Resilient Distributed Datasets:
A Fault Tolerant Abstraction for In-Memory Cluster Computing

By Matei Zahariz, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Best Paper award **@ NSDI 2012**

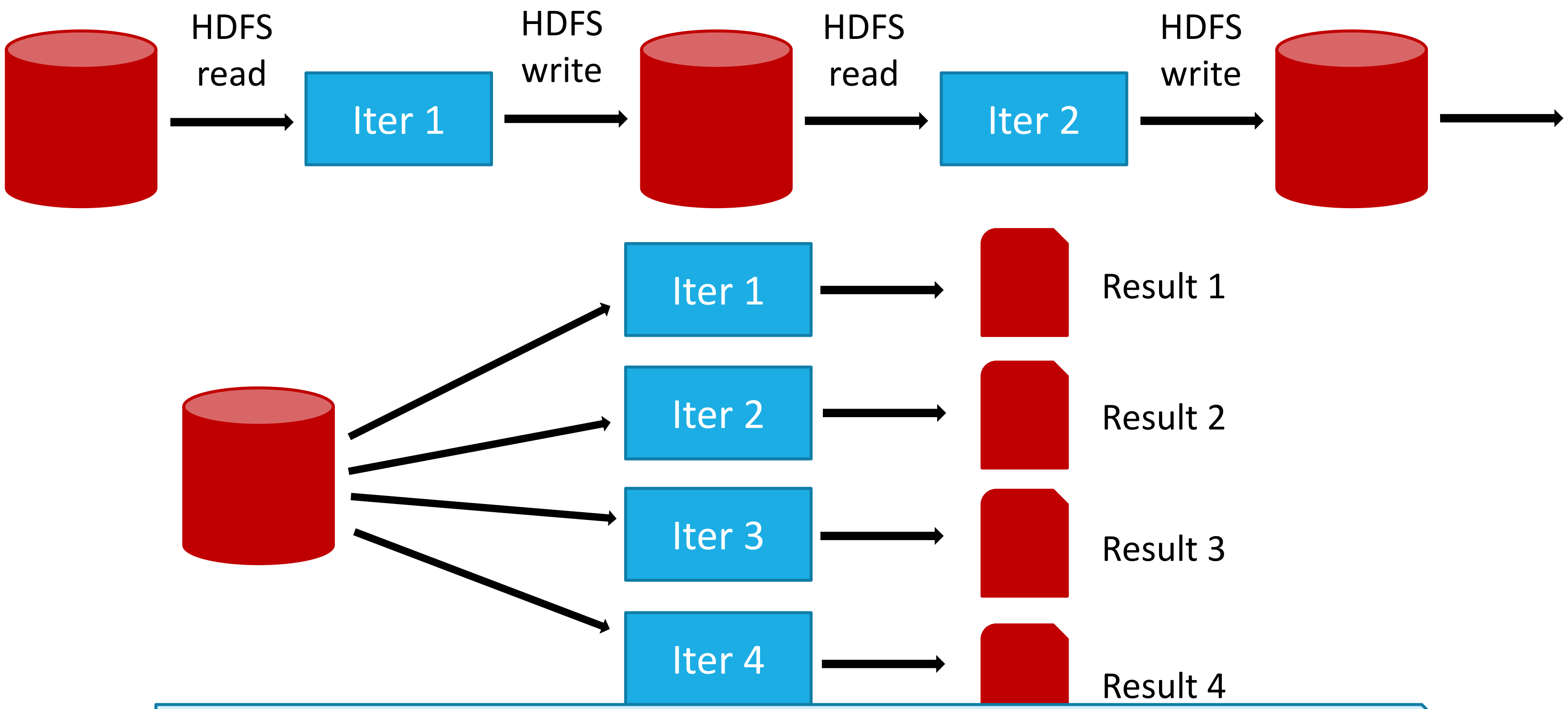
PRESENTED BY CAROLYN LIANG
CMPT843

Motivation

- MapReduce greatly simplified “big data” analysis on large, unreliable clusters.

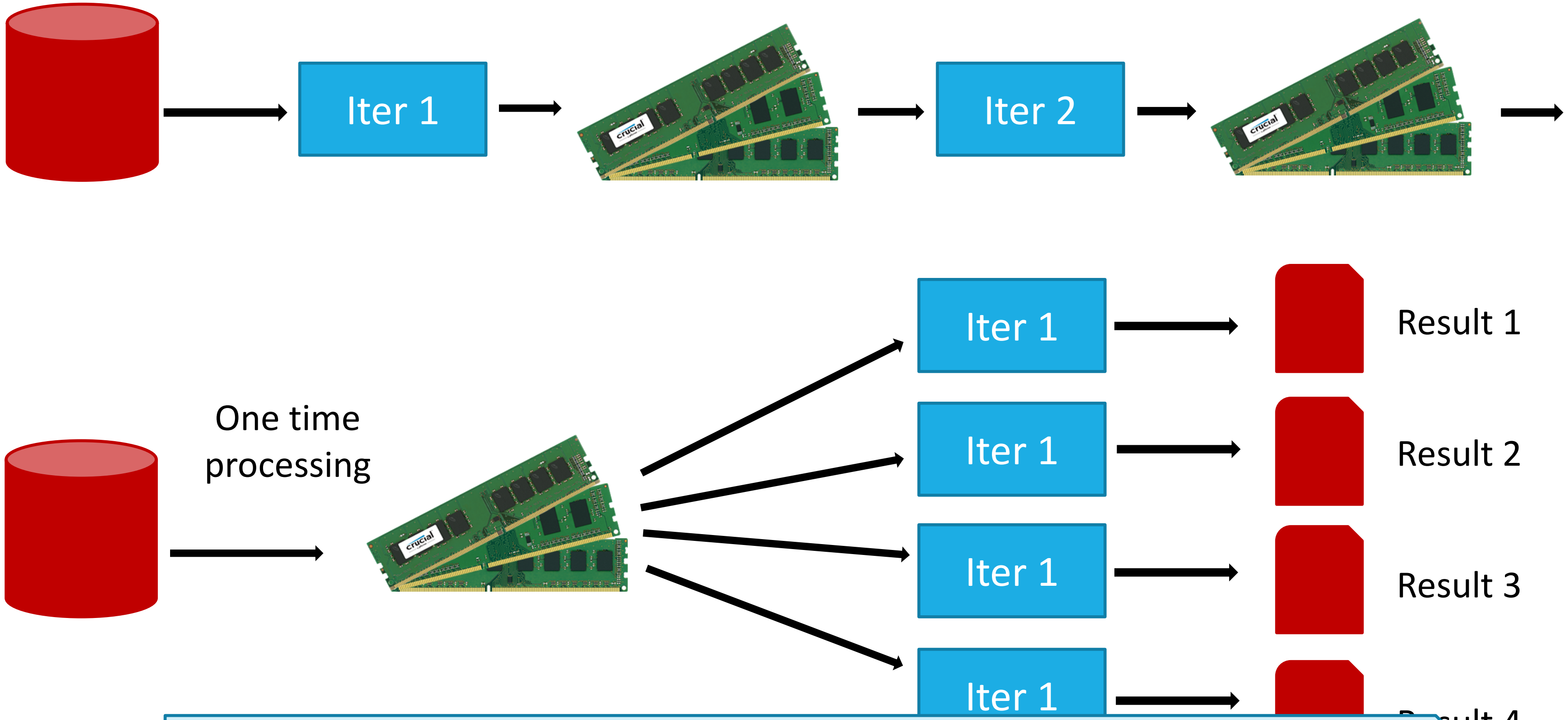
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage applications (e.g. iterative machine learning & graph processing)
  - More interactive ad-hoc queries

Hadoop MapReduce



Slow due to data replication, disk I/O,
But necessary for fault tolerance.

Want: in-memory data sharing



10-100 faster than network/disk, but how to get fault tolerance

Challenge

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

Challenge

- Existing storage abstractions have interfaces based on fine-grained updates to mutable state
  - RAMCloud, databases, distributed mem, Piccolo

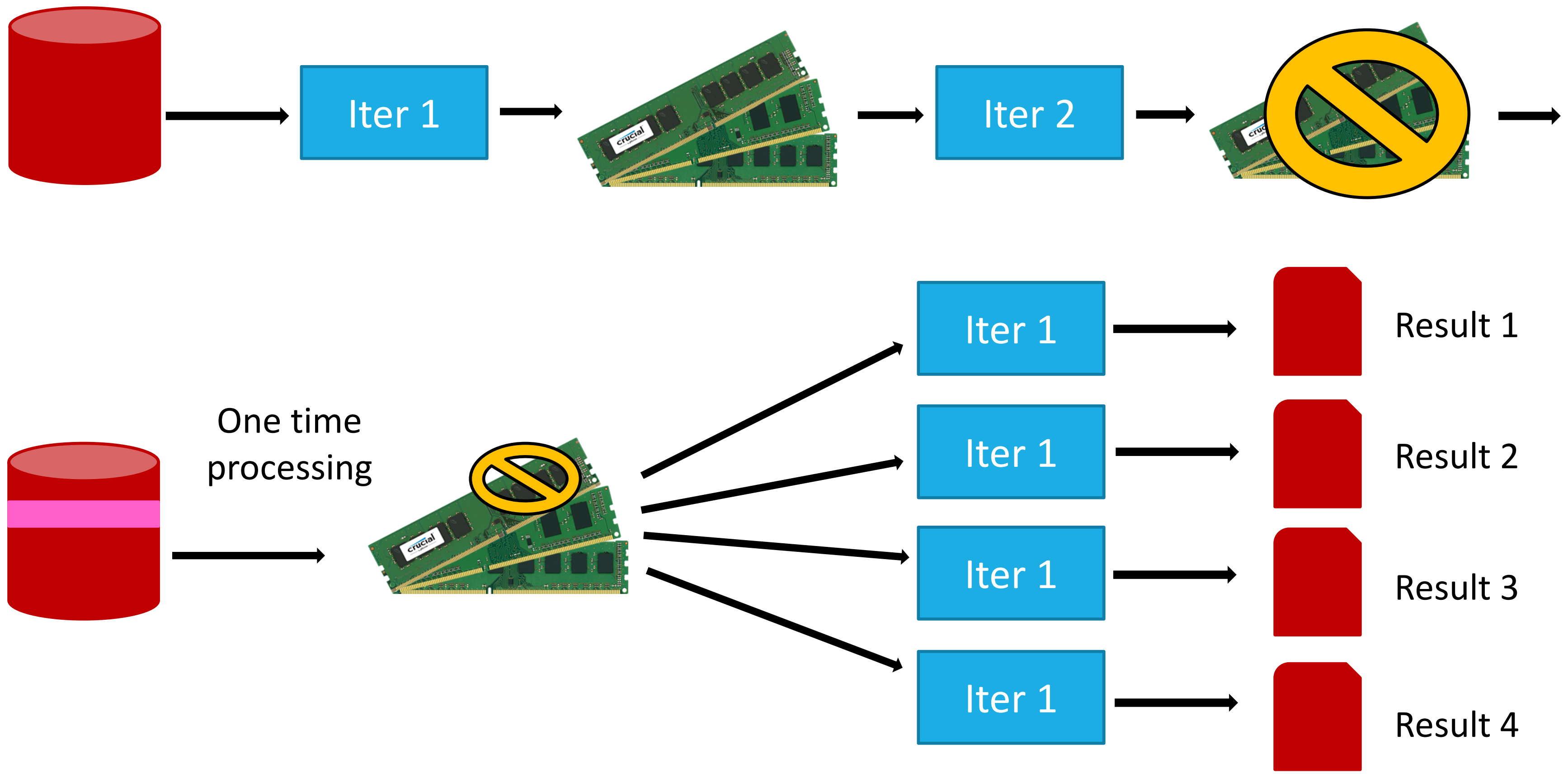
- Fault Tolerance:
  - Requires replicating data
  - logs across nodes

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
  - Recomputed lost partitions on failure
  - No cost if nothing fails

RDDs – Fault tolerant



Outline

- Spark Programming Interface
  - Example: Log Mining
  - Example: PageRank
- Implementation
- Conclusion

Spark Programming Interface

Provides:

- Resilient distributed datasets (RDDs)
- Operations on RDDs:
  - transformations (build new RDDs)
  - actions (compute and output results)
- Control of each RDD's partitioning (layout across nodes) and persistence (storage in RAM, on disk, etc)

RDD on Spark

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Example: Log Mining

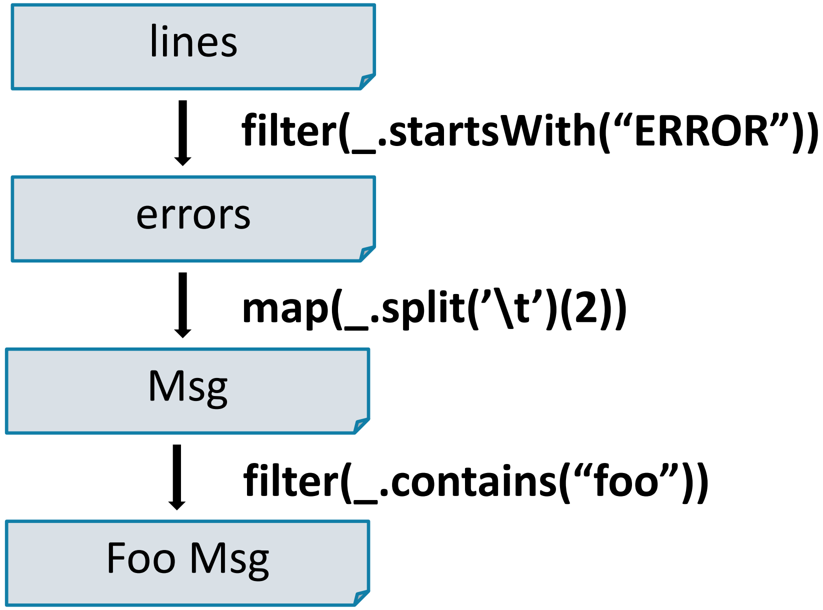
Load error messages from a log into memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://…")

errors = lines.filter(_.startsWith("ERROR"))
Msg = error.map(_.split('\t')(2))
Msg.persist()


Msg.filter(_.contains('foo')) .count()
Msg.filter(_.contains('bar')) .count()
```

transformation

action

Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

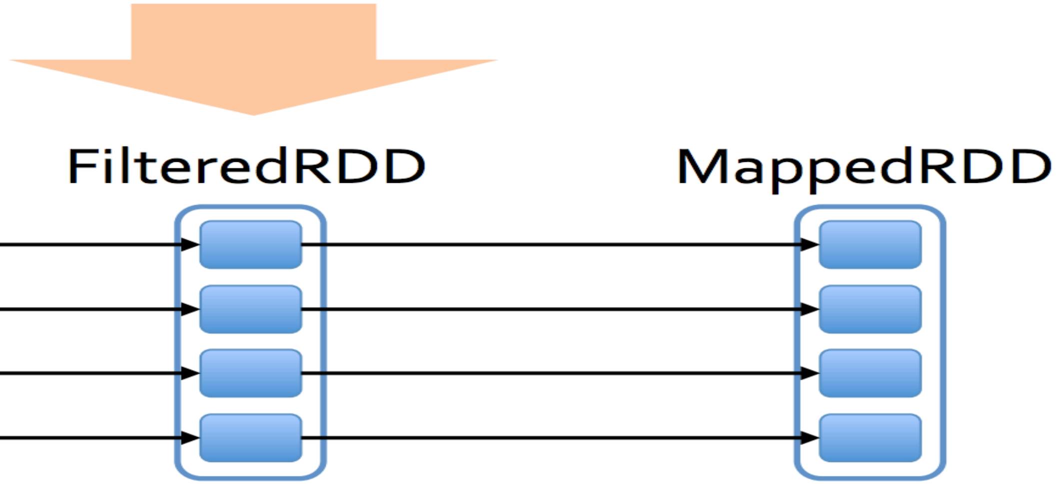


Lineage graph for 3rd query

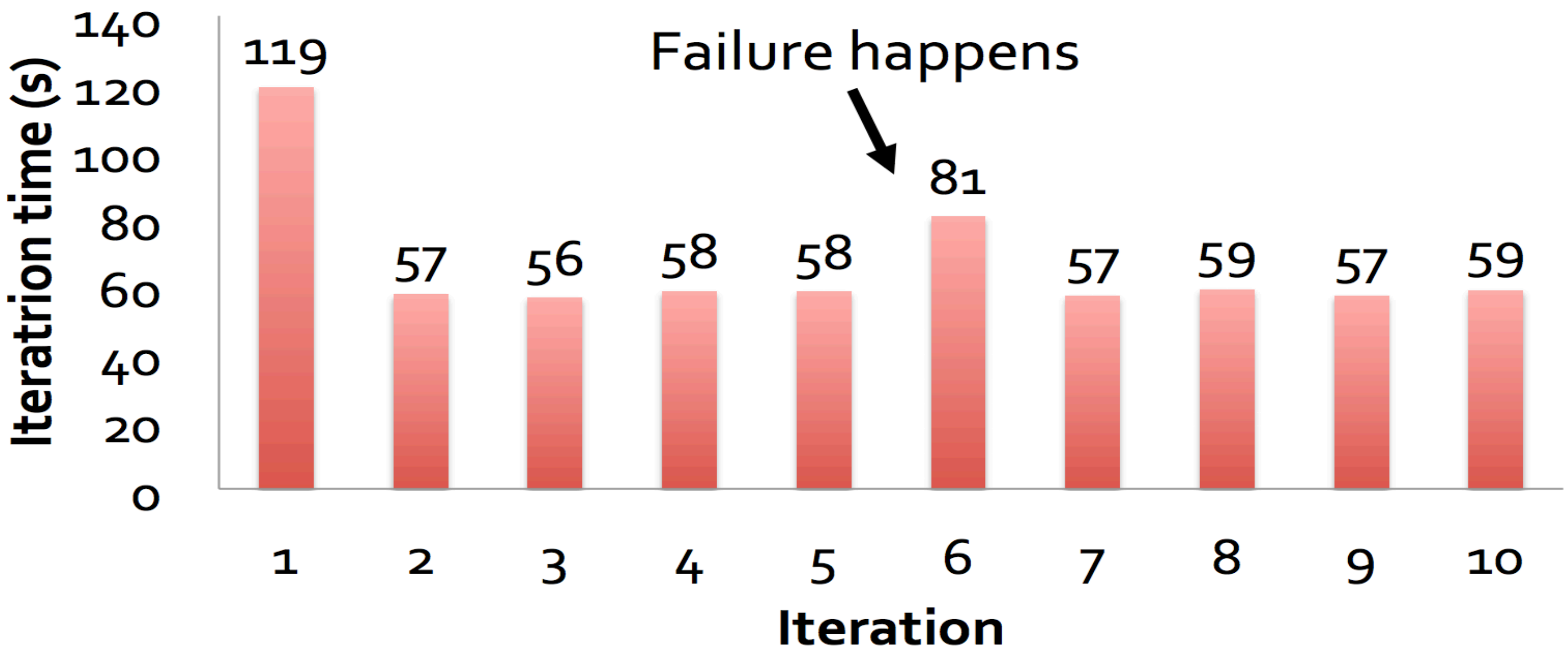
Fault Recovery

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



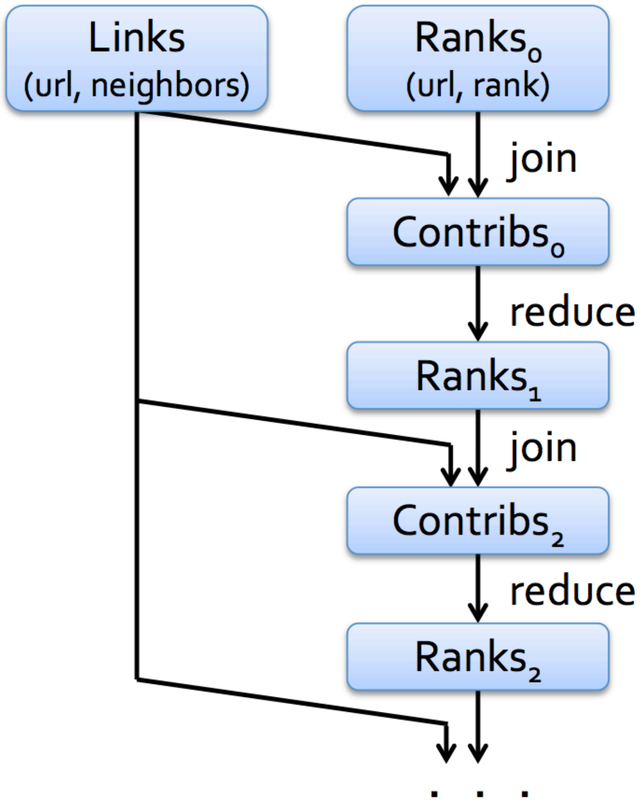
Example: PageRank

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

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

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

Optimizing Placement



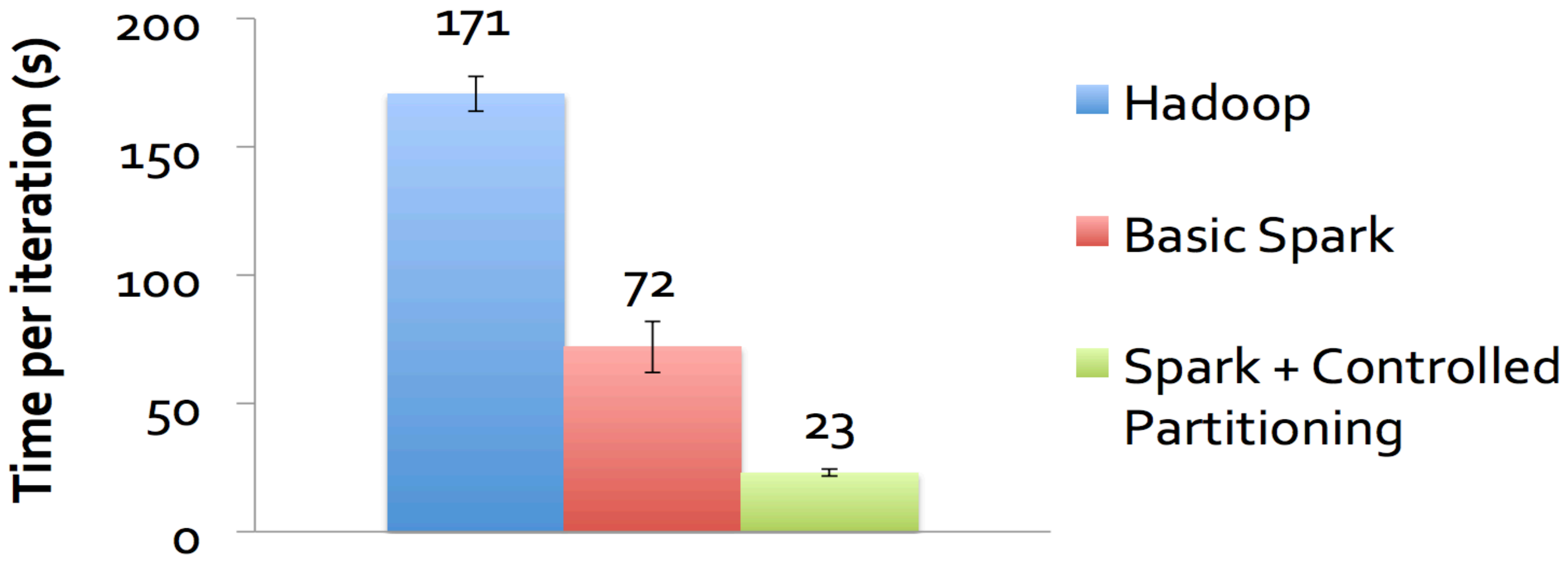
links & ranks repeatedly joined

-Can co-partition them (e.g. hash both on URL) to avoid shuffles.

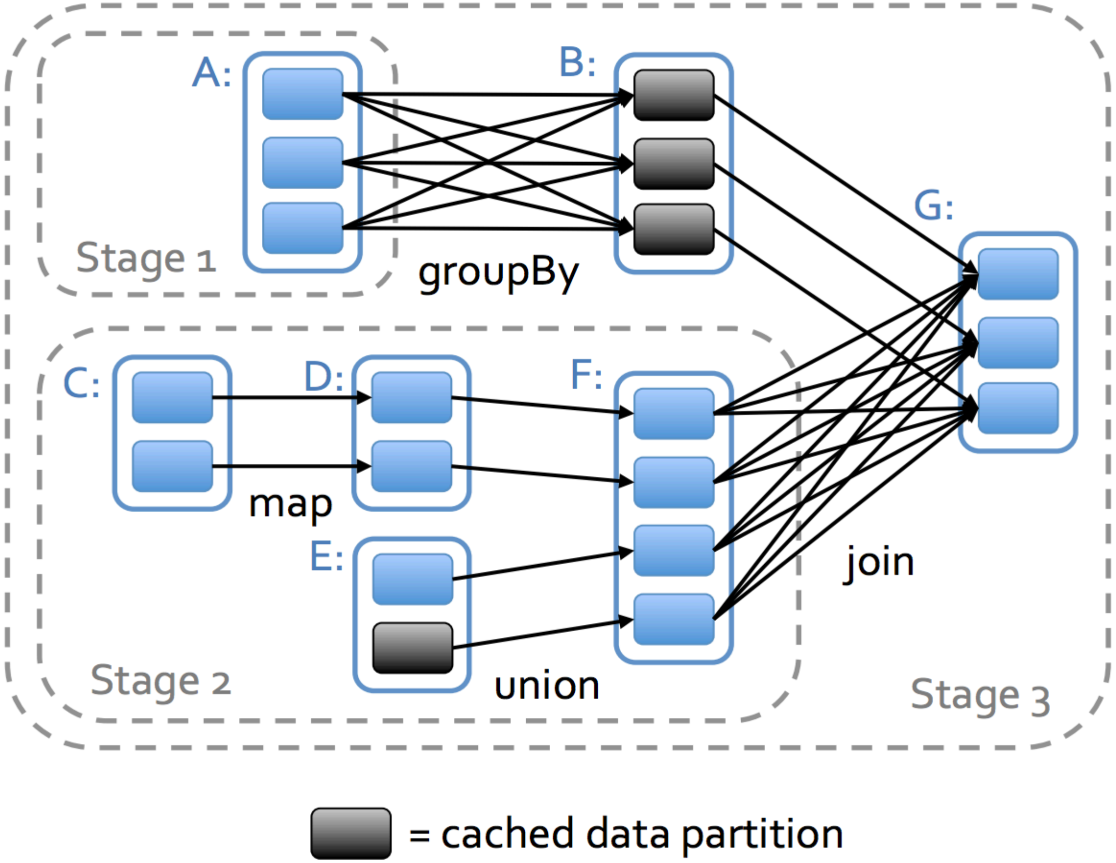
-Can also use app knowledge, e.g., hash on DNS name

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

PageRank Performance



Implementation- Job scheduler



- Each stage contains as many narrow dependencies as possible
- boundaries of the stages are the shuffle operations required for wide dependence

Implementation- Memory Management

-Three ways for storage of persistent RDDs:

- In-memory storage as desterilized Java Objects
- In-memory storage as serialized data
- On-disk storage

- Use LRU eviction policy

Conclusion

- RDDs offer a simple and efficient programming model for a broad range of applications

- Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery

Questions?