Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

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

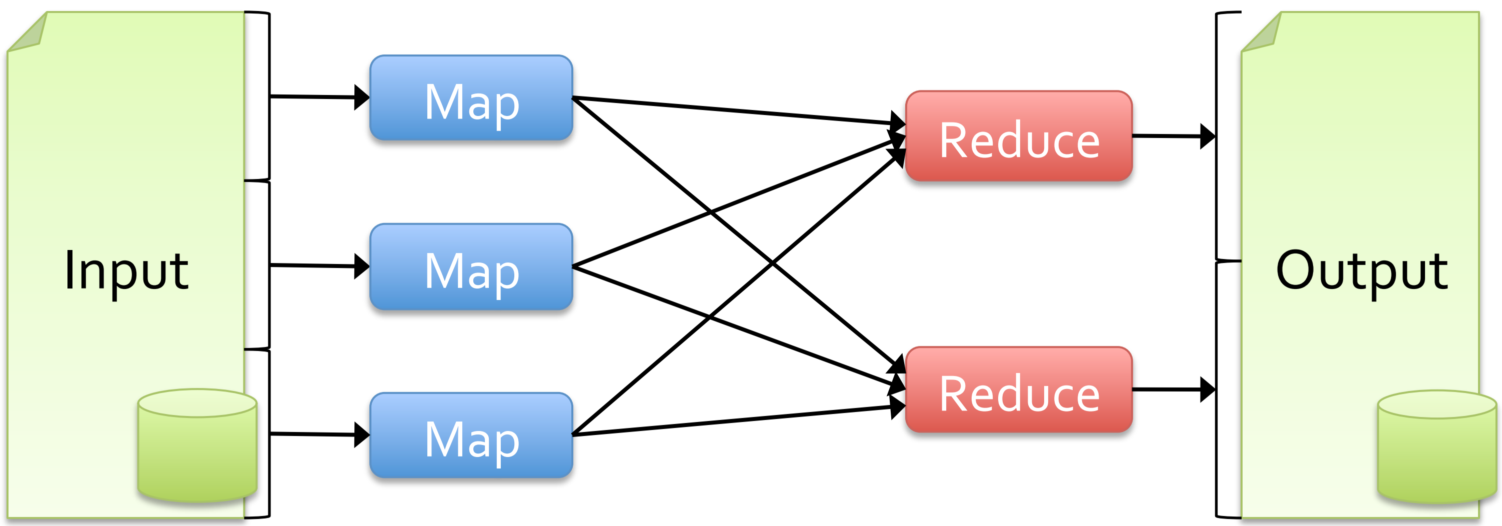
amplab
UC BERKELEY

Environment

amazon
web services™

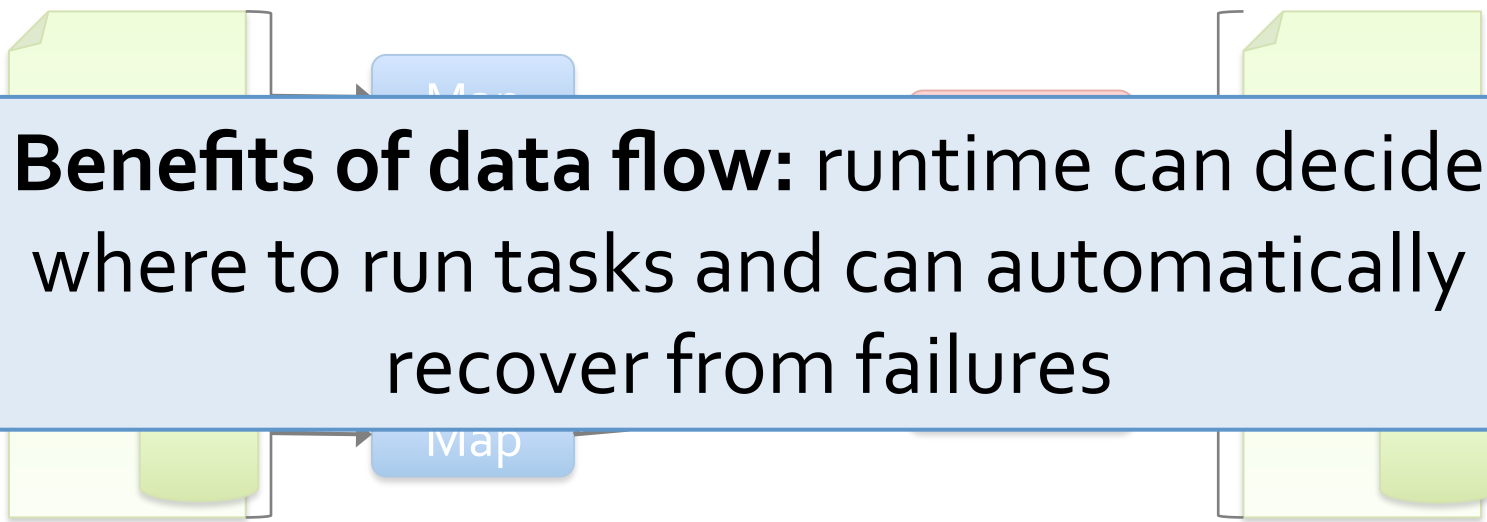
Motivation

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

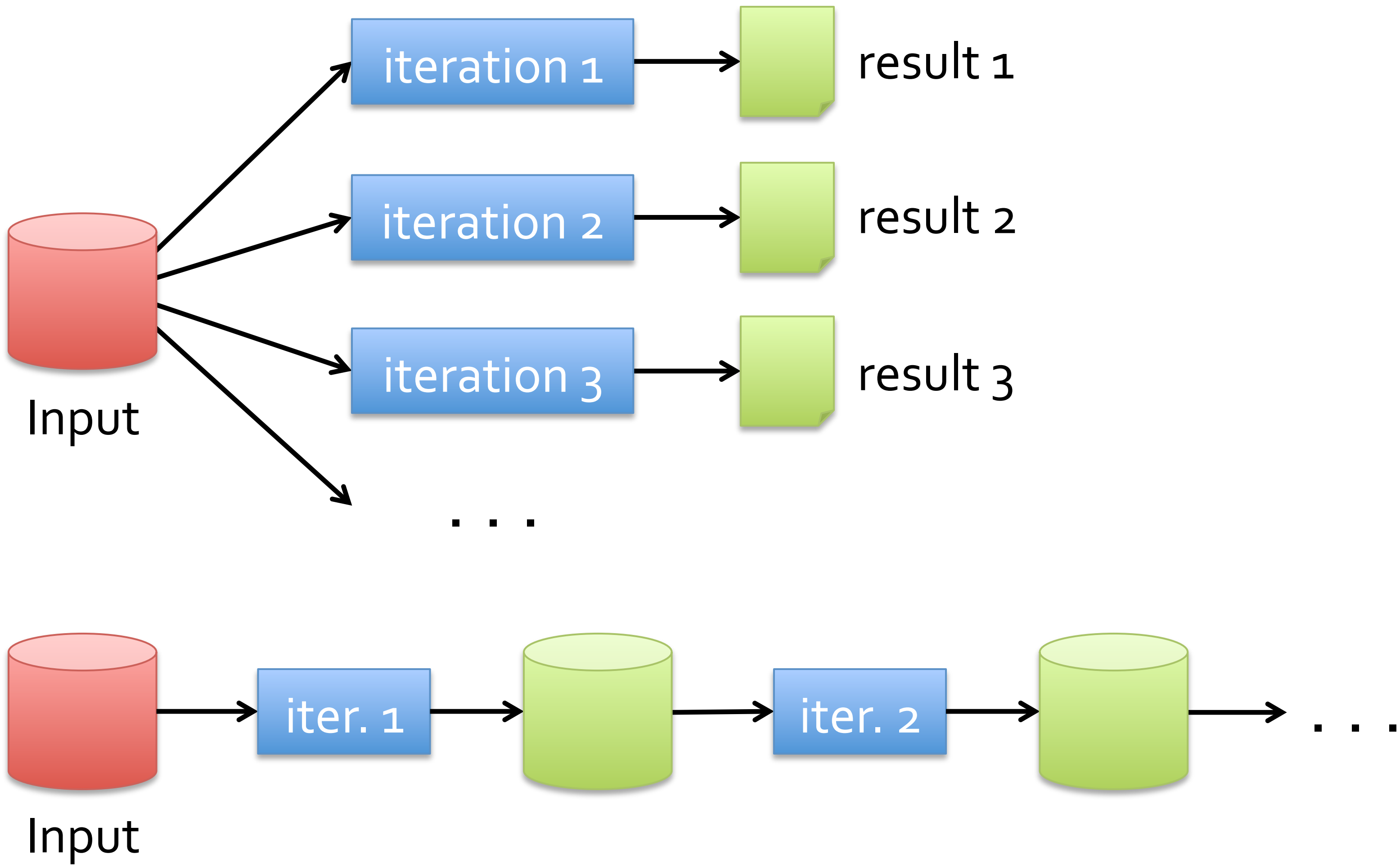
Acyclic data flow is inefficient for applications that repeatedly *reuse* a working set of data:

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

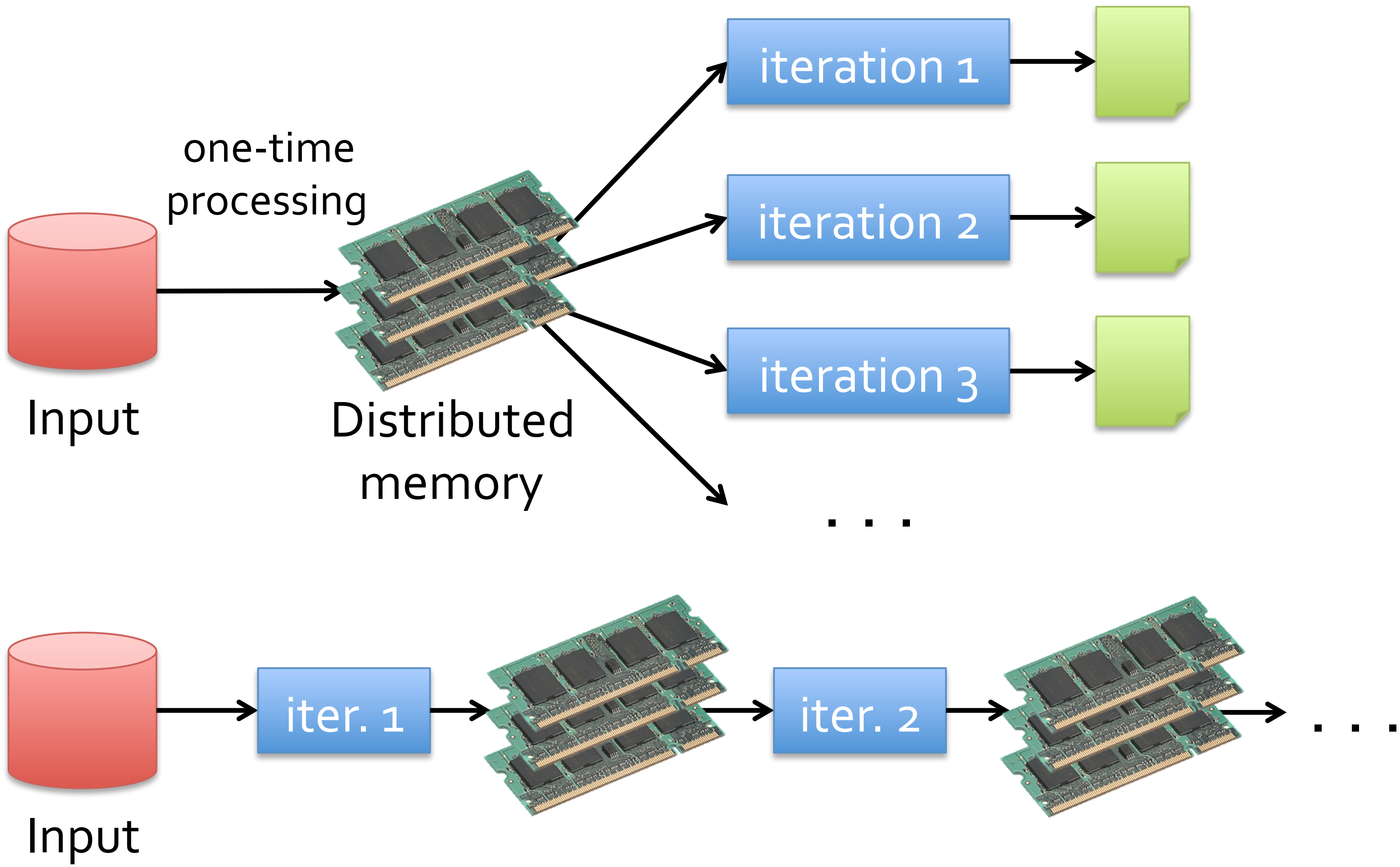
» **Interactive** data mining tools (R, Excel, Python)

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

Example: Iterative Apps



Goal: Keep Working Set in RAM



Challenge

How to design a distributed memory abstraction that is both ***fault-tolerant*** and ***efficient***?

Challenge

Existing distributed storage abstractions have interfaces based on *fine-grained* updates

» Reads and writes to cells in a table

» E.g. databases, key-value stores, distributed memory

Require replicating data or logs across nodes for fault tolerance ➔ expensive!

Solution: Resilient Distributed Datasets (RDDs)

Provide an interface based on *coarse-grained* transformations (map, group-by, join, …)

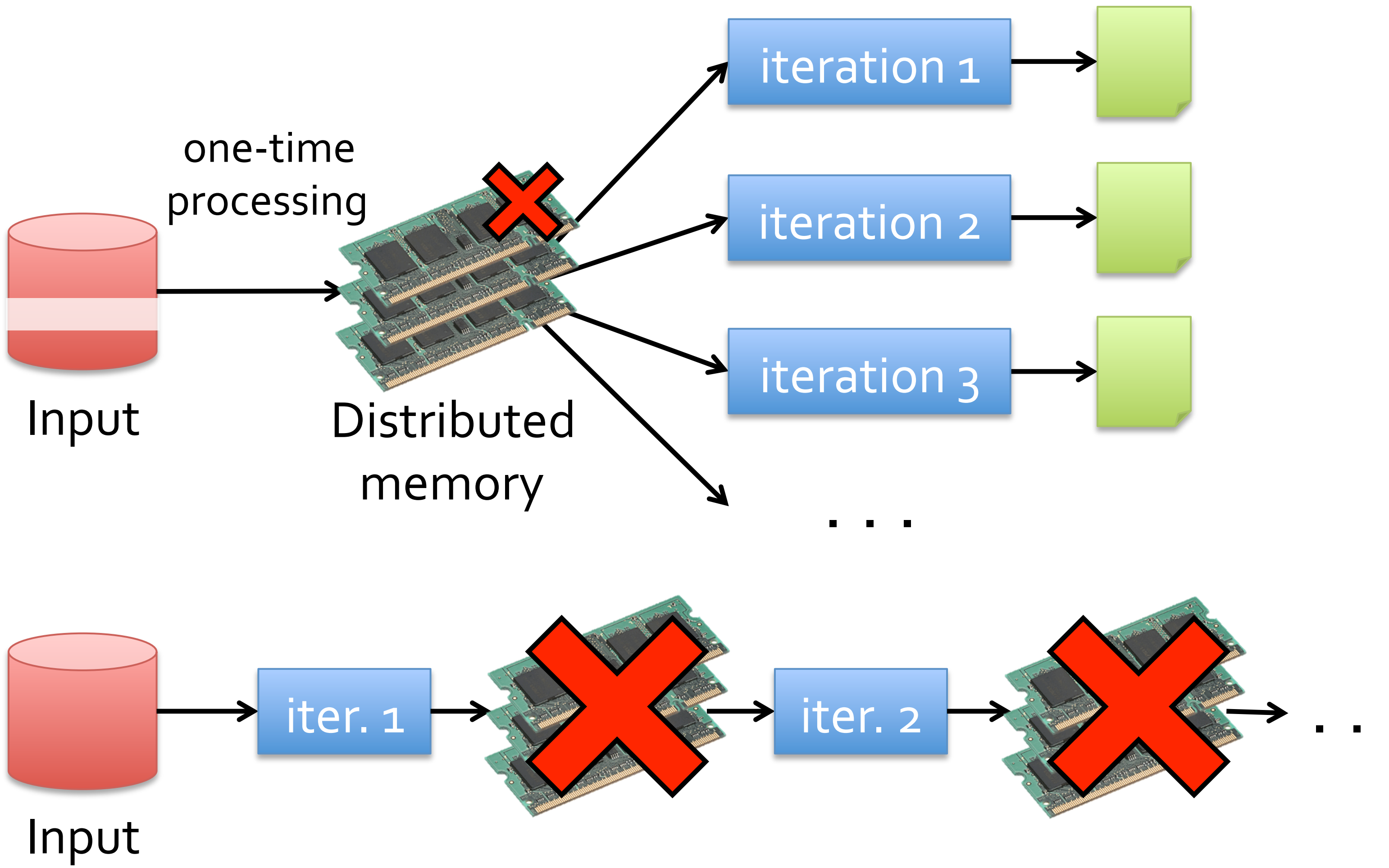
Efficient fault recovery using *lineage*

» Log one operation to apply to many elements

» Recompute lost partitions on failure

» No cost if nothing fails

RDD Recovery



Generality of RDDs

Despite coarse-grained interface, RDDs can express surprisingly many parallel algorithms

» These naturally *apply the same operation to many items*

Capture many current programming models

» **Data flow models**: MapReduce, Dryad, SQL, …

» **Specialized models** for iterative apps: BSP (Pregel), iterative MapReduce, bulk incremental

» Also support new apps that these models don’t

Outline

Programming interface

Applications

Implementation

Demo

Spark Programming Interface

Language-integrated API in Scala

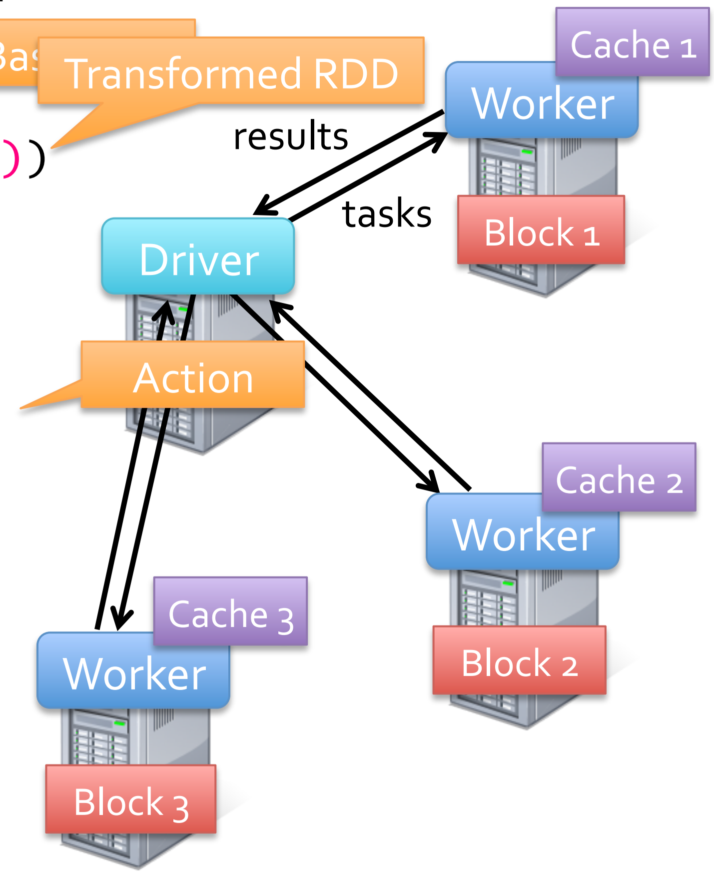
Provides:

- » Resilient distributed datasets (RDDs)
  - Partitioned collections with controllable caching
- » Operations on RDDs
  - Transformations (define RDDs), actions (compute results)
- » Restricted shared variables (broadcast, accumulators)

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
. . .
```

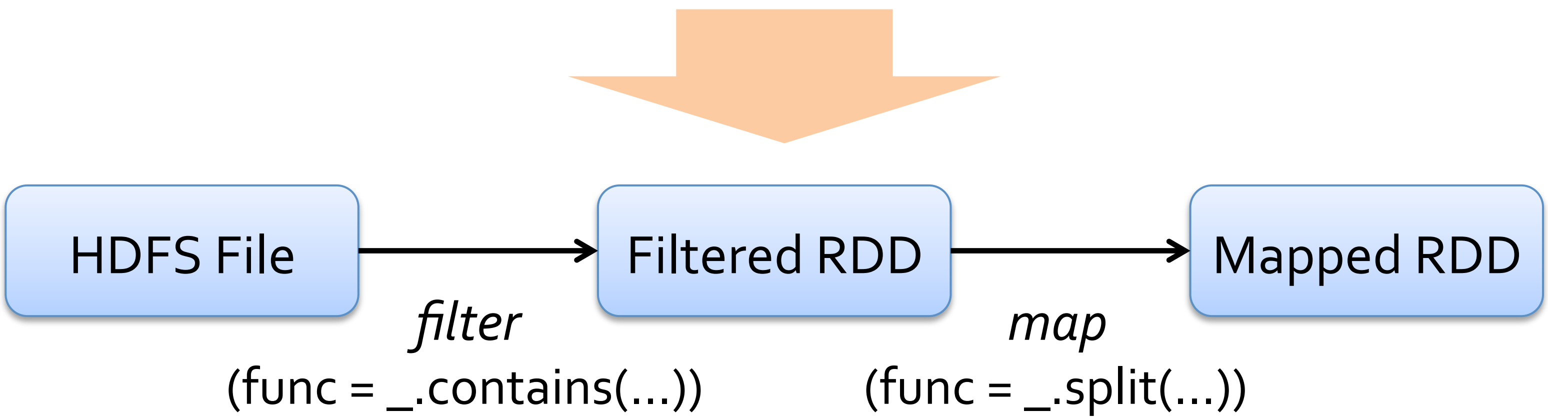


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

Fault Tolerance

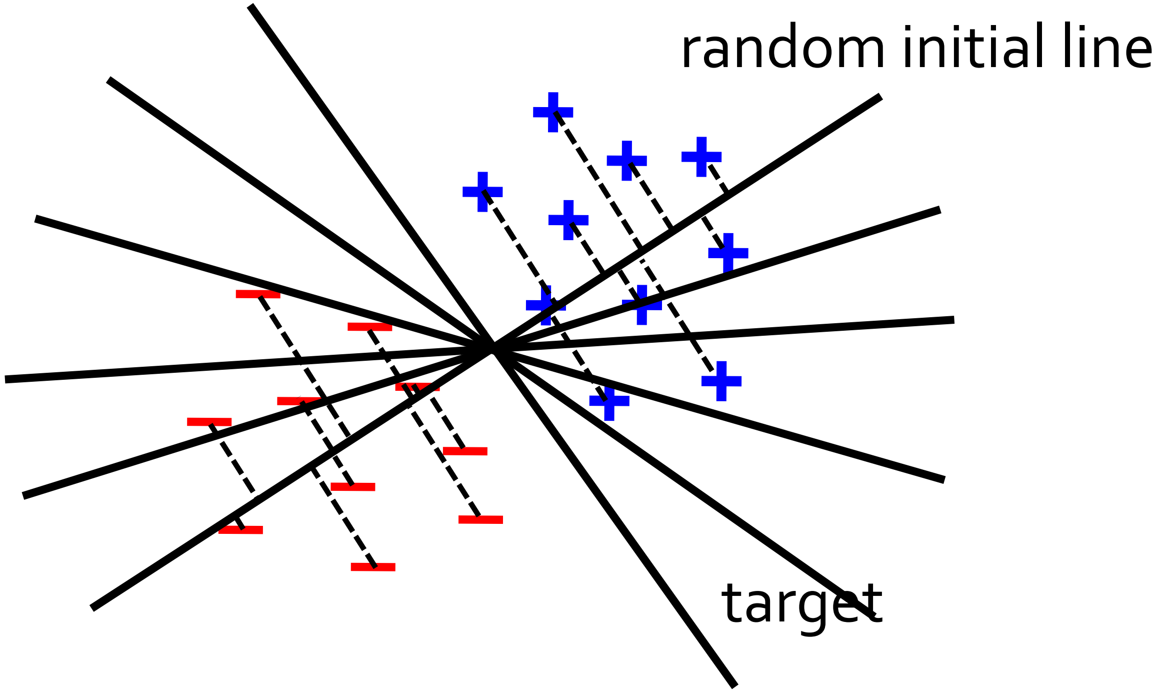
RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex:
```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



Example: Logistic Regression

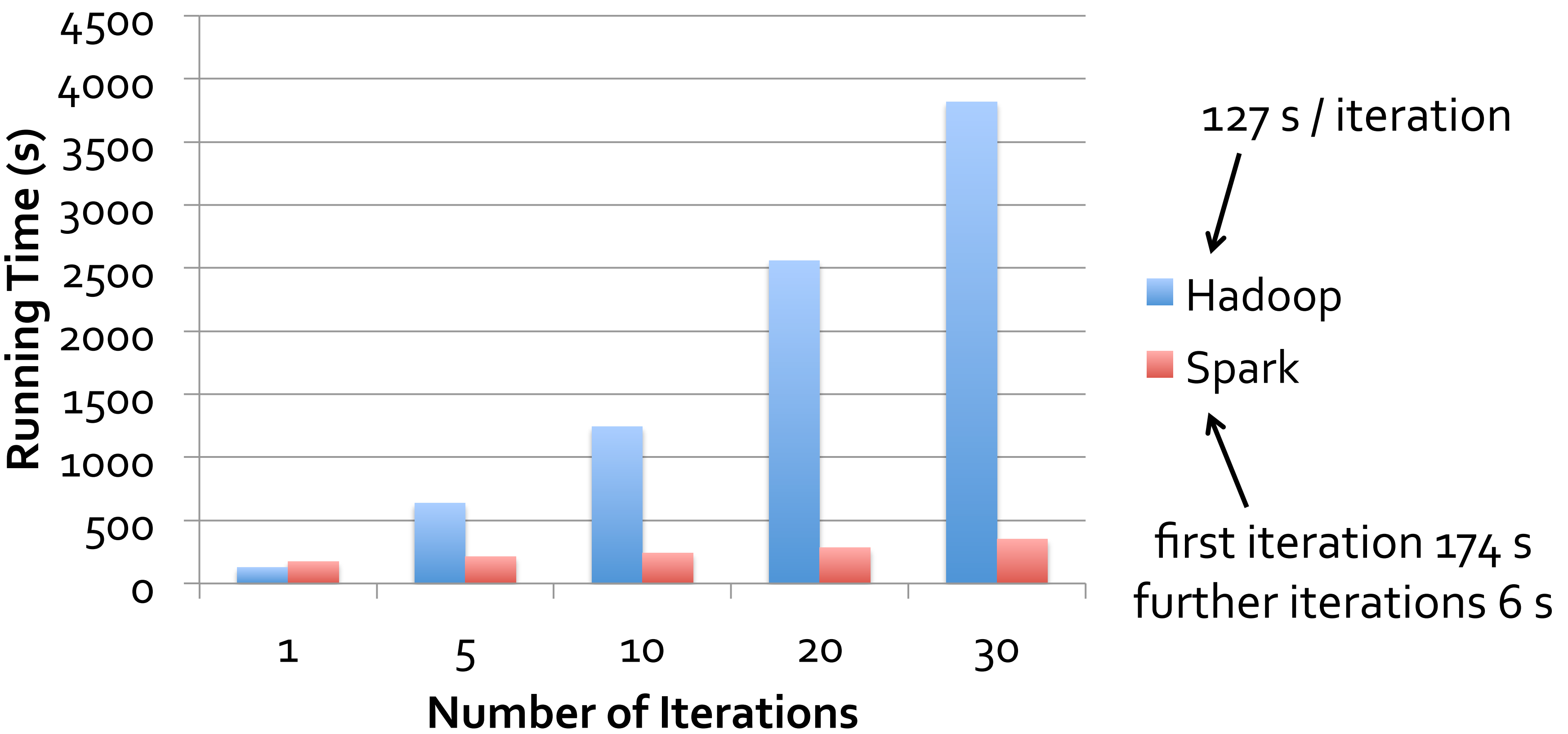
Goal: find best line separating two sets of points



Example: Logistic Regression

```
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: Collaborative Filtering

Goal: predict users' movie ratings based on past ratings of other movies

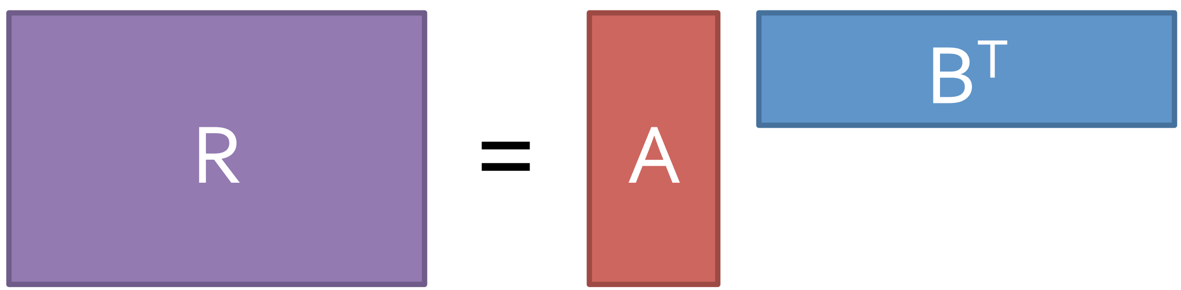
$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

Users

Movies

Model and Algorithm

Model R as product of user and movie feature matrices A and B of size U×K and M×K



Alternating Least Squares (ALS)

» Start with random A & B

» Optimize user vectors (A) based on movies

» Optimize movie vectors (B) based on users

» Repeat until converged

Serial ALS

```
var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
  A = (0 until U).map(i => updateUser(i, B, R))
  B = (0 until M).map(i => updateMovie(i, A, R))
}
```

Range objects

Naïve Spark ALS

```
var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updateUser(i, B, R))
           .collect()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R))
           .collect()
}
```

Problem:
R re-sent
to all nodes
in each
iteration

Efficient Spark ALS

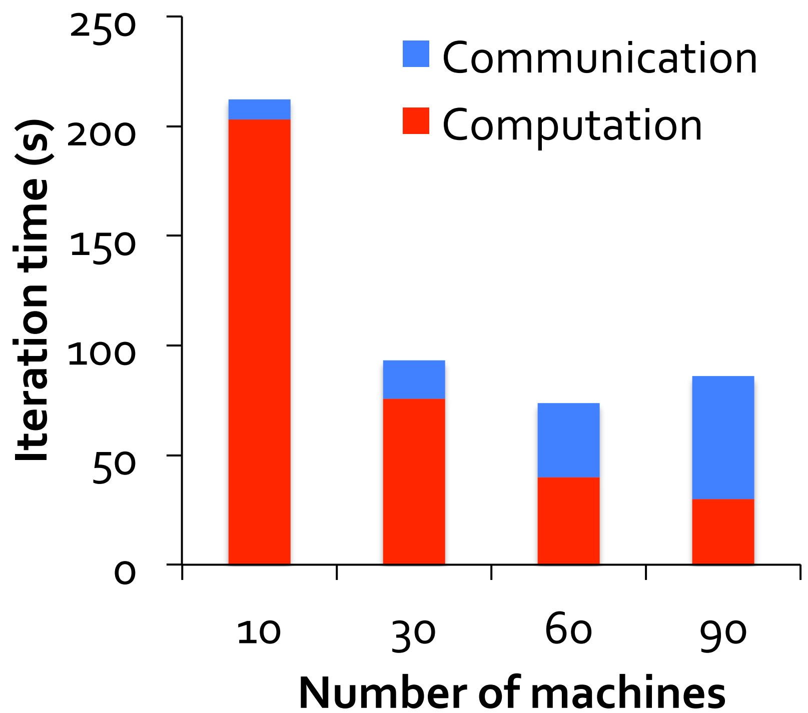
```
var R = spark.broadcast(readRatingsMatrix(...))

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updateUser(i, B, R.value))
           .collect()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R.value))
           .collect()
}
```

Solution: mark R as broadcast variable

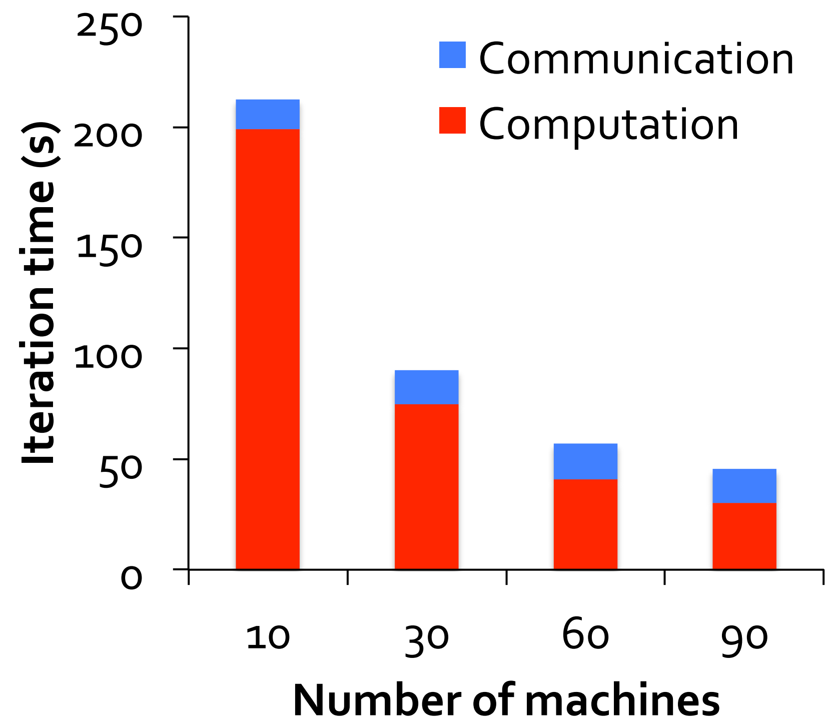
Result: 3× performance improvement

Scaling Up Broadcast

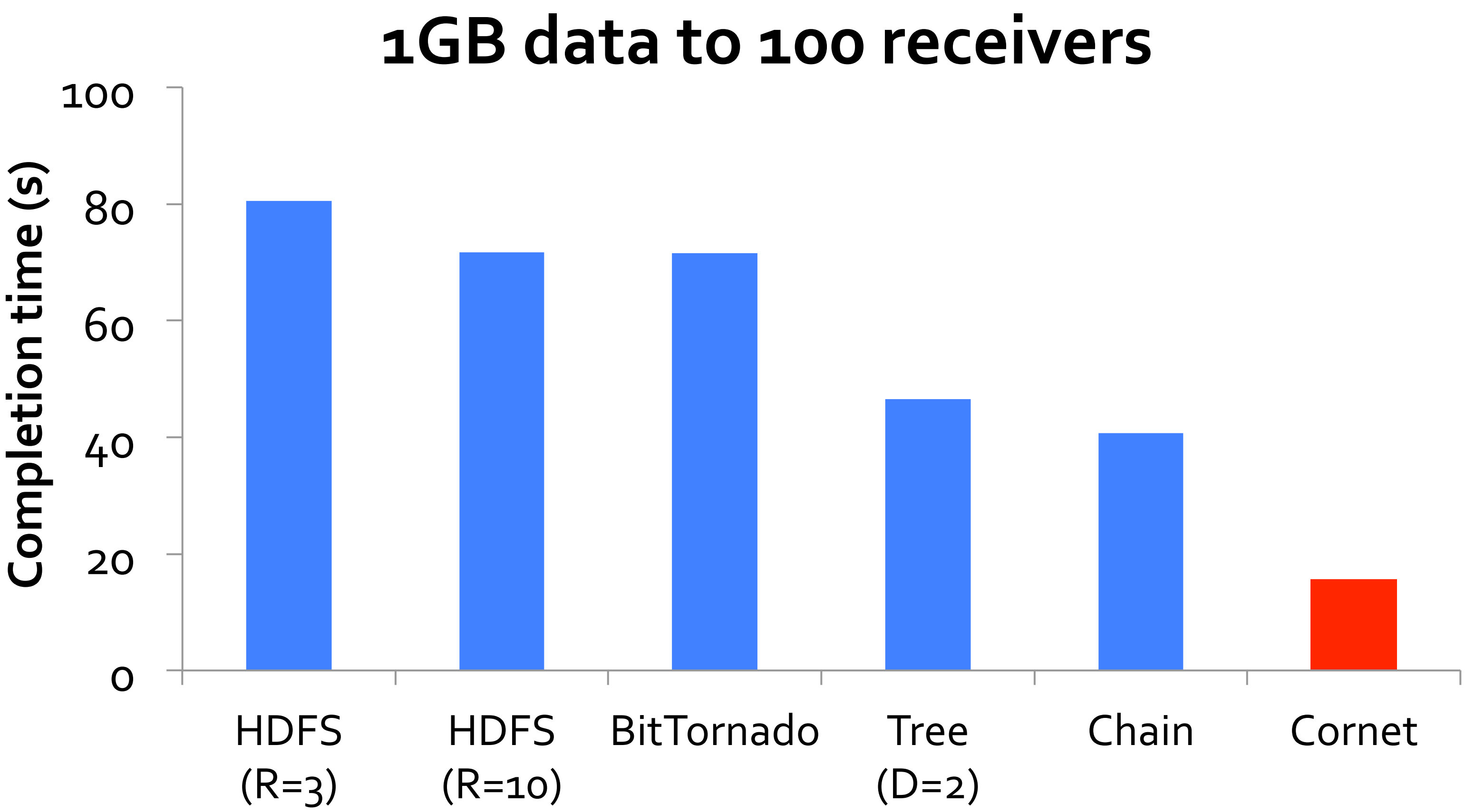
Initial version (HDFS)



Cornet broadcast



Cornet Performance



[Chowdhury et al, SIGCOMM 2011]

Spark Applications

EM alg. for traffic prediction (Mobile Millennium)

Twitter spam classification (Monarch)

In-memory OLAP & anomaly detection (Conviva)

Time series analysis

Network simulation

…

Mobile Millennium Project

Estimate city traffic using GPS observations from probe vehicles
(e.g. SF taxis)

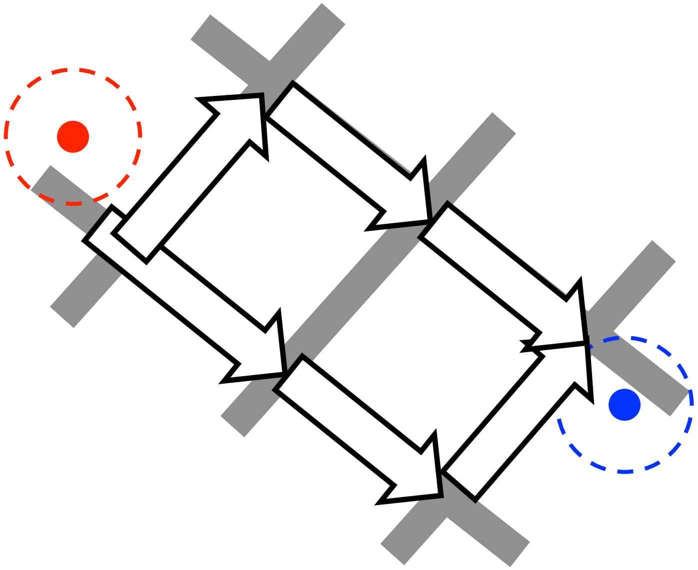
Sample Data

Credit: Tim Hunter, with support of the Mobile Millennium team; P.I. Alex Bayen; traffic.berkeley.edu

Challenge

Data is noisy and sparse (1 sample/minute)

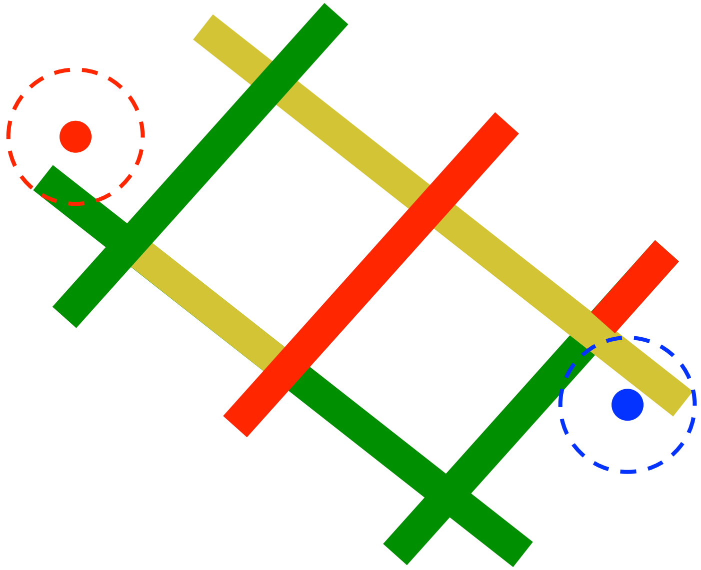
Must infer path taken by each vehicle in addition to travel time distribution on each link



Challenge

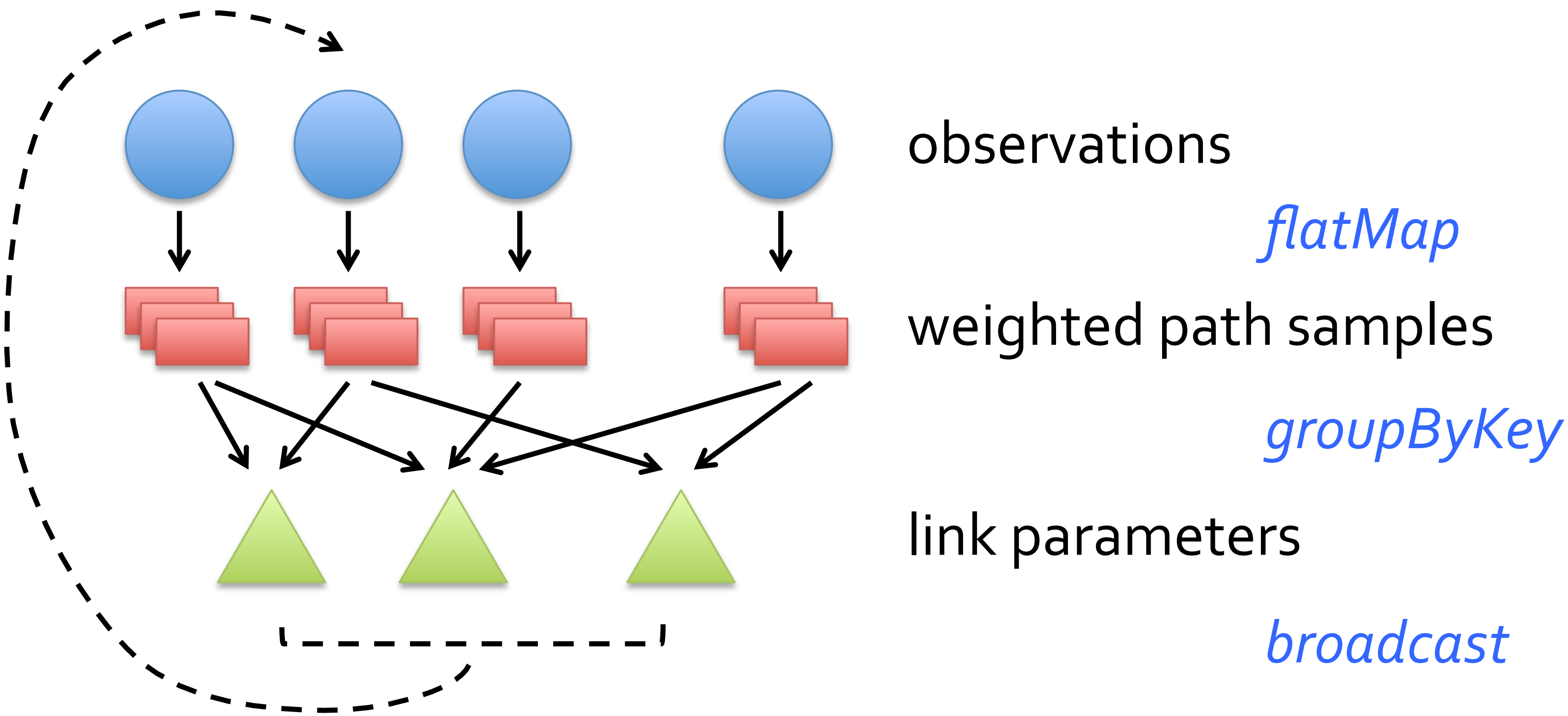
Data is noisy and sparse (1 sample/minute)

Must infer path taken by each vehicle in addition to travel time distribution on each link



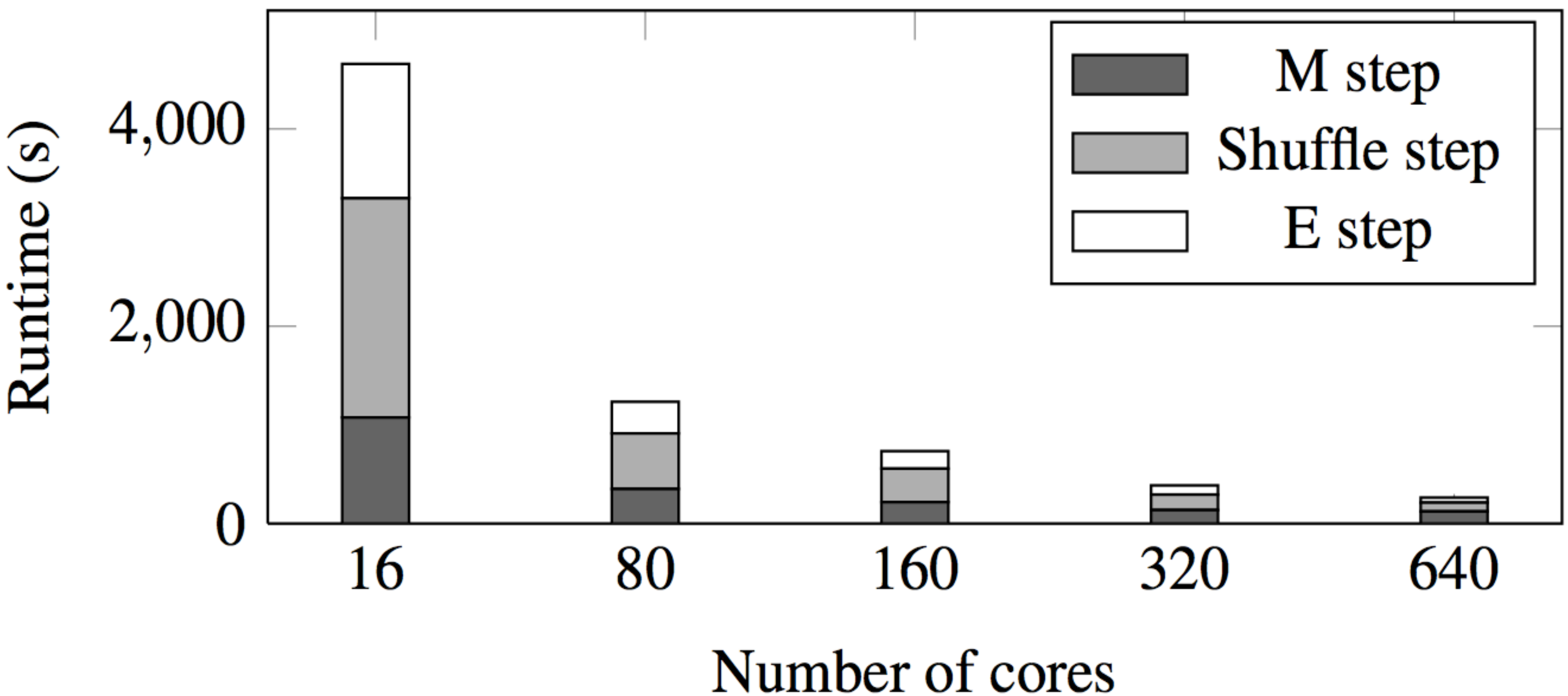
Solution

EM algorithm to estimate paths and travel time distributions simultaneously



Results

[Hunter et al, SOCC 2011]



3× speedup from caching, 4.5x from broadcast

Cluster Programming Models

RDDs can express many proposed data-parallel programming models

- » **MapReduce, DryadLINQ**
- » **Bulk incremental processing**
- » **Pregel** graph processing
- » **Iterative MapReduce** (e.g. Haloop)
- » **SQL**

Allow apps to efficiently *intermix* these models

Models We Have Built

Pregel on Spark (Bagel)

» 200 lines of code

Haloop on Spark

» 200 lines of code

Hive on Spark (Shark)

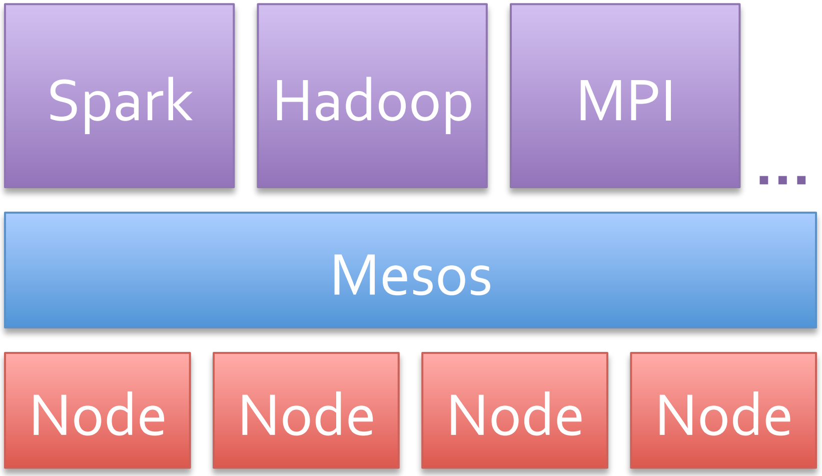
» 3000 lines of code

» Compatible with Apache Hive

» ML operators in Scala

Implementation

Spark runs on the Mesos cluster manager [NSDI 11], letting it share resources with Hadoop & other apps



Can read from any Hadoop input source (HDFS, S3, …)

No changes to Scala language & compiler

Outline

Programming interface

Applications

Implementation

Demo

Conclusion

Spark's RDDs offer a simple and efficient programming model for a broad range of apps

Solid foundation for higher-level abstractions

Join our open source community:

www.spark-project.org

Related Work

DryadLINQ, FlumeJava
- » Similar “distributed collection” API, but cannot reuse datasets efficiently *across* queries

GraphLab, Piccolo, BigTable, RAMCloud
- » Fine-grained writes requiring replication or checkpoints

Iterative MapReduce (e.g. Twister, HaLoop)
- » Implicit data sharing for a fixed computation pattern

Relational databases
- » Lineage/provenance, logical logging, materialized views

Caching systems (e.g. Nectar)
- » Store data in files, no explicit control over what is cached

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

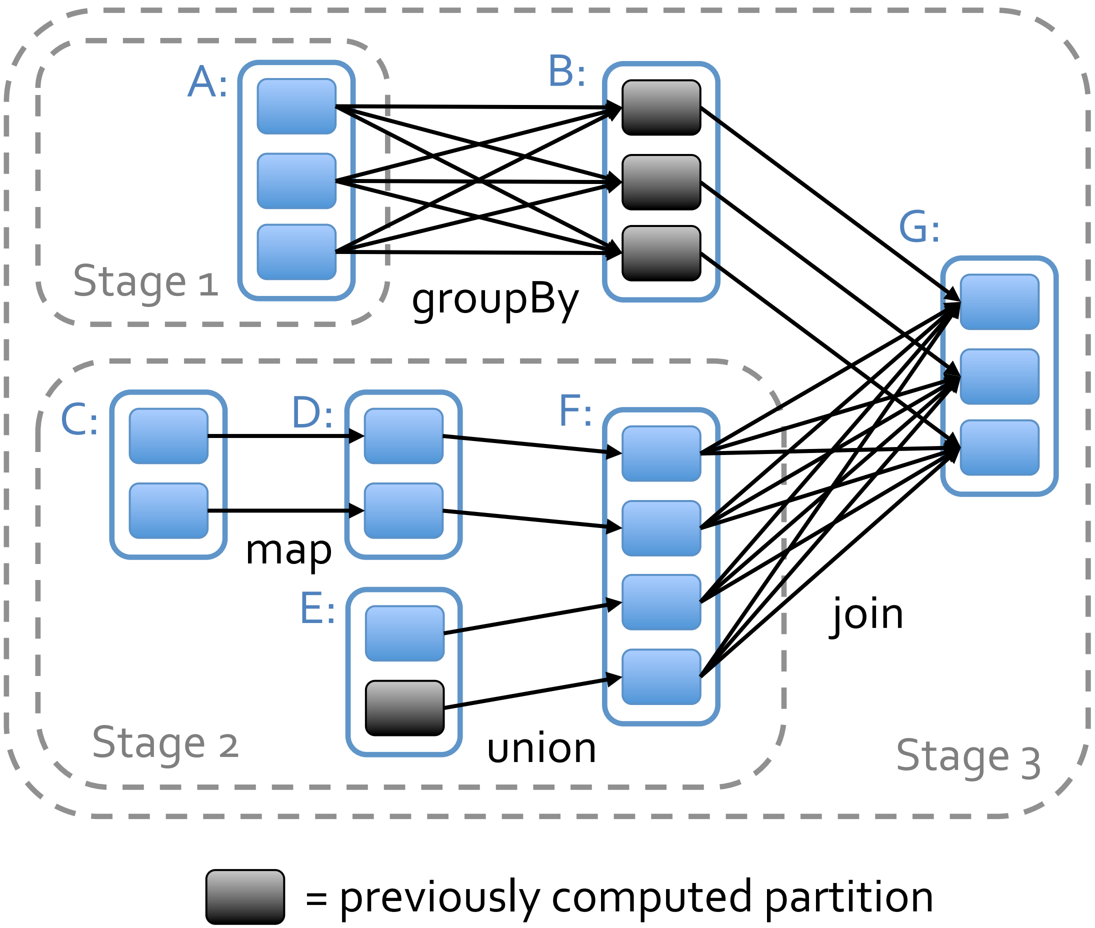
Job Scheduler

Dryad-like task DAG

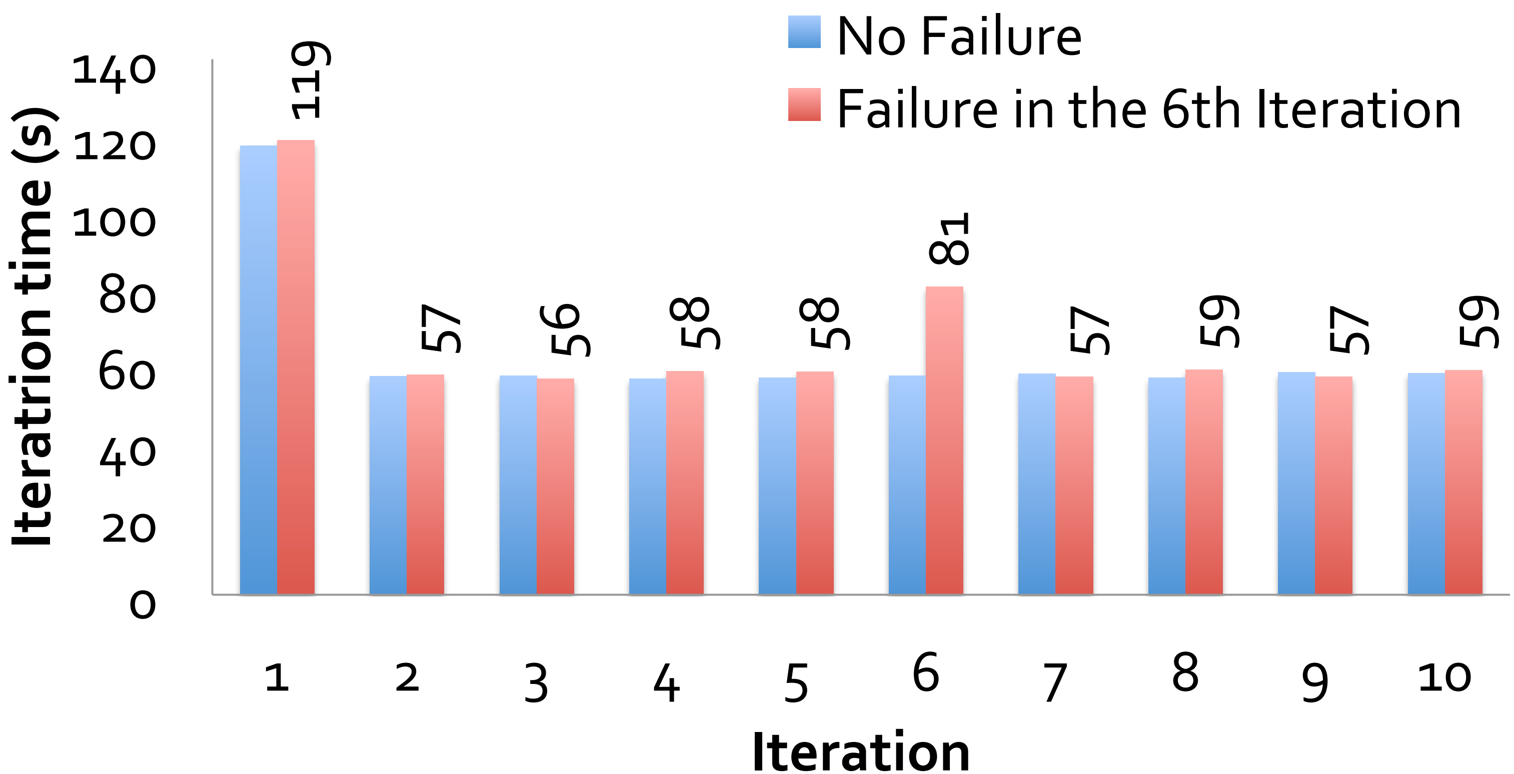
Reuses previously computed data

Partitioning-aware to avoid shuffles

Automatic pipelining



Fault Recovery Results



Behavior with Not Enough RAM