Spark

CS6450: Distributed Systems

Lecture 18

Ryan Stutsman

Problems with Map-Reduce

- Scaled analytics to thousands of machines
- Eliminated fault-tolerance as a concern

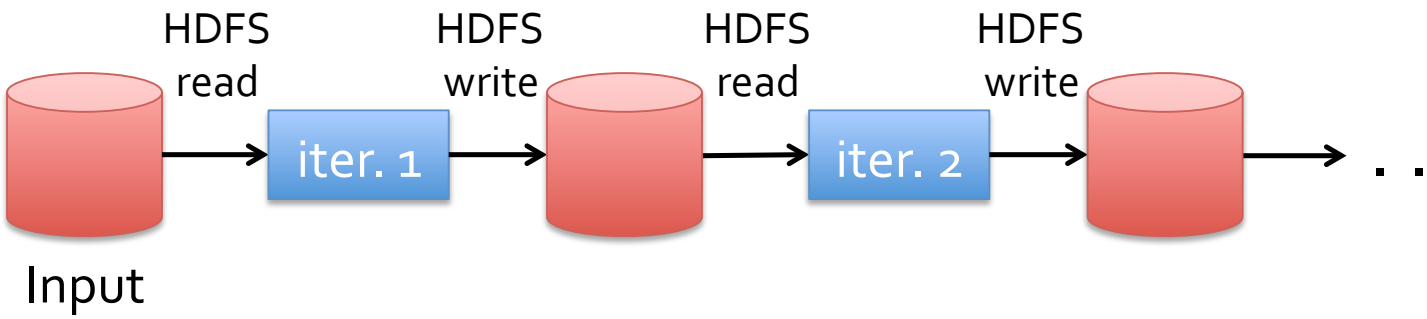
- Not very expressive
  - Iterative algorithms
    (PageRank, Logistic Regression, Transitive Closure)
  - Interactive and ad-hoc queries
    (Interactive Log Debugging)

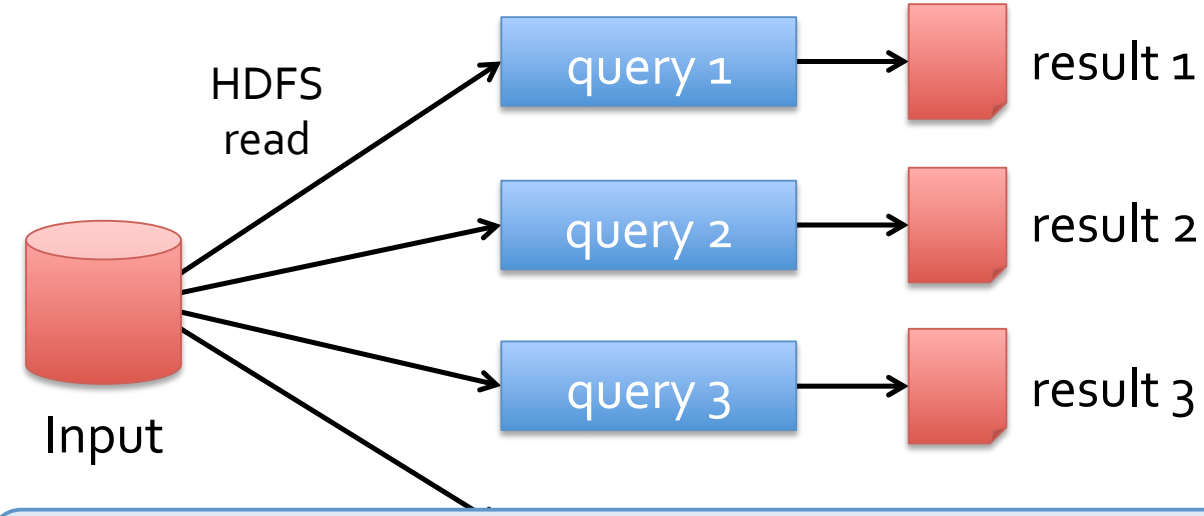
- Lots of specialized frameworks
  - Pregel, GraphLab, PowerGraph, DryadLINQ, HaLoop…

Sharing Data between Iterations/Ops

- Only way to share data between iterations/phases is through shared storage
- Allow operations to feed data to one another
  - Ideally, through memory instead of disk-based storage
- Need the ”chain” of operations to be exposed to make this work
- Also, does this break the MR fault-tolerance scheme?
  - Retry any Map or Reduce task since idempotent

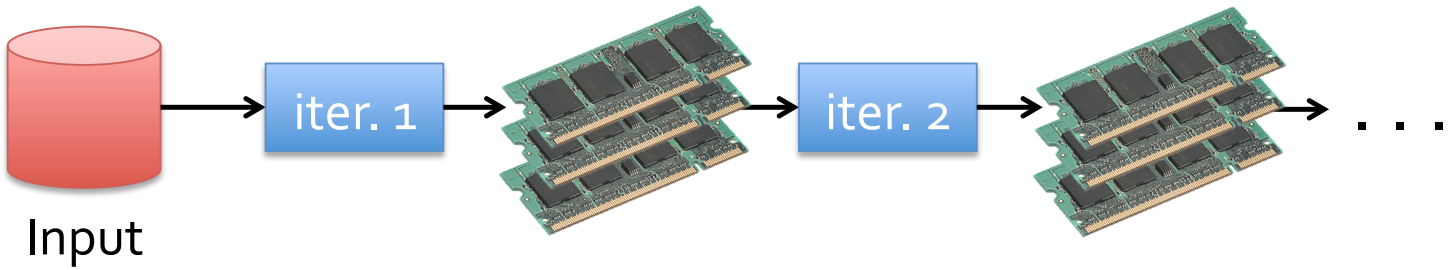
Examples

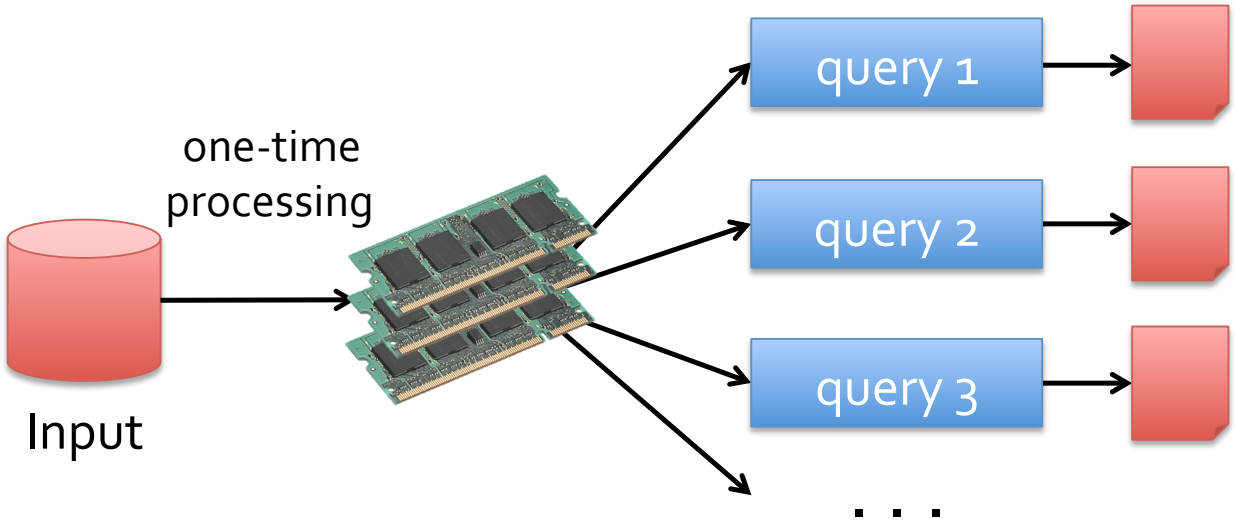




Slow due to replication and disk I/O, but necessary for fault tolerance

Goal: In-memory Data Sharing





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

Challenges

- Want distributed memory abstraction that is both fault-tolerant and efficient
- Existing storage allow fine-grained mutation to state
  - In-memory Key-value stores
  - But, they require costly on-the-fly replication for mutations
- Insight: leverage similar coarse-grained approach that transforms whole data set per op, like MR

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

Spark Programming Interface

- Scala API, exposed within interpreter as well
- RDDs
- Transformations on RDDs (RDD → RDD)
- Actions on RDDs (RDD → output)
- Control over RDD partitioning (how items are split over nodes)
- Control over RDD persistence (in RAM, on disk, or recompute on loss)

Transformations

$$
\begin{aligned}
\mathit{map}(f : \mathrm{T} \Rightarrow \mathrm{U}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]} \\
\mathit{filter}(f : \mathrm{T} \Rightarrow \mathrm{Bool}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]} \\
\mathit{flatMap}(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]} \\
\mathit{sample}(\mathit{fraction} : \mathrm{Float}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]} \ \text{(Deterministic sampling)} \\
\mathit{groupByKey}() &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]} \\
\mathit{reduceByKey}(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]} \\
\mathit{union}() &: (\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]} \\
\mathit{join}() &: (\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]} \\
\mathit{cogroup}() &: (\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]} \\
\mathit{crossProduct}() &: (\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]} \\
\mathit{mapValues}(f : \mathrm{V} \Rightarrow \mathrm{W}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]} \ \text{(Preserves partitioning)} \\
\mathit{sort}(c : \mathrm{Comparator[K]}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]} \\
\mathit{partitionBy}(p : \mathrm{Partitioner[K]}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}
\end{aligned}
$$

RDDs in terms of Scala types → Scala semantics at workers

Transformations are lazy ”thunks”; cause ***no*** cluster action

Actions

$$
\begin{aligned}
count() &: \text{RDD[T]} \Rightarrow \text{Long} \\
collect() &: \text{RDD[T]} \Rightarrow \text{Seq[T]} \\
reduce(f : (\text{T}, \text{T}) \Rightarrow \text{T}) &: \text{RDD[T]} \Rightarrow \text{T} \\
lookup(k : \text{K}) &: \text{RDD[(K, V)]} \Rightarrow \text{Seq[V]} \ \ \text{(On hash/range partitioned RDDs)} \\
save(path : \text{String}) &: \text{Outputs RDD to a storage system, } e.g.,\text{ HDFS}
\end{aligned}
$$

Consumes an RDD to produce output

either to storage (save) or

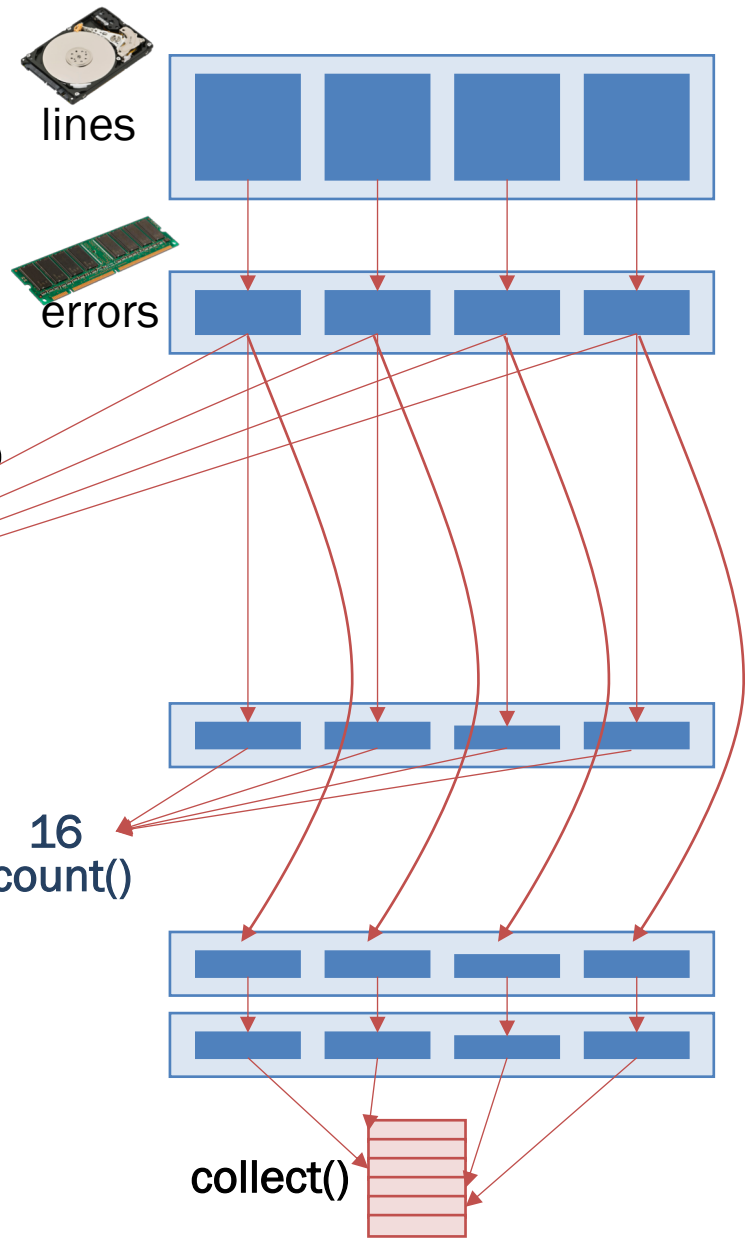
to interpreter/Scala (count, collect, reduce)

Causes RDD lineage chain to get executed on the cluster to produce the output
(for any missing piece of the computation)

Interactive Debugging

```
lines = textFile("hdfs://foo.log")
errors = lines.filter(
            _.startsWith("ERROR"))
errors.persist()

errors.count()
errors.filter(
      _.contains("MySQL")).count()
errors.filter(_.contains("HDFS"))
      .map(_.split("\t")(3))
      .collect()
```

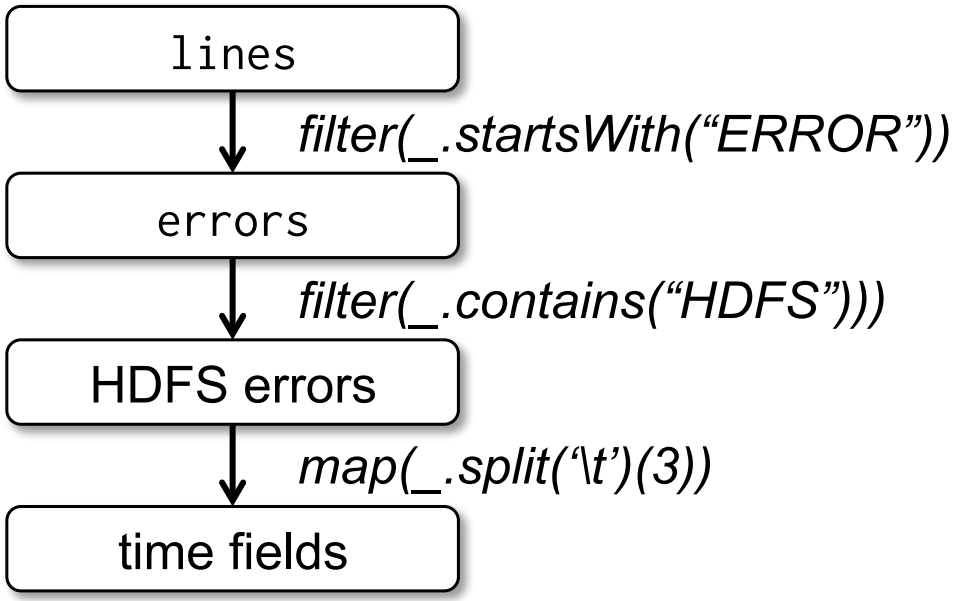


persist()?

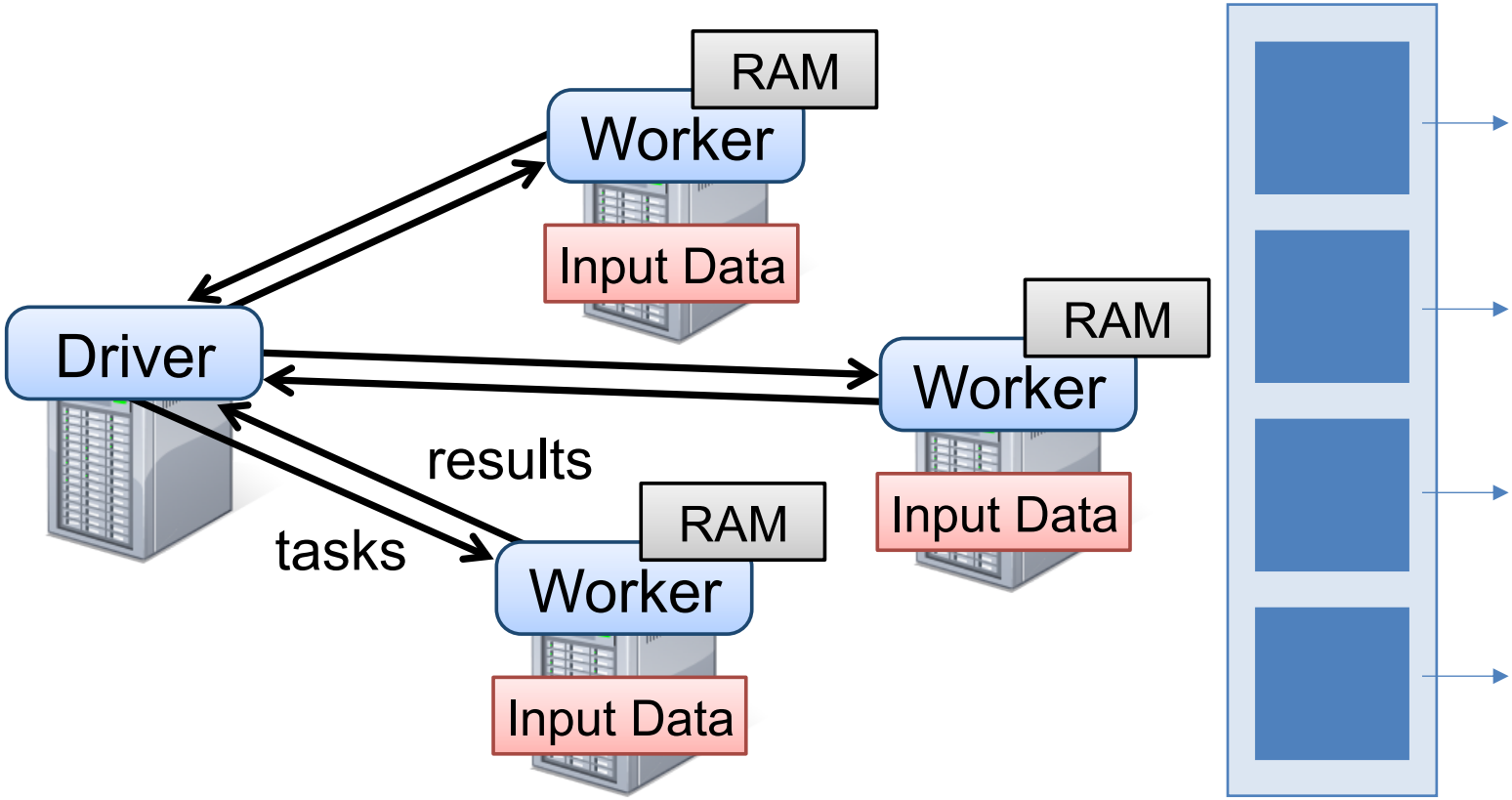
- Not an action and not a transformation
- A scheduler hint
- Tells which RDDs the Spark schedule should *materialize* and whether in memory or storage
- Gives the user control over reuse/recompute/recovery tradeoffs

- Q: If persist() asks for the materialization of an RDD why isn't it an action?

Lineage Graph of RDDs



Physical Execution of Tasks over RDDs



Example: PageRank

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

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

RDD[(URL, Seq[URL])]

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

RDD[(URL, Rank)]

RDD[(URL, (Seq[URL], Rank))]

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

Reduce to RDD[(URL, Rank)]

For each neighbor in links, emits (URL, RankContrib)

Join (⋈)

Ryan	5
Claire	6
Elliott	3

⋈

Ryan	M
Claire	F
Elliott	M

=

Ryan	5	M
Claire	6	F
Elliott	3	M

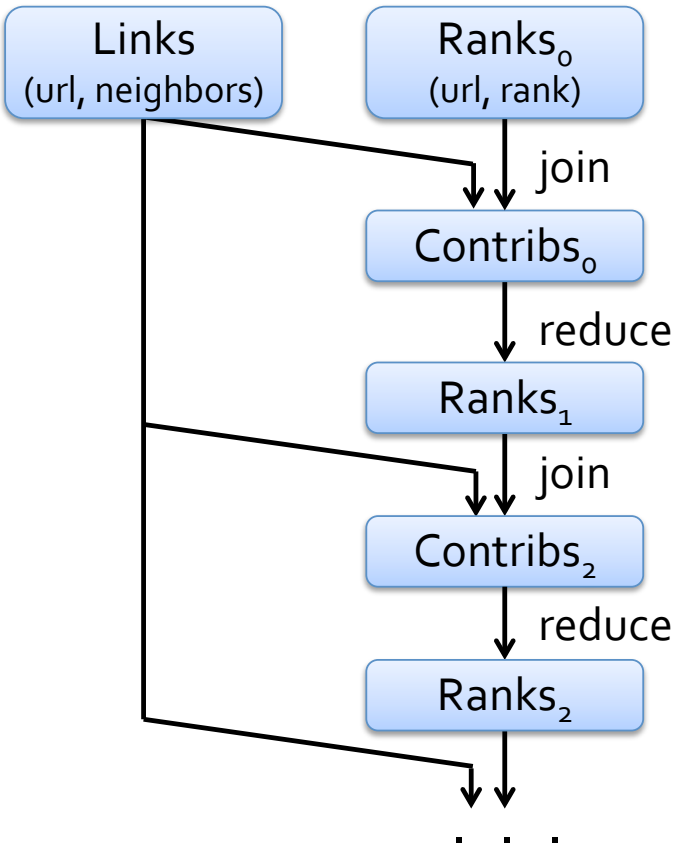
A	5
A	3
A	5
B	4
B	7
C	1
C	9

⋈

C	M
B	M
A	M
B	F
A	F
C	F
B	F

If partitioning doesn't match, then need to reshuffle to match pairs. Same problem in reduce() for Map-Reduce.

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())
```

Q: Where might we have placed persist()?

Co-partitioning Example

From join of "top" partitions

From join of "bottom" partitions

foo.com	5
foo.com	4
widget.com	3
bar.com	2
foo.com	2

bar.com	2
bad.com	0
foo.com	11
widget.com	3

bar.com	foo.com
bad.com	foo.com
foo.com	widget.com
widget.com	bar.com foo.com

bar.com	5
bad.com	4
foo.com	3
widget.com	4

- Can avoid shuffle on join
- But, fundamentally a shuffle on reduceByKey
- Optimization: custom partitioner on domain

PageRank Performance

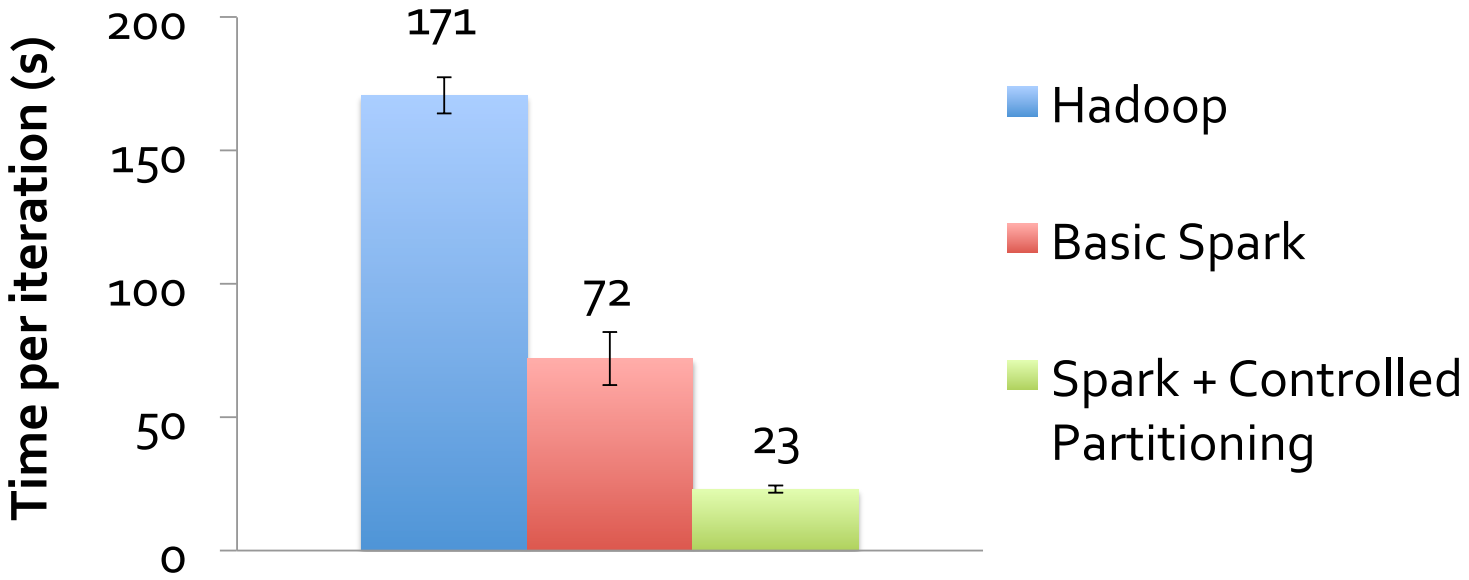
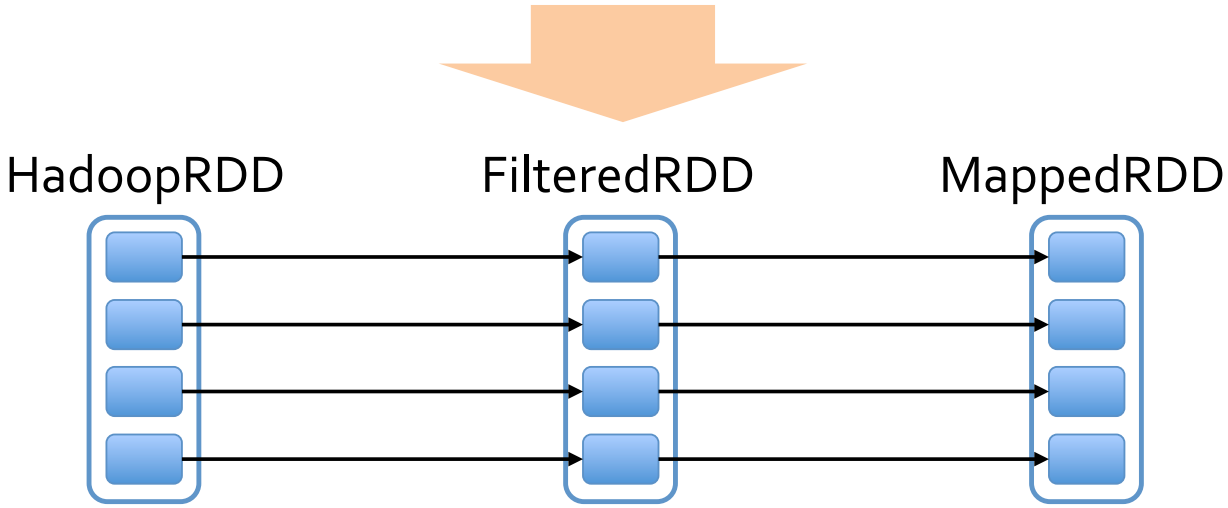


Figure 10a: 30 machines on 54 GB of Wikipedia data computing PageRank

Fault Recovery

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

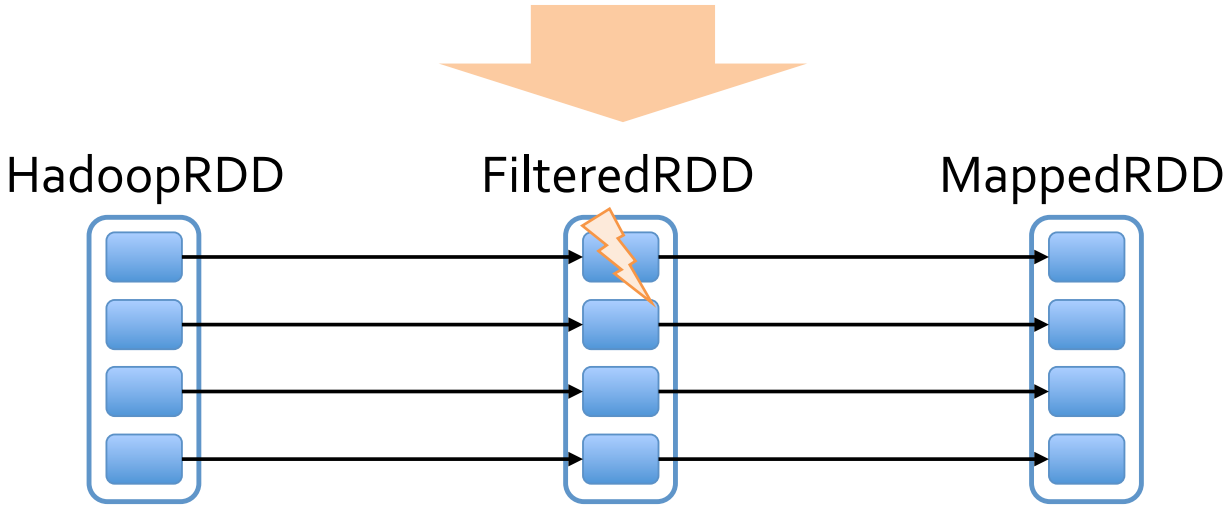
E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                         .map(_.split('\t')(2))
```



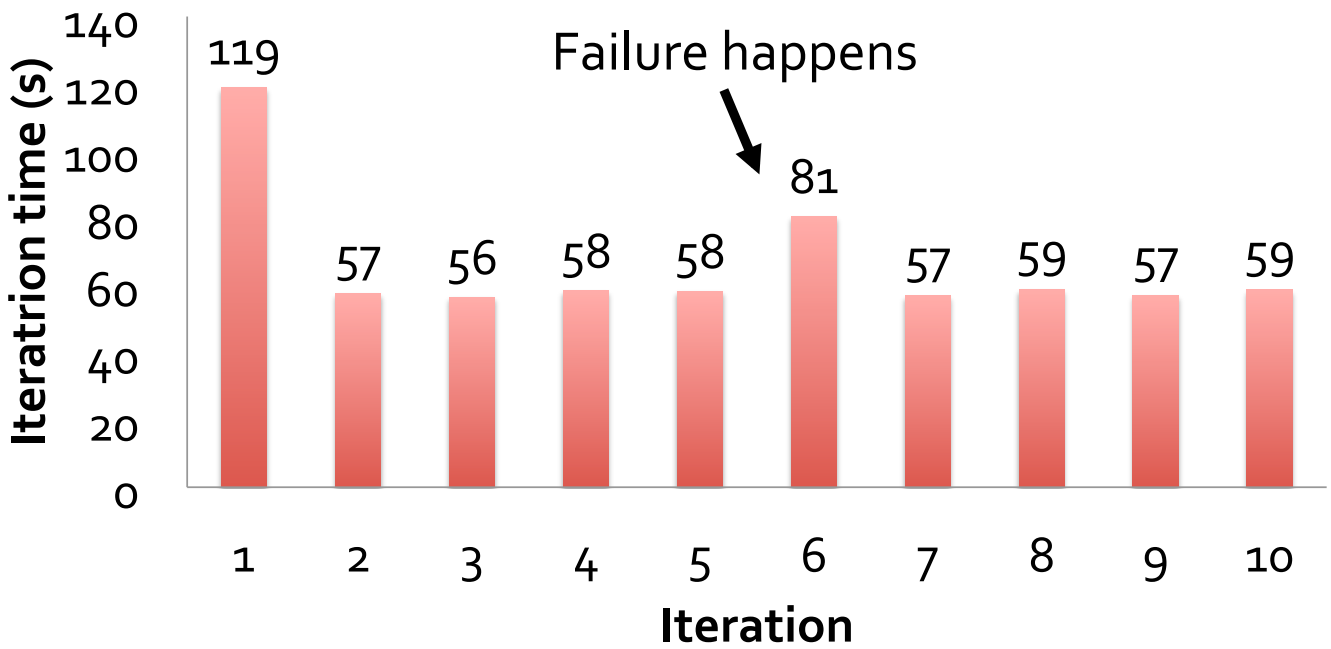
Fault Recovery

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))
```

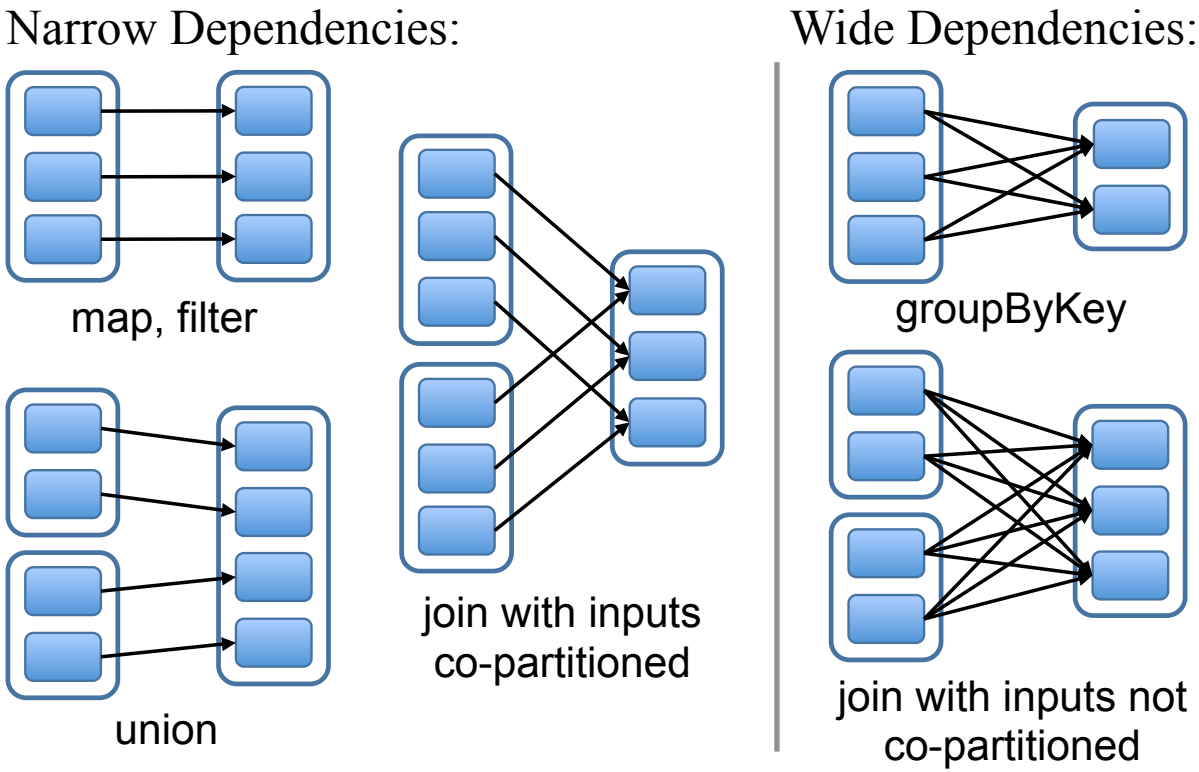


Fault Recovery



100 GB K-means Fig 11, partial reconstruction in step 6 much less than cost to write back results at each step with MR

Narrow & Wide Dependencies



Wide: multiple child partitions depend on partition

Must stall for all parent data, loss of child requires whole parent RDD (not just a small # of partitions)

Narrow: can pipeline on one machine

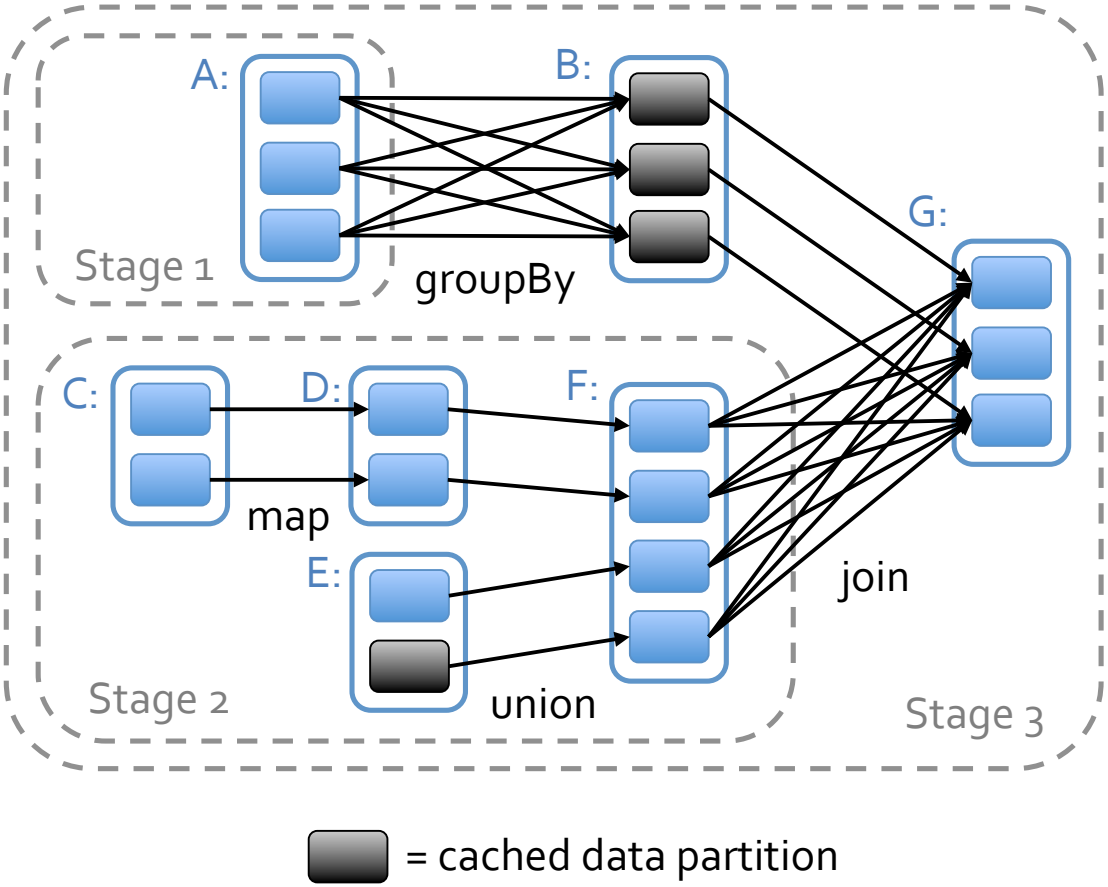
Task Scheduler

Dryad-like DAGs

Pipelines functions within a stage

Locality & data reuse aware

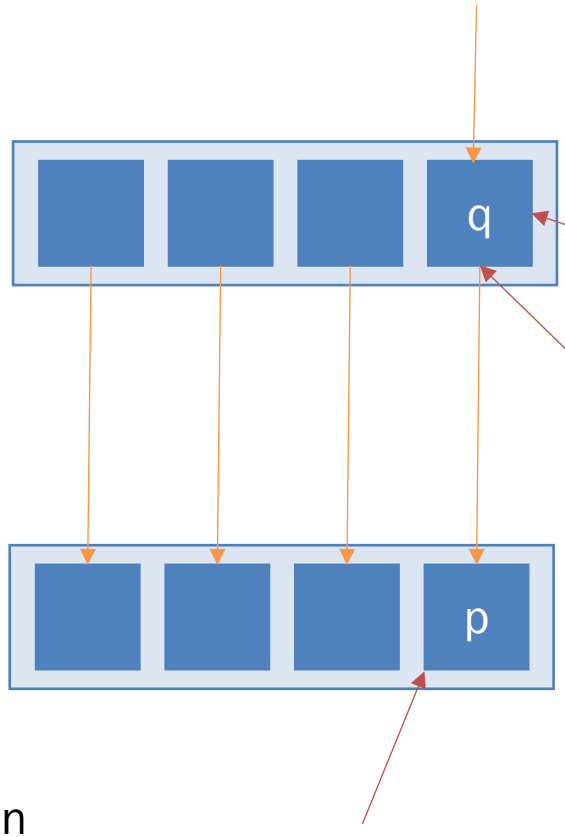
Partitioning-aware to avoid shuffles



RDD Implementation

- partitions(): set of partitions (ranges/hash range)
- dependencies(): set of parent RDDs
- closure for computing the transformation
- preferredLocations(p): returns a set of locations where partition p can access parent data locally
- partitioner(): metadata about RDD partitioning scheme

Volcano Model



dependencies()

parent.partitioner() ==
child.partitioner()
and no shuffle,
so narrow

preferredLocations(p)

Iterator accesses are
nested/recursive to pipeline
work and avoid materialization
while avoiding communication

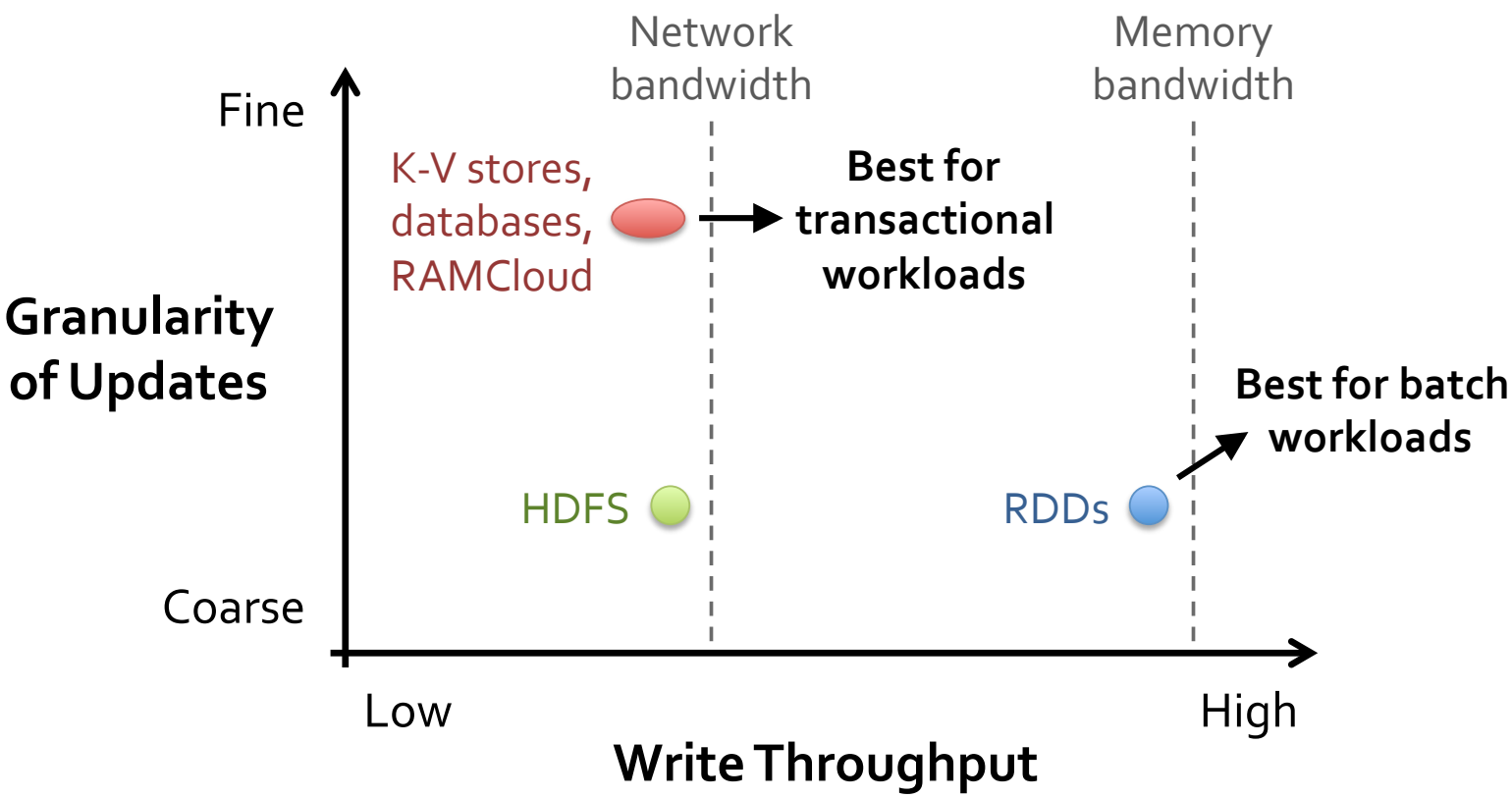
i = iterator(p, qIter))

child.count() { return sum([row for row in i]) }

Generality of RDDs

- RDDs can express many parallel algorithms
  - They already apply the same operation to many items
- Unifies many programming models
  - Data flow models: MapReduce, Dryad, SQL, …
  - Specialized models for iterative apps: BSP/Pregel, iterative MapReduce (Haloop), bulk incremental, ..
- Supports new applications that these models don't

Tradeoff Space



Programming Models as Libraries

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

Memory Reuse Impact

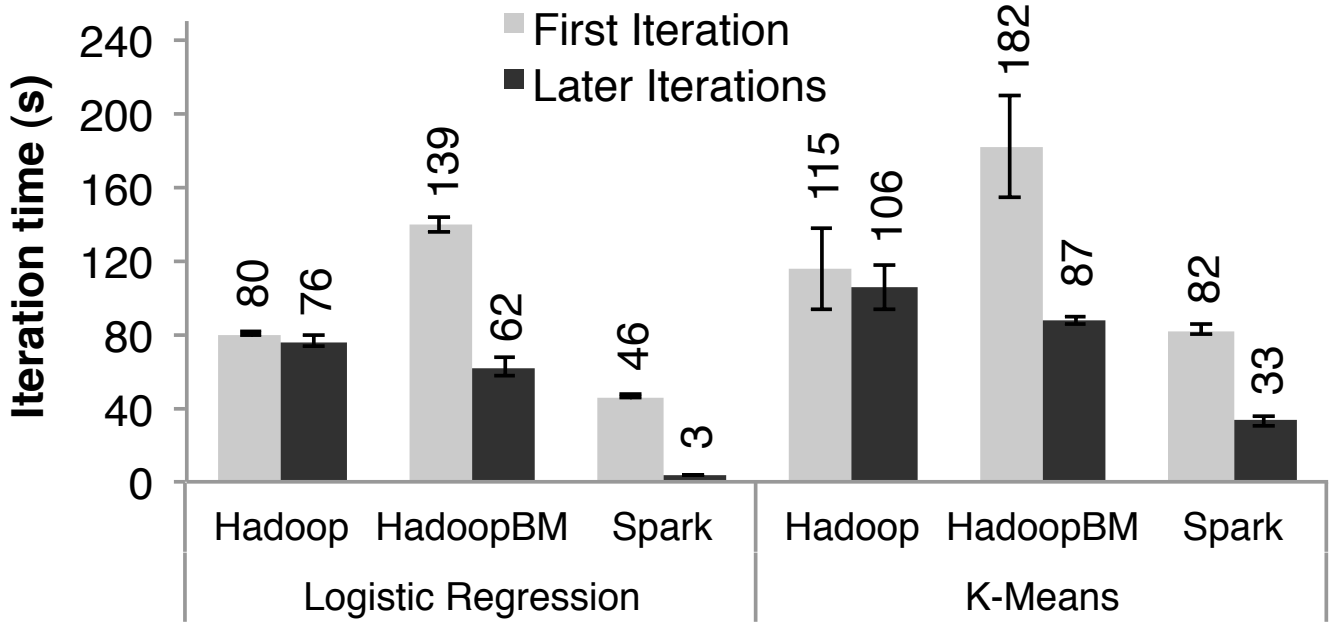


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

Analysis of Speedup

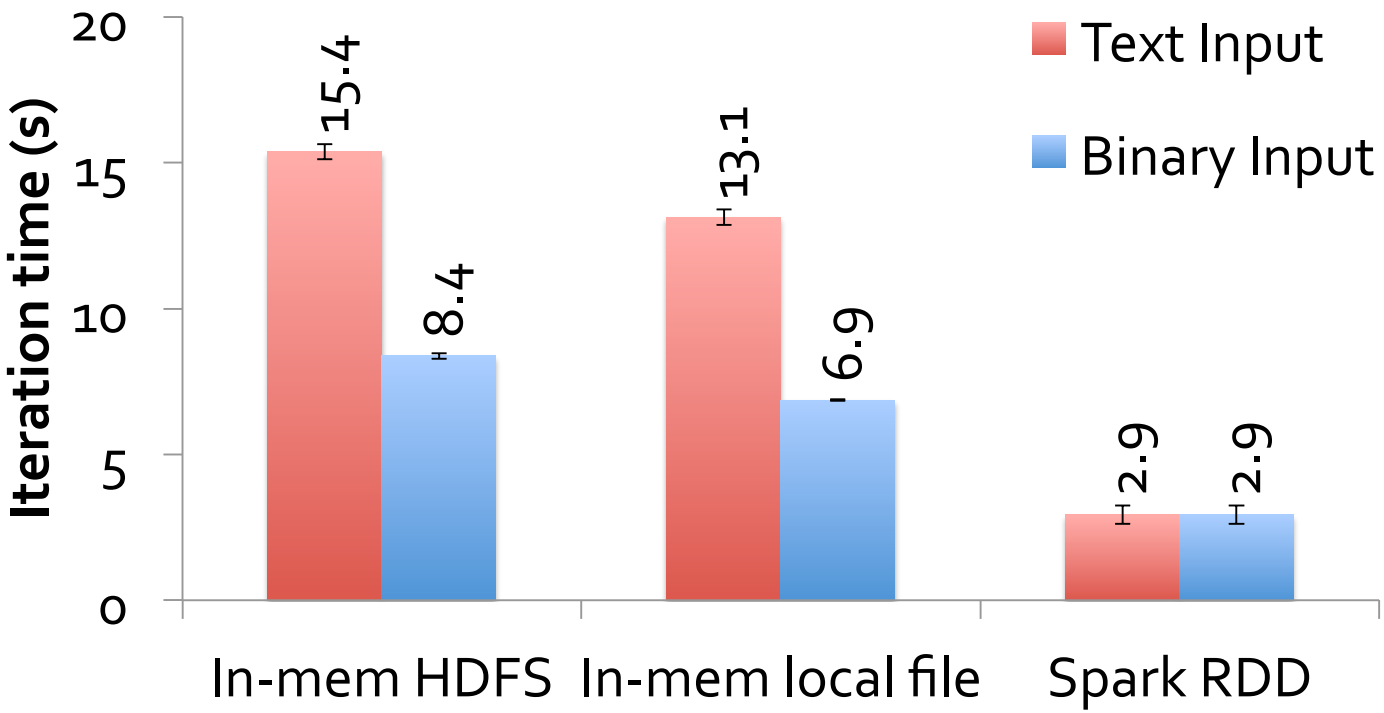


Fig 9: 256 GB on a single machine

Eviction and Working Sets

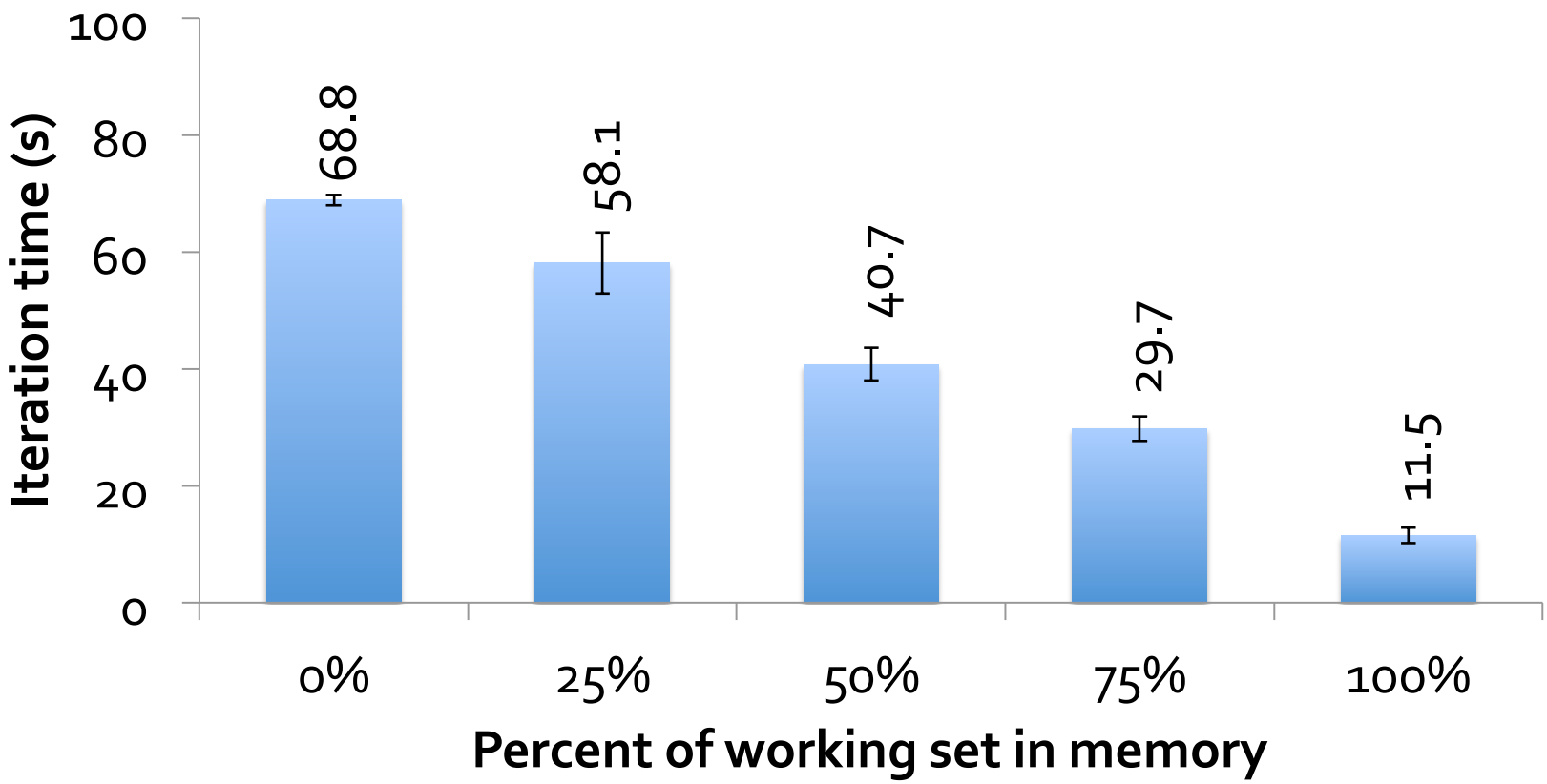
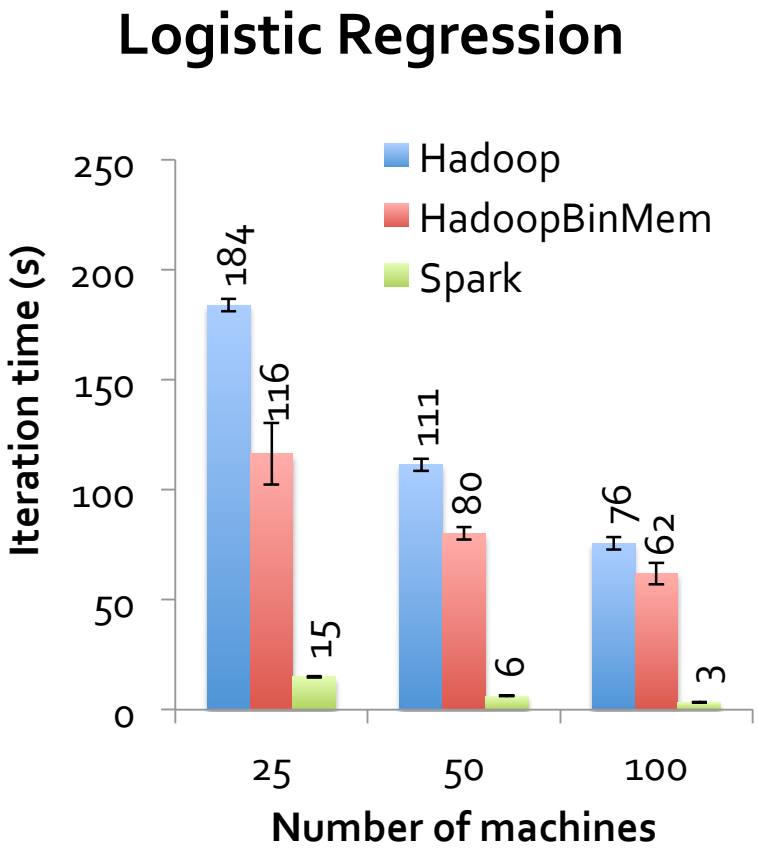
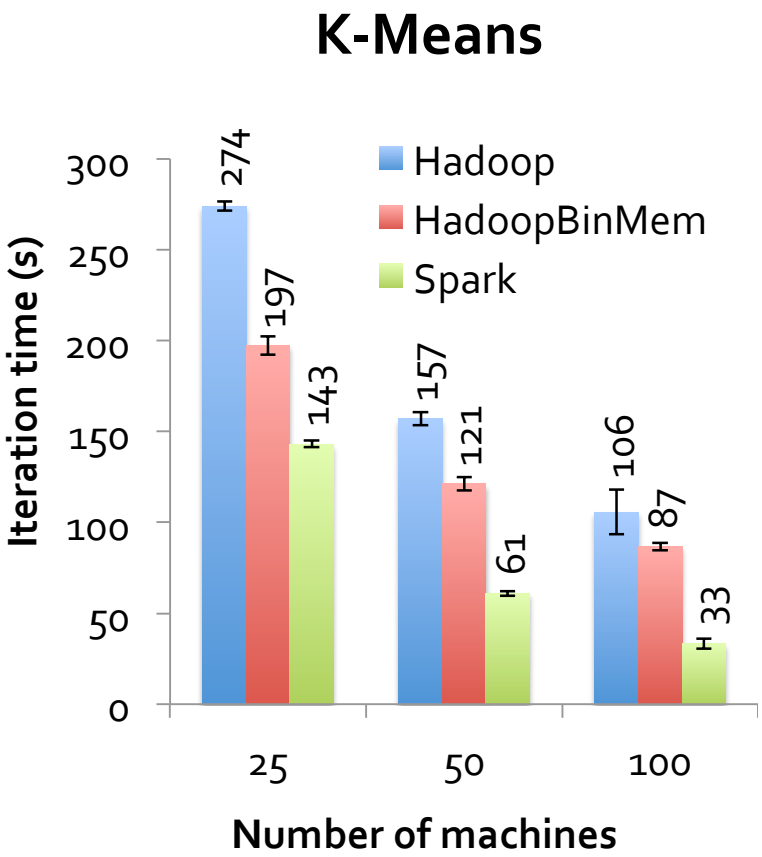


Fig 12: 100 GB LR on 25 machines

Scalability





100 GB datasets each

Evolution of Spark

- Want to write Spark programs in different languages
  - Not everyone loves Scala, or JVM-based languages
- Problem: RDD semantics are bound to JVM
- Move toward Dataframes
  - Effectively, DB relations
  - Can manipulate representation w/o ser/des
  - Can use bindings in any language
- Capturing closures in many languages? Mismatching language/dataset semantics…

Conclusions

- M-R expressivity and performance have been a central point of sadness
- Several attempts to make improvements
- Spark improves expressivity, which also improves performance since scheduler can “think” across the whole pipeline
- Still preserves a lot of M-R fault-tolerance
- Does force users to reason a bit about fault-tolerance, though through careful persist() calls
- Not dead yet: TensorFlow and many more …