Imperial College London

LSDS Large-Scale Data & Systems Group

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

Peter Pietzuch
prp@doc.ic.ac.uk

Department of Computing
Imperial College London
http://lsds.doc.ic.ac.uk

Slides based on the RDD NSDI'12 talk

Autumn 2020

Spark Motivation

MapReduce greatly simplified "big data" analysis on large, unreliable clusters

But as soon as it got popular, users wanted more:

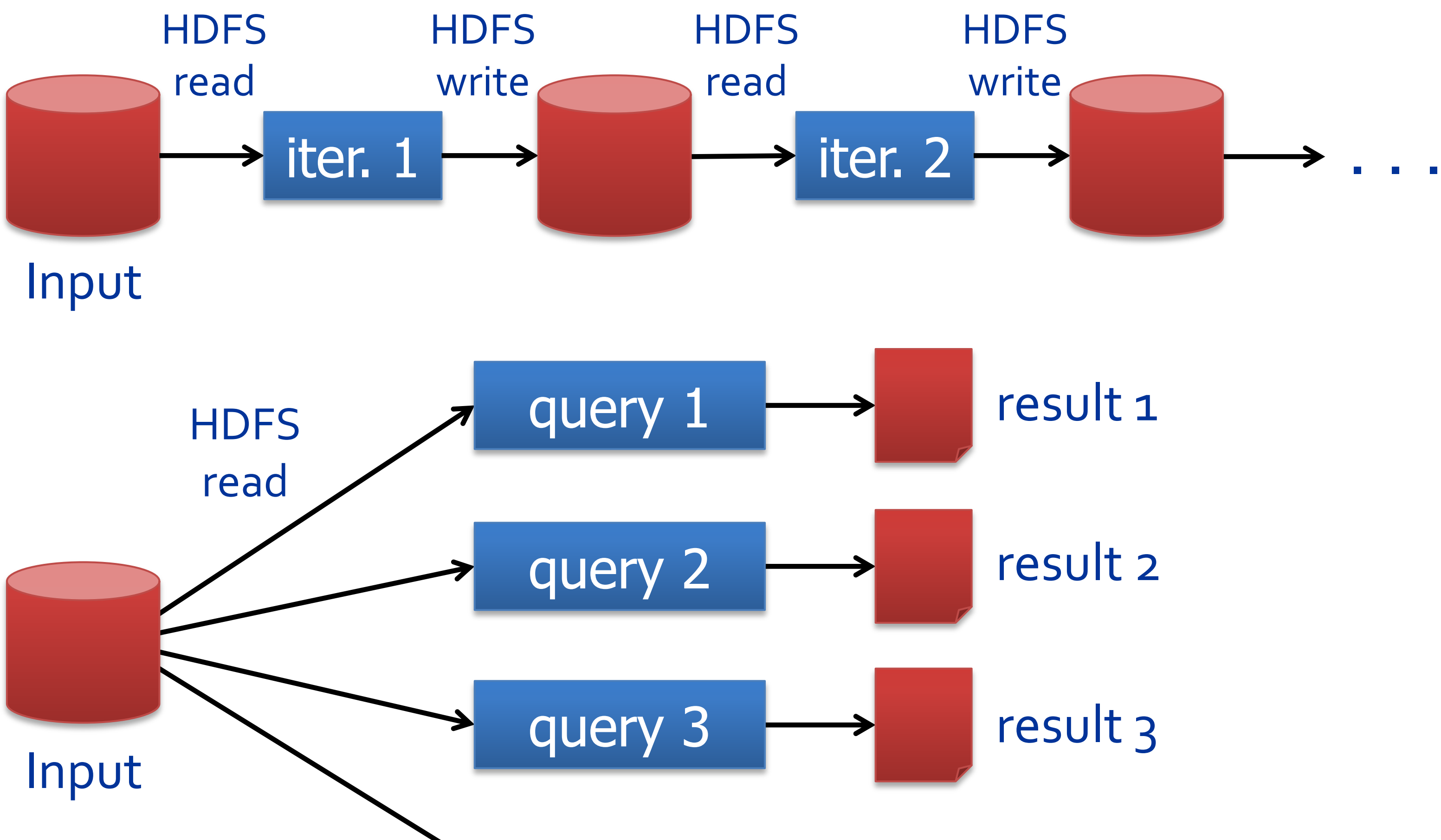
- More **complex**, multi-stage applications
  (e.g. iterative machine learning & graph processing)
- More **interactive** ad-hoc queries

Complex apps and interactive queries both need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

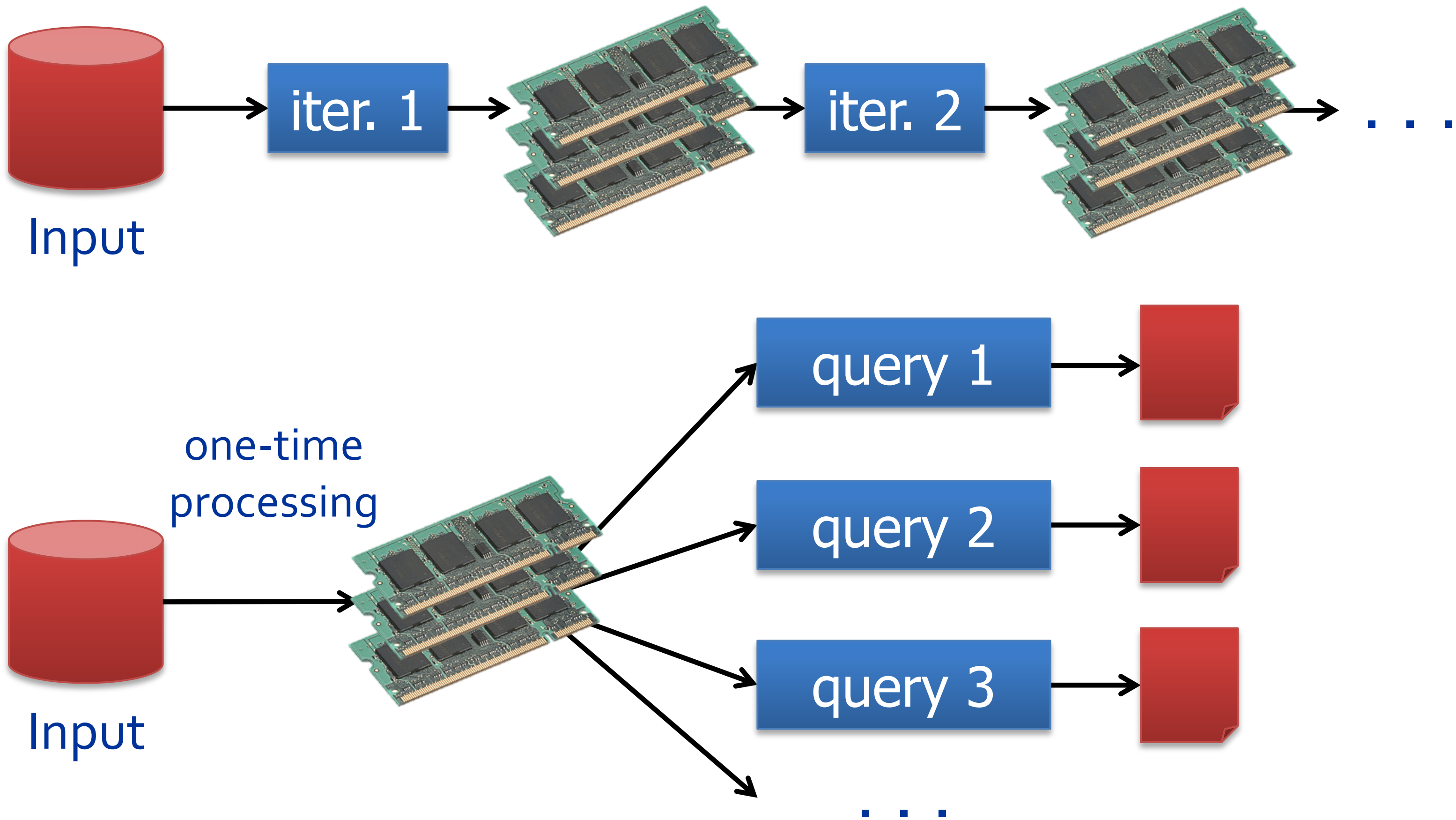
In MapReduce, the only way to share data across jobs is stable storage ➔ slow!

Disk-based Data Sharing



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

Goal: In-Memory Data Sharing



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

Challenge

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

Existing storage abstractions have interfaces based on fine-grained updates to mutable state

- RAMCloud, databases, distributed mem, Piccolo

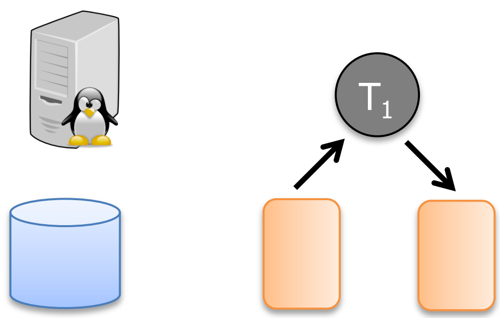
Requires replicating data or logs across nodes for fault tolerance

- Costly for data-intensive apps
- 10-100x slower than memory write

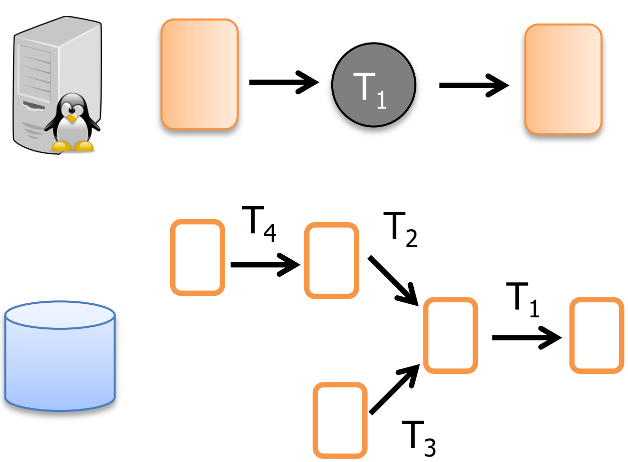
Berkeley,
USENIX
NSDI'12

Resilient Distributed Datasets (RDDs)

Map
Reduce



Spark



Lineage graph

Idea:

- Store data lineage instead of data
- Recompute data based on lineage

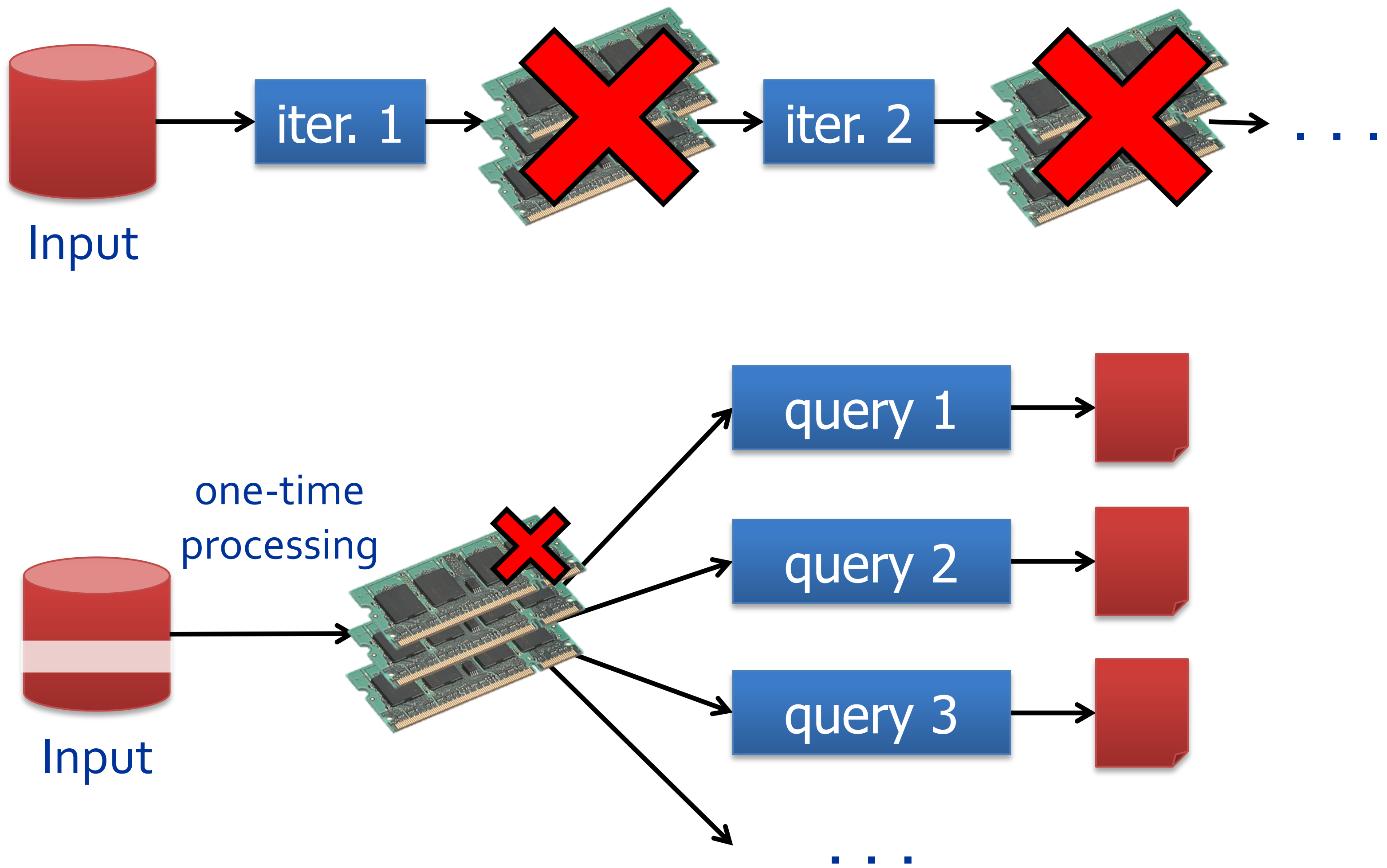
Resilient Distributed Datasets (RDD)

- Partitioned across nodes
- Immutable to simplify lineage tracking
- Can only be built through coarse-grained deterministic transformations (map, filter, join, …)
- Checkpointing to disk to avoid unbounded lineage

Enables efficient in-memory processing

- Order of magnitude faster iterative algorithms

RDD Recovery



Generality of RDDs

Despite their restrictions, RDDs can express surprisingly many parallel algorithms

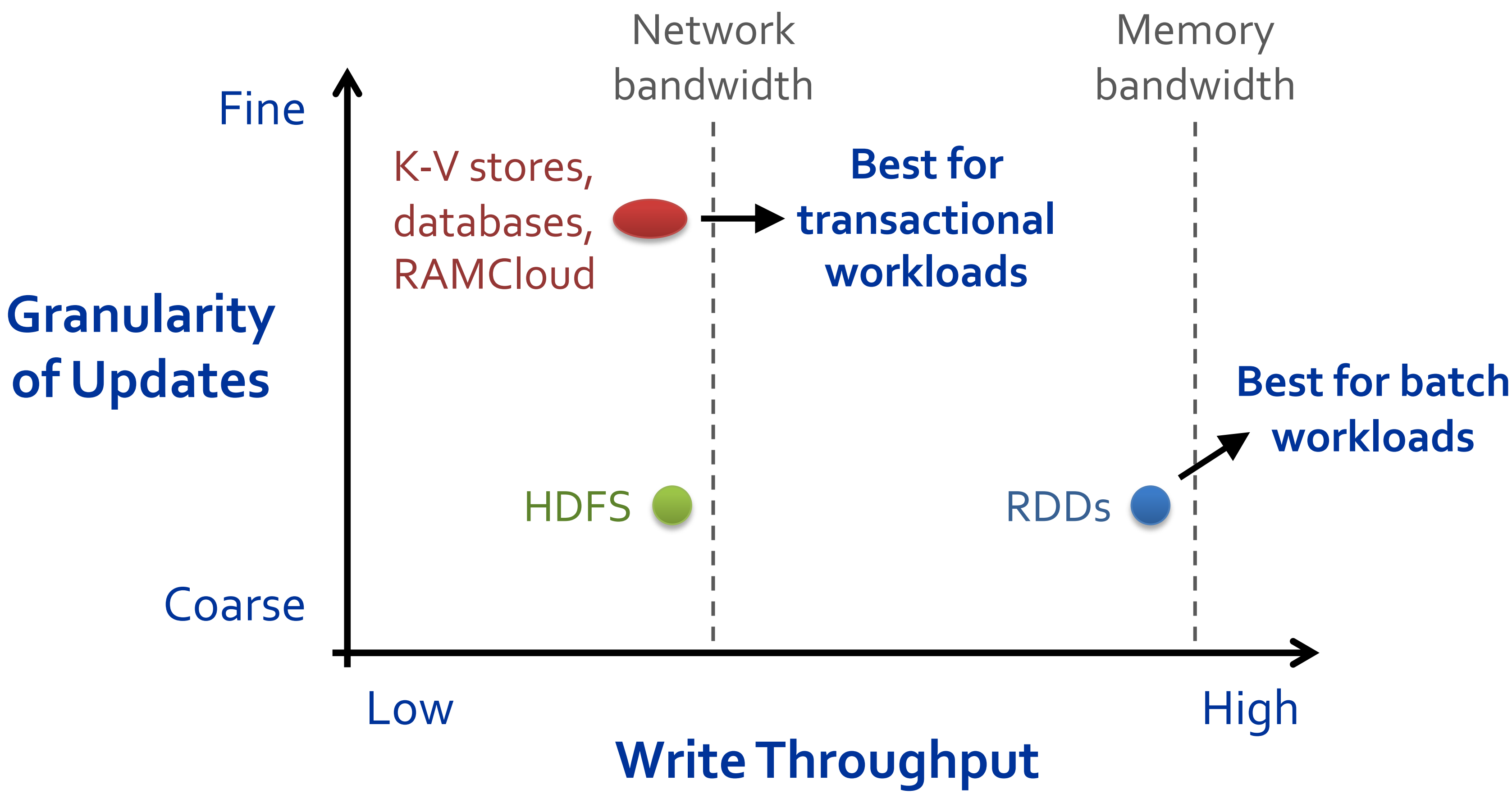
- These naturally apply the same operation to many items

Unify many current programming models

- Data flow models: MapReduce, Dryad, SQL, …
- Specialized models for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …

Support new apps that these models don't

Tradeoff Space



Spark Programming Interface

DryadLINQ-like API in the Scala language

Usable interactively from Scala interpreter

Provides:

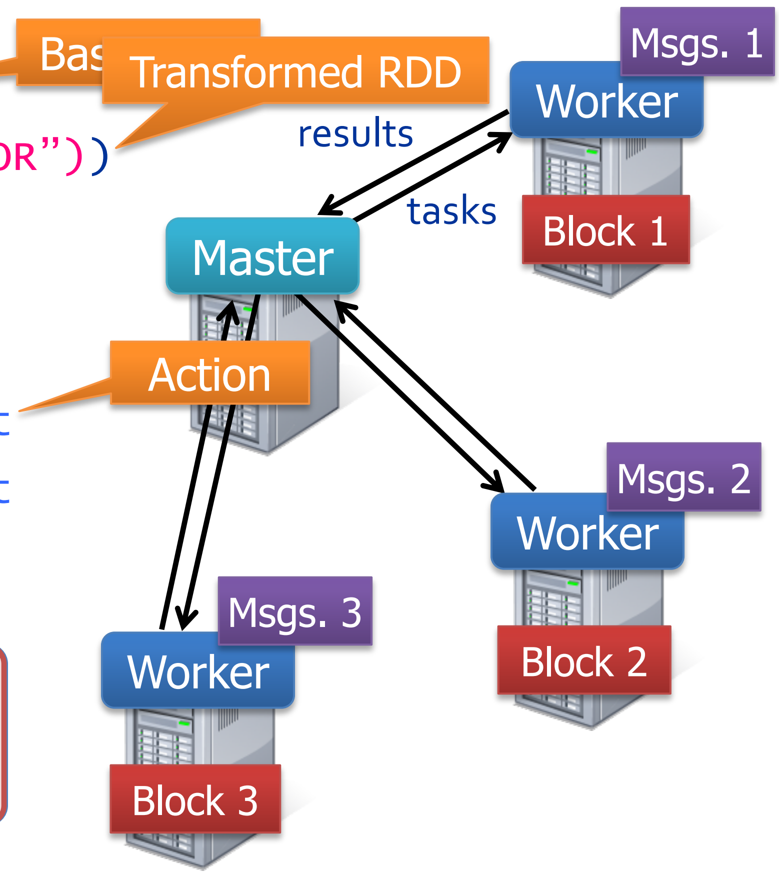
- Resilient distributed datasets (RDDs)
- Operations on RDDs: transformations (build new RDDs), actions (compute and output results)
- Control of each RDD’s partitioning (layout across nodes) and persistence (storage in RAM, on disk, etc)

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”))
messages = errors.map(_.split(‘\t’)(2))
messages.persist()

messages.filter(_.contains(“foo”)).count
messages.filter(_.contains(“bar”)).count
```



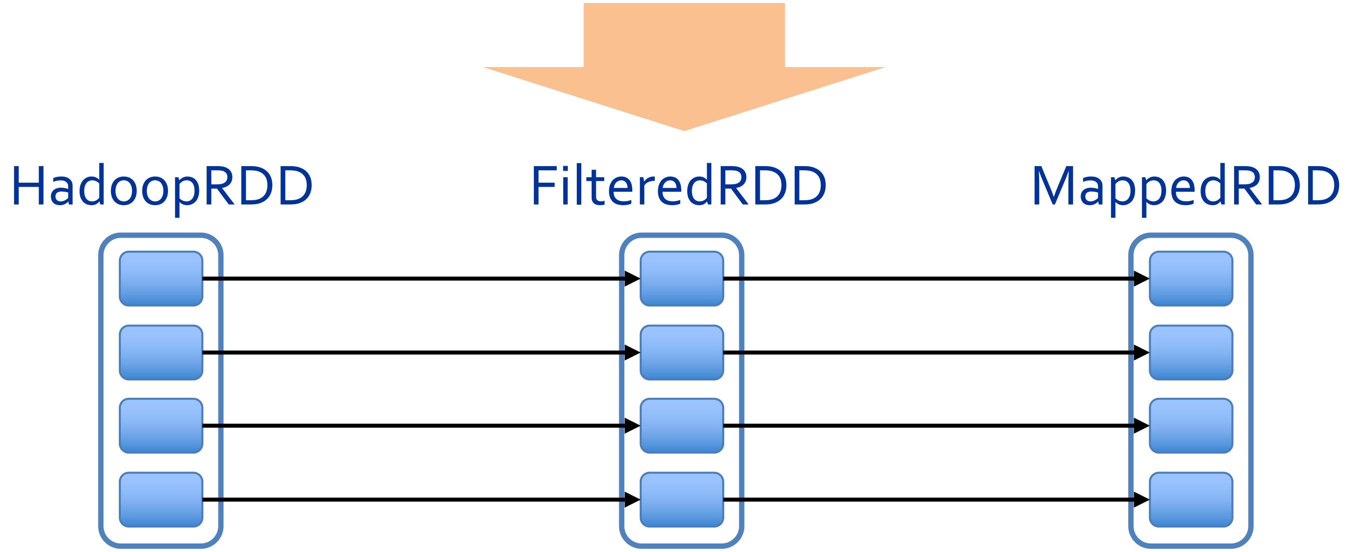
Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

Fault Recovery

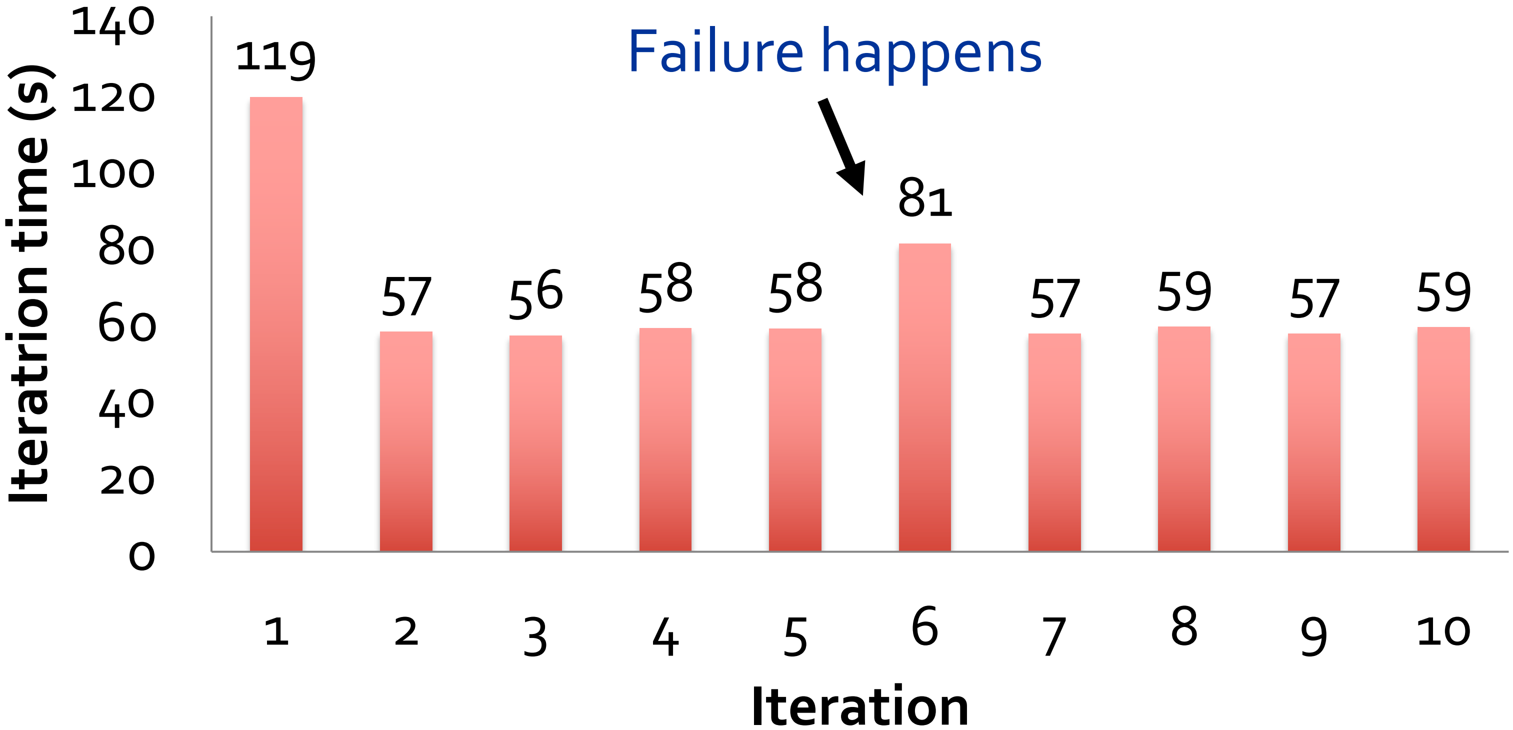
RDDs track the graph of transformations that built them (their lineage) to rebuild lost data

E.g.:

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



Fault Recovery Results



Example: PageRank

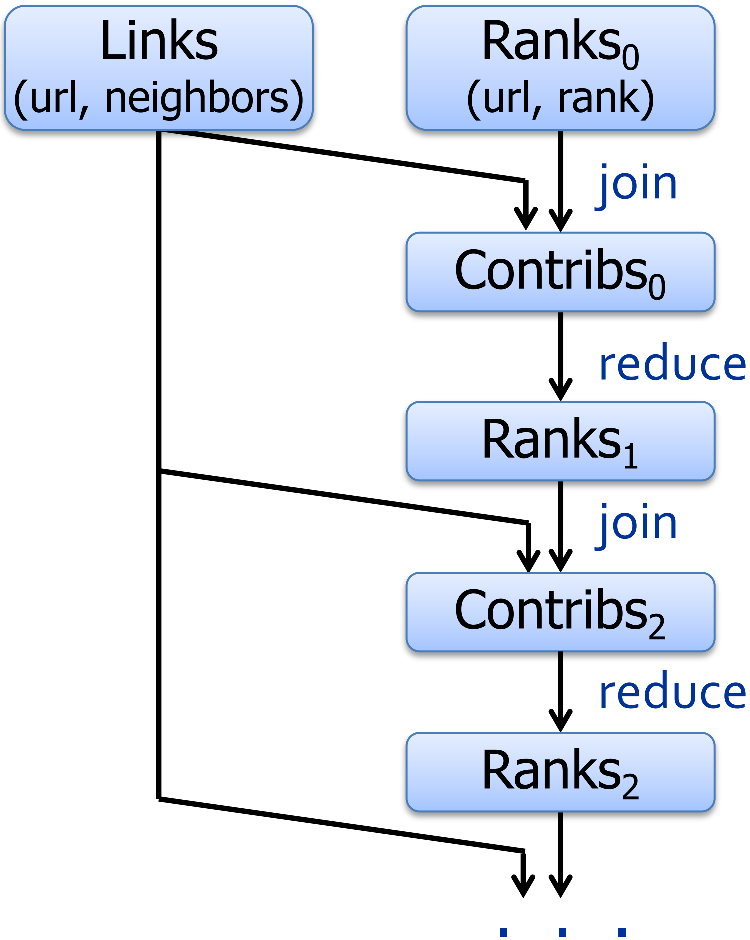
```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

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

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

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

Optimising 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