Introduction to Parallel & Distributed Computing

Parallel Programming with Spark

Lecture 16, Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

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

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

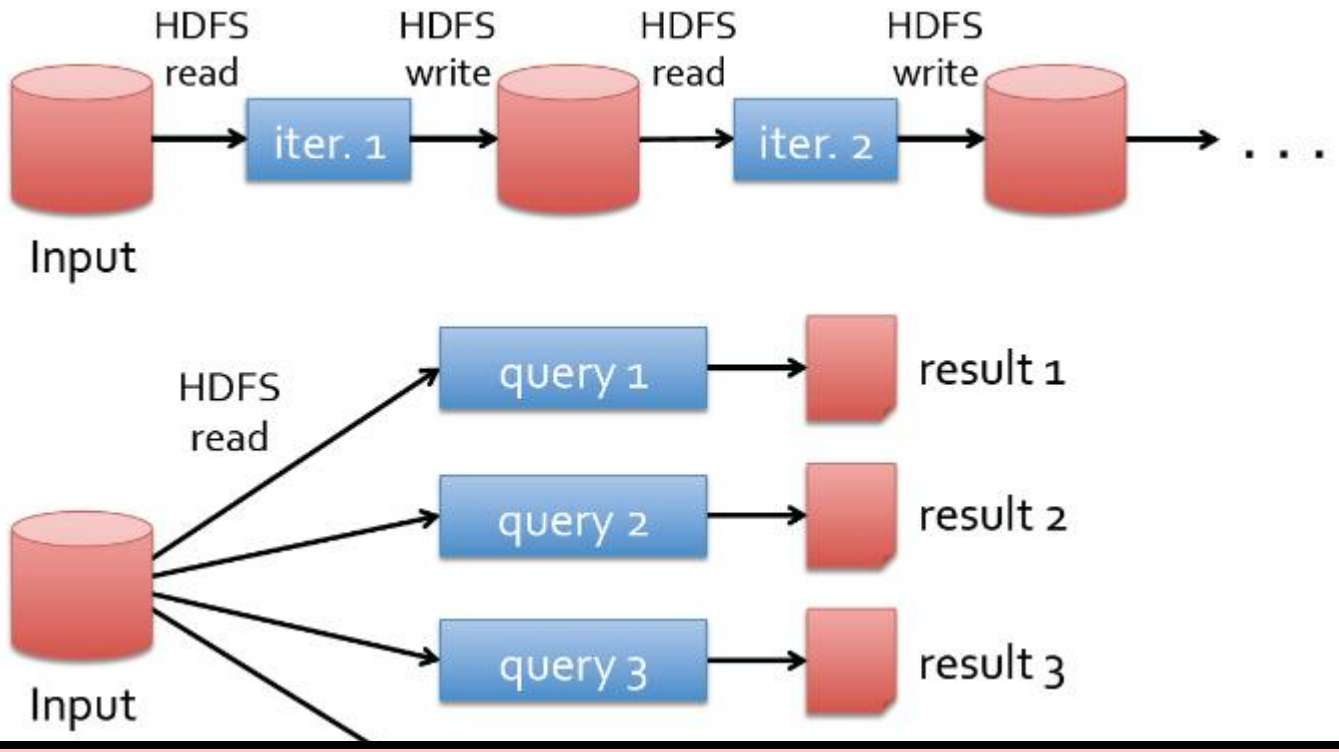
Matei Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" NSDI 2012. (Best Paper)

Spark: Motivation

- **Complex apps and interactive queries both need one thing that MapReduce lacks:**
  - Efficient primitives for data sharing

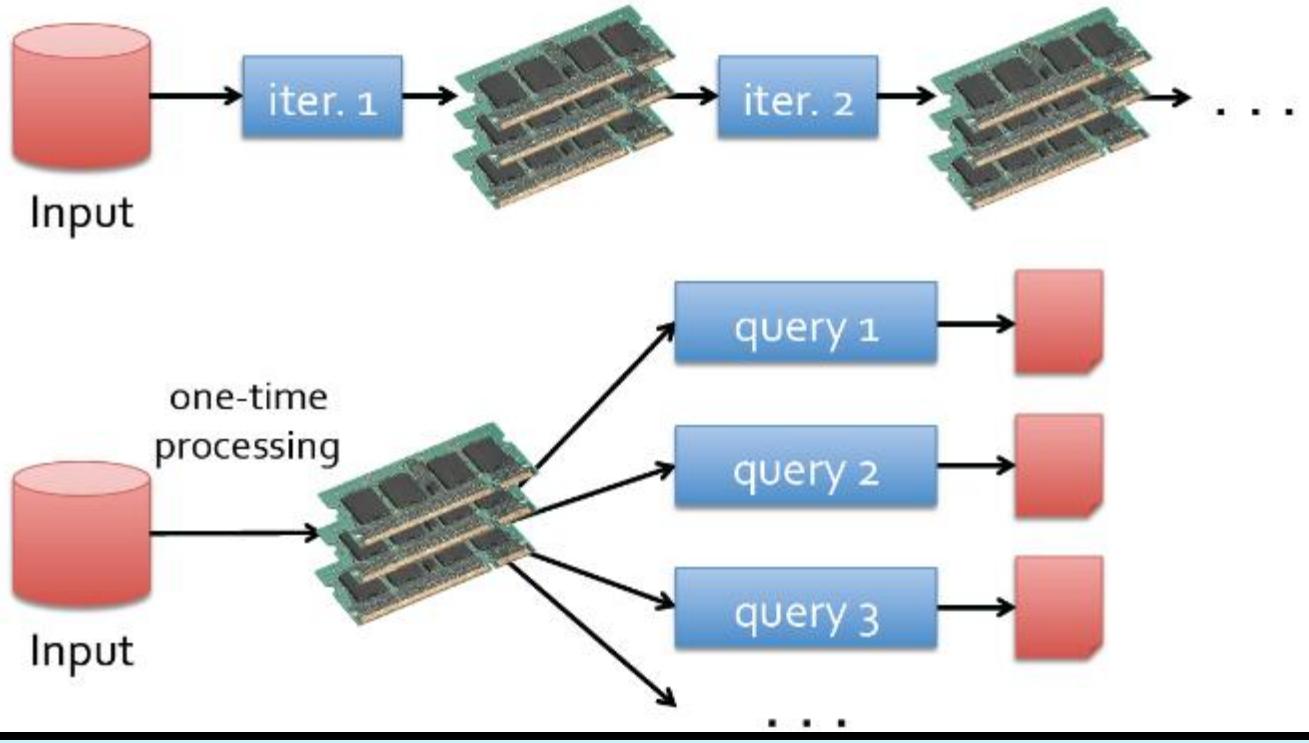
In MapReduce, the only way to share data across jobs is stable storage → slow!

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?ç

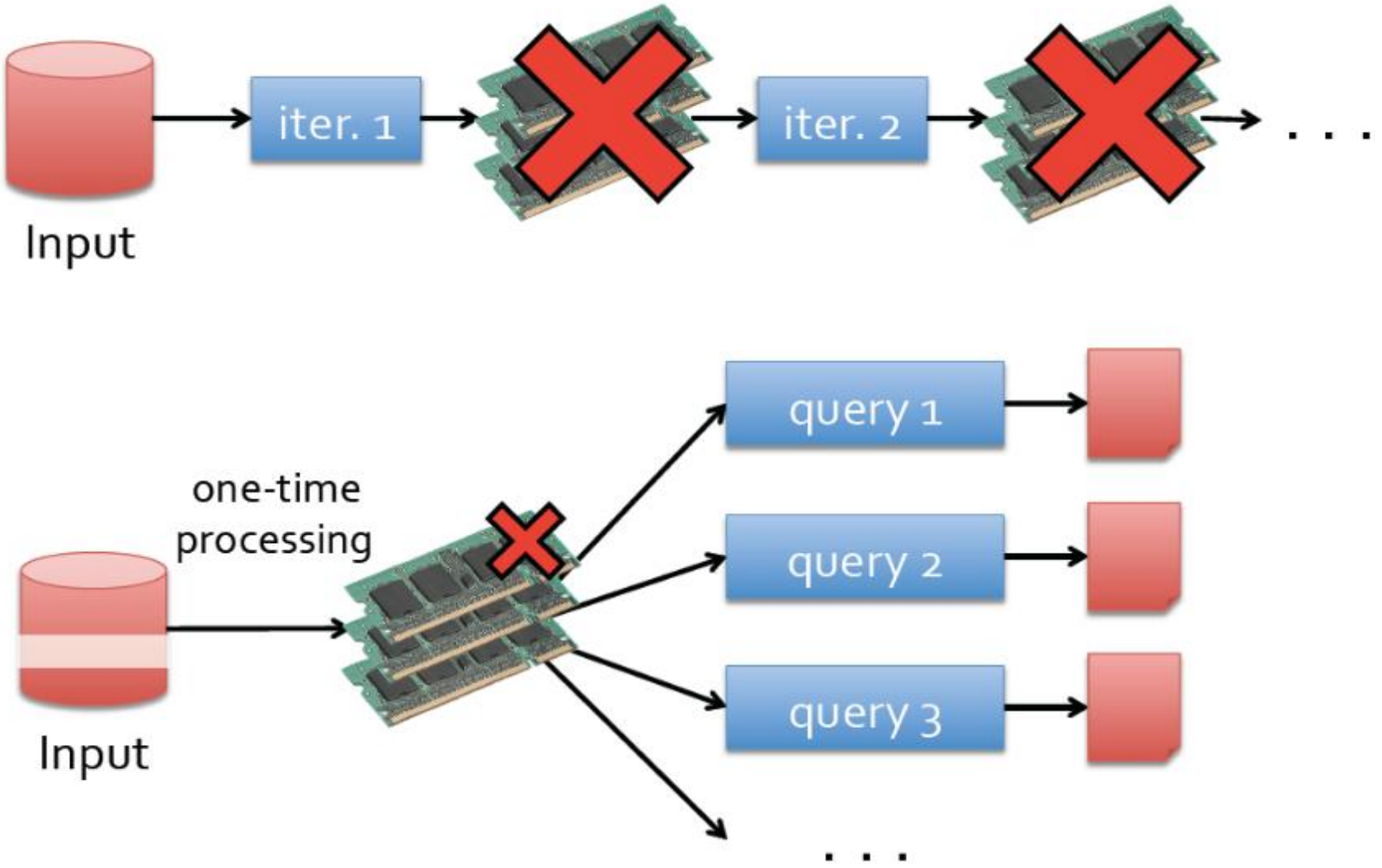
Challenge

- **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
  - 10-100× slower than memory write

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

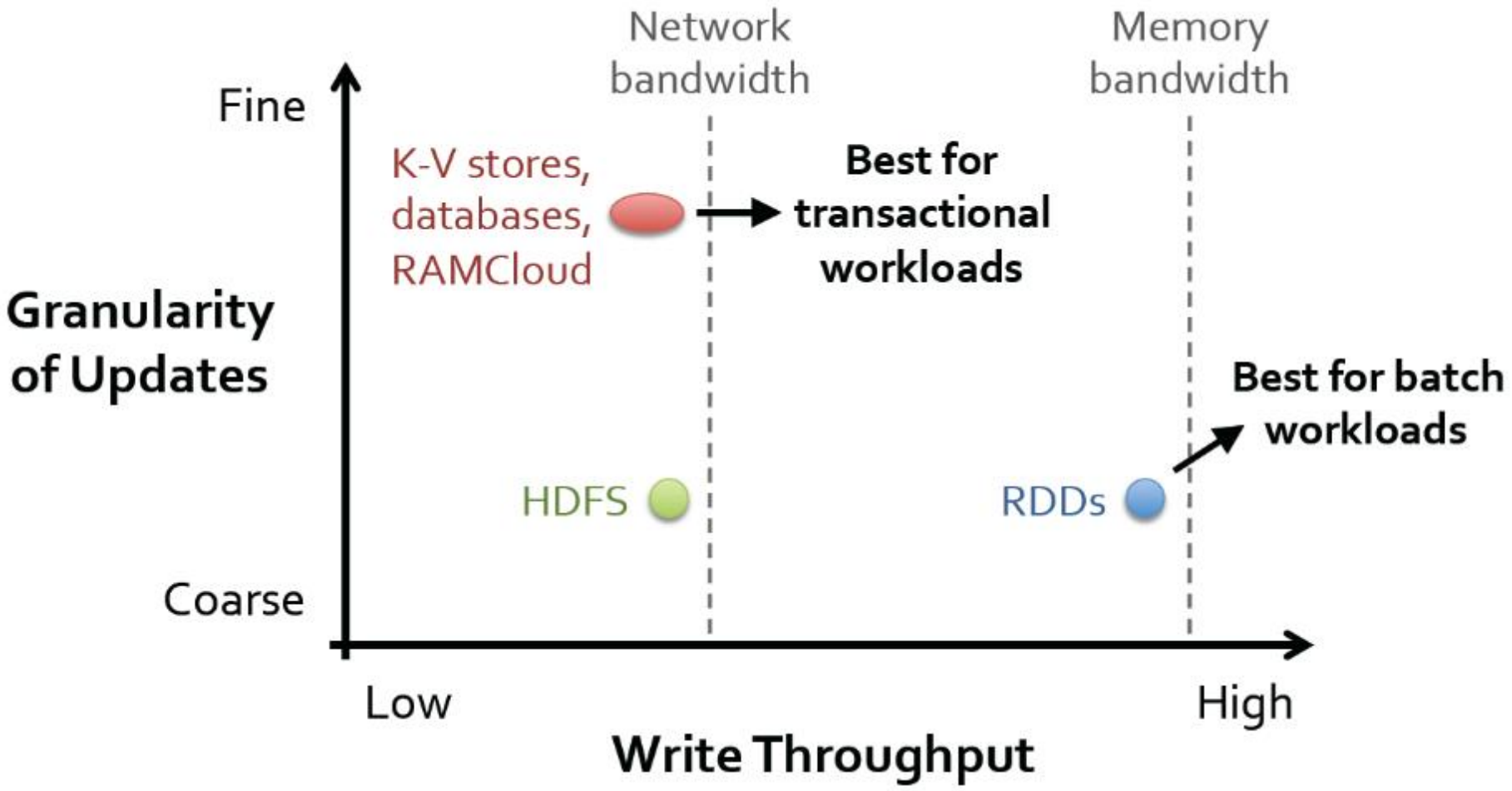
RDD Recovery



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**
  - Instead of specialized APIs for one type of app, give user first-class control of distributed datasets

Tradeoff Space



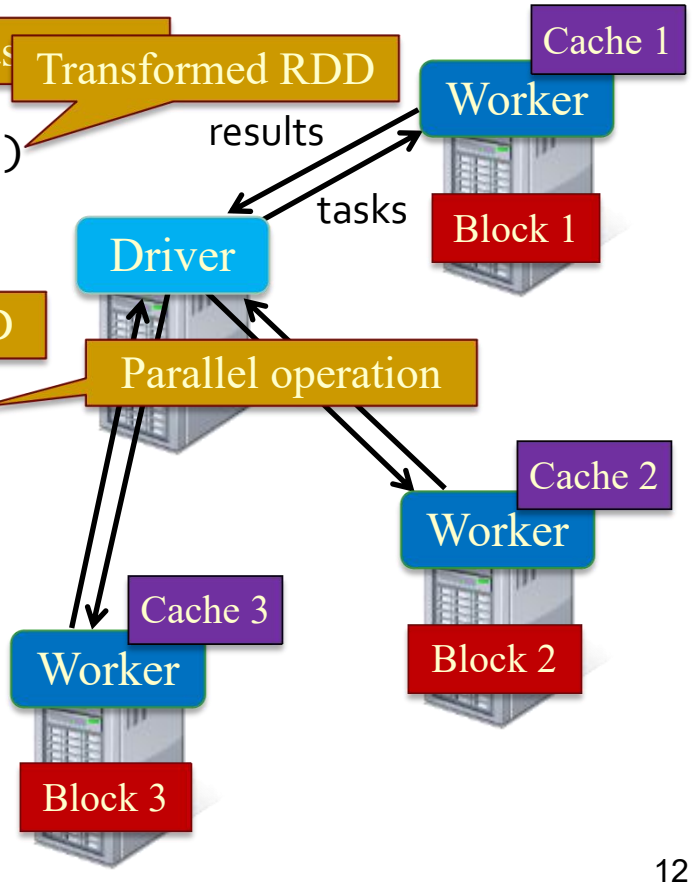
Spark Programming Interface

- **DryadLINQ-like API in the Scala language**
- **Usable iteratively from Scale 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))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



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

RDDs in More Detail

- **An RDD is an immutable, partitioned, logical collection of records**
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- **Partitioning can be based on a key in each record (using hash or range partitioning)**
- **Built using bulk transformations on other RDDs**
- **Can be cached for future reuse**

RDD Operations

Transformations (define a new RDD)	**Parallel operations** (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

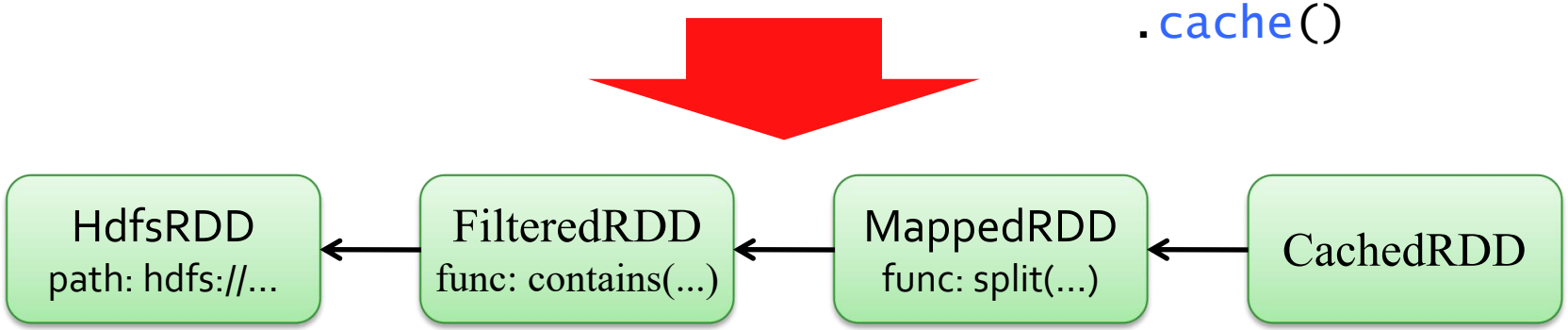
RDD Operations

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

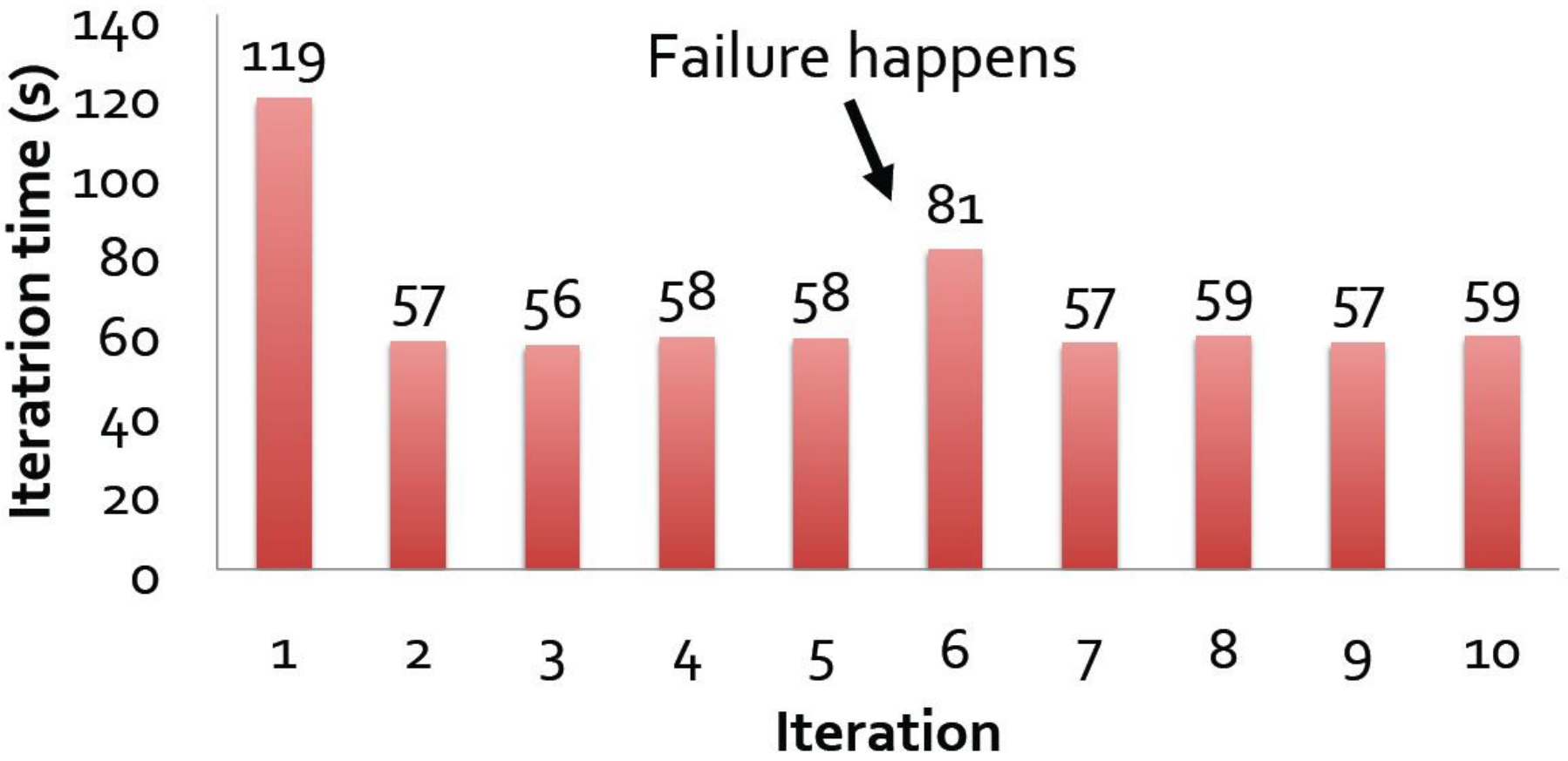
RDD Fault Tolerance

- **RDDs maintain lineage information that can be used to reconstruct lost partitions**
- **Ex:**

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



Fault Recovery Results



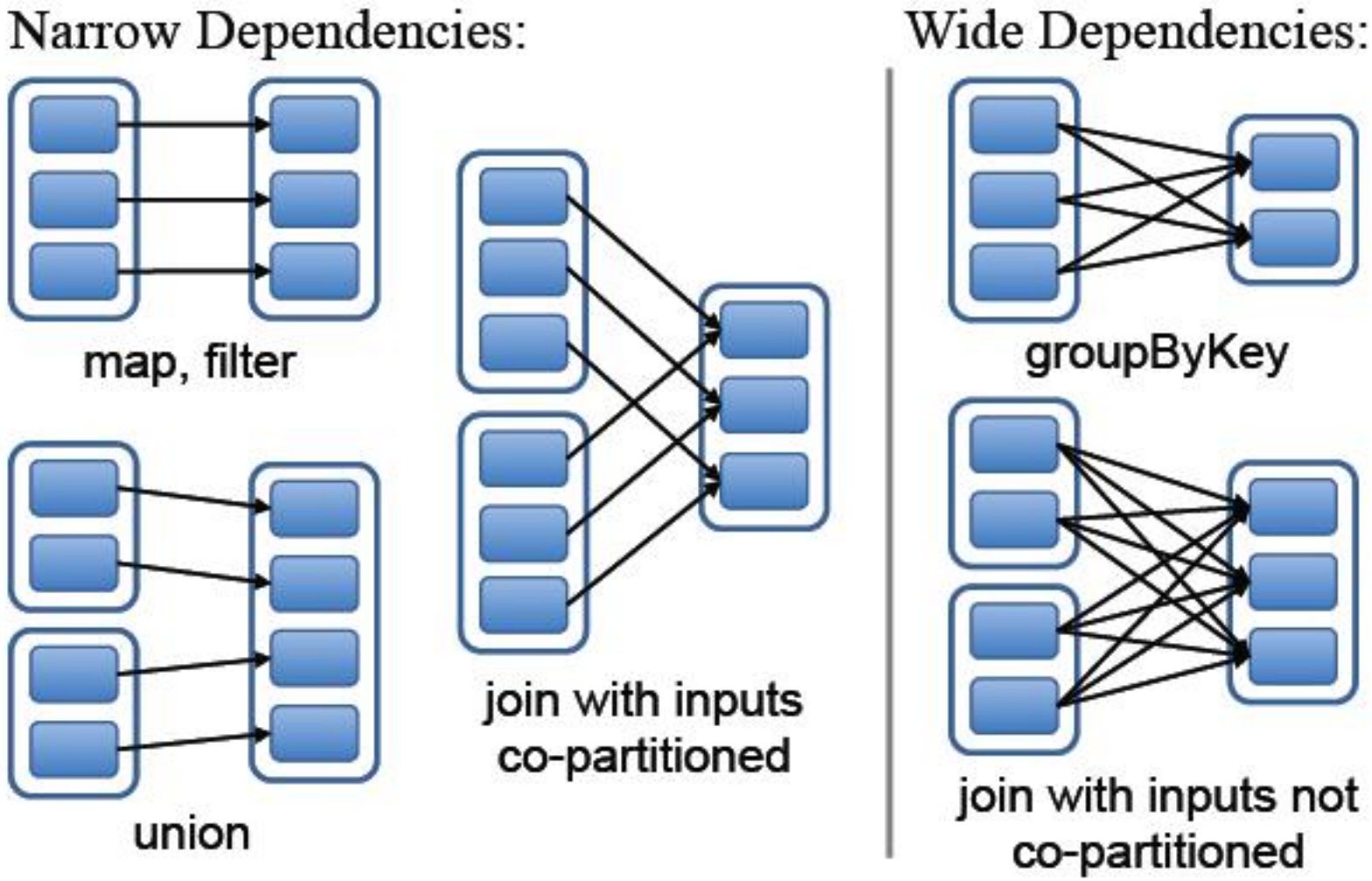
Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

RDDs vs Distributed Shared Memory

Concern	RDDs	Distr. Shared Mem.
Reads	Fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using speculative execution	Difficult
Work placement	Automatic based on data locality	Up to app (but runtime aims for transparency)

Representing RDDs

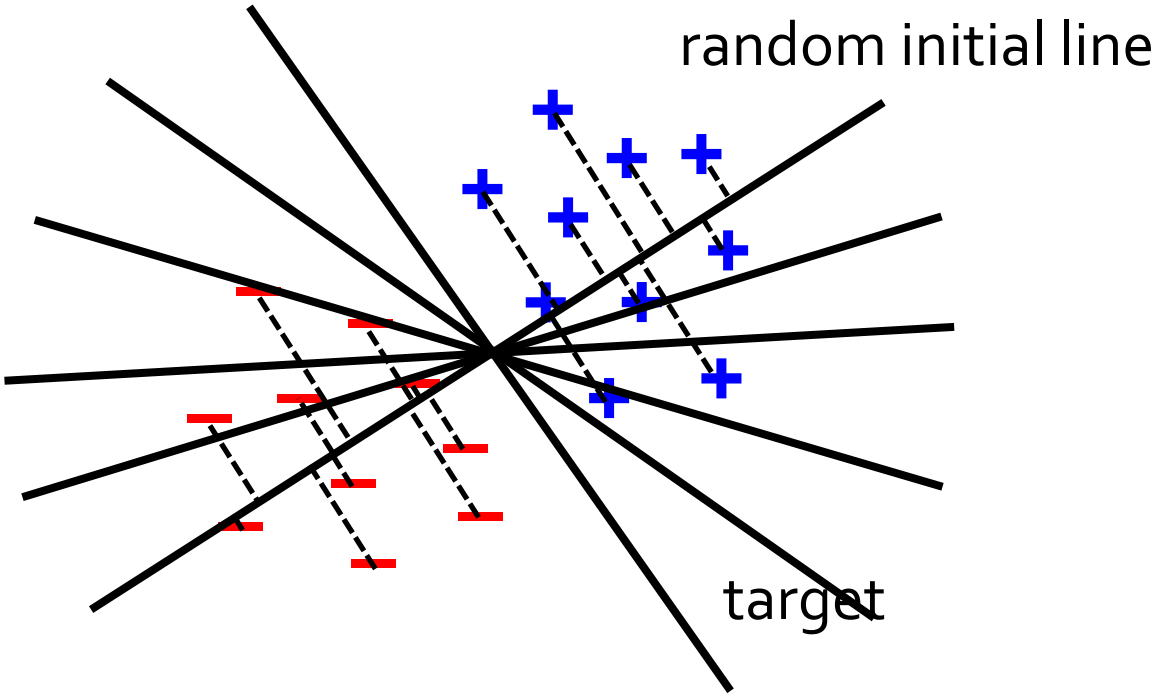


Related Work

- **DryadLINQ**
  - Language-integrated API with SQL-like operations on lazy datasets
  - Cannot have a dataset persist *across* queries
- **Relational databases**
  - Lineage/provenance, logical logging, materialized views
- **Piccolo**
  - Parallel programs with shared distributed tables; similar to distributed shared memory
- **Iterative MapReduce (Twister and HaLoop)**
  - Cannot define multiple distributed datasets, run different map/reduce pairs on them, or query data interactively
- **RAMCloud**
  - Allows random read/write to all cells, requiring logging much like distributed shared memory systems

Example: Logistic Regression

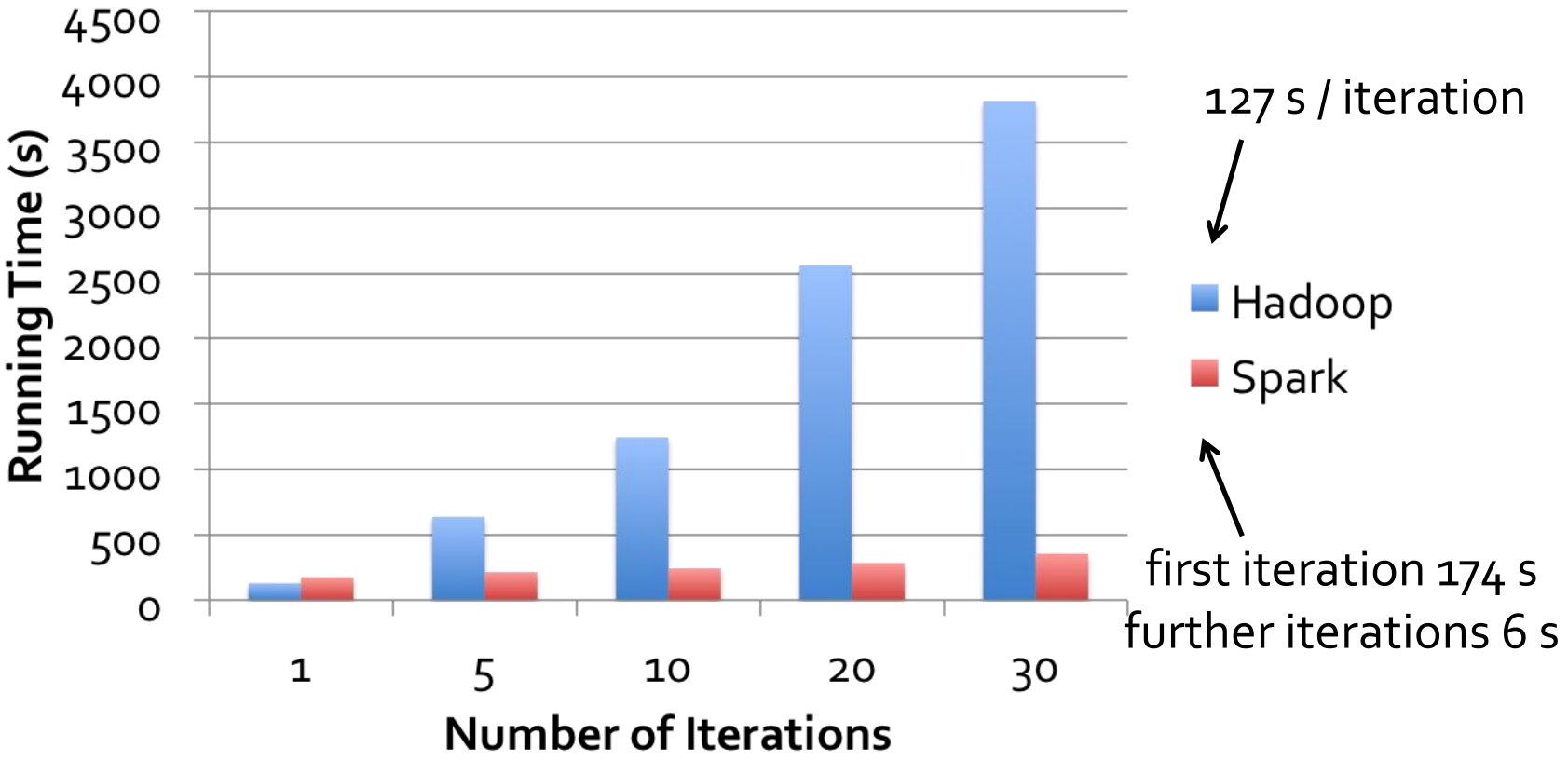
- **Goal: find best line separating two sets of points**



Logistic Regression Code

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Logistic Regression Performance



Example: MapReduce

- **MapReduce data flow can be expressed using RDD transformations**

```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

Word Count in Spark

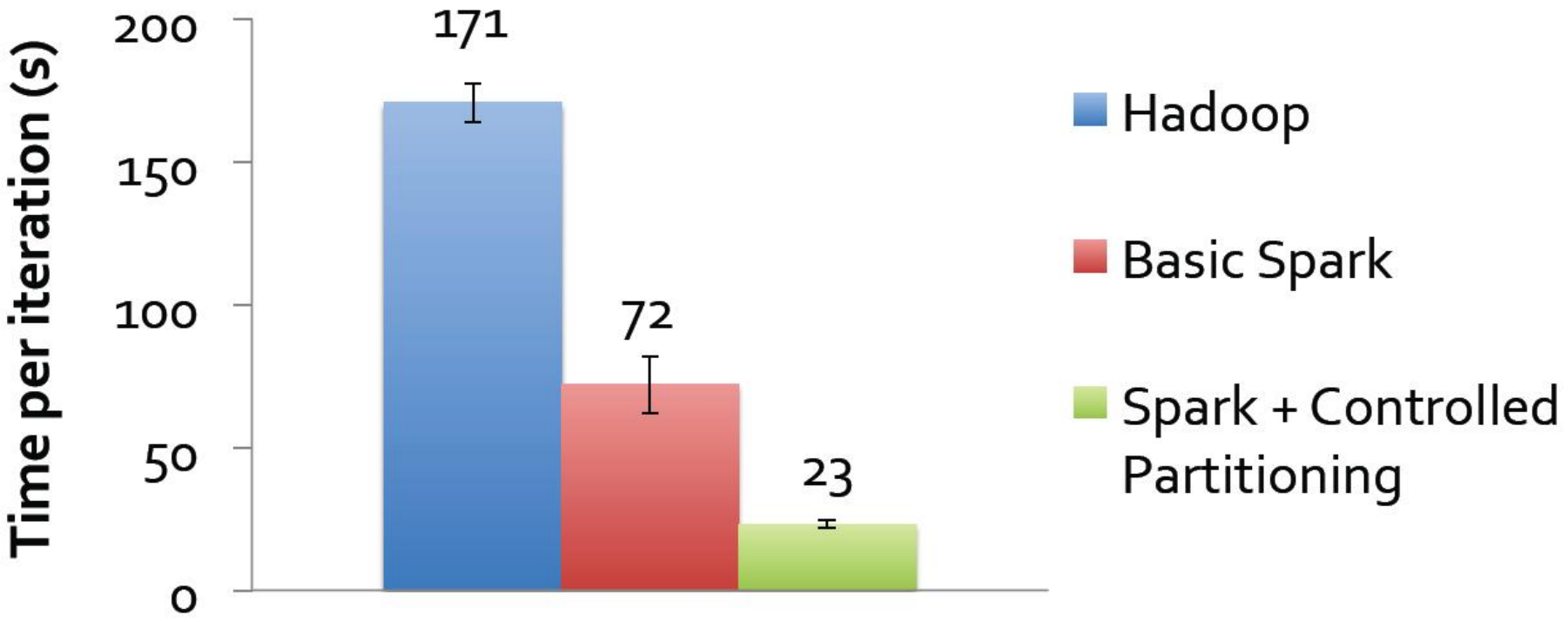
```
val lines = spark.textFile("hdfs://...")

val counts = lines.flatMap(_.split("\\s"))
                  .reduceByKey(_ + _)

counts.save("hdfs://...")
```

Example: PageRank

- **Start each page with a rank of 1**
- **On each iteration, update each page's rank to** $\sum_{i \in neighbors} rank_i / |neighbors_i|$

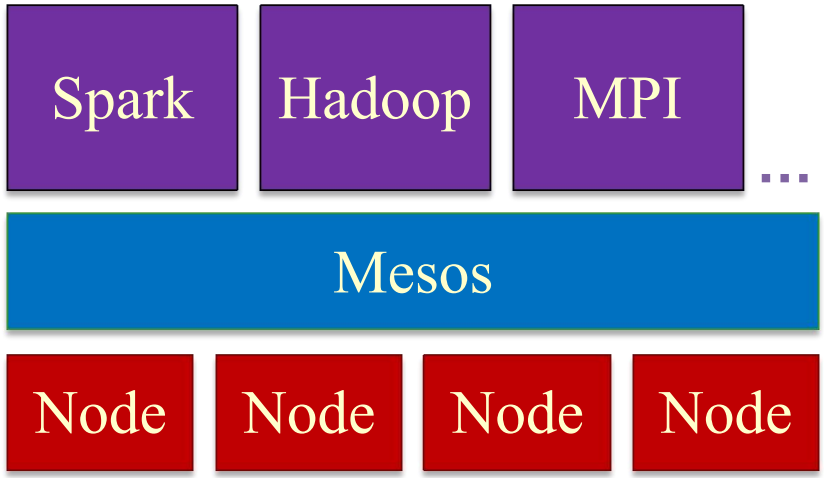
```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pair
for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

PageRank Performance



Implementation

- **Runs on Mesos [NSDI'11] to share cluster w/ Hadoop**
- **Can read from any Hadoop input source (HDFS, S3, …)**
- **No changes to Scala language or compiler**
  - Reflections + bytecode analysis to correctly ship code



Language Integration

- **Scala closures are Serializable Java objects**
  - Serialize on driver, load & run on workers
- **Not quite enough**
  - Nested closures may reference entire outer scope
  - May pull in non-Serializable variables not used inside
  - Solution: bytecode analysis + reflection
- **Shared variables implemented using custom serialized form (e.g. broadcast variable contains pointer to BitTorrent tracker)**

Interactive Spark

- **Modified Scala interpreter to allow Spark to be used interactively from the command line**
- **Required two changes:**
  - Modified wrapper code generation so that each "line" typed has references to objects for its dependencies
  - Place generated classes in distributed filesystem
- **Enables in-memory exploration of big data**

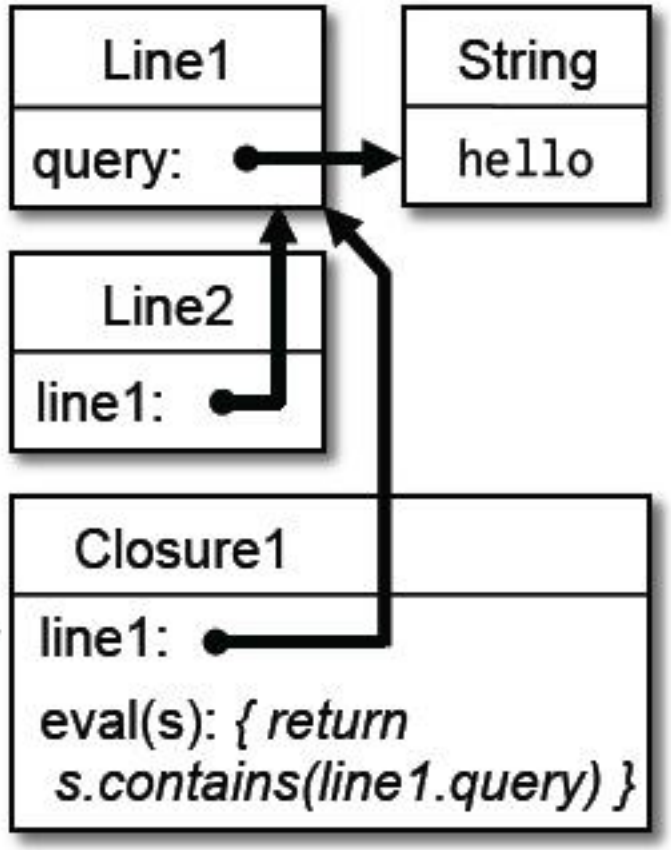
Interactive Spark

Line 1:

```
var query = "hello"
```

Line 2:

```
rdd.filter(_.contains(query))
   .count()
```



a) Lines typed by user

b) Resulting object graph

Spark: Conclusion

- **Spark offers a simple and efficient programming model for stateful data analytics**
  - By making distributed datasets a first-class primitive
  - Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery
- **RDDs provides**
  - Lineage info for fault recovery and debugging
  - Adjustable in-memory caching
  - Locality-aware parallel operations
- **http://spark.apache.org/**