MapReduce
Spark

Some slides are adapted from those of Jeff Dean and Matei Zaharia

What have we learnt so far?

- Distributed storage systems
  - consistency semantics
  - protocols for fault tolerance
    - Paxos, Raft, Viewstamp
- Transactional Online processing
  - distributed transactions
- Today: Offline batch processing

Why distributed computations?

- How long to sort 1 TB on one computer?
  - One computer can read ~50MB from disk
  - Takes 5.5 hours!
- Google indexes 60 trillion web pages
  - 60 * 10^12 pages * 10KB/page = 600 PB
- Large Hadron Collider is expected to produce 15 PB every year!

Solution: use many nodes!

- Data Centers at Amazon/Facebook/Google
  - Hundreds of thousands of PCs connected by high speed LANs
- Cloud computing
  - Any programmer can rent nodes in Data Centers for cheap
- The promise:
  - 1000 nodes ➔ 1000X speedup

Distributed computations are difficult to program

- Sending data to/from nodes
- Coordinating among nodes
- Recovering from node failure
- Optimizing for locality
- Debugging

Same for
all problems

The world before MapReduce comes along

- Dominant philosophy in systems research
  - programming many machines should be "the same" that of a single multi-core machine
  - *distributed shared memory*
  - *automatic parallelization of existing programs*
- MPI for high performance computing
  - a collection of communication/synchronization primitives to simplify message passing
- No systems handle failures

MapReduce

- A programming model for large-scale computations
  - Process large amounts of input, produce output
  - No side-effects or persistent state (unlike file system)

- MapReduce is implemented as a runtime library:
  - automatic parallelization
  - load balancing
  - locality optimization
  - handling of machine failures

MapReduce design

- Input data is partitioned into M splits
- Map: extract information on each split
  - Each Map produces R partitions
- Shuffle and sort
  - Bring M partitions to the same reducer
- Reduce: aggregate, summarize, filter or transform
- Output is in R result files

More specifically…

- Programmer specifies two methods:
  - map(k, v) → <k', v'>*
  - reduce(k', <v'>*) → <k', v'>*
- All v' with same k' are reduced together

- Usually also specify:
  - partition(k’, total partitions) -> partition for k’
    - often a simple hash of the key
    - allows reduce operations for different k’ to be parallelized

Example: Count word frequencies in web pages

- Input is files with one doc per record
- Map parses documents into words
  - key = document URL
  - value = document contents
- Output of map:

“doc1”, “to be or not to be” →

“to”, “1”
“be”, “1”
“or”, “1”
…

Example: word frequencies

- Reduce: computes sum for a key

key = “be”
values = “1”, “1”
→ “2”

key = “not”
values = “1”
→ “1”

key = “or”
values = “1”
→ “1”

key = “to”
values = “1”, “1”
→ “2”

- Output of reduce saved

“be”, “2”
“not”, “1”
“or”, “1”
“to”, “2”

Example: Pseudo-code

```
Map(String input_key, String input_value):
  //input_key: document name
  //input_value: document contents
  for each word w in input_values:
    EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
  //key: a word, same for input and output
  //intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce is widely applicable

- Distributed grep
- Document clustering
- Web link graph reversal
- Detecting duplicate web pages
- …

MapReduce implementation

- Input data is partitioned into M splits
- Map: extract information on each split
  - Each Map produces R partitions
- Shuffle and sort
  - Bring M partitions to the same reducer
- Reduce: aggregate, summarize, filter or transform
- Output is in R result files, stored in a replicated, distributed file system (GFS).

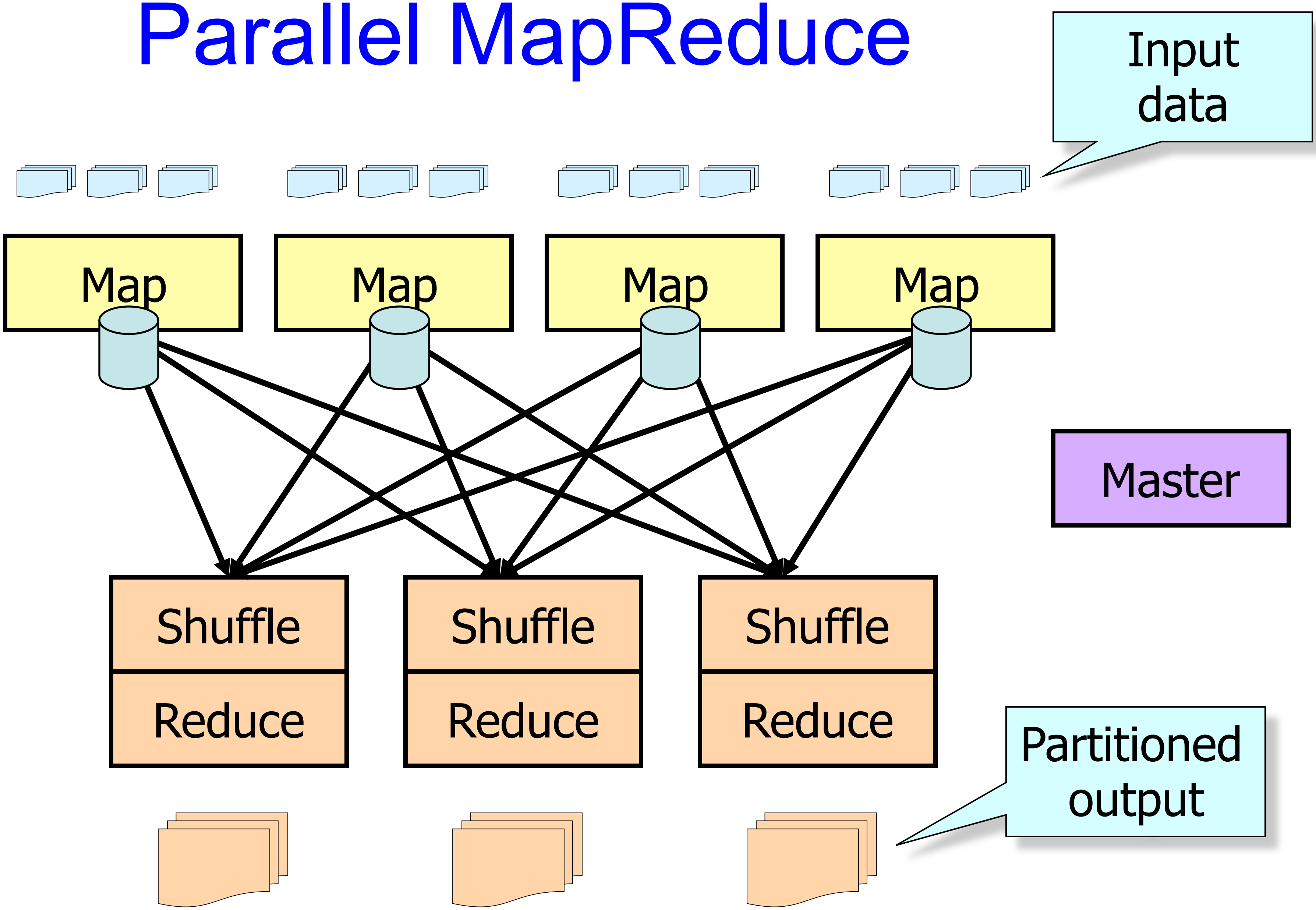
MapReduce scheduling

- One master, many workers
  - Input data split into *M* map tasks
  - *R* reduce tasks
  - Tasks are assigned to workers dynamically

MapReduce scheduling

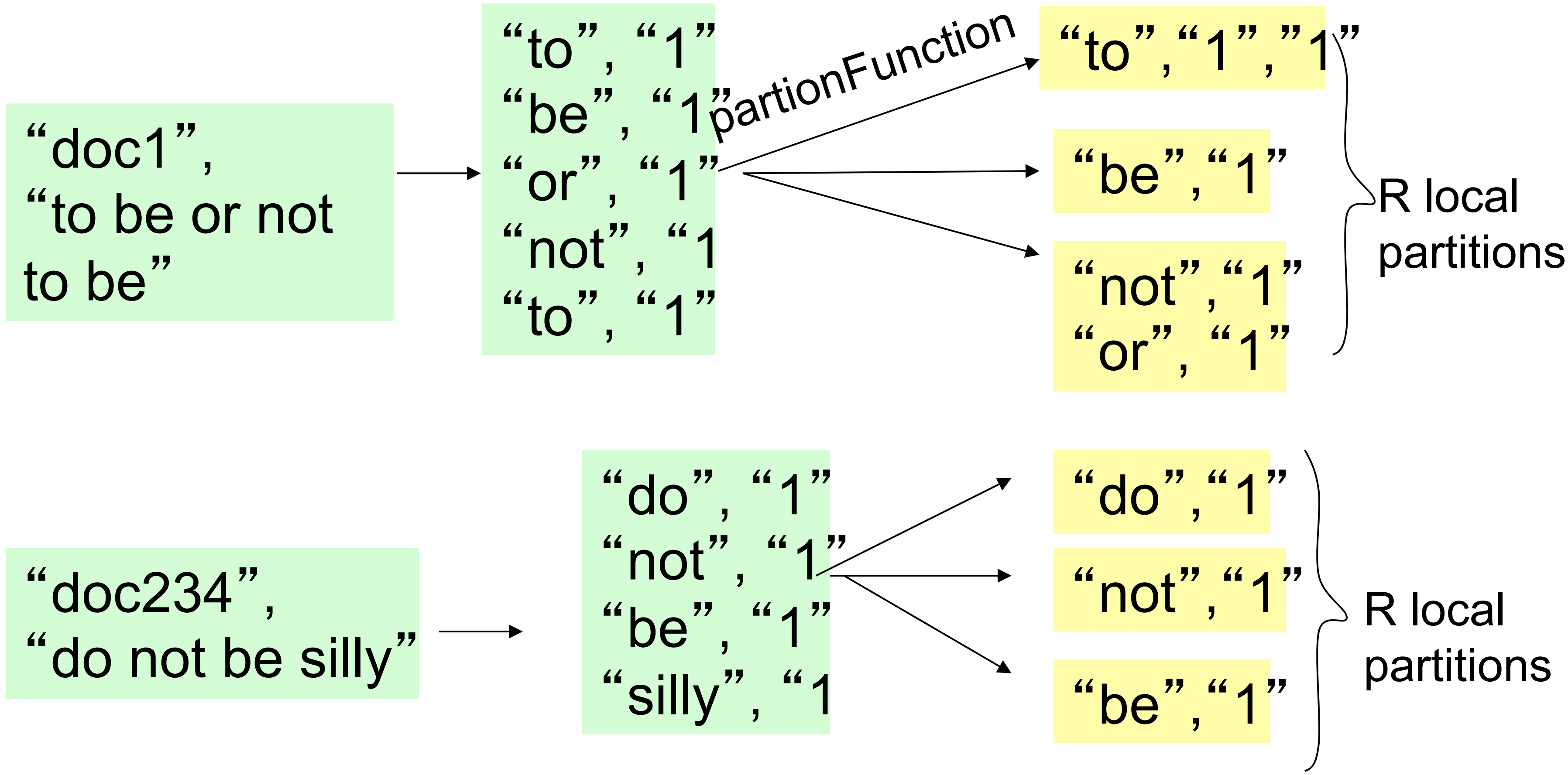
- Master assigns a map task to a free worker
  - Prefers "close-by" workers when assigning task
  - Worker reads task input (often from local disk!)
  - Worker produces R **local files** containing intermediate k/v pairs
- Master assigns a reduce task to a free worker
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's *Reduce* op to produce the output

Parallel MapReduce



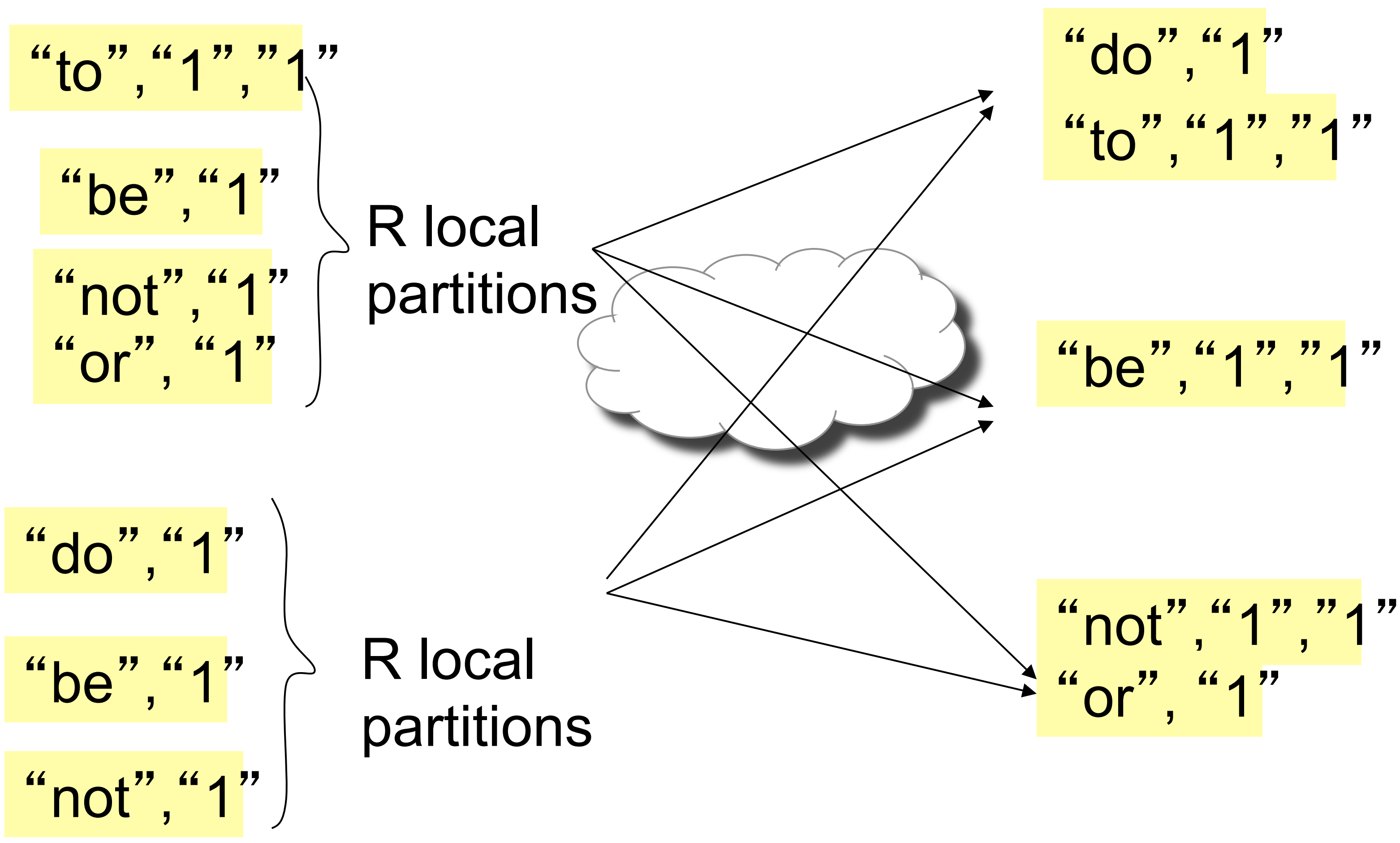
WordCount Internals

- Input data is split into M map jobs
- Each map job generates in R local partitions



WordCount Internals

- Shuffle brings same partitions to same reducer



WordCount Internals

- Reduce aggregates sorted key values pairs

“do”,“1”
“to”,“1”,”1” → “do”,“1”
“to”, “2”

“be”,“1”,”1” → “be”,“2”

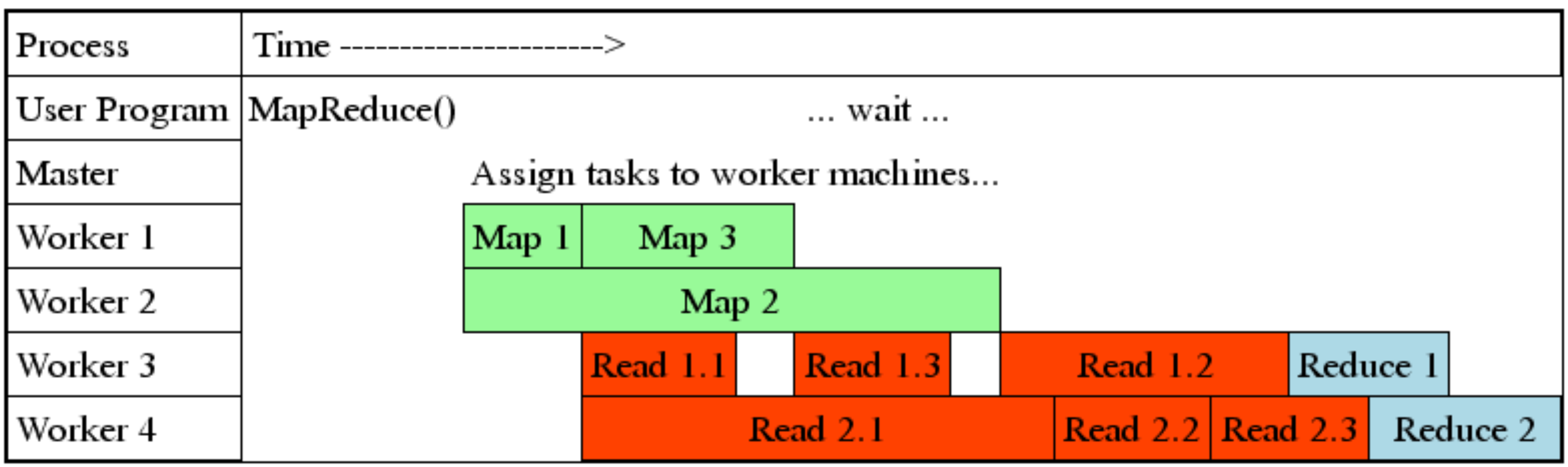
“not”,“1”,”1”
“or”, “1” → “not”,“2”
“or”, “1”

The importance of partition function

- partition(k', total partitions) -> partition for k'
  - e.g. hash(k') % R
- What is the partition function for sort?

Load Balance and Pipelining

- Fine granularity tasks: many more map tasks than machines
  – Minimizes time for fault recovery
  – Can pipeline shuffling with map execution



Fault tolerance

- What are the potential failure cases?
  - Lost packets
  - Temporary network disconnect
  - Servers crash and rebooted
  - Servers fail permanently (disk wipe)

Fault tolerance via re-execution

On master failure:

- Lab3 does not require handing master failure

On worker failure:

- Re-execute in-progress map tasks
- Re-execute in-progress reduce tasks
- Task completion committed through master

Is it possible a task is executed twice?

How to handle stragglers

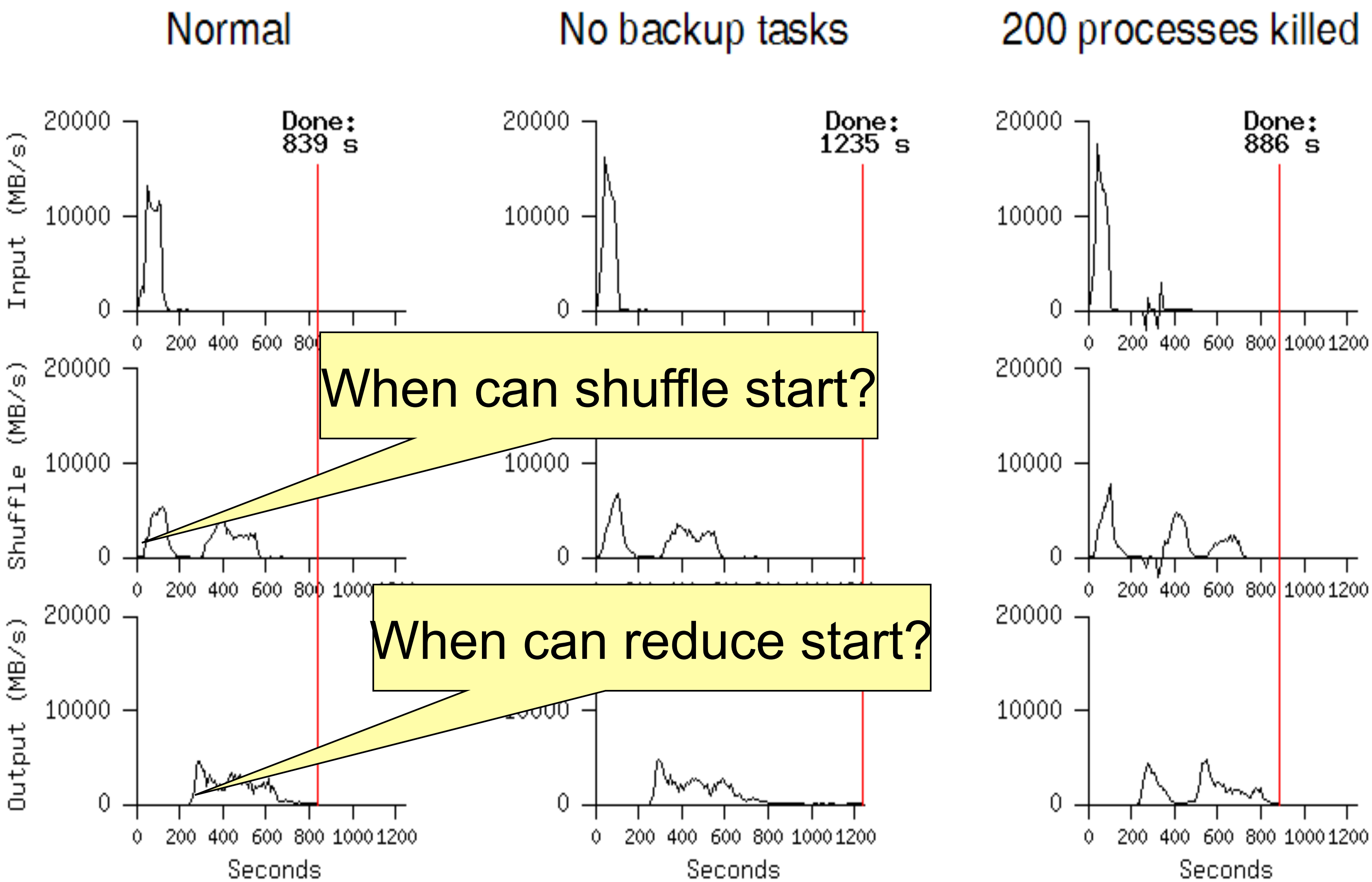
- Ideal speedup on N Machines?
- Why no ideal speedup in practice?
- Straggler: Slow workers drastically increase completion time
  – Other jobs consuming resources on machine
  – Bad disks with soft errors transfer data very slowly
  – Weird things: processor caches disabled (!!)
  – An unusually large reduce partition

- Solution: Near end of phase, spawn backup copies of tasks
  – Whichever one finishes first "wins"

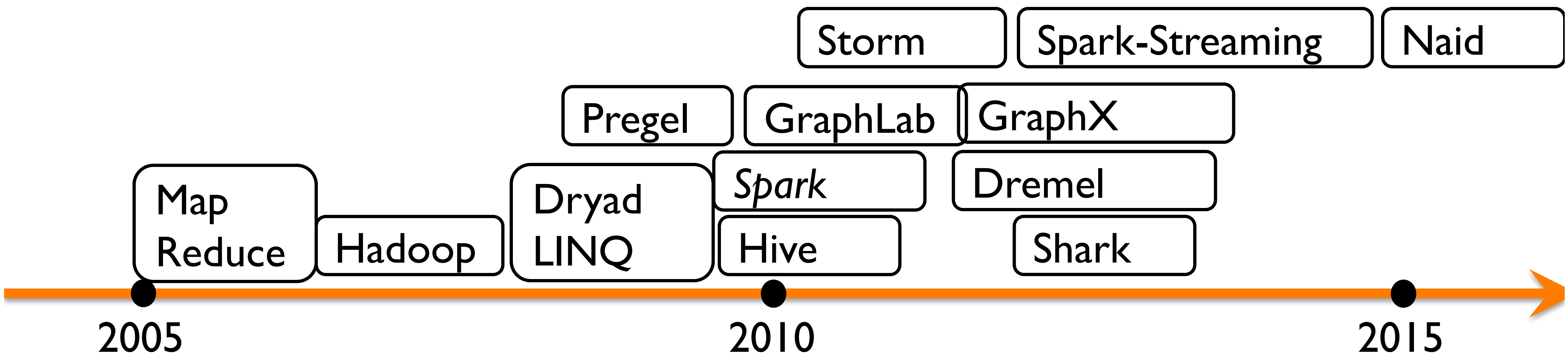
MapReduce Sort Performance

- 1TB (100-byte record) data to be sorted
- 1700 machines
- M=15000 R=4000

MapReduce Sort Performance



Big Data Computation



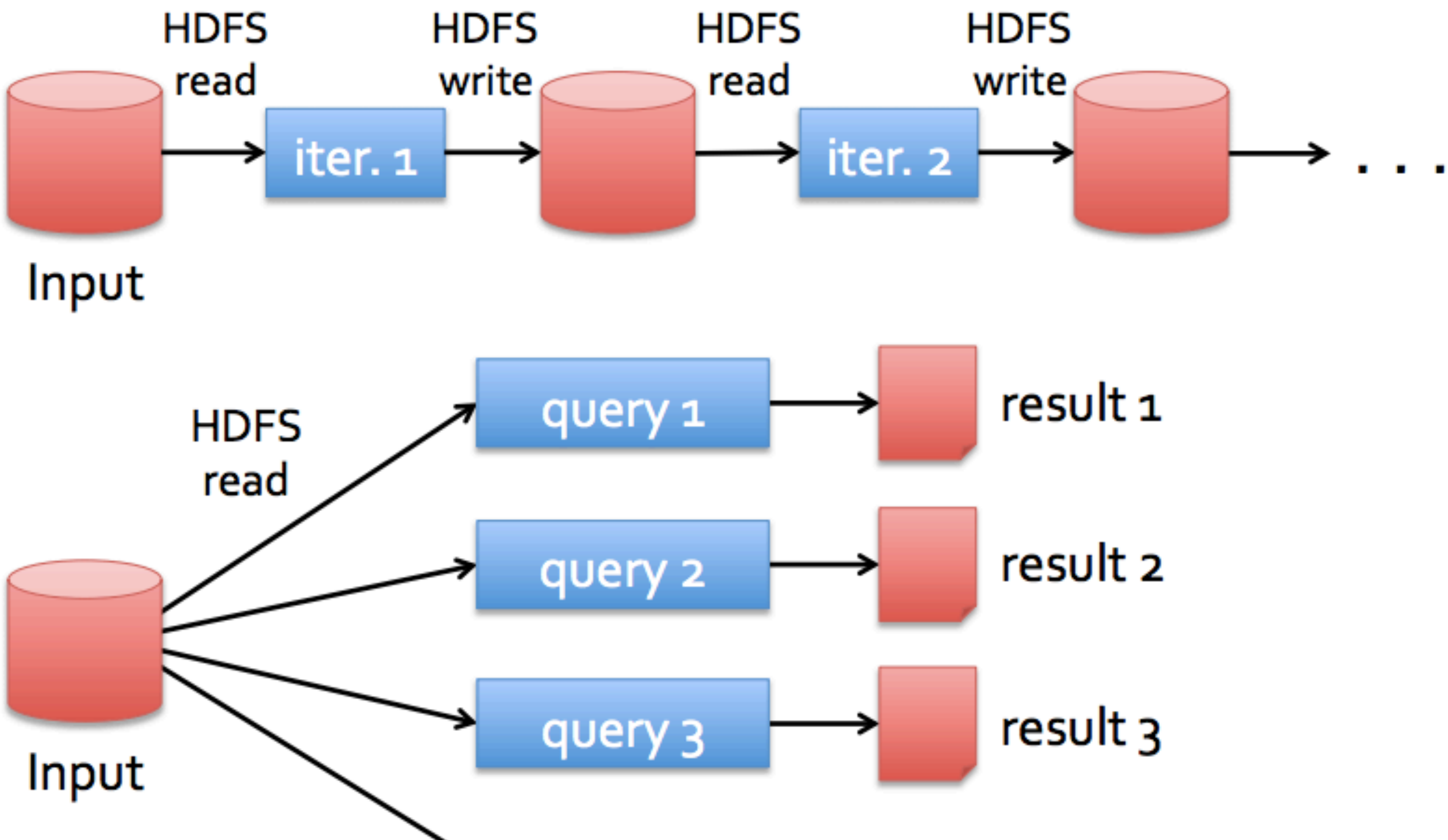
Spark's motivation

- More Complex Analytics
  - multi-stage processing
    - iterative machine learning
    - iterative graph processing
- Better performance
  - lots of application's dataset can fit in the aggregate memory of many machines

What MapReduce lacks

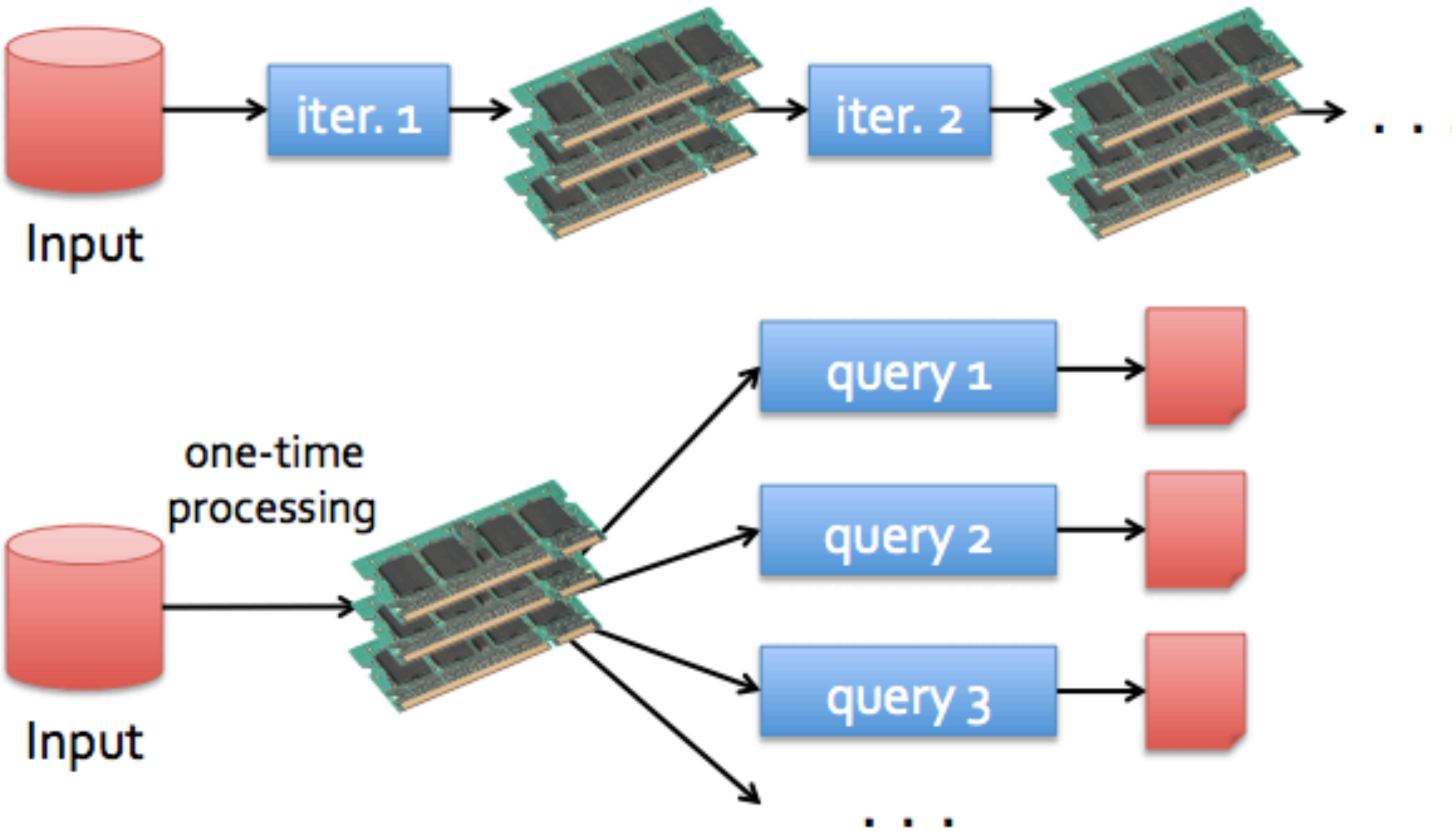
- Efficient data sharing primitive for multi-staging processing
  - output of the previous stage is stored on GFS
  - input of the current stage is read from GFS

Multi-stage MapReduce job



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

Spark's goal

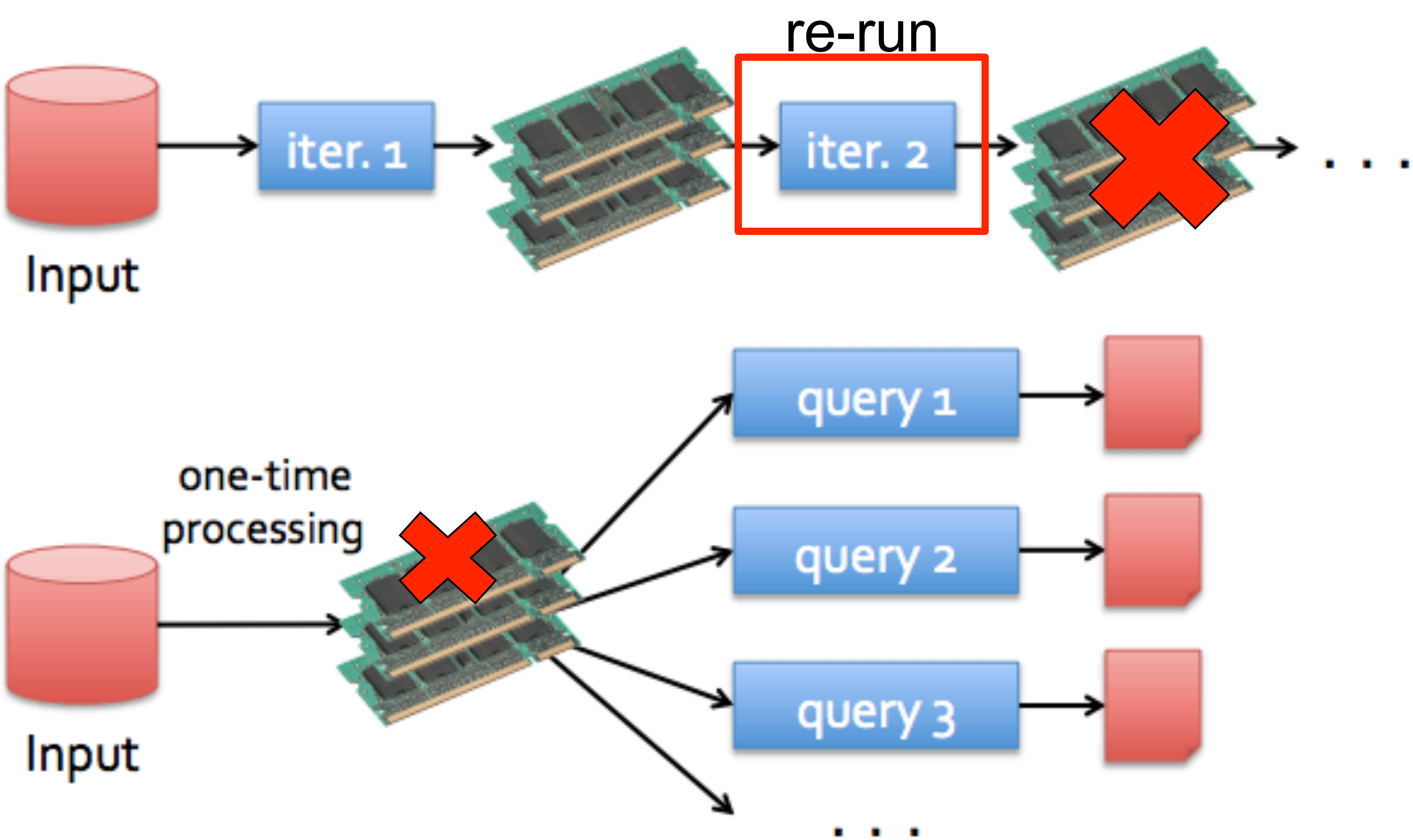


10-100× faster than network/disk, but how to get FT?

Spark's solution

- 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

RDD recovery



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

Spark API

- DryadLINQ-like API in Scala language

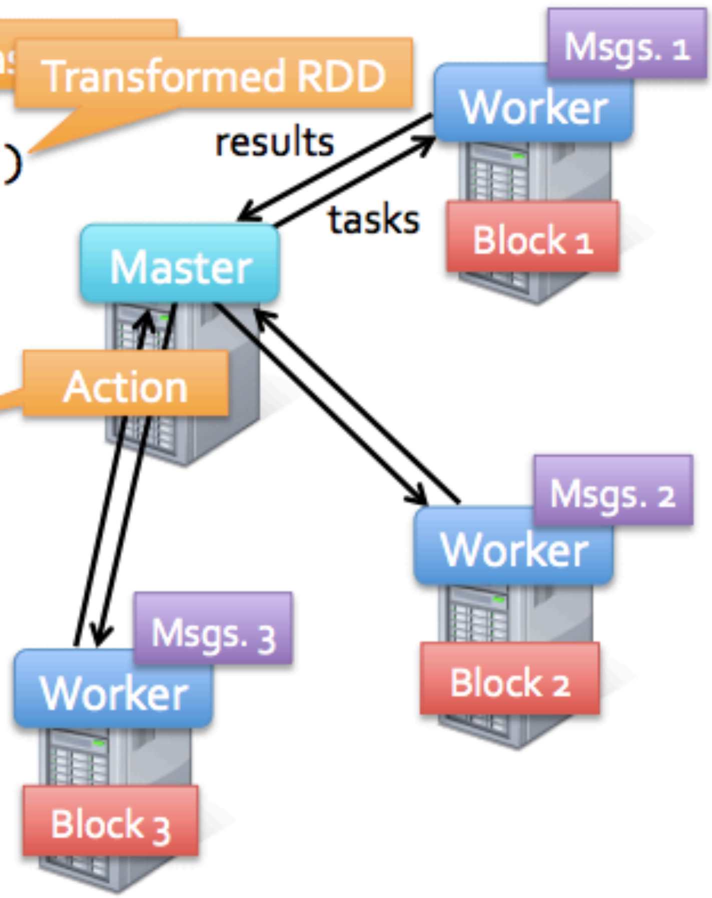
$\textit{map}(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
$\textit{filter}(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
$\textit{flatMap}(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow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

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
```

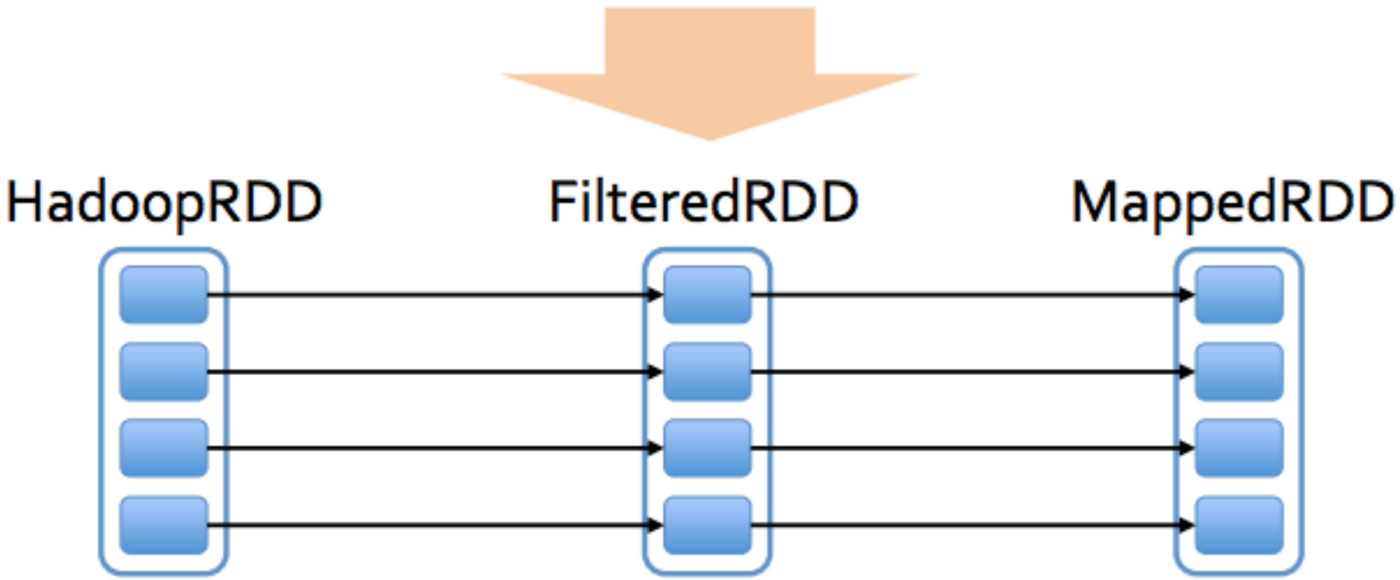


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

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))
```



Another example: PageRank

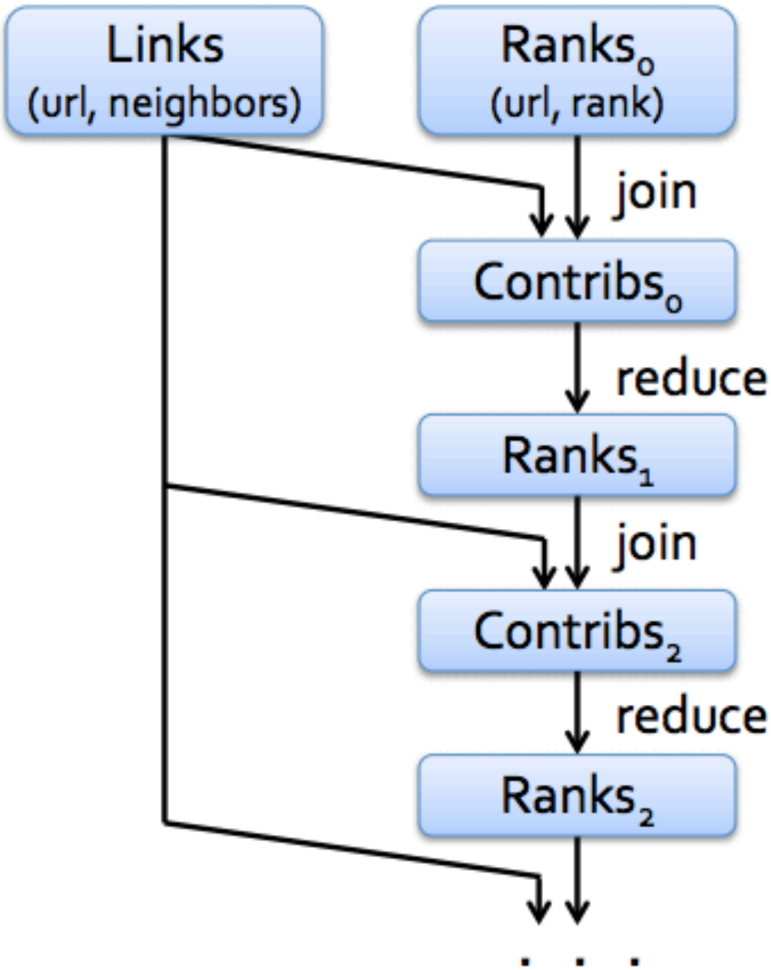
1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

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

```
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



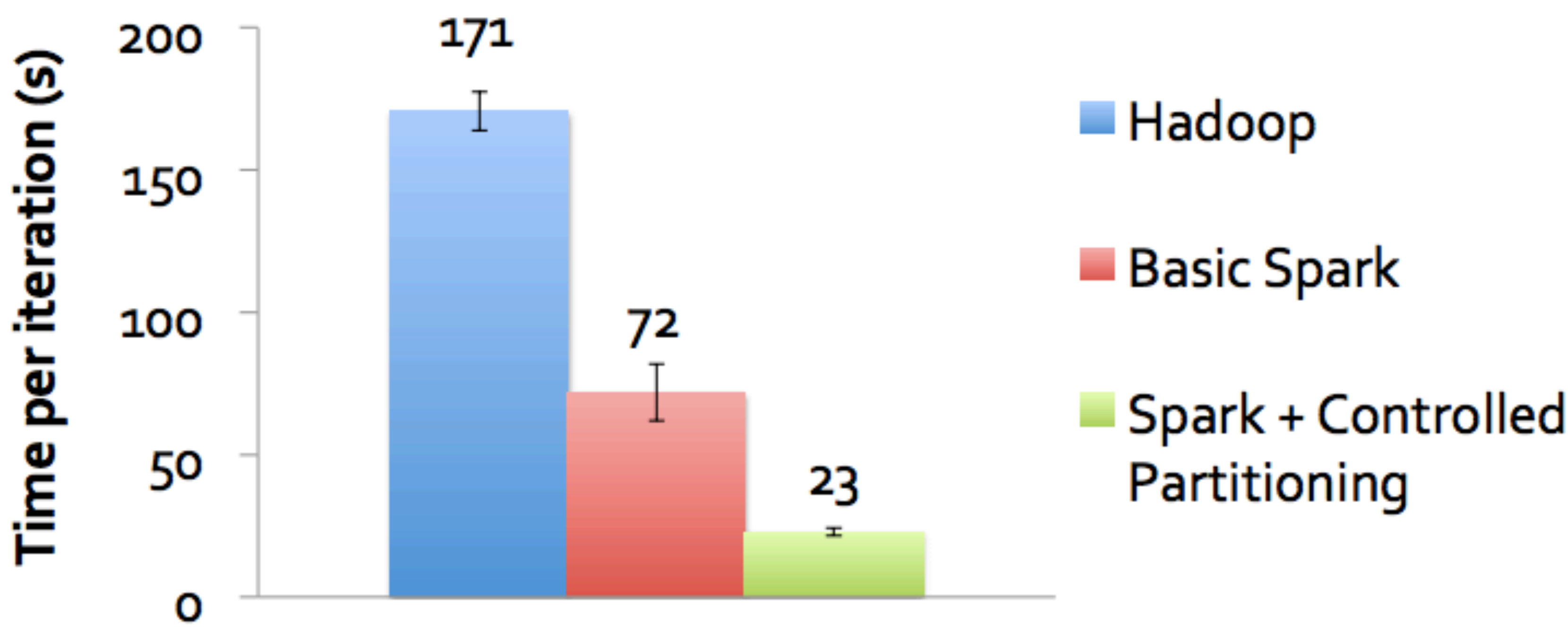
`links` & `ranks` repeatedly joined

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 Optimization



Summary

- MapReduce
  - The interface Map + Reduce let programmers write applications that can be automatically parallelized/distributed
  - Re-execution to handle failure / stragglers
- Spark
  - Enable multi-stage MR jobs to pass data via memory
  - RDD handles fault-tolerance at a coarse-granularity by tracking lineage.