MapReduce:
Simplified Data Processing on Large Clusters

Jeff Dean, Sanjay Ghemawat
Google, Inc.

Motivation: Large Scale Data Processing

- Many tasks: Process lots of data to produce other data

Want to use hundreds or thousands of CPUs

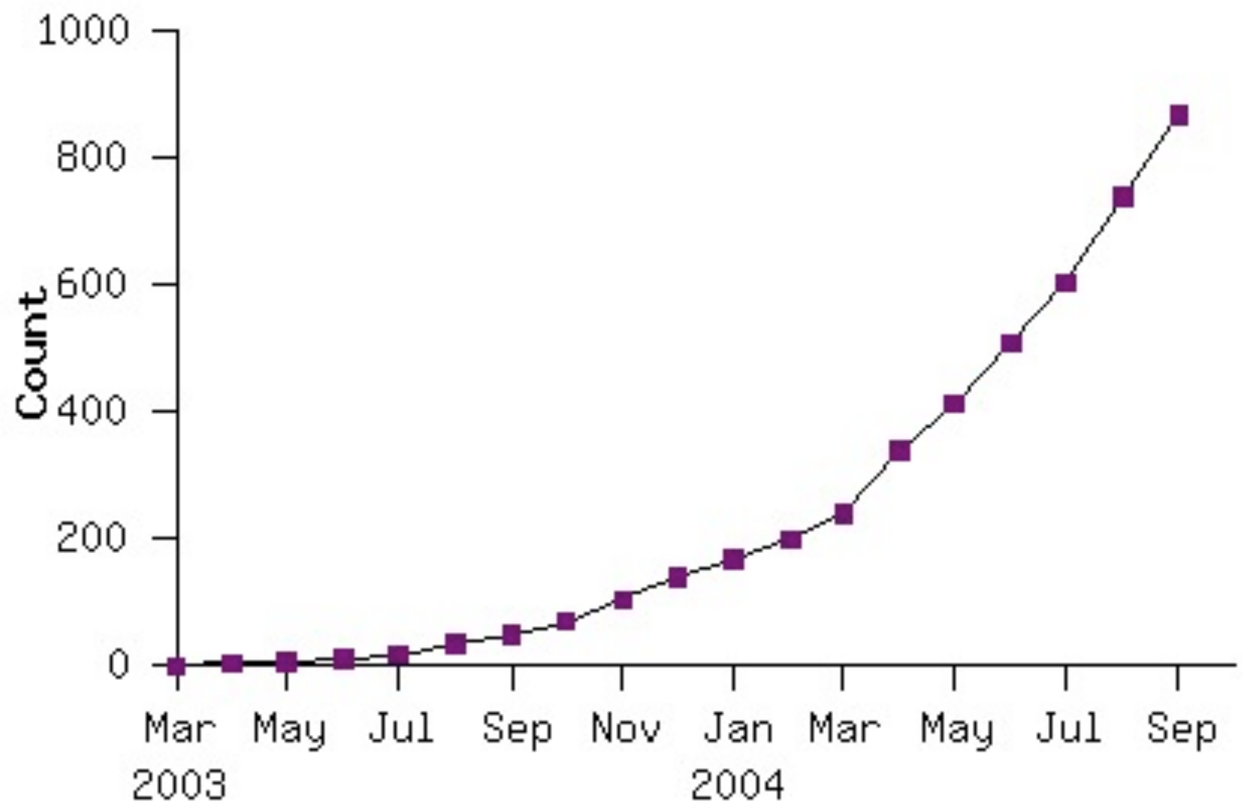
... but this needs to be easy

MapReduce provides:

- Automatic parallelization and distribution
- Fault-tolerance
- I/O scheduling
- Status and monitoring

Model is Widely Applicable

MapReduce Programs In Google Source Tree



Programming model

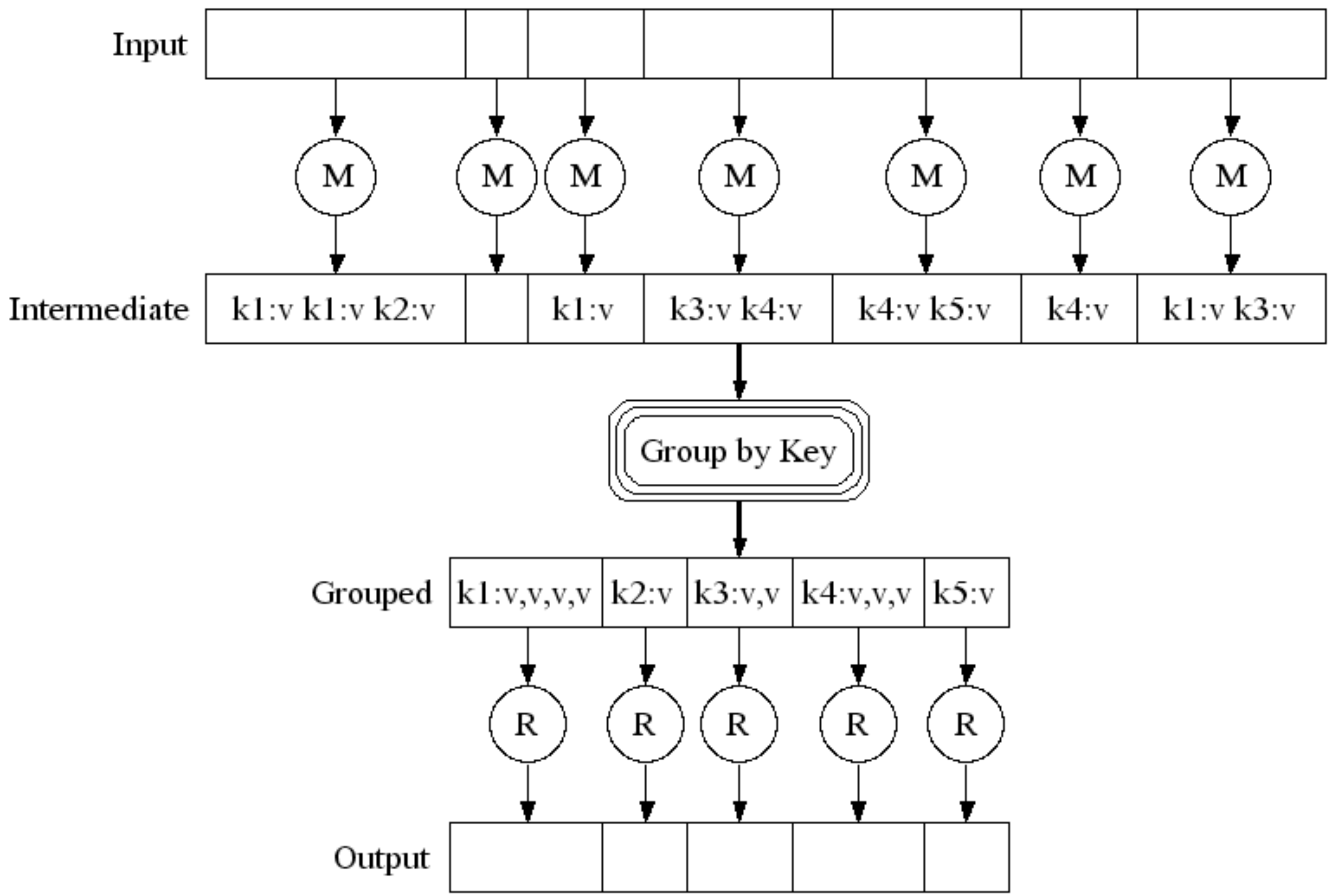
Input & Output: each a set of key/value pairs

Programmer specifies two functions:

```
map (in_key, in_value) ->
        list(out_key, intermediate_value)
reduce (out_key, list(intermediate_value)) ->
        list(out_value)
```

Inspired by similar primitives in LISP and other languages

Execution



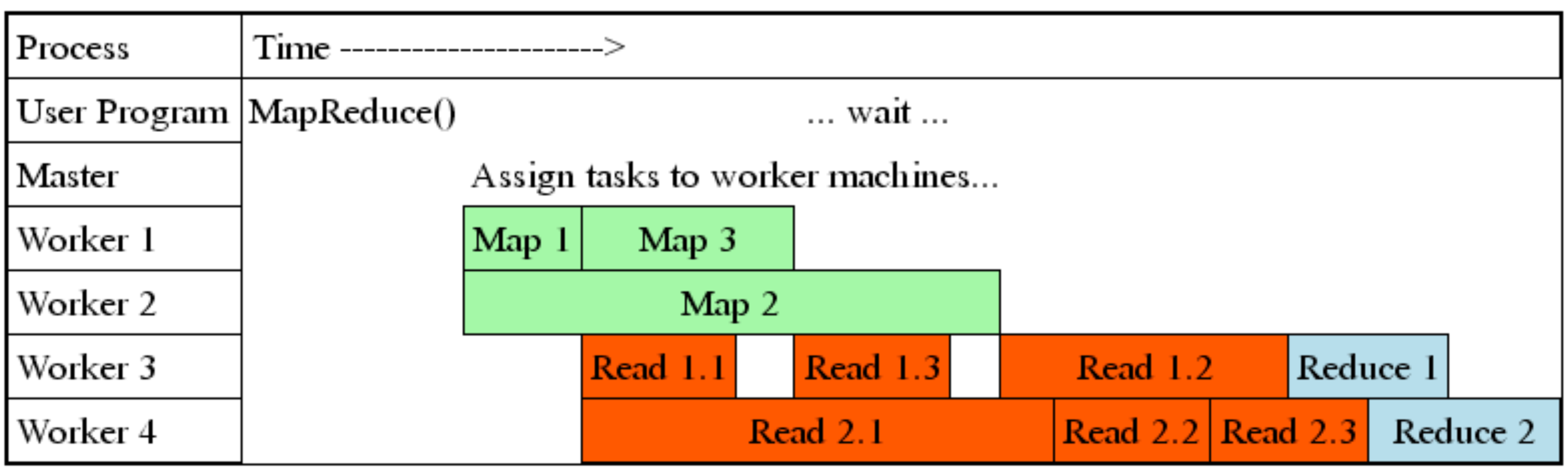
Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Task Granularity And Pipelining

Fine granularity tasks: many more map tasks than machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing

Often use 200,000 map/5000 reduce tasks w/ 2000 machines



Fault tolerance: Handled via re-execution

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master
- Master failure:
  - Could handle, but don't yet (master failure unlikely)

Robust: lost 1600 of 1800 machines once, but finished fine

Refinements

- **Redundant Execution**
- **Locality Optimization**
- **Skipping Bad Records**

- Sorting guarantees within each reduce partition
- Compression of intermediate data
- Combiner: useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Experience: Rewrite of Production Indexing System

Rewrote Google's production indexing system using MapReduce

- Set of 24 MapReduce operations
- New code is simpler, easier to understand
- MapReduce takes care of failures, slow machines
- Easy to make indexing faster by adding more machines

Resilient Distributed Datasets

A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica

UC Berkeley

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

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

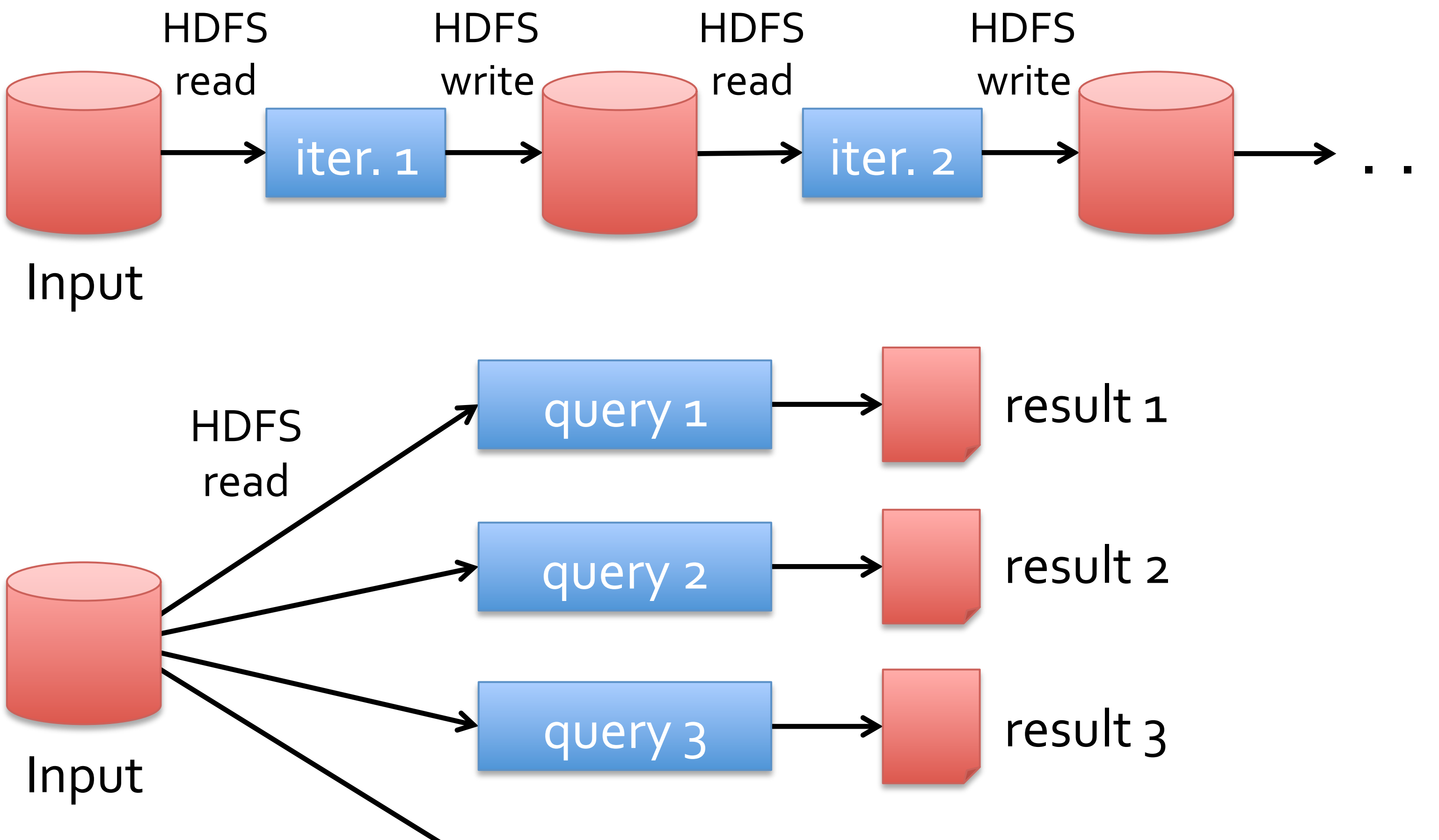
Motivation

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

Efficient primitives for **data sharing**

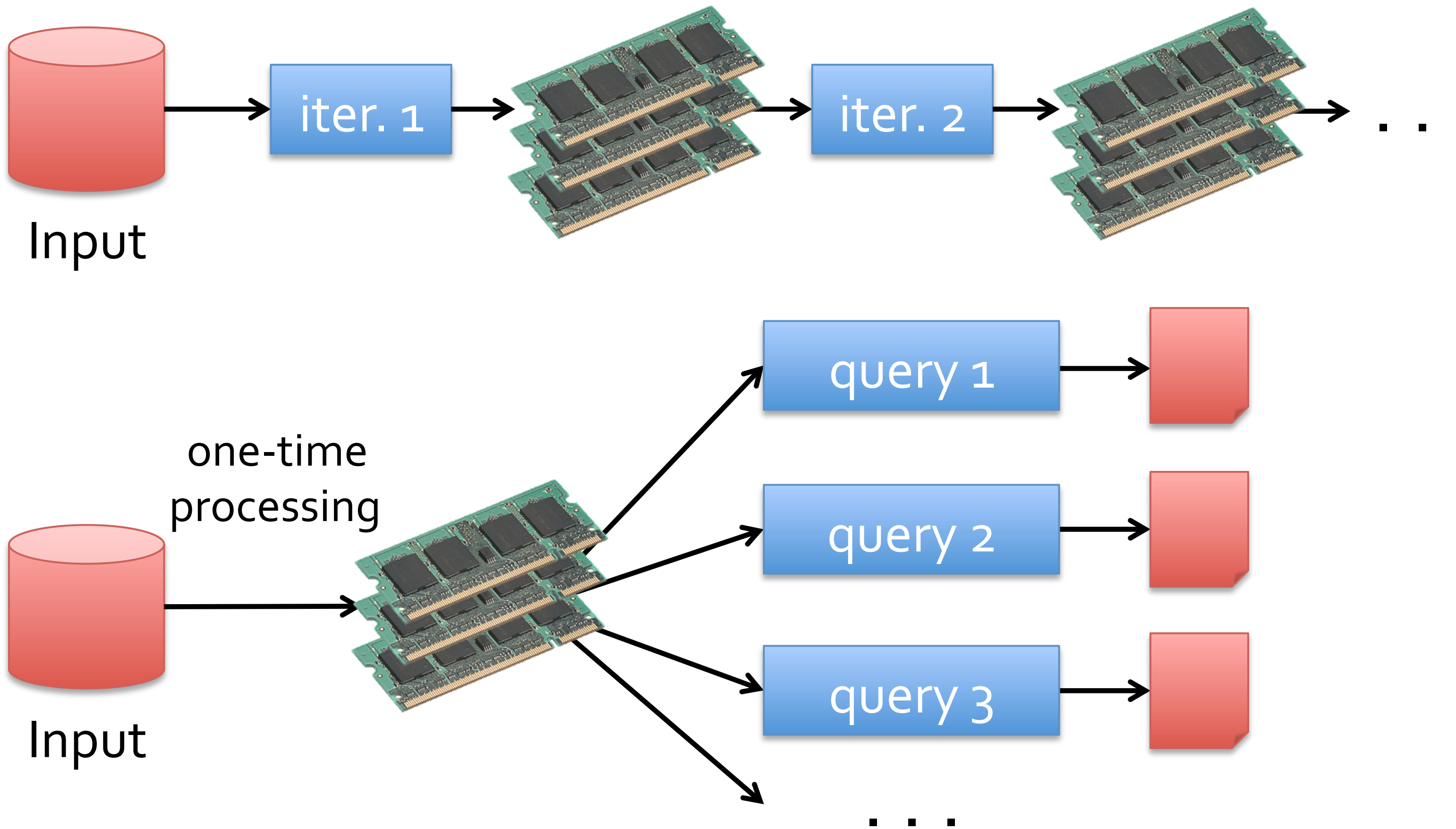
In MapReduce, the only way to share data across jobs is stable storage → slow!

Examples



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**?

Challenge

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

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, …)

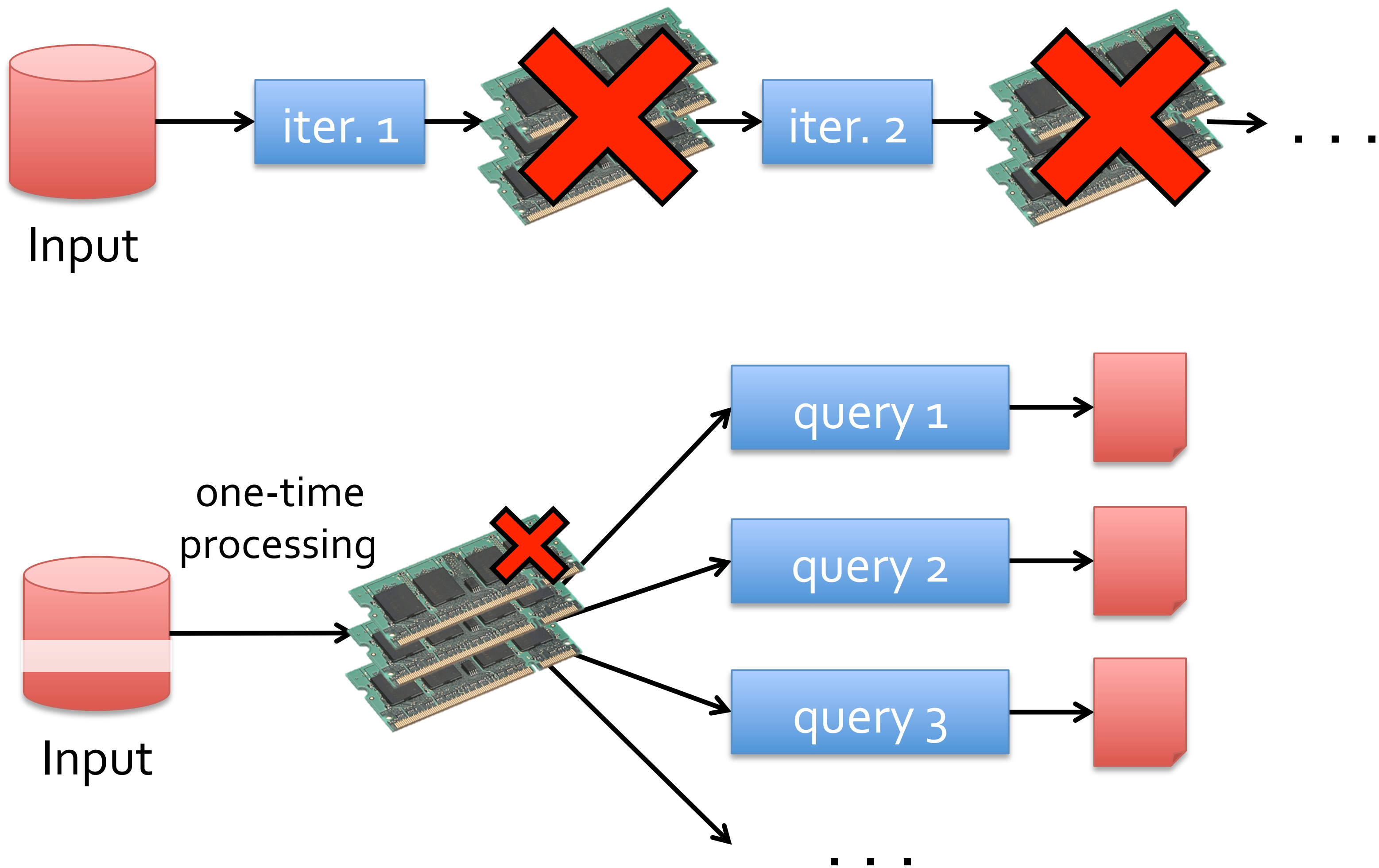
Efficient fault recovery using *lineage*

» Log one operation to apply to many elements

» Recompute lost partitions on failure

» No cost if nothing fails

RDD Recovery



Generality of RDDs

Despite their restrictions, RDDs can express surprisingly many parallel algorithms

» These naturally *apply the same operation to many items*

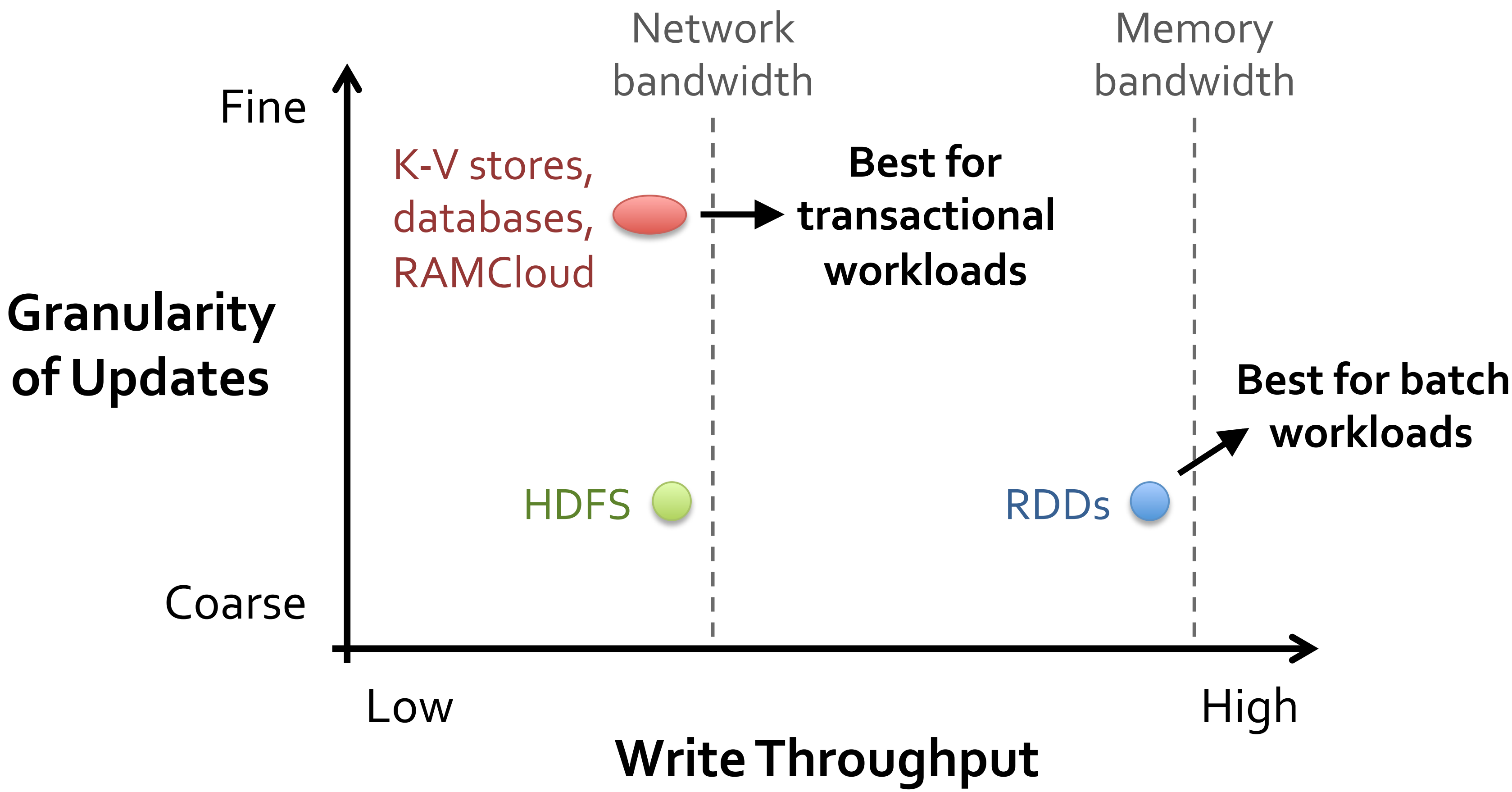
Unify many current programming models

» *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)

Spark Operations

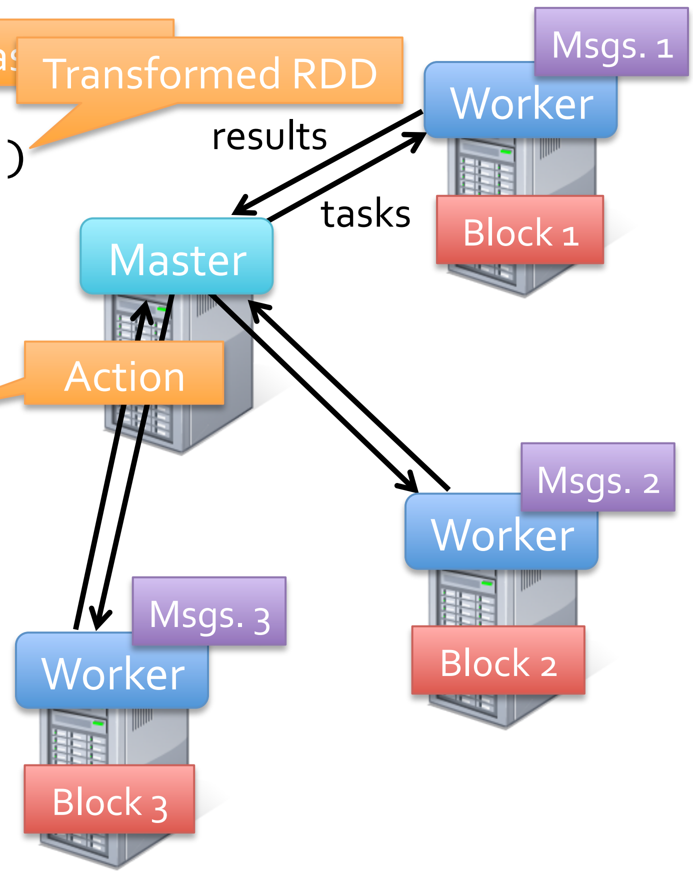
Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

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
```

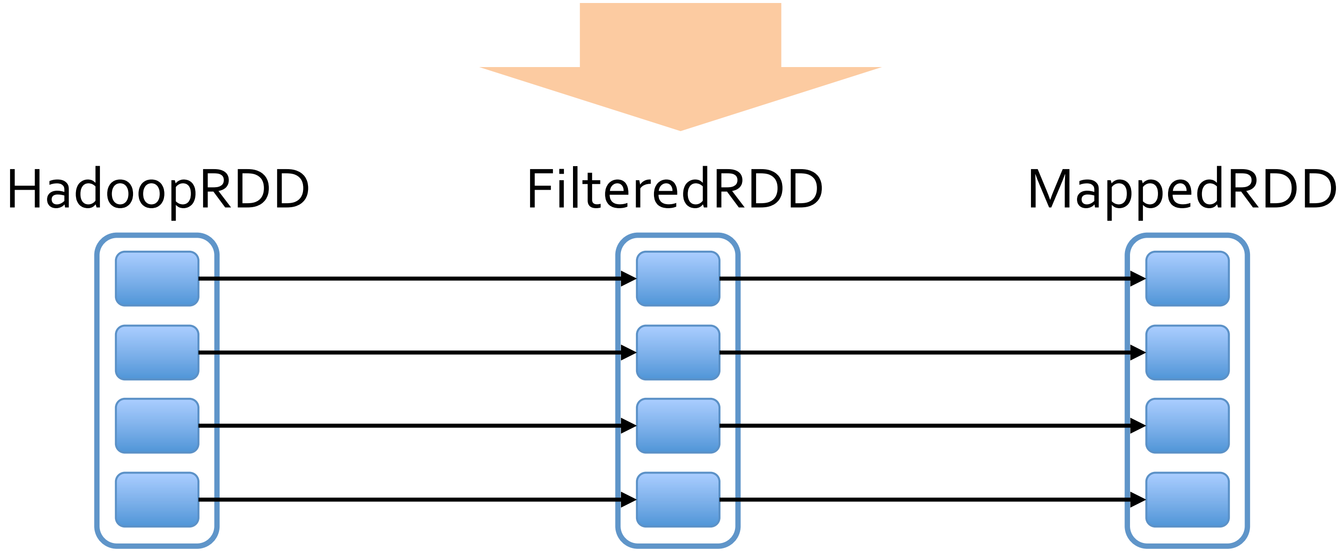


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

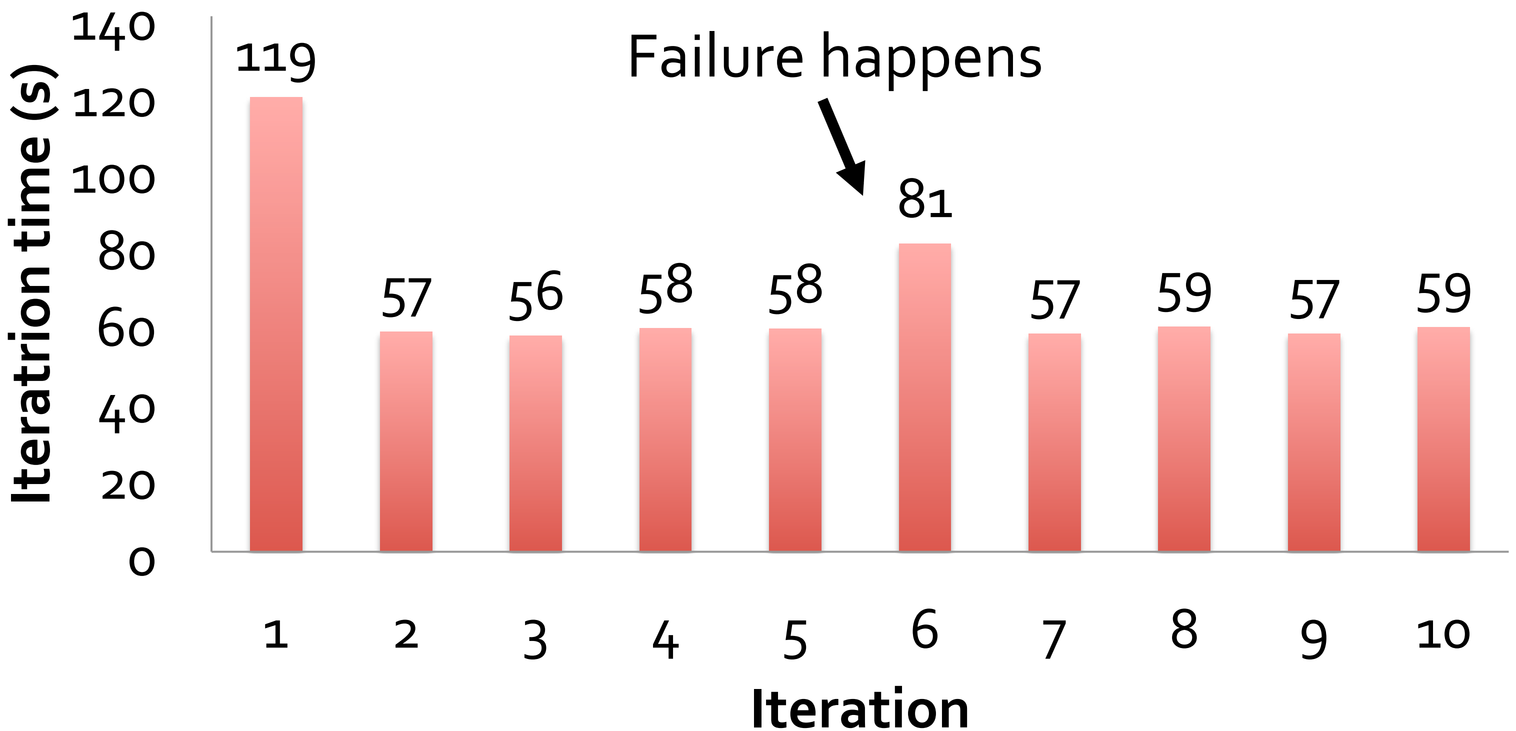
Fault Recovery

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

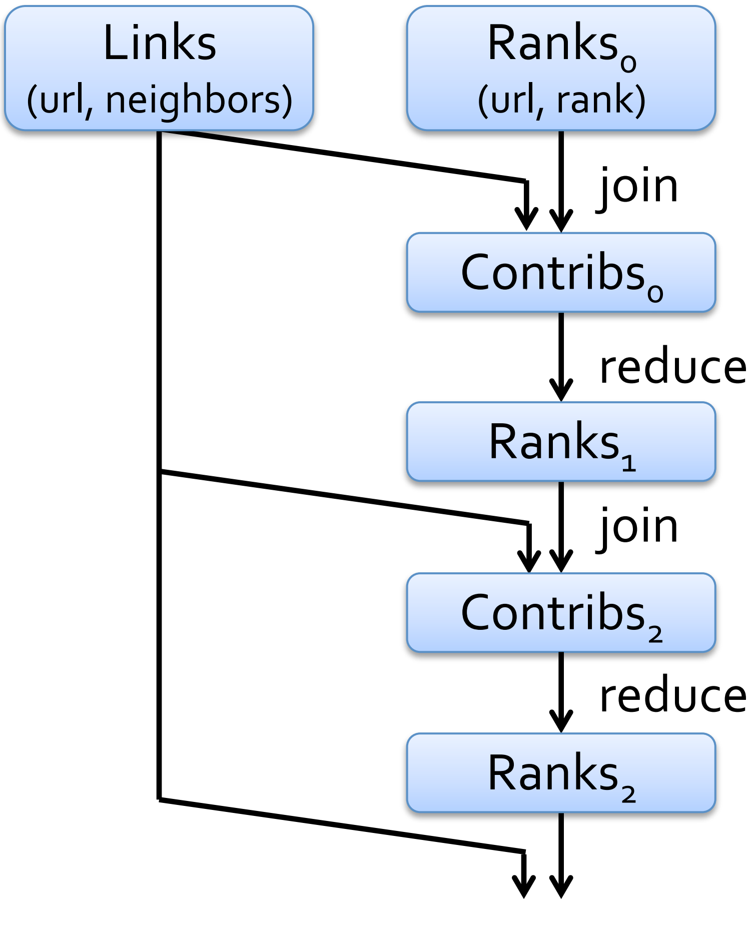
1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

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

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

Optimizing Placement



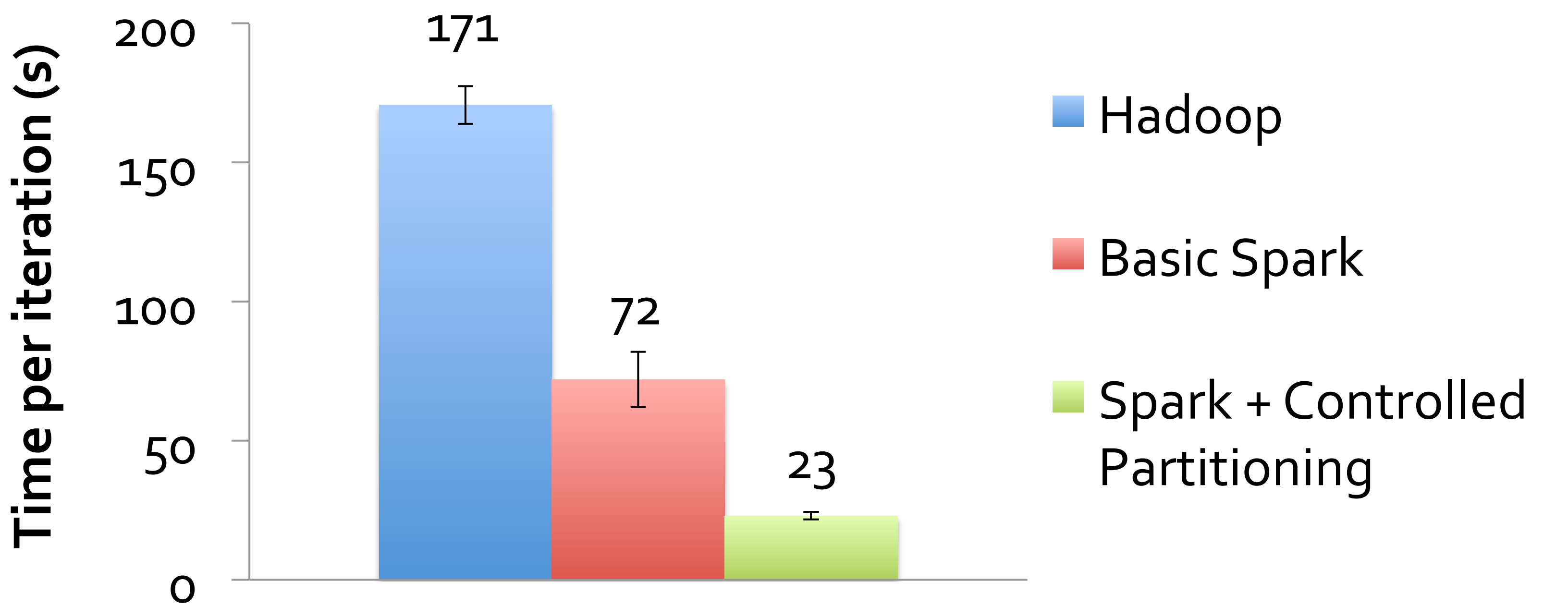
`links` & `ranks` repeatedly joined

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

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

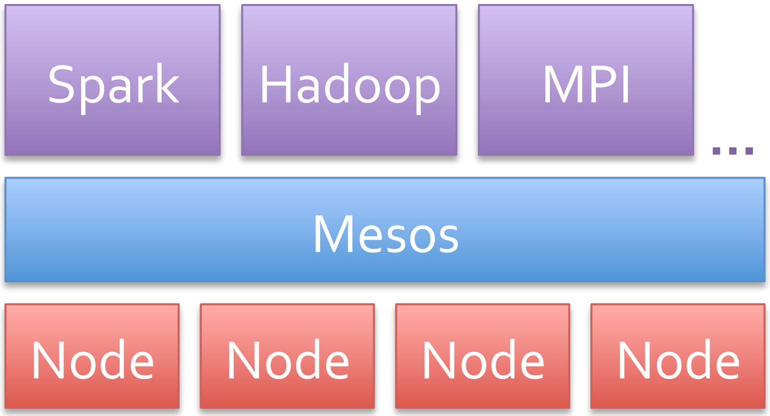
```
links = links.partitionBy(
        new URLPartitioner())
```

PageRank Performance



Implementation

Runs on Mesos [NSDI 11]
to share clusters w/ Hadoop



Can read from any Hadoop
input source (HDFS, S3, …)

No changes to Scala language or compiler

» Reflection + bytecode analysis to correctly ship code

www.spark-project.org

Programming Models Implemented on Spark

RDDs can express many existing parallel models

- » **MapReduce, DryadLINQ**
- » **Pregel** graph processing [200 LOC]
- » **Iterative MapReduce** [200 LOC]
- » **SQL**: Hive on Spark (Shark) [in progress]

All are based on coarse-grained operations

Enables apps to efficiently *intermix* these models

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

Try it out at **www.spark-project.org**