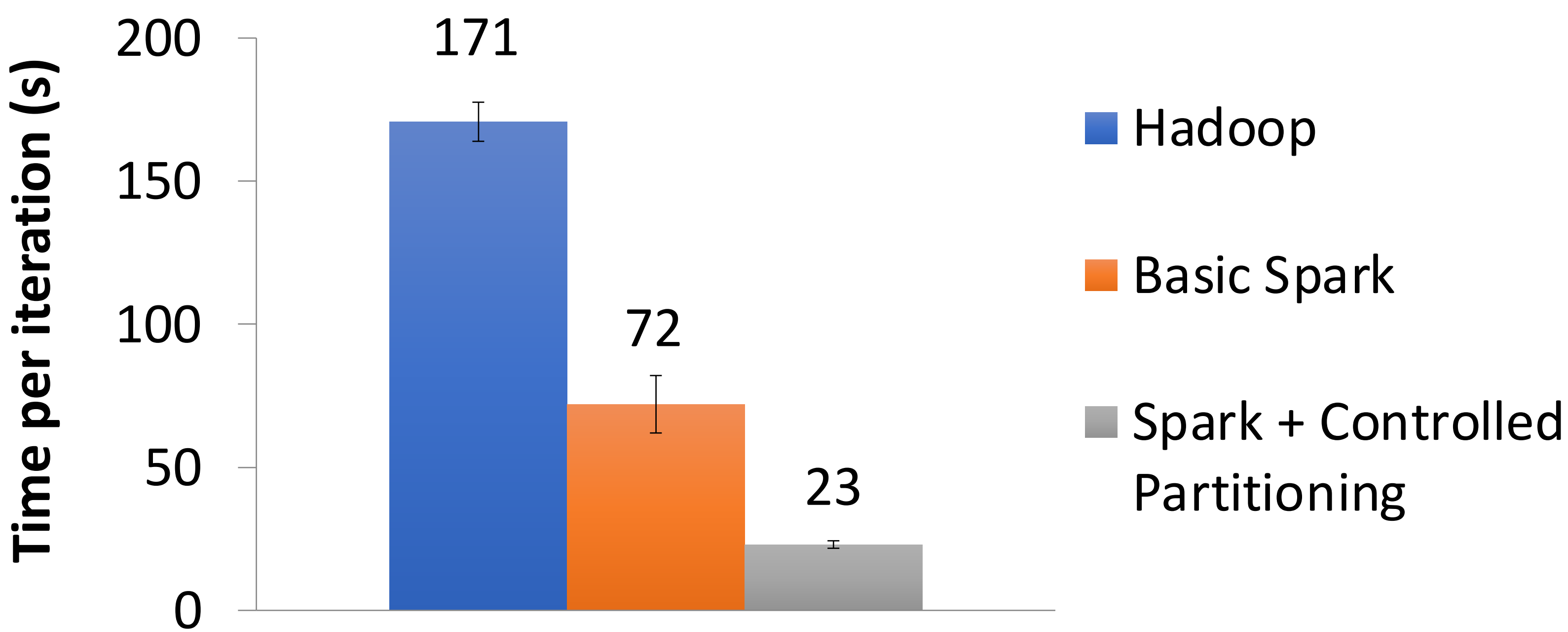
Co-partitioning example

Co-partitioning can avoid shuffle on join

But, fundamentally a shuffle on `reduceByKey`

Optimization: custom partitioner on domain

PageRank performance



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

Tradeoff space