Lightning-fast cluster computing

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 J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

UC BERKELEY

NSDI 2012

Contents

Motivation

MapReduce greatly simplifies “big data” analysis on large, unreliable clusters

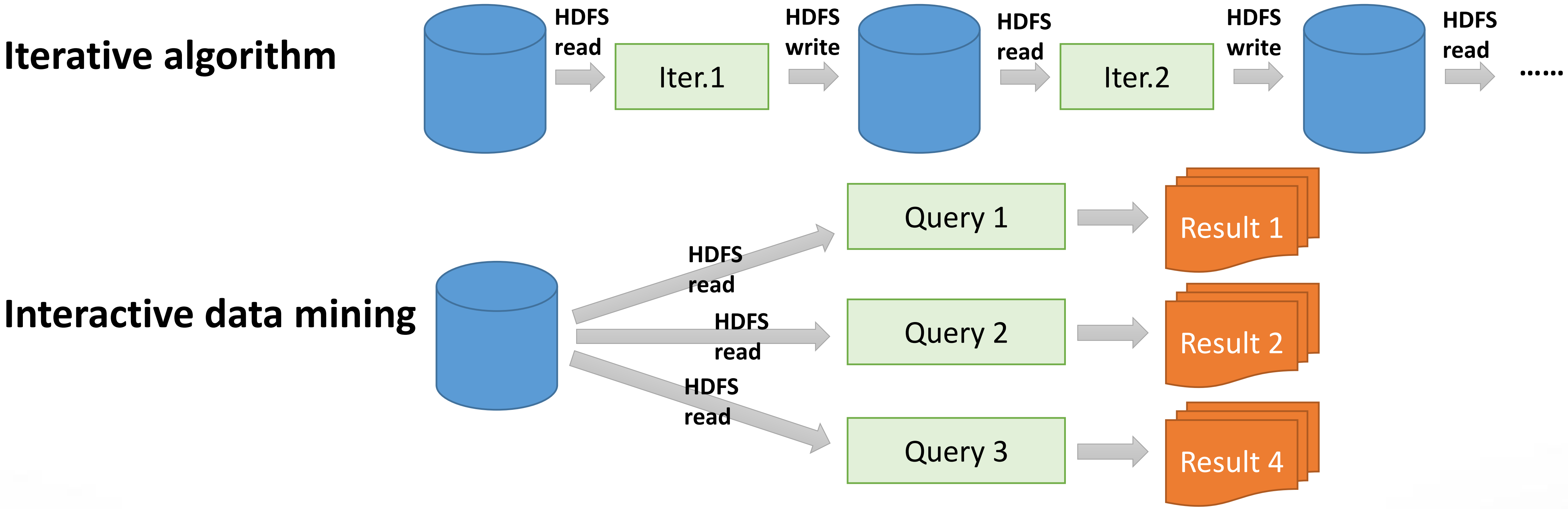
Problems:

- Iterative algorithm & graph processing
- Interactive data mining

inefficient

Cause:
Inefficient data sharing

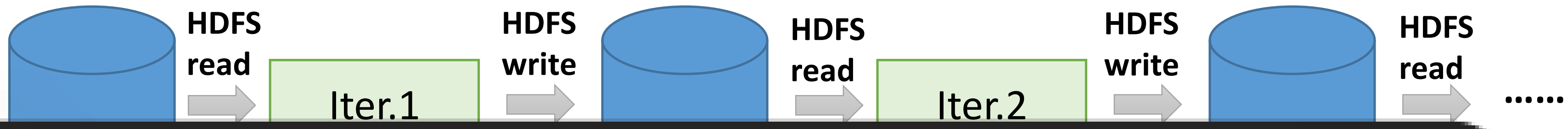
Motivation



Replication & Disk I/O

Motivation

Iterative algorithm



GOAL DESIGN

- Efficient in-memory cluster computing (iterative algorithm & interactive data mining)
- Fault-tolerant
- General-purpose abstraction
- Explicitly persist intermediate results in memory
- Control partitioning to optimize data placement

Replication & Disk I/O

Resilient Distributed Datasets (RDDs)

An RDD is a read-only, partitioned collection of records, which provides a restricted form of shared memory.

Restricted conditions:

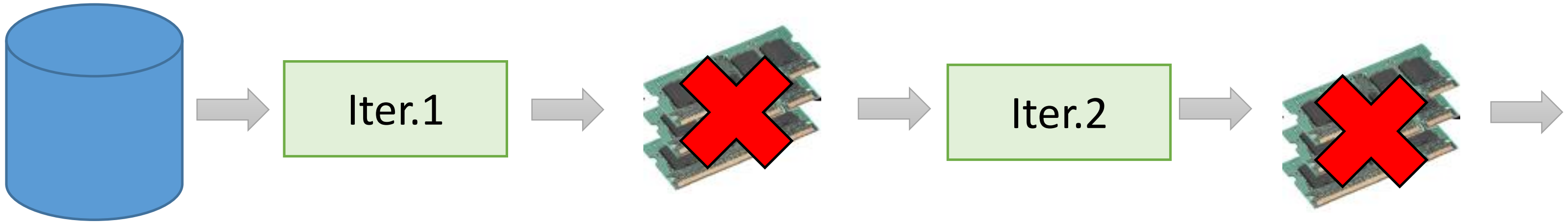
- Read-only, partitioned
- Built through coarse-grained transformations (e.g., map, filter and join)

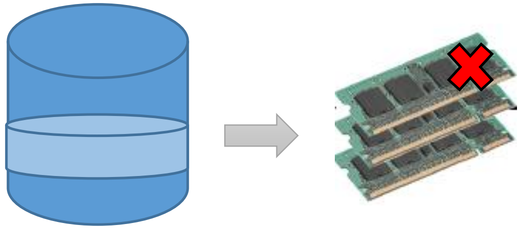
Powerful properties:

- Generality of RDDs —Abstraction
- Running backup copies of tasks from the slow nodes
- Automatic schedule based on data locality
- Keeping intermediate data in memory
- Efficient fault recovery using lineage

Resilient Distributed Datasets (RDDs)

Fault recovery





Spark programming

Language-integrated API in Scala language

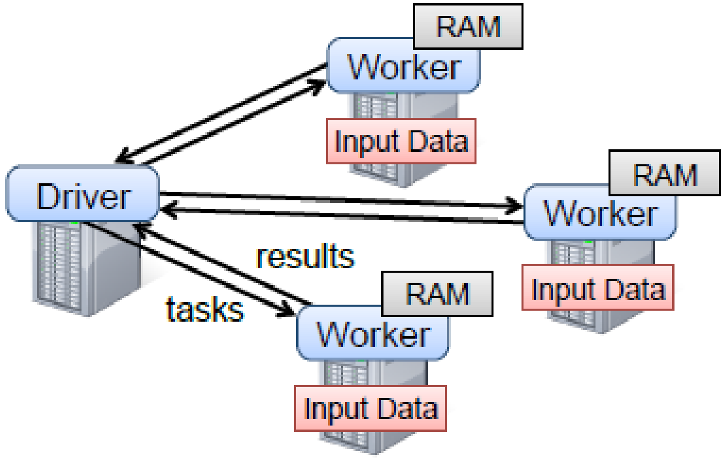
Runtime: a driver connects to a cluster of workers

Two type of operations:

- Transformations ——create RDDs
- Actions ——compute and output results

Control partitioning

Control persistence (storage in RAM, on disk, etc)



Spark programming

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

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Spark programming

Example 1: Log Mining

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
messages.persist()
messages.count()
messages.filter(_.contains("MySQL")).count()
```

Base RDD

Transformed RDD

Declare RDD persist in memory

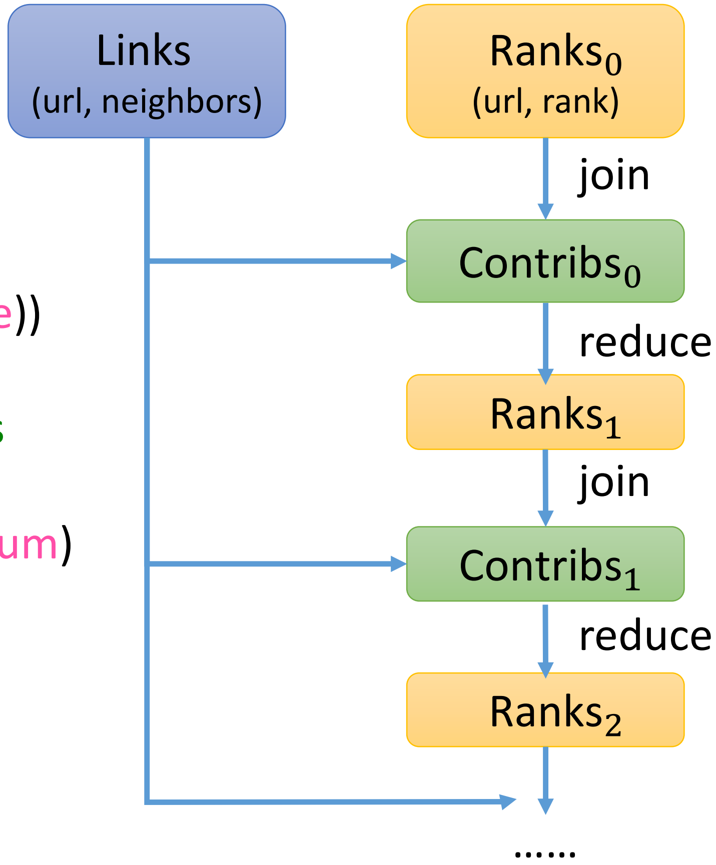
Action

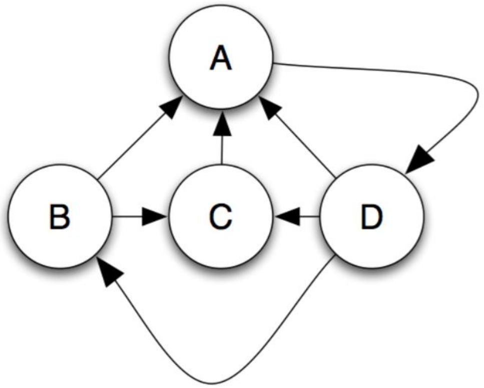
Spark programming

Example 2: PageRank

```
val links = spark.textFile(...).map(...).persist() // RDD of (URL, neighbors) pairs
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
   // Build an RDD of (targetURL, float) pairs
   // with the contributions sent by each page
   val contribs = links.join(ranks).flatMap {
   (url, (links, rank)) =>
               links.map(dest => (dest, rank/links.size))
   }
   // Sum contributions by URL and get new ranks
   ranks = contribs.reduceByKey((x,y) => x+y)
                     .mapValues(sum => a/N + (1-a)*sum)
}
```

$$PR(A) = \frac{a}{N} + (1-a)\sum_i \frac{PR(I_i)}{C(I_i)}$$





Optimizing RDD placement

- hash-partition the links and ranks by URL
- custom Partitioner using partitionBy

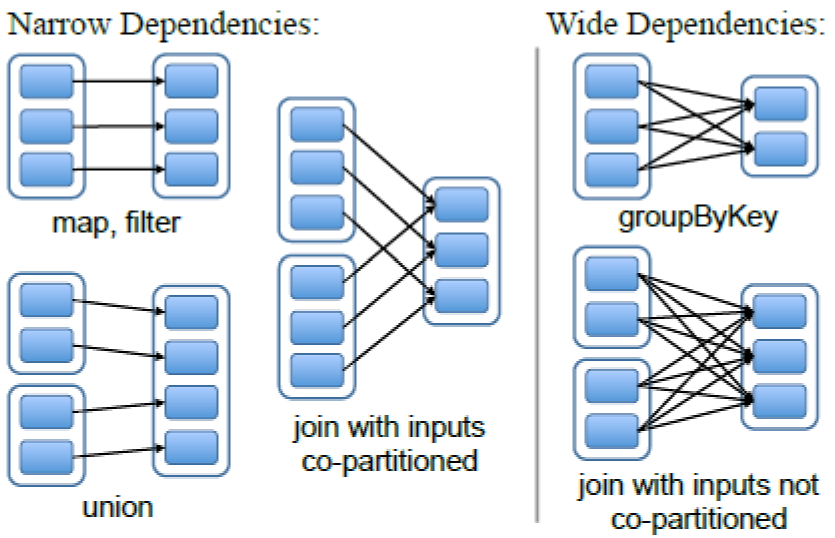
e.g., links = spark.textFile(...).map(...) .partitionBy(myPartFunc).persist()

Spark programming

Implementation

- Representing RDDs

  Interface exposes five pieces of information:
  - a set of partitions
  - a set of dependencies on parent RDDs
  - a function for computing the dataset based on its parents
  - the metadata about its partitioning scheme
  - the data placement



- Job Scheduling

  Delay scheduling
  Schedule based on data locality

- Memory Management
  - in-memory storage as deserialized Java objects
  - in-memory storage as serialized data
  - on-disk storage

  limited memory available —— LRU

- Support for Checkpointing

  Persist——REPLICATE flag

Evaluation

Iterative Machine Learning Applications

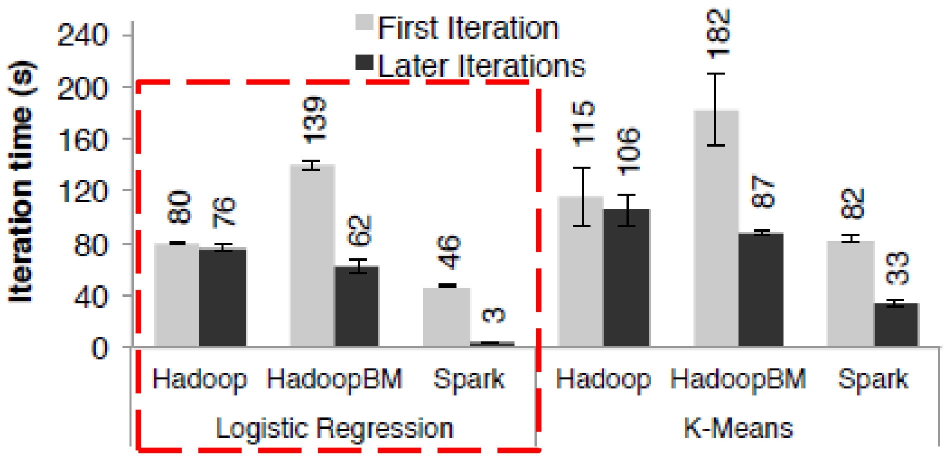


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

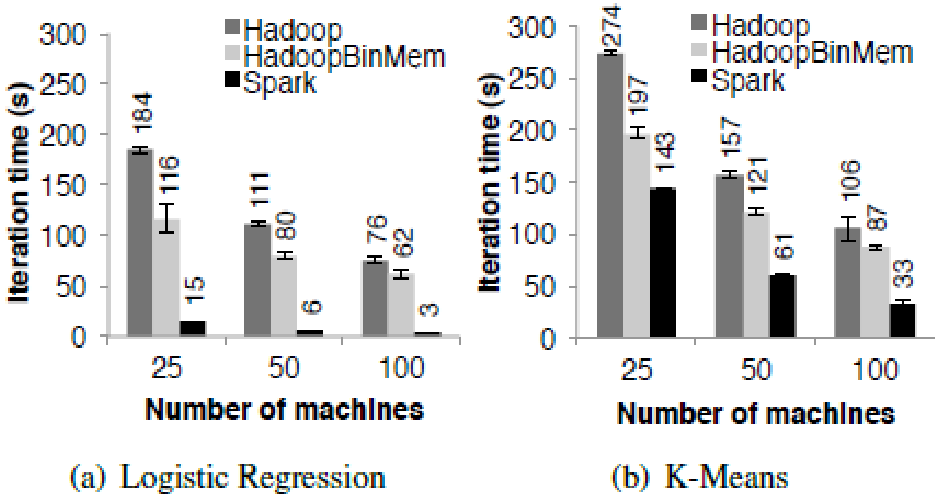
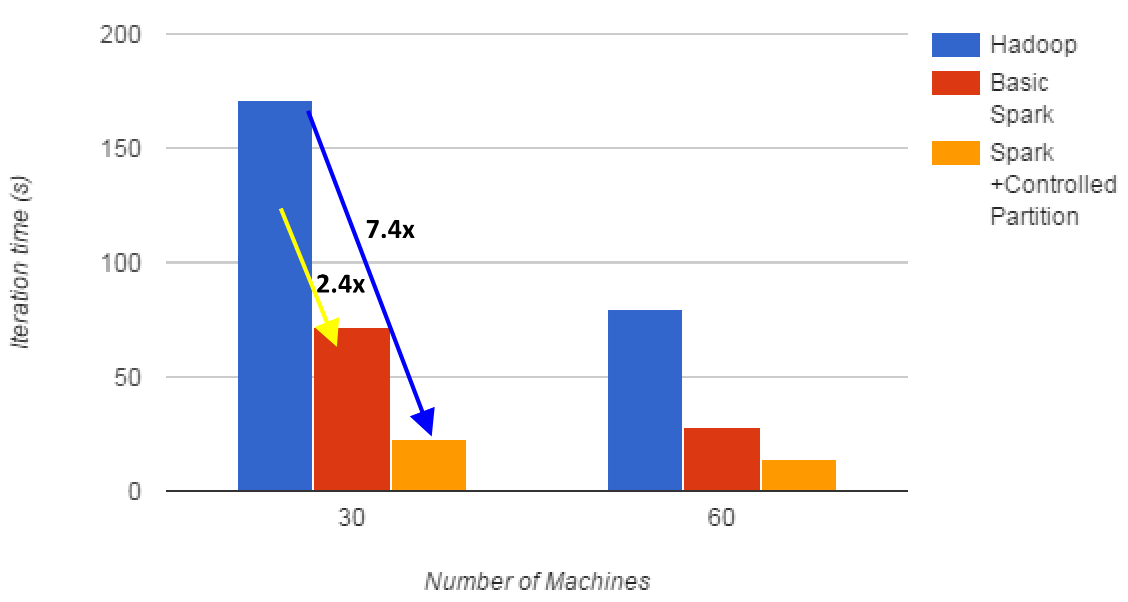


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

- ✓ Hadoop: The Hadoop 0.20.2 stable release.
- ✓ HadoopBinMem: A Hadoop deployment that converts the input data into a low-overhead binary format in the first iteration and stores it in an in-memory HDFS instance.
- ✓ Spark: Our implementation of RDDs.

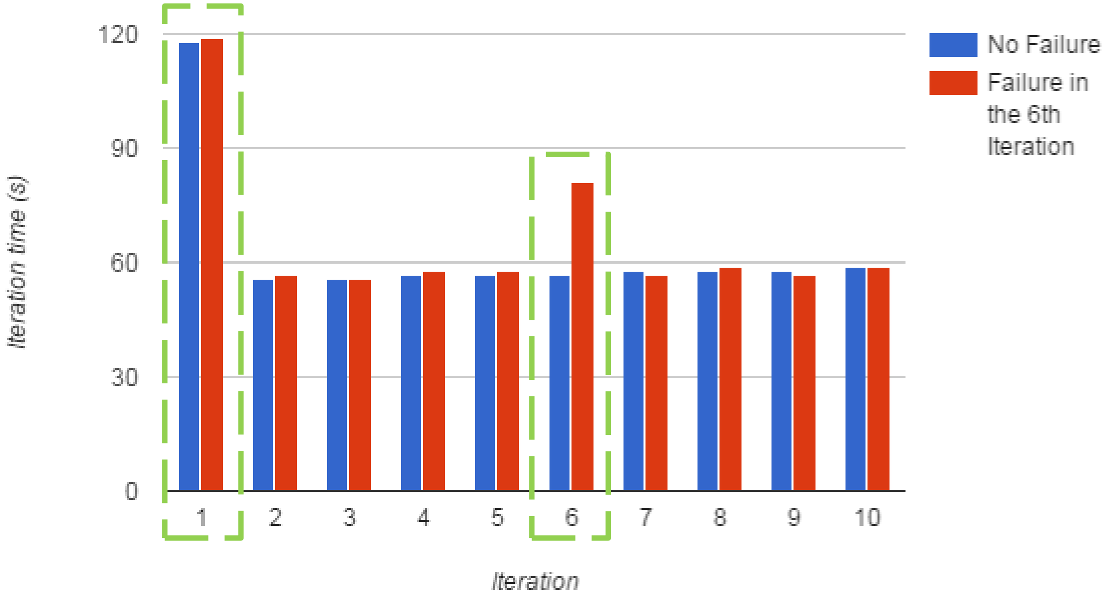
Evaluation

PageRank



Performance of PageRank on Hadoop and Spark

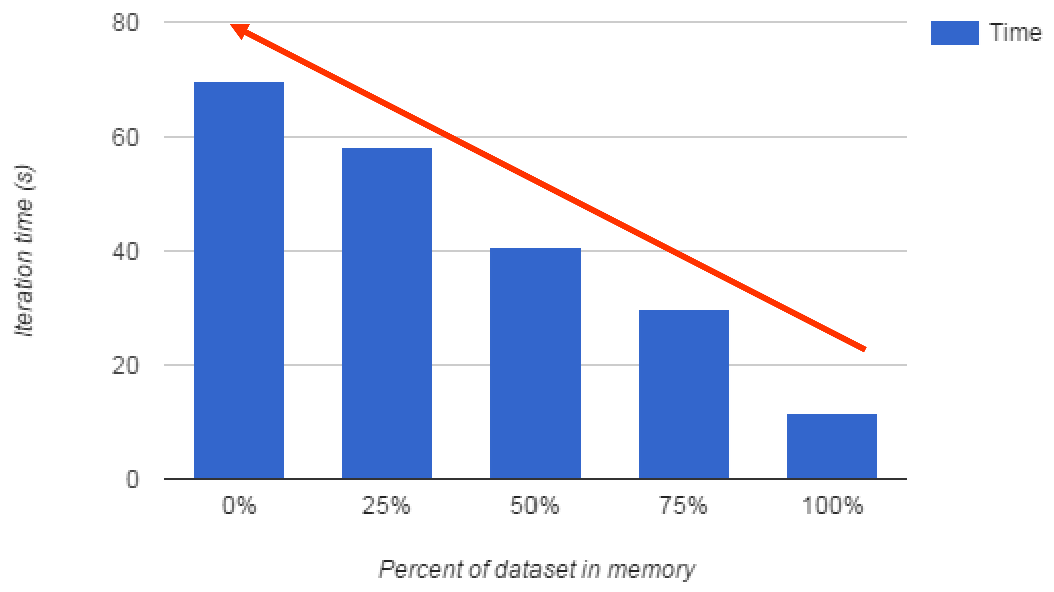
Fault Recovery



Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage

Evaluation

Behavior with Insufficient Memory



Performance of logistic regression using 100GB data on 25 machines with varying amounts of data in memory

Conclusion

- Resilient distributed datasets (RDDs) is an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications.
- RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage.
- Implement RDDs in Spark that outperforms Hadoop by up to 20x in iterative applications.