Spark

Fast, Interactive, Language-Integrated Cluster Computing

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

www.spark-project.org

amplab UC BERKELEY

Project Goals

Extend the MapReduce model to better support two common classes of analytics apps:

» **Iterative** algorithms (machine learning, graphs)

» **Interactive** data mining

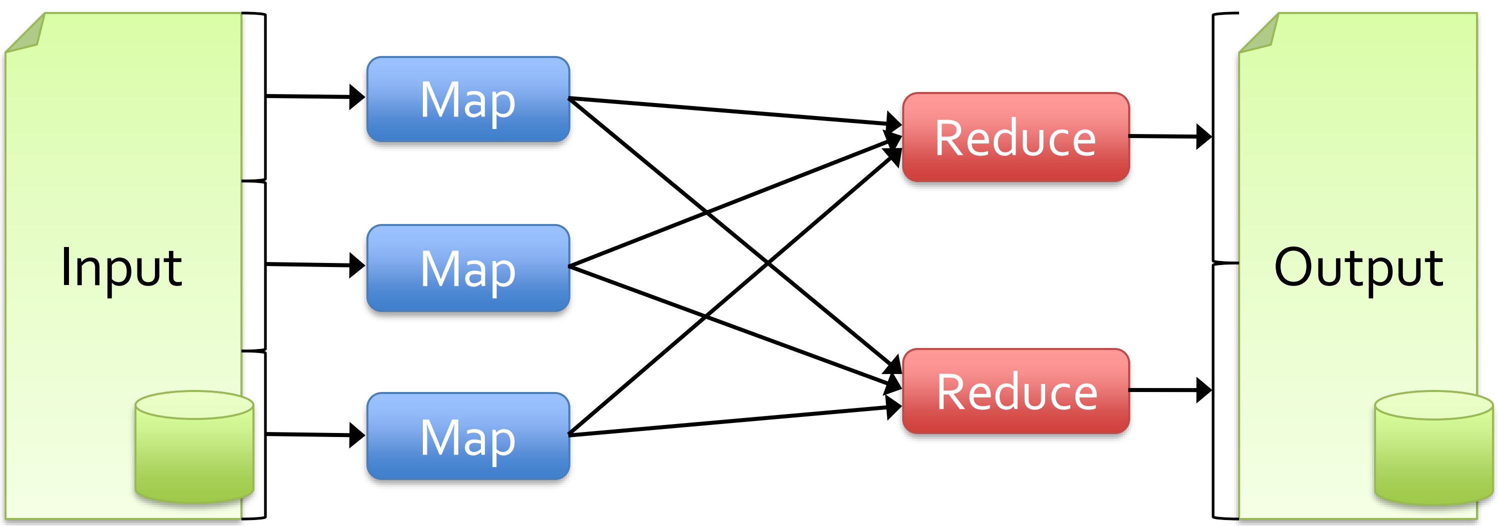
Enhance programmability:

» Integrate into Scala programming language

» Allow interactive use from Scala interpreter

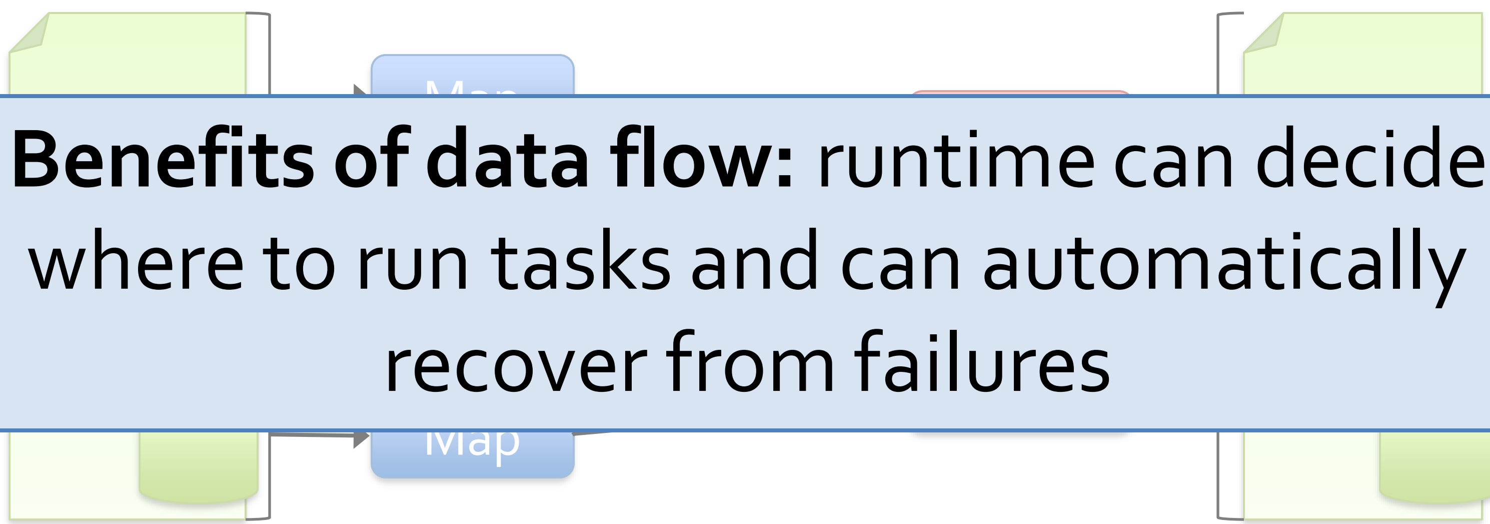
Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation

Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:

» **Iterative** algorithms (machine learning, graphs)
» **Interactive** data mining tools (R, Excel, Python)

With current frameworks, apps reload data from stable storage on each query

Solution: Resilient Distributed Datasets (RDDs)

Allow apps to keep working sets in memory for efficient reuse

Retain the attractive properties of MapReduce

» Fault tolerance, data locality, scalability

Support a wide range of applications

RDD

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing (2012)
- Most machine learning algorithms require iterative computation.
- The iterations on MapReduce cause big overhead between Map and Reduce
  - Data replication
  - Disk I/O
  - Serialization

RDD

- The iterations are computationally expensive since Hadoop uses HDFS for sharing data
- HDFS causes frequent file I/O → Slow
- Solutions
    - Reduce uses of file I/O
    - Use RAM

RDD

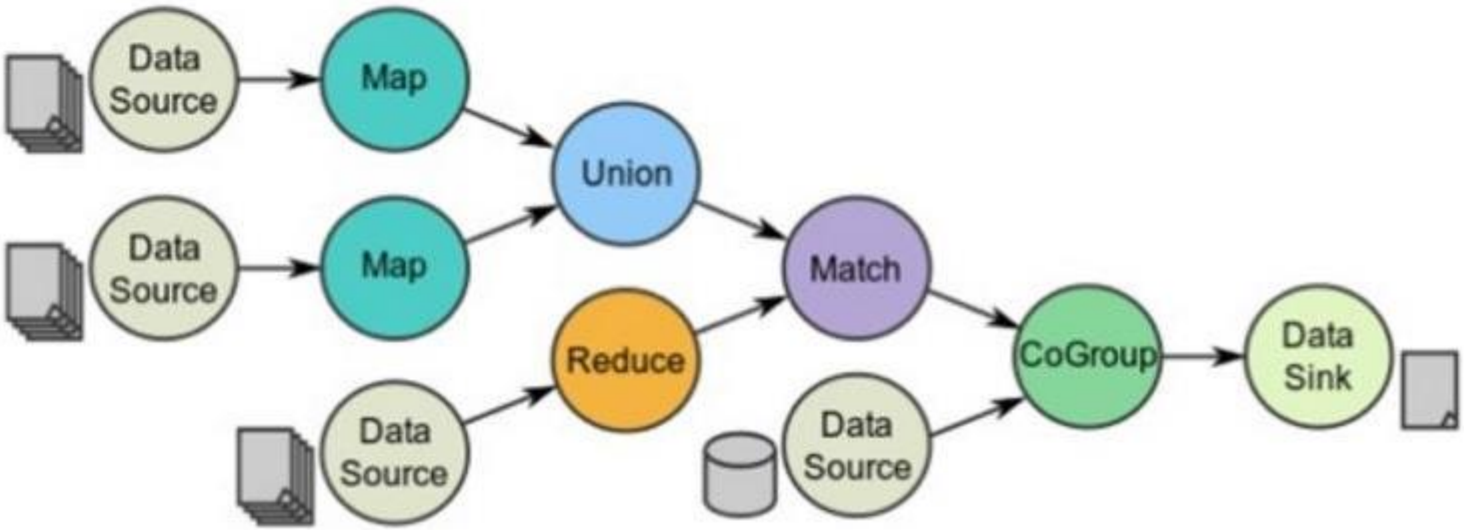
- Using RAM is much more efficient
  - However, how to handle fault-tolerant?
  - Need to load the data again into memory?
- Instead update data in RAM, make all data in RAM as read-only.

RDD

- Designed by Lineage and Directed Acyclic Graph (DAG)
  - RDD records all history of the process of the data
  - Fault-tolerant happens, RDD checks the lineage of the data and roll back → Fast recovery
  - All data is stored as DAG, so efficient.

RDD

- Lineage and DAG



Outline

Spark programming model

Implementation

Demo

User applications

Programming Model

Resilient distributed datasets (RDDs)

- » Immutable, partitioned collections of objects
- » Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
- » Can be *cached* for efficient reuse

Actions on RDDs

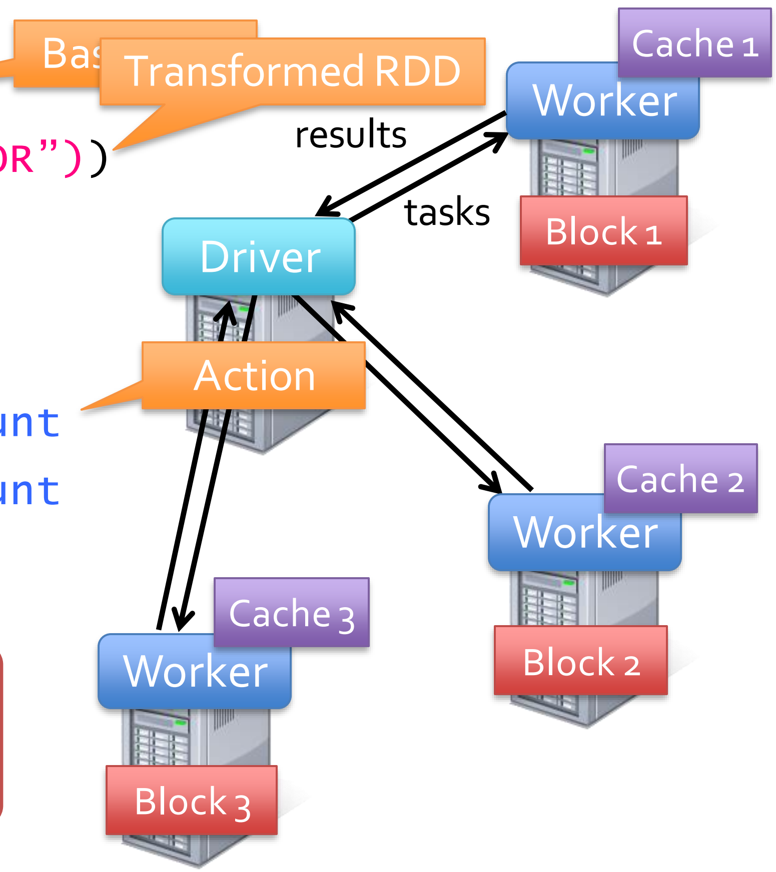
- » Count, reduce, collect, save, …

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
. . .
```

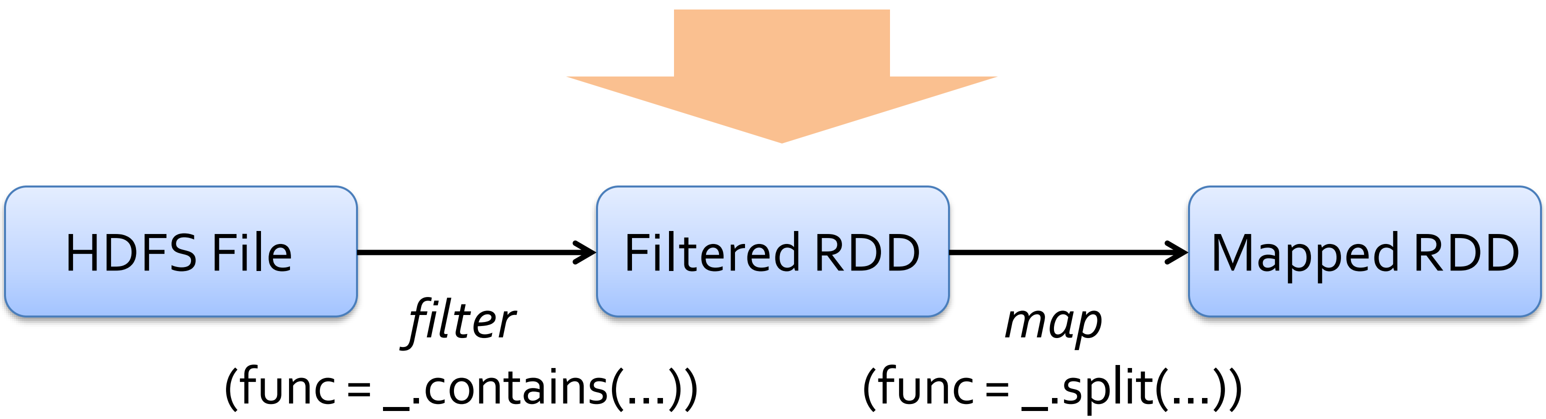


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

RDD Fault Tolerance

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

Ex:
```
messages = textFile(...).filter(_.startsWith(“ERROR”))
                        .map(_.split(‘\t’)(2))
```



Word Count

Use a few transformations to build a dataset of (String, int) pairs called counts and then save it to a file.

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

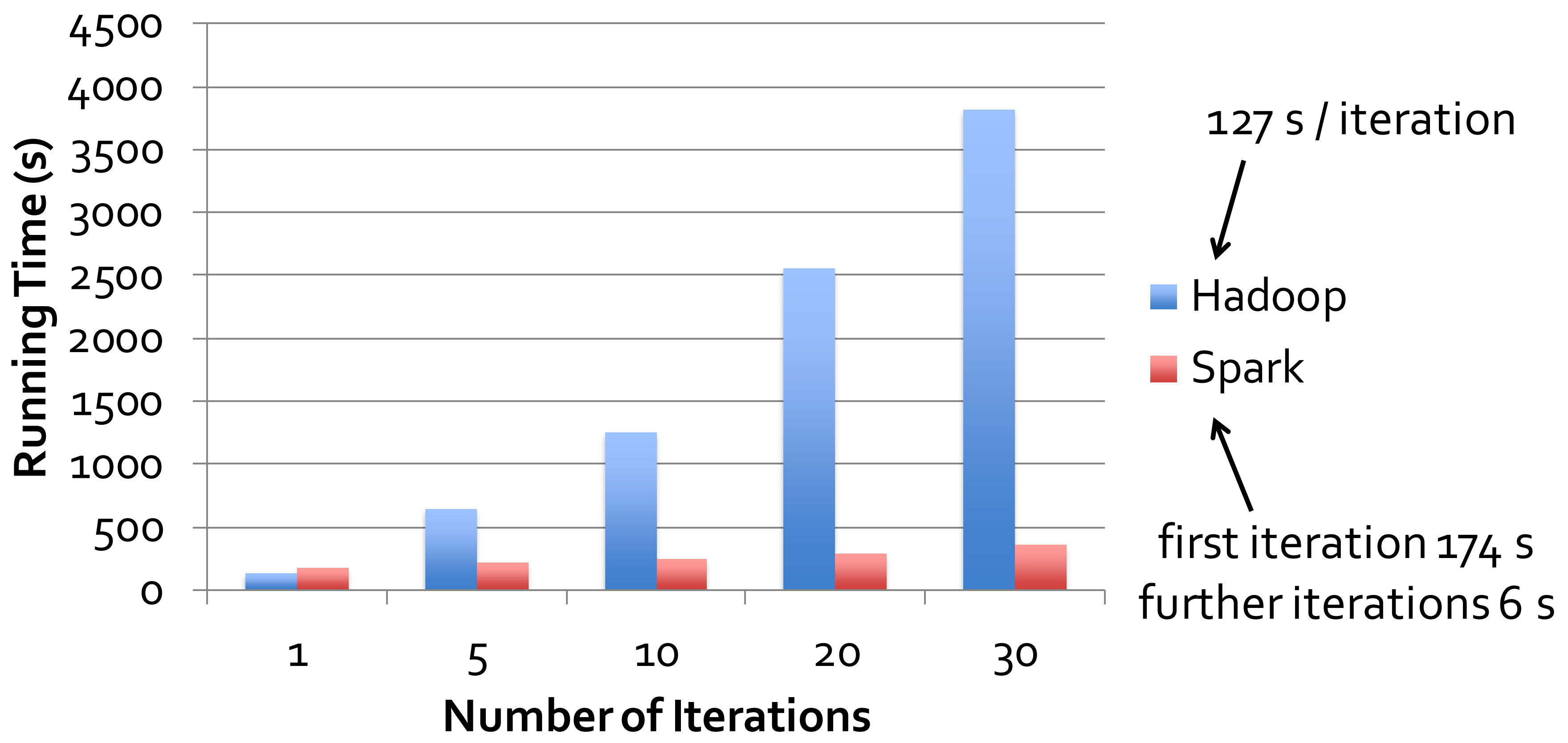
The map operation produces one output value for each input value, whereas the flatMap operation produces an arbitrary number (zero or more) values for each input value

Pi estimation

This code estimates π by "throwing darts" at a circle. We pick random points in the unit square ((0, 0) to (1,1)) and see how many fall in the unit circle. The fraction should be π / 4, so we use this to get our estimate.

```
val count = sc.parallelize(1 to NUM_SAMPLES).map{i =>
  val x = Math.random()
  val y = Math.random()
  if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / NUM_SAMPLES)
```

Logistic Regression Performance



Spark Applications

In-memory data mining on Hive data (Conviva)

Predictive analytics (Quantifind)

City traffic prediction (Mobile Millennium)

Twitter spam classification (Monarch)

Collaborative filtering via matrix factorization

…

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
- » Distribute generated classes over the network

Conclusion

Spark provides a simple, efficient, and powerful programming model for a wide range of apps

Download our open source release:

www.spark-project.org

matei@berkeley.edu

Related Work

DryadLINQ, FlumeJava

» Similar "distributed collection" API, but cannot reuse datasets efficiently *across* queries

Relational databases

» Lineage/provenance, logical logging, materialized views

GraphLab, Piccolo, BigTable, RAMCloud

» Fine-grained writes similar to distributed shared memory

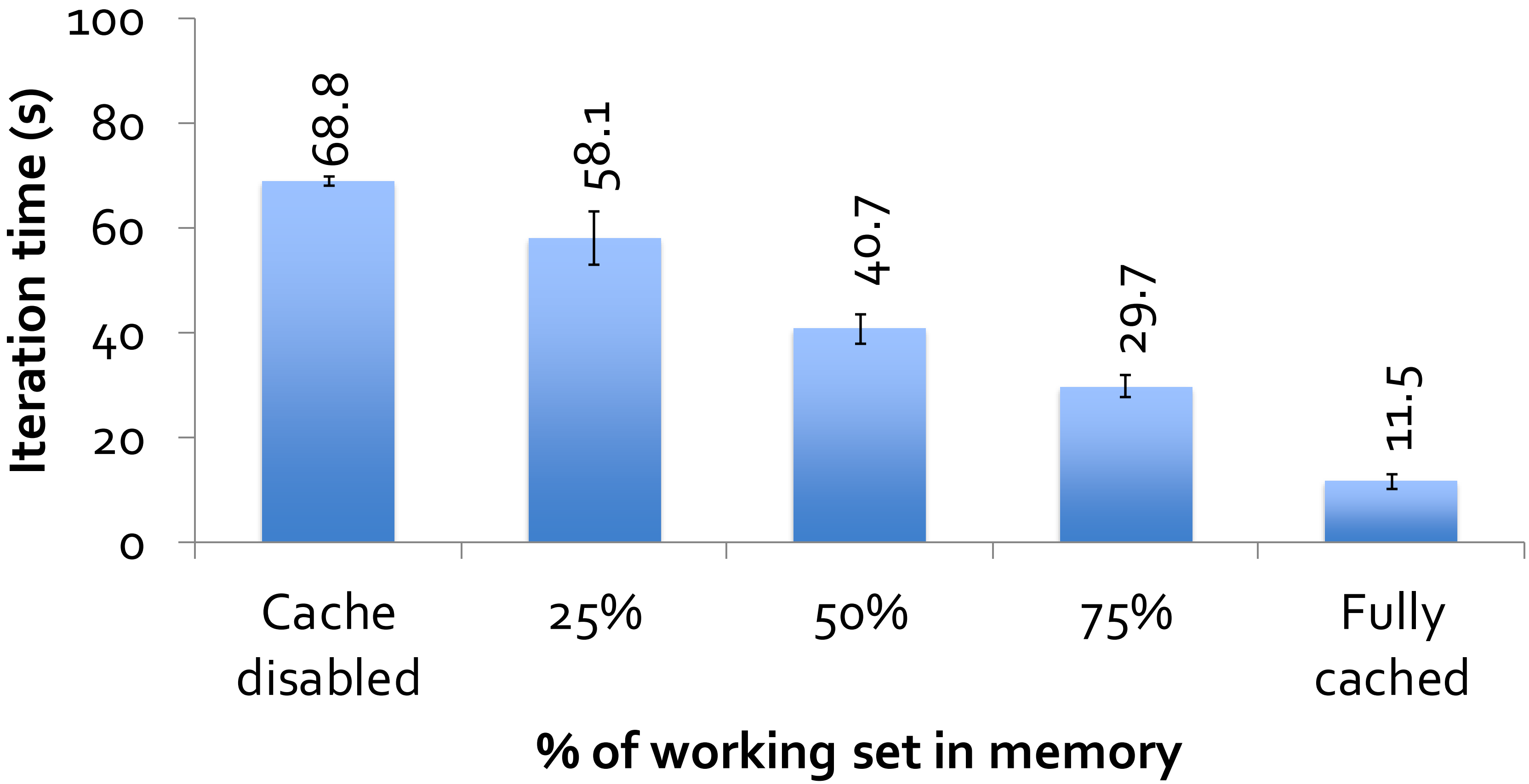
Iterative MapReduce (e.g. Twister, HaLoop)

» Implicit data sharing for a fixed computation pattern

Caching systems (e.g. Nectar)

» Store data in files, no explicit control over what is cached

Behavior with Not Enough RAM



Fault Recovery Results