CE 528 Cloud Computing

Lecture 13: Data Processing II
Spring 2026

Prof. Yigong Hu

Slides courtesy of Yue Cheng

Administrivia

All groups did a great job on the progress

- Keep up the good work

TA is currently testing your tools

One improvement: improve clarity in the design document

Recap

Applications

Scalable computing engines

Scalable storage systems

What's Good with Mapreduce

Scaled analytics to thousands of machines

Eliminated fault tolerance as a concern

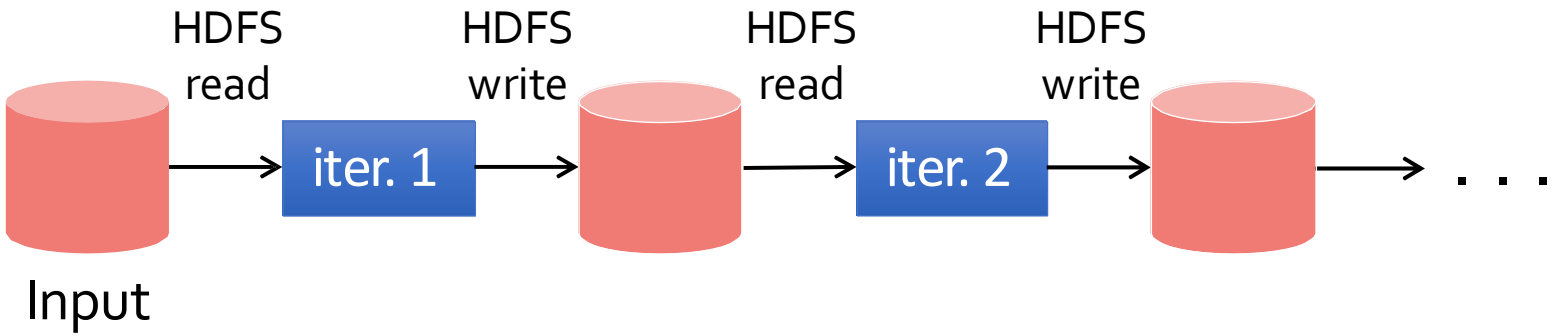
Not very expressive

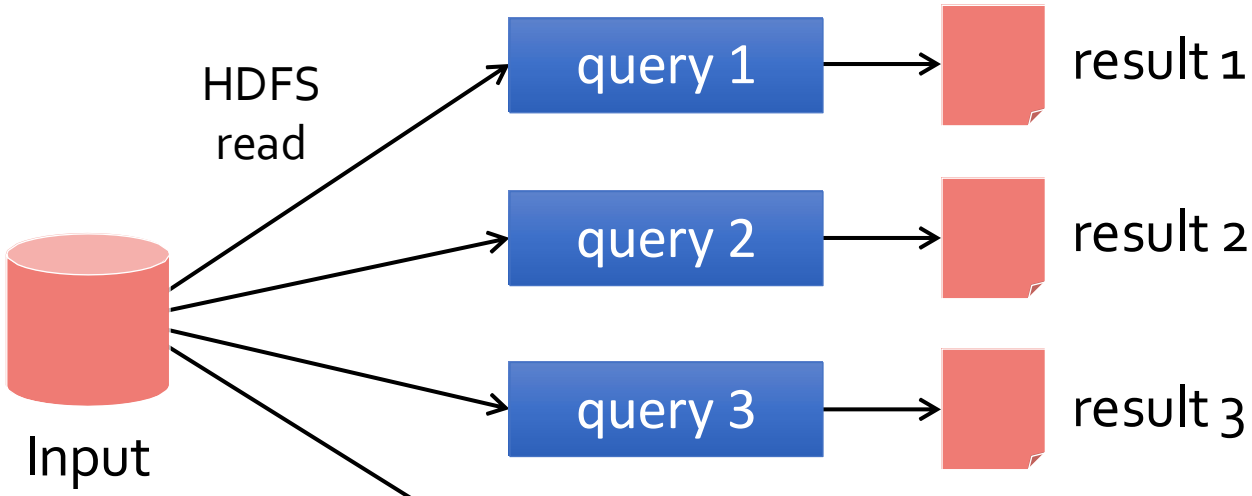
- Iterative algorithms
  (PageRank, Logistic Regression, …)
- Interactive and ad-hoc queries
  (Interactive Log Debugging)

Lots of specialized frameworks

- Pregel, GraphLab, PowerGraph, DryadLINQ, HaLoop…

Examples





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

Sharing Data Between Stages/Iterations

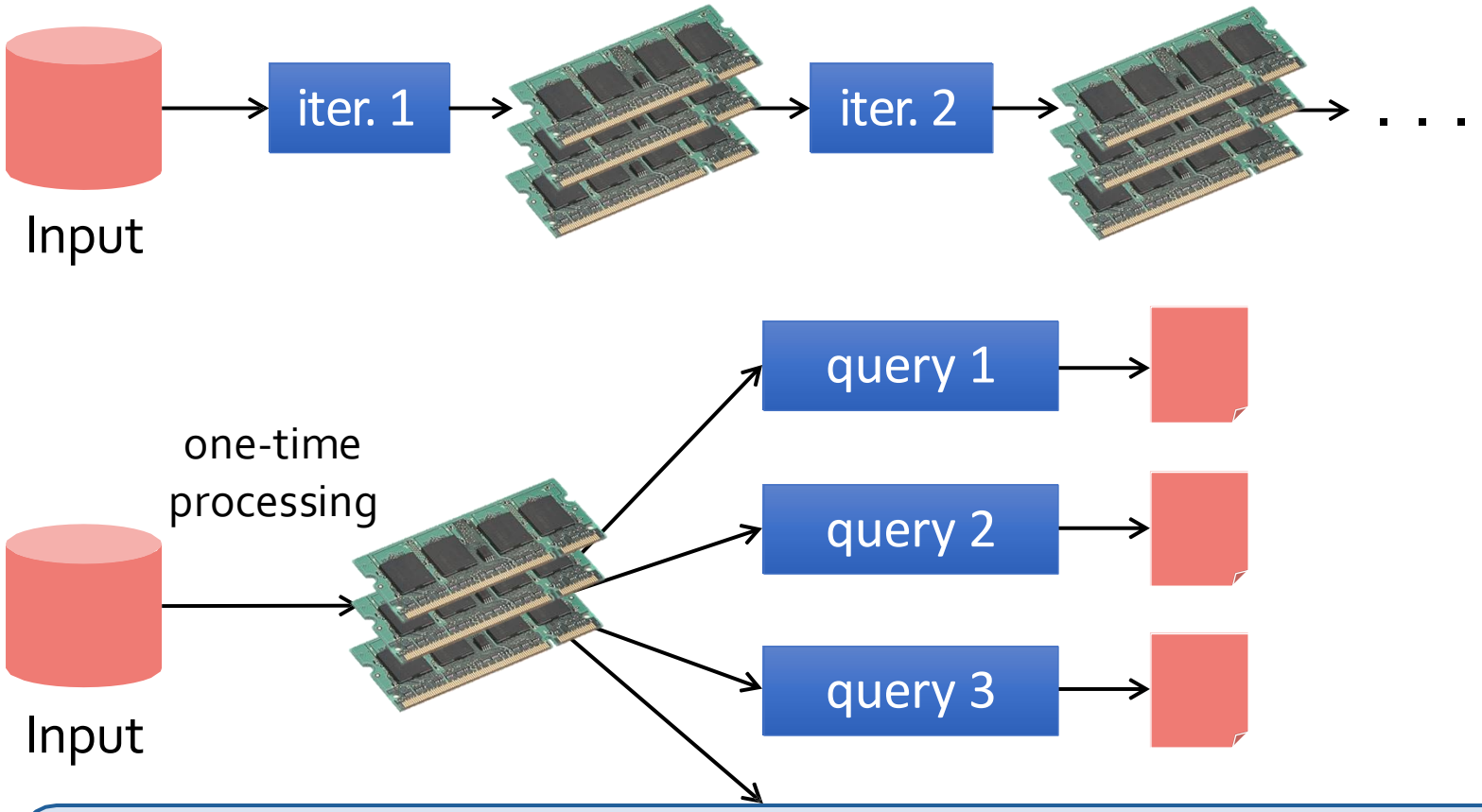
Only way to share data between iterations/phases is through shared storage

- Slow!

Allow operations to feed data to one another

- Ideally, through memory instead of disk-based storage

Goal: In-memory Data Sharing



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

Sharing Data Between Stages/Iterations

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

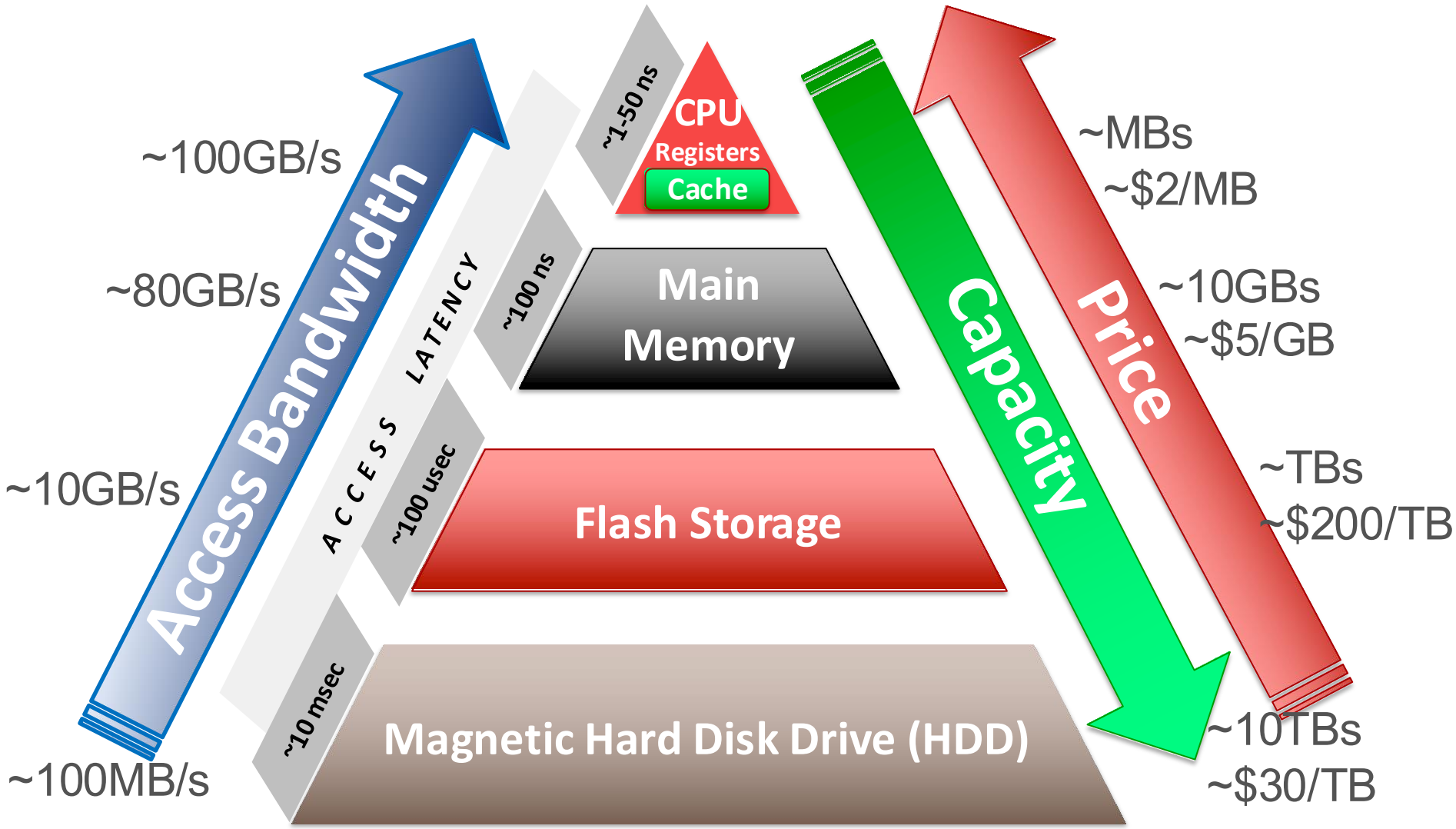
Challenges

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

Existing storage systems allow fine-grained mutation to state

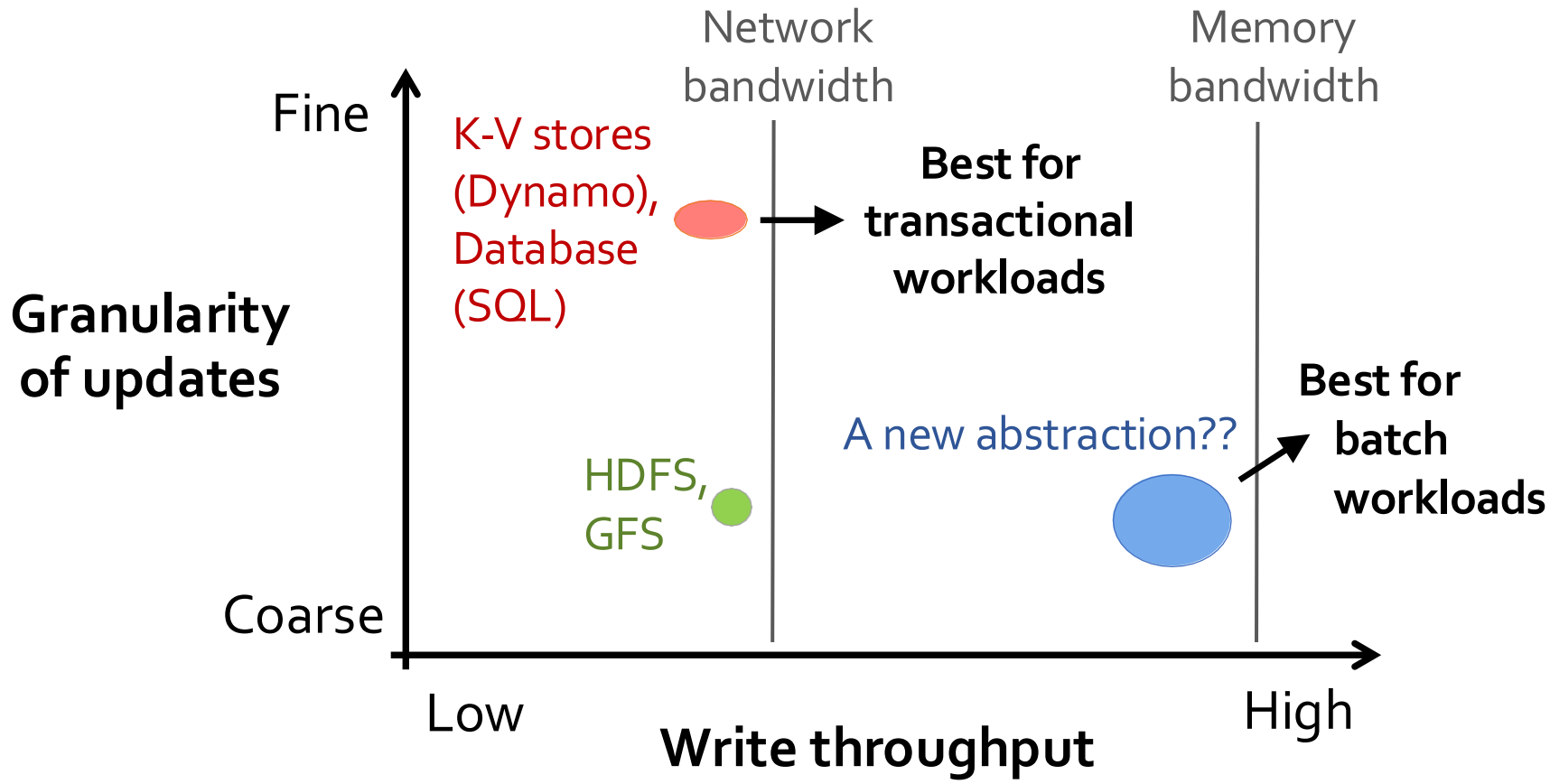
- In-memory key-value stores
- Requires replicating data or logs across nodes for fault tolerance
  - Costly for data-intensive apps
  - 10-100x slower than memory write
- They also require costly on-the-fly replication for mutations

Memory-storage Hierachy



*UCSD DSC 102: Systems for scalable analysis. Arun Kumar

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

- In-memory key-value stores
- Requires replicating data or logs across nodes for fault tolerance
  - Costly for data-intensive apps
- Th… …ons

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

Solution: Resilient Distributed Datasets

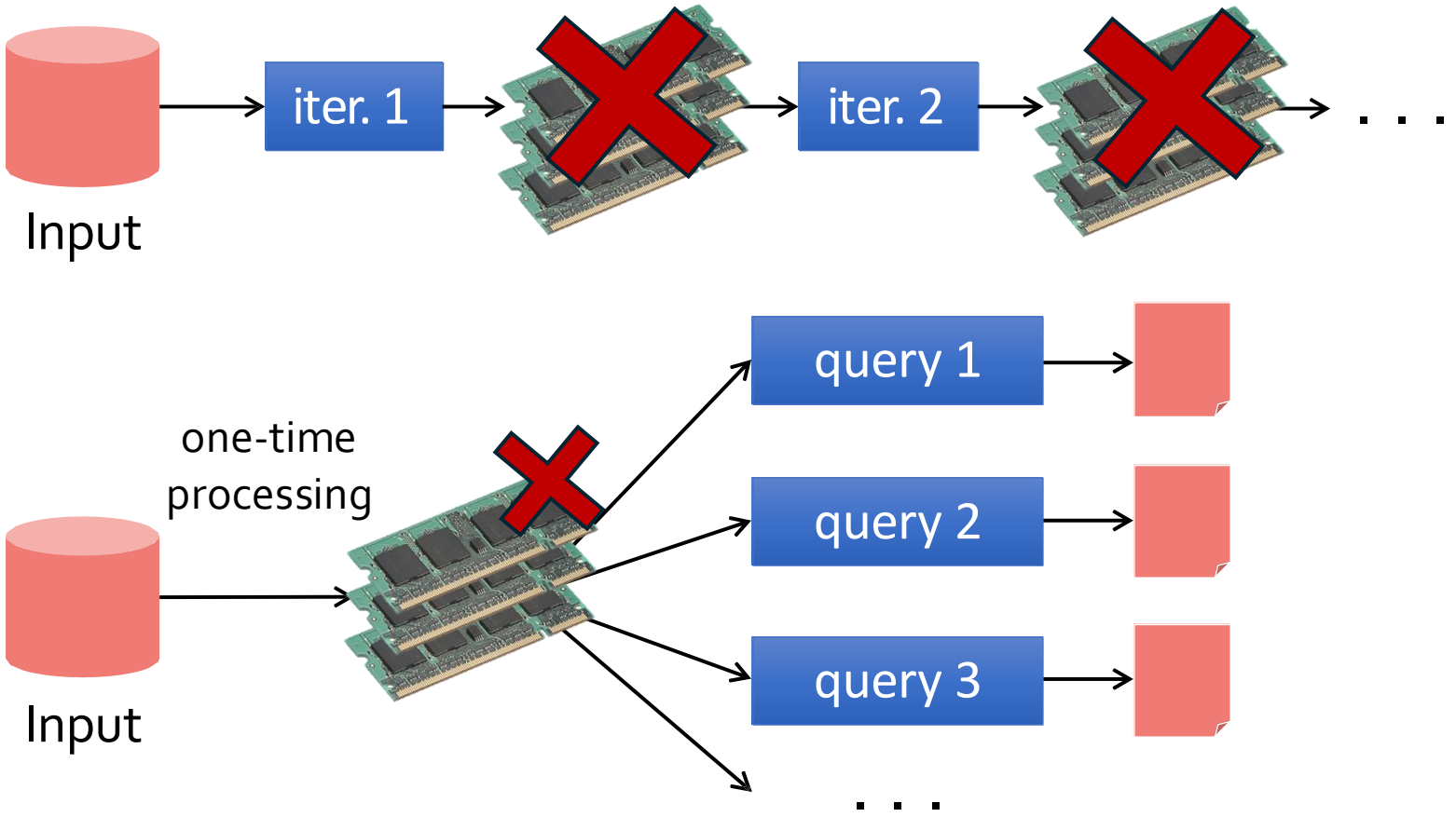
Restricted form of distributed shared memory

- 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

Goal: In-memory Data Sharing



Spark Programming interface

Scala API, now have multi-language bindings such as Python, Java, etc.

Managing RDDs

- Transformations on RDDs (RDD1 → RDD2)
- 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 operations; 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

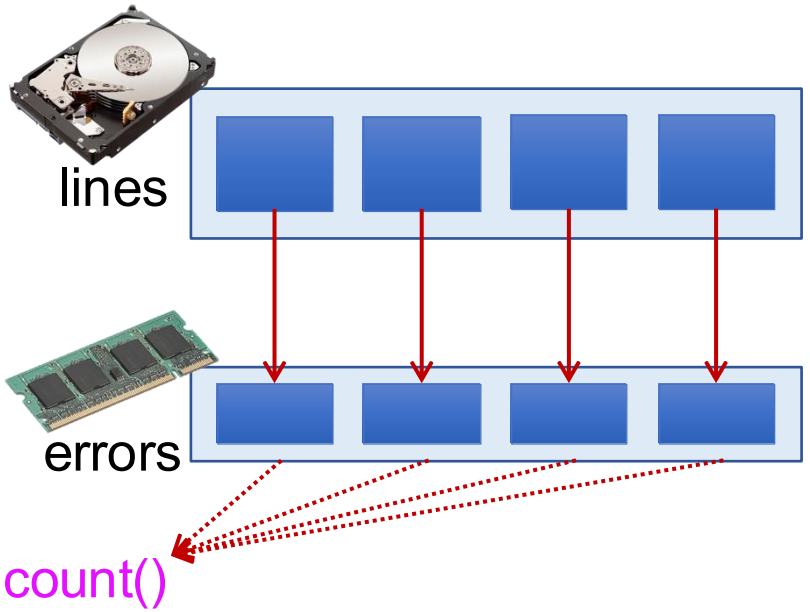
to interpreter/Scala (count, collect, reduce)

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

(for any missing pieces of the computation)

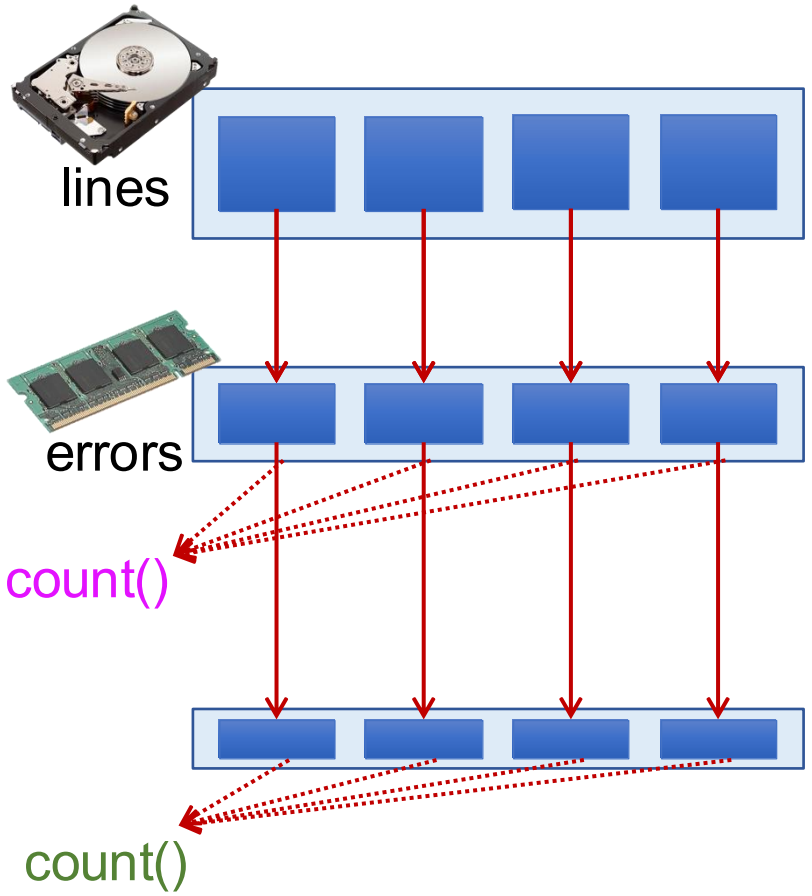
Interactive Debugging

```
lines = textFile(“hdfs://foo.log”)
errors = lines.filter(
         _.startsWith(“ERROR”)
errors.persist()
errors.count()
errors.filter(
    _.contains(“MySQL”))
```



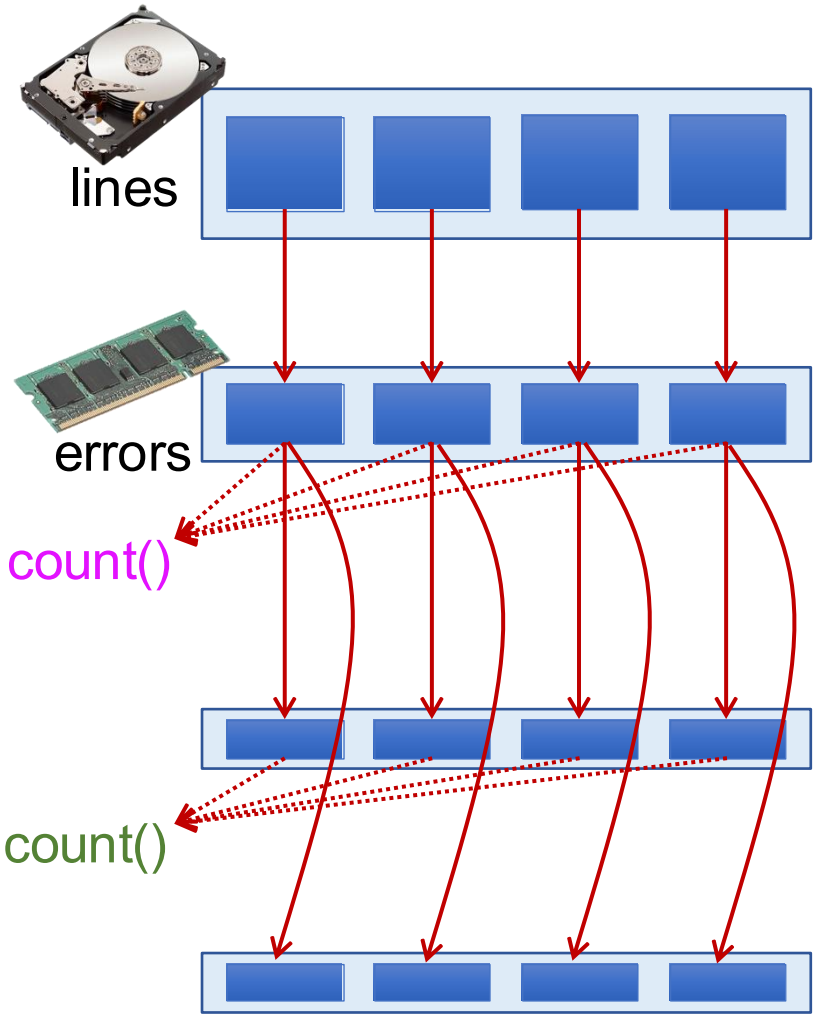
Interactive Debugging

```
lines = textFile(“hdfs://foo.log”)
errors = lines.filter(
         _.startsWith(“ERROR”)
errors.persist()
errors.count()
errors.filter(
     _.contains(“MySQL”)).count()
errors.filter(
     _.contains(“HDFS”))
     _.map(_.split(“\t”)(3))
     .collect()
```



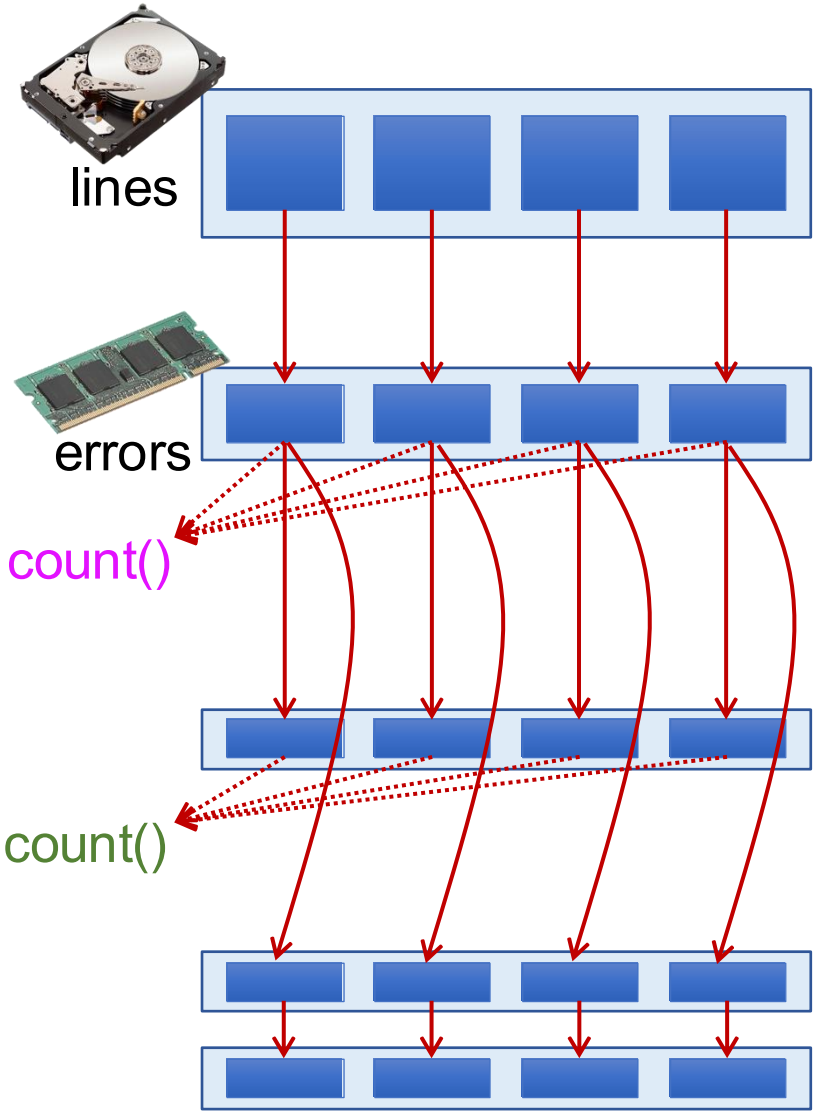
Interactive Debugging

```
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()
```



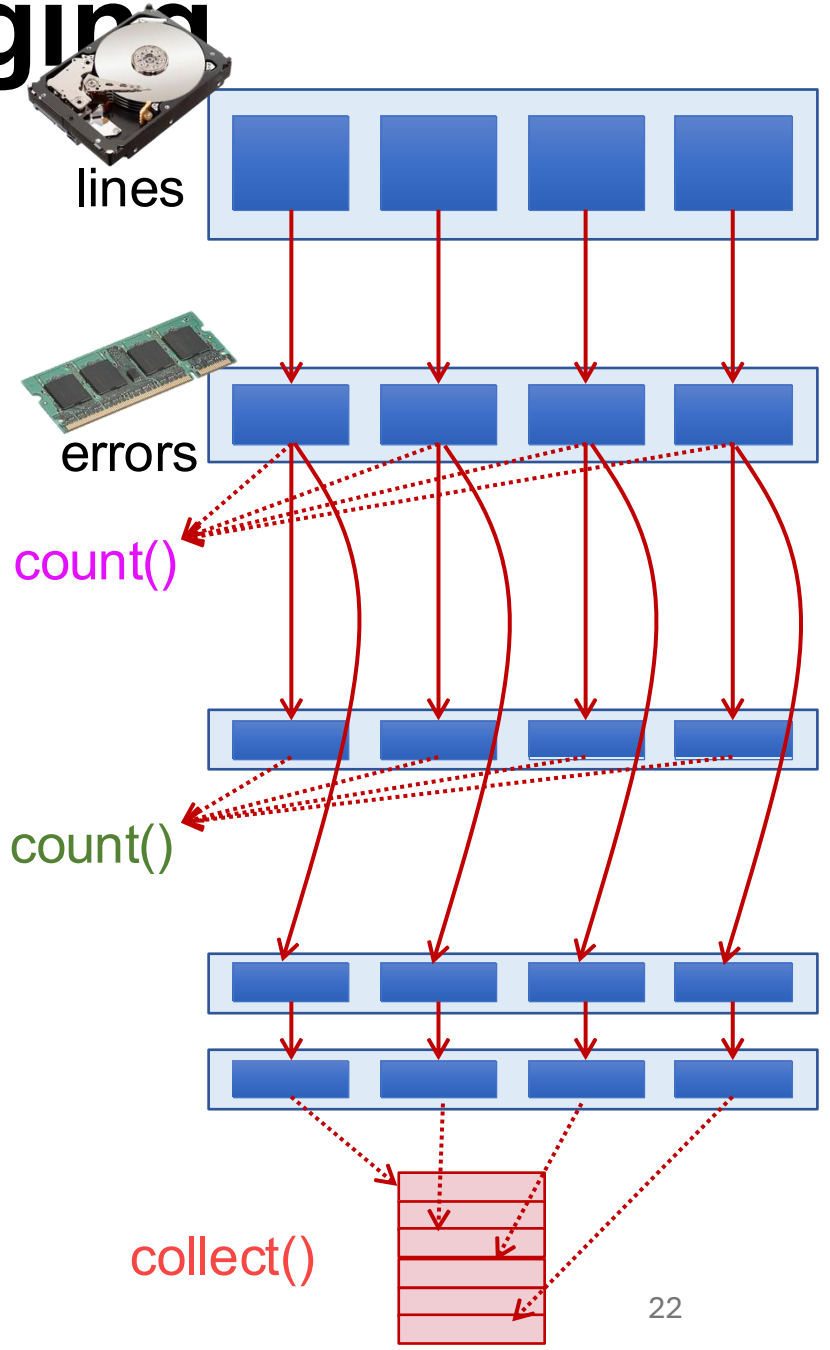
Interactive Debugging

```
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

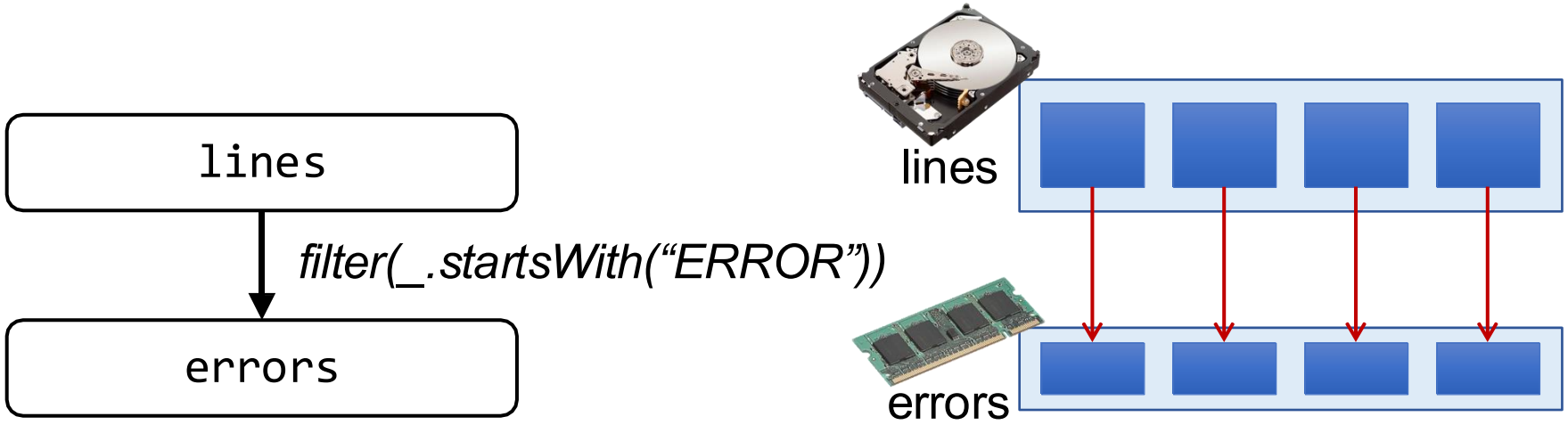
Tells which RDDs the Spark schedule should materialize and whether in memory or storage

Gives the user control over reuse/recompute/recovery tradeoffs

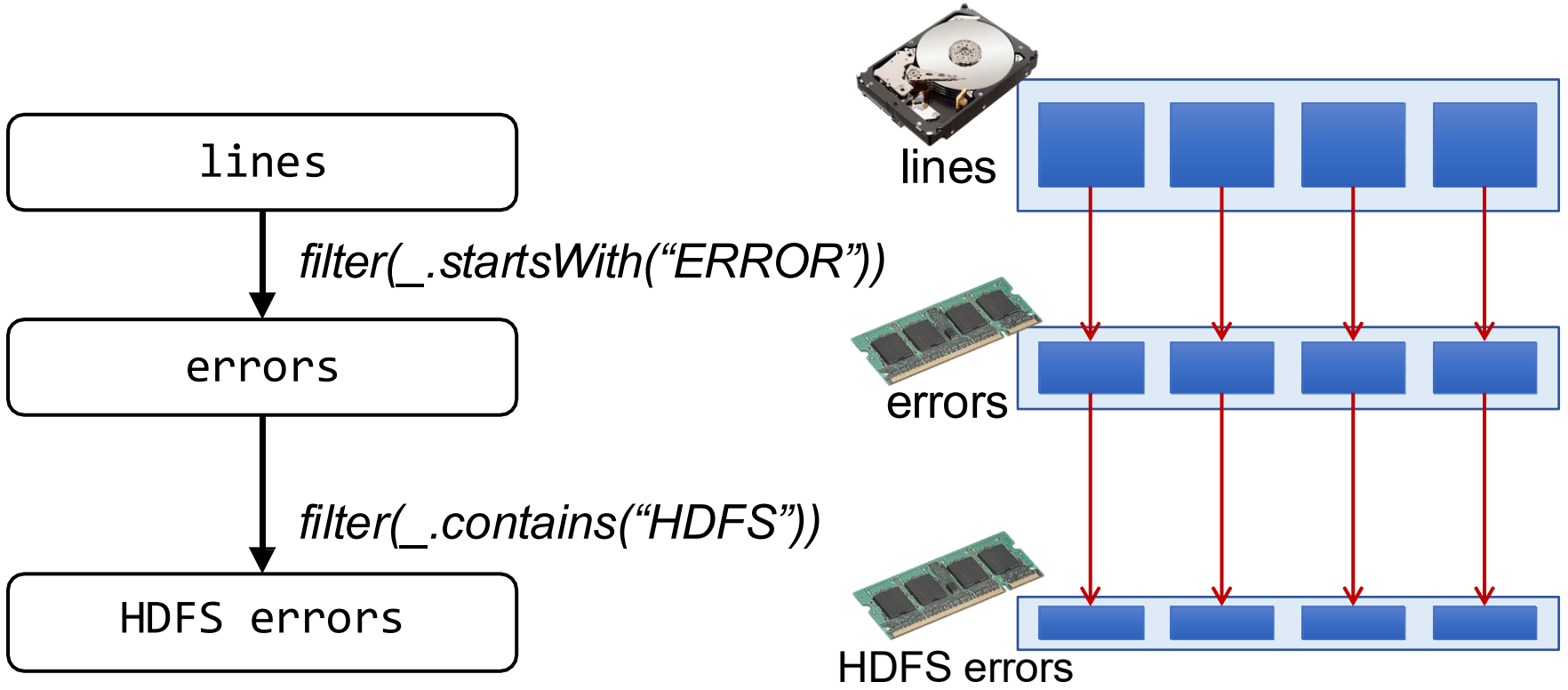
Lineage Graph of RDDs

`lines`

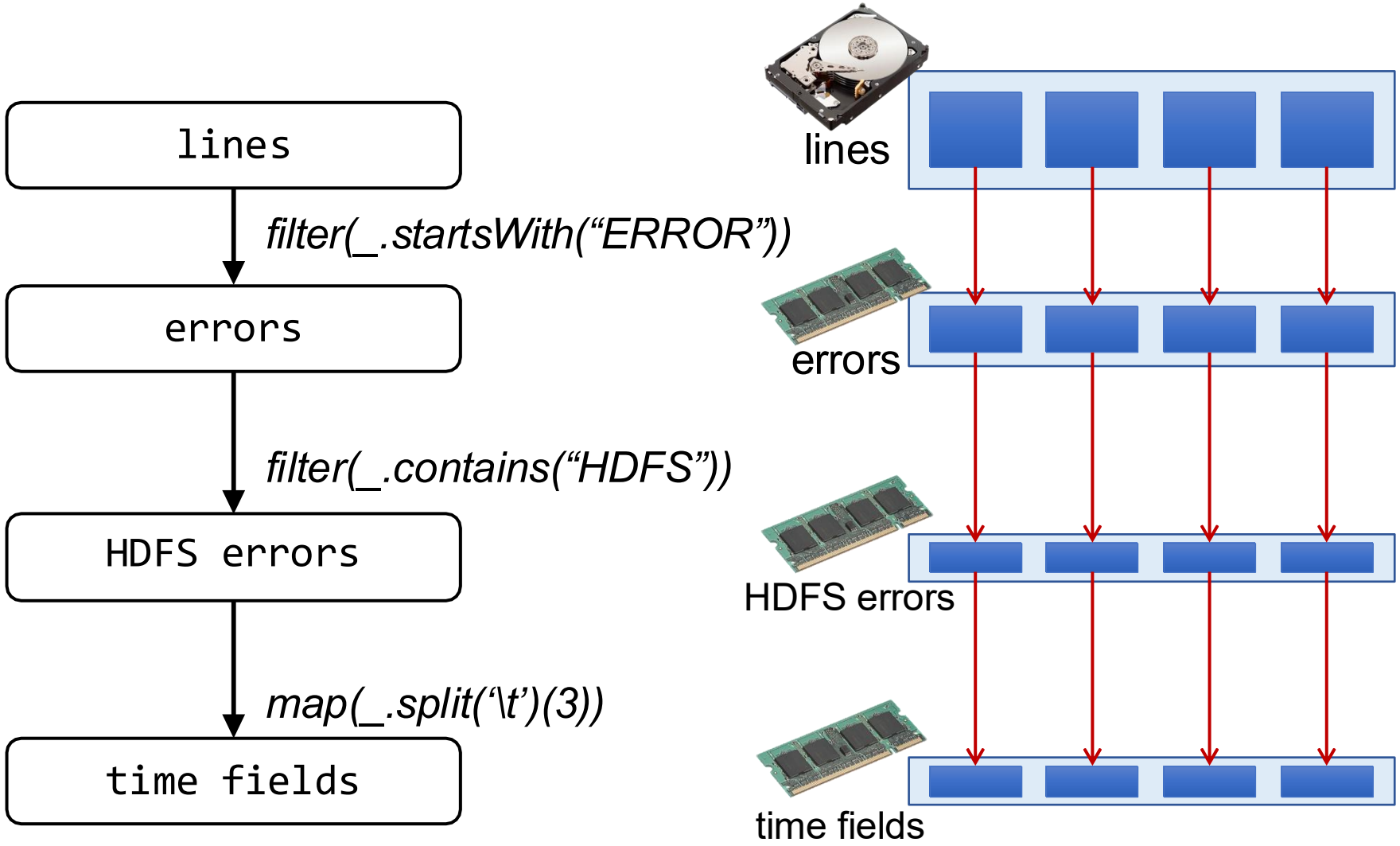
Lineage Graph of RDDs



Lineage Graph of RDDs



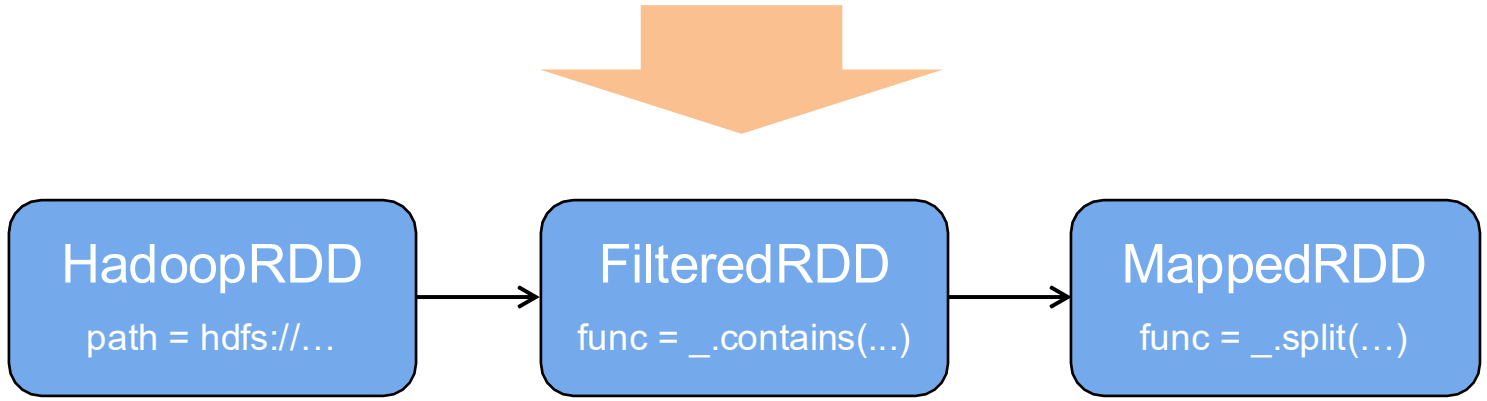
Lineage Graph of RDDs



Fault Recovery

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

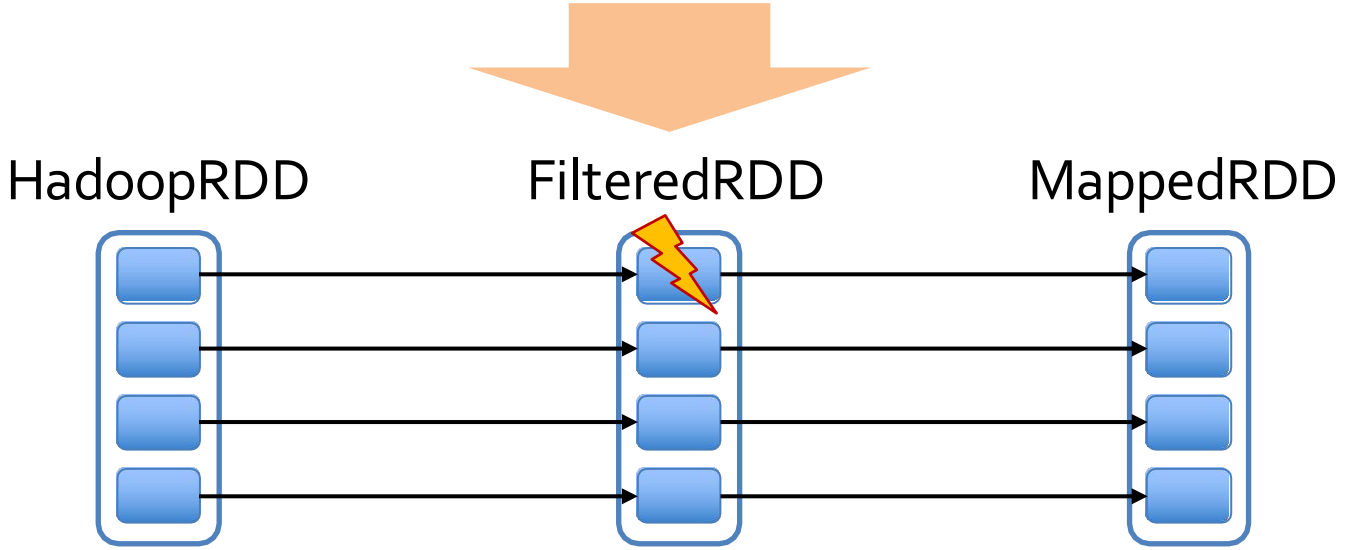
E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



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))
```



Example: PageRank

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

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

```
val lines = spark.read.textFile("in").rdd
val links1 = lines.map{ s =>
    val parts = s.split("\\s+")
   (parts(0), parts(1))
}
val links2 = links1.distinct()
val links3 = links2.groupByKey()
val links4 = links3.cache()
var ranks = links4.mapValues(v => 1.0)

for (i <- 1 to 10) { 12 val jj = links4.join(ranks) 13 val contribs =
jj.values.flatMap{ 14 case (urls, rank) => 15 urls.map(url => (url,
rank / urls.size)) 16 } 17 ranks = contribs.reduceByKey(_ +
_).mapValues(0.15 + 0.85 * _) 18 } 19 20 val output =
ranks.collect() 21 output.foreach(tup => println(s"${tup._1} has
```

```
val lines = spark.read.textFile("in").rdd
val links1 = lines.map{ s =>
    val parts = s.split("\\s+")
   (parts(0), parts(1))
}
val links2 = links1.distinct()
val links3 = links2.groupByKey()
val links4 = links3.cache()
var ranks = links4.mapValues(v => 1.0)

for (i <- 1 to 10) {
   val jj = links4.join(ranks)
   val contribs = jj.values.flatMap{
   case (urls, rank) =>
      urls.map(url => (url, rank / urls.size))
   }
   ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 *_)
}

val output = ranks.collect()
output.foreach(tup => println(s"${tup._1} has rank: ${tup._2} ."))
```

Input:

u1 u3

u1 u1

u2 u3

u2 u2

u3 u1

Demo

```
val lines = spark.read.textFile("in").rdd
val links1 = lines.map{ s =>
    val parts = s.split("\\s+")
   (parts(0), parts(1))
}
val links2 = links1.distinct()
val links3 = links2.groupByKey()
val links4 = links3.cache()
var ranks = links4.mapValues(v => 1.0)

for (i <- 1 to 10) {
   val jj = links4.join(ranks)
   val contribs = jj.values.flatMap{
      case (urls, rank) =>
         urls.map(url => (url, rank / urls.size))
      }
   ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 *_)
}

val output = ranks.collect()
output.foreach(tup => println(s"${tup._1} has rank: ${tup._2} ."))
```

Input:

u1 u3
u1 u1
u2 u3
u2 u2
u3 u1

Join(𝔚)

Alice	5
Bob	6
Claire	4

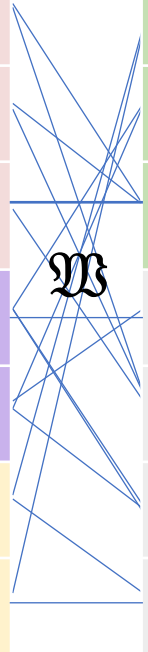
𝔚

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

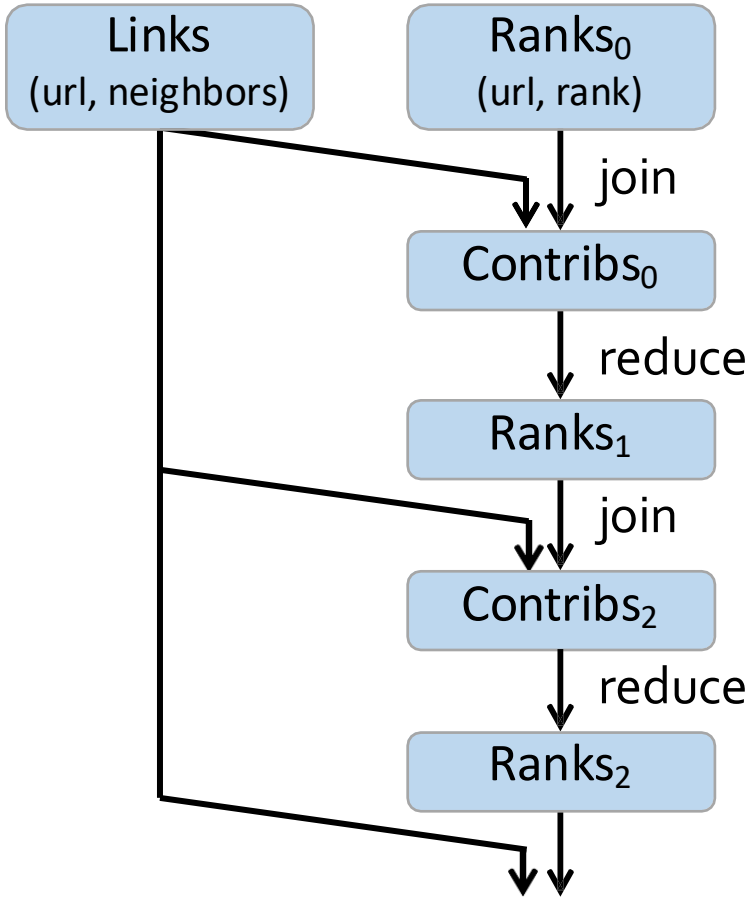


𝔚

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.

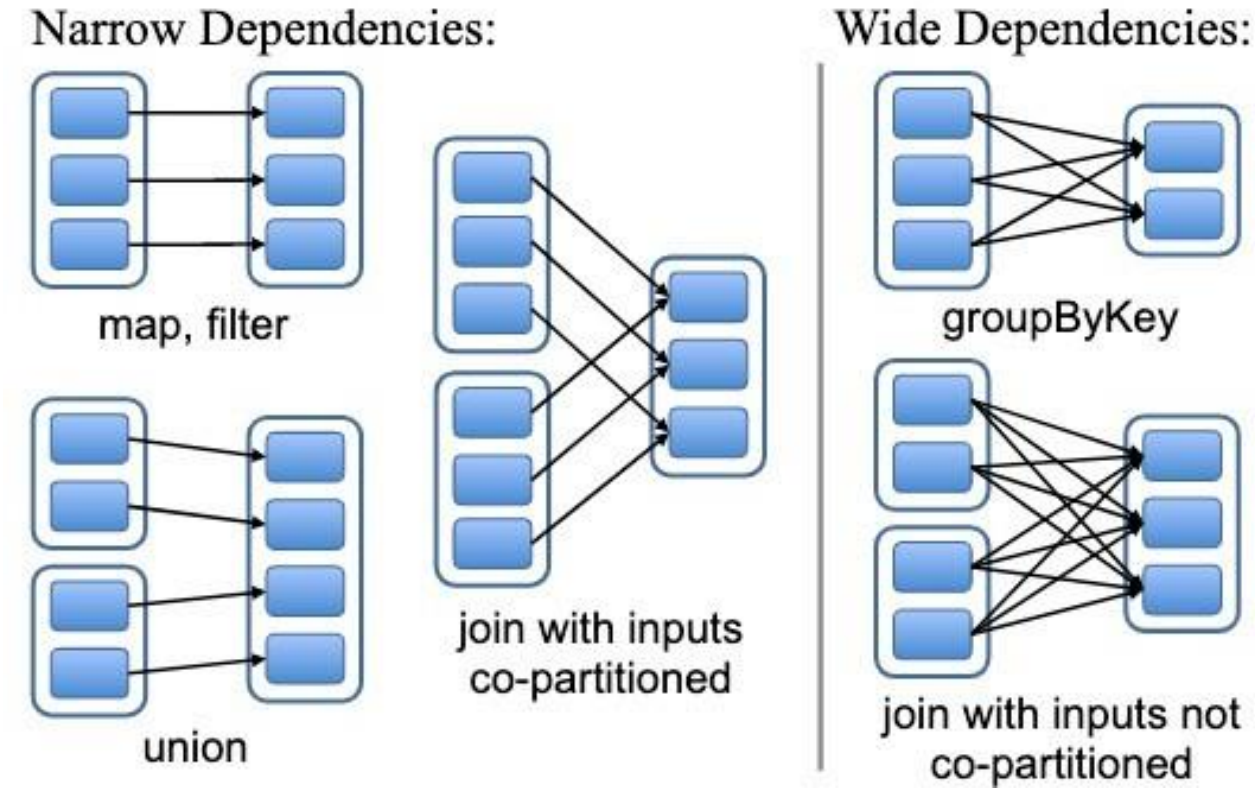
Optimizing placement



- 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())`

Q: Where might we have placed `persist()`?

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