Distributed Computing with Spark

Reza Zadeh

Stanford

DATABRICKS

Spark

Thanks to Matei Zaharia

Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Numerical computing on Spark

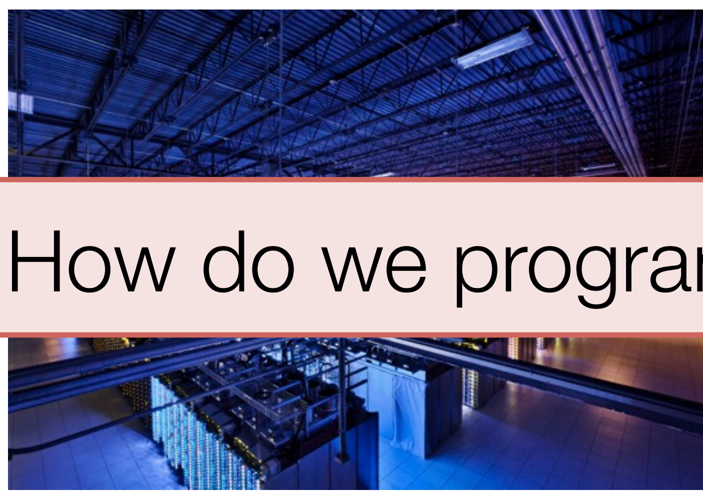
Ongoing work

Problem

Data growing faster than processing speeds

Only solution is to parallelize on large clusters

- » Wide use in both enterprises and web industry



How do we program these things?

Traditional Network Programming

Message-passing between nodes (e.g. MPI)

Very difficult to do at scale:

» How to split problem across nodes?

- Must consider network & data locality

» How to deal with failures? (inevitable at scale)

» Even worse: stragglers (node not failed, but slow)

» Ethernet networking not fast

Rarely used in commodity datacenters

Data Flow Models

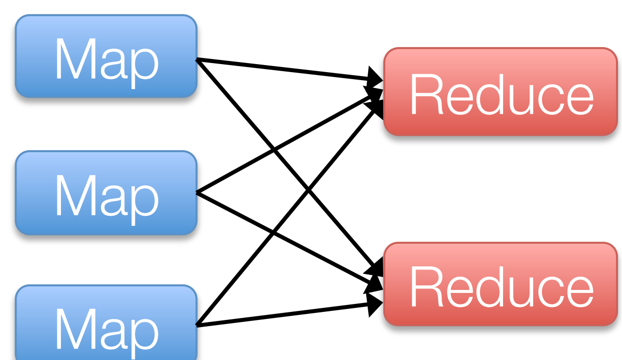
Restrict the programming interface so that the system can do more automatically

Express jobs as graphs of high-level operators

» System picks how to split each operator into tasks and where to run each task

» Run parts twice fault recovery

Biggest example: MapReduce



MapReduce Numerical Algorithms

Matrix-vector multiplication

Power iteration (e.g. PageRank)

Gradient descent methods

Stochastic SVD

Tall skinny QR

Many others!

Why Use a Data Flow Engine?

Ease of programming

» High-level functions instead of message passing

Wide deployment

» More common than MPI, especially “near” data

Scalability to very largest clusters

» Even HPC world is now concerned about resilience

Examples: Pig, Hive, Scalding, Storm

Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Numerical computing on Spark

Ongoing work

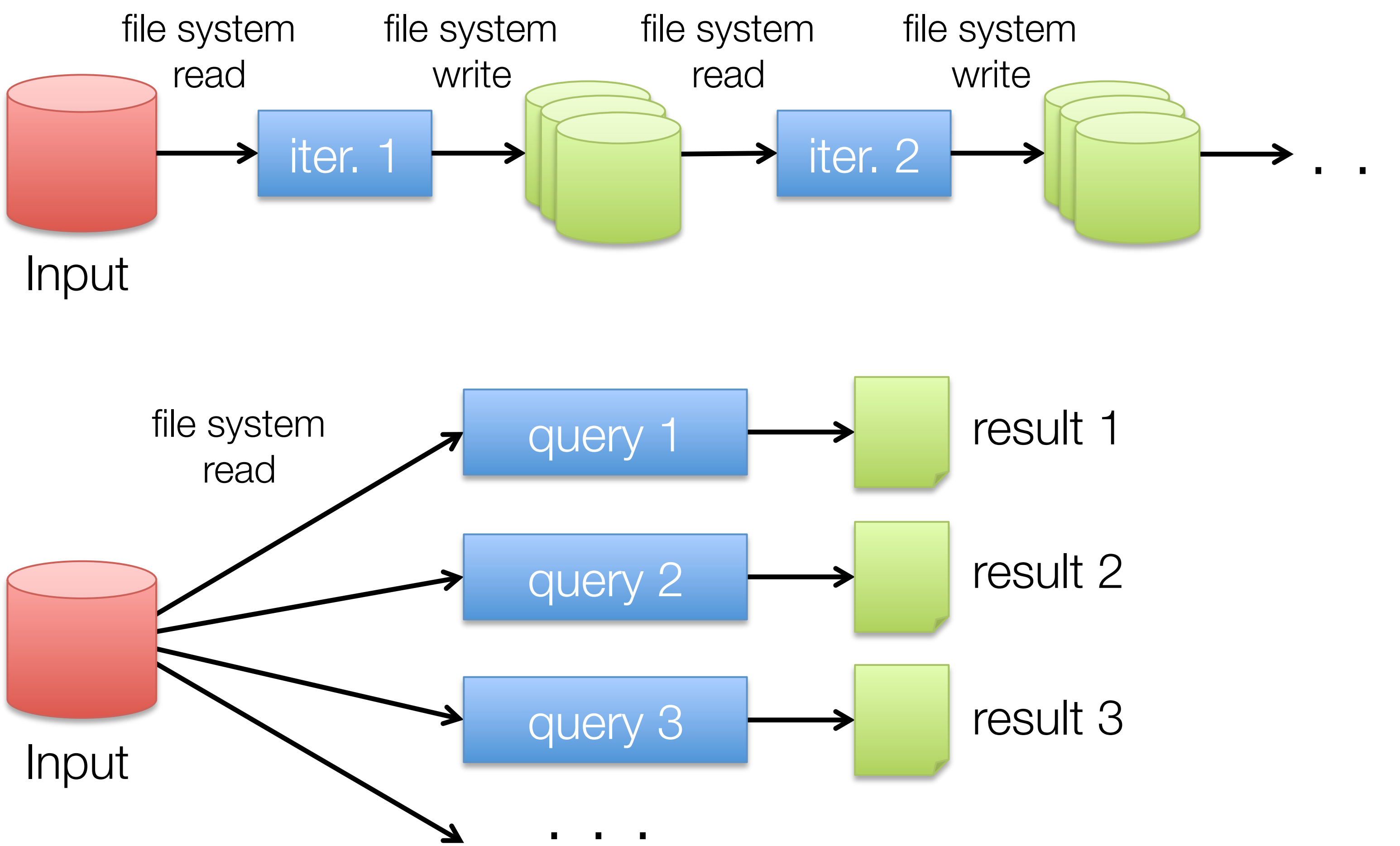
Limitations of MapReduce

MapReduce is great at one-pass computation, but inefficient for *multi-pass* algorithms

No efficient primitives for data sharing

- State between steps goes to distributed file system
- Slow due to replication & disk storage

Example: Iterative Apps

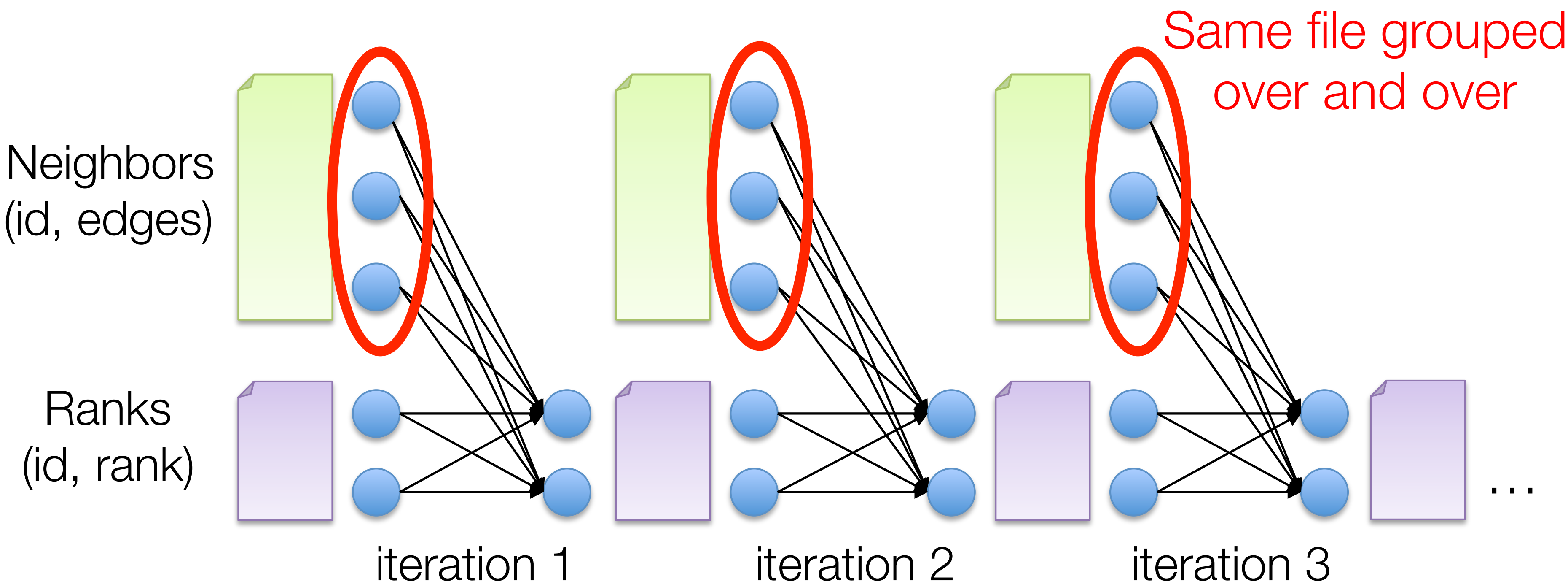


Commonly spend 90% of time doing I/O

Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector



Result

While MapReduce is simple, it can require *asymptotically* more communication or I/O

Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Numerical computing on Spark

Ongoing work

Spark Computing Engine

Extends MapReduce model with primitives for efficient data sharing

- » "Resilient distributed datasets"

Open source at Apache

- » Most active community in big data, with 50+ companies contributing

Clean APIs in Java, Scala, Python

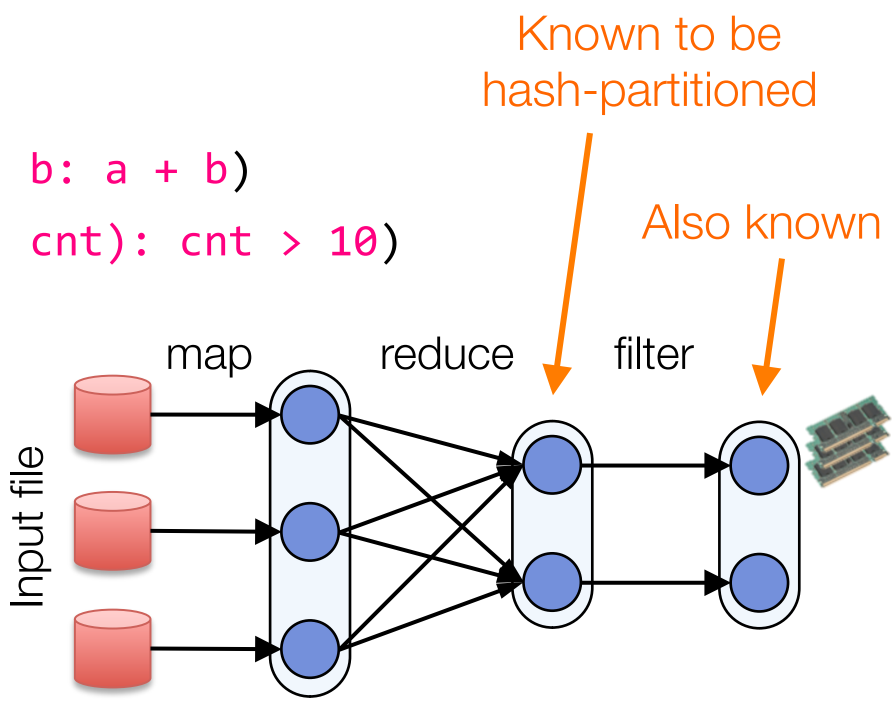
Resilient Distributed Datasets (RDDs)

Collections of objects stored across a cluster

User-controlled partitioning & storage (memory, disk, …)

Automatically rebuilt on failure

```
urls = spark.textFile(“hdfs://...”)
records = urls.map(lambda s: (s, 1))
counts = records.reduceByKey(lambda a, b: a + b)
bigCounts = counts.filter(lambda (url, cnt): cnt > 10)


bigCounts.cache()

bigCounts.filter(
  lambda (k,v): “news” in k).count()

bigCounts.join(otherPartitionedRDD)
```



Key Idea

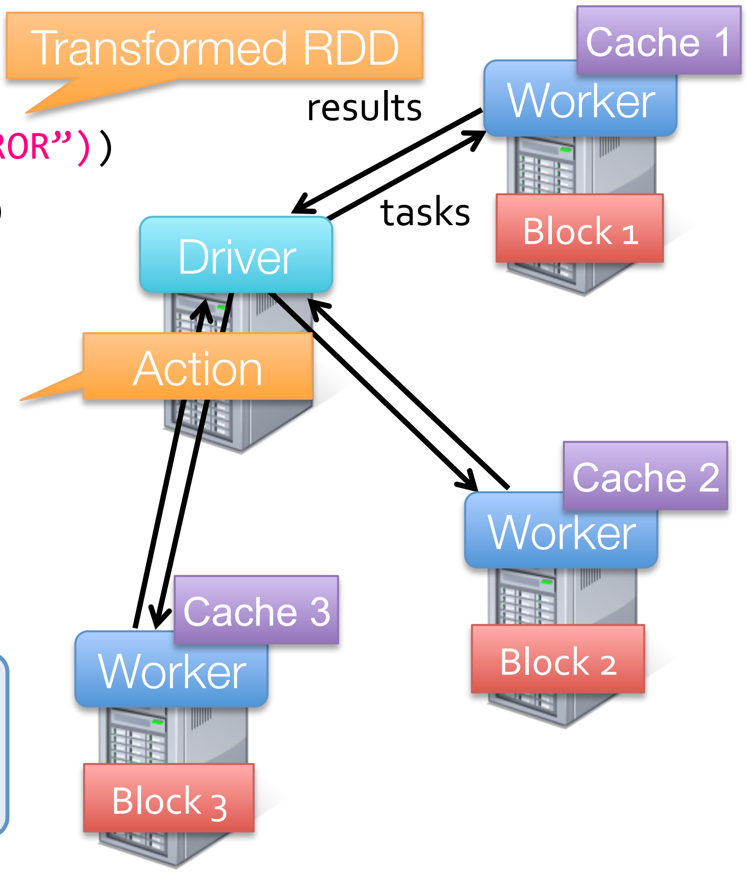
Resilient Distributed Datasets (RDDs)

- » Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
- » Built via parallel transformations (map, filter, …)
- » Automatically rebuilt on failure

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```

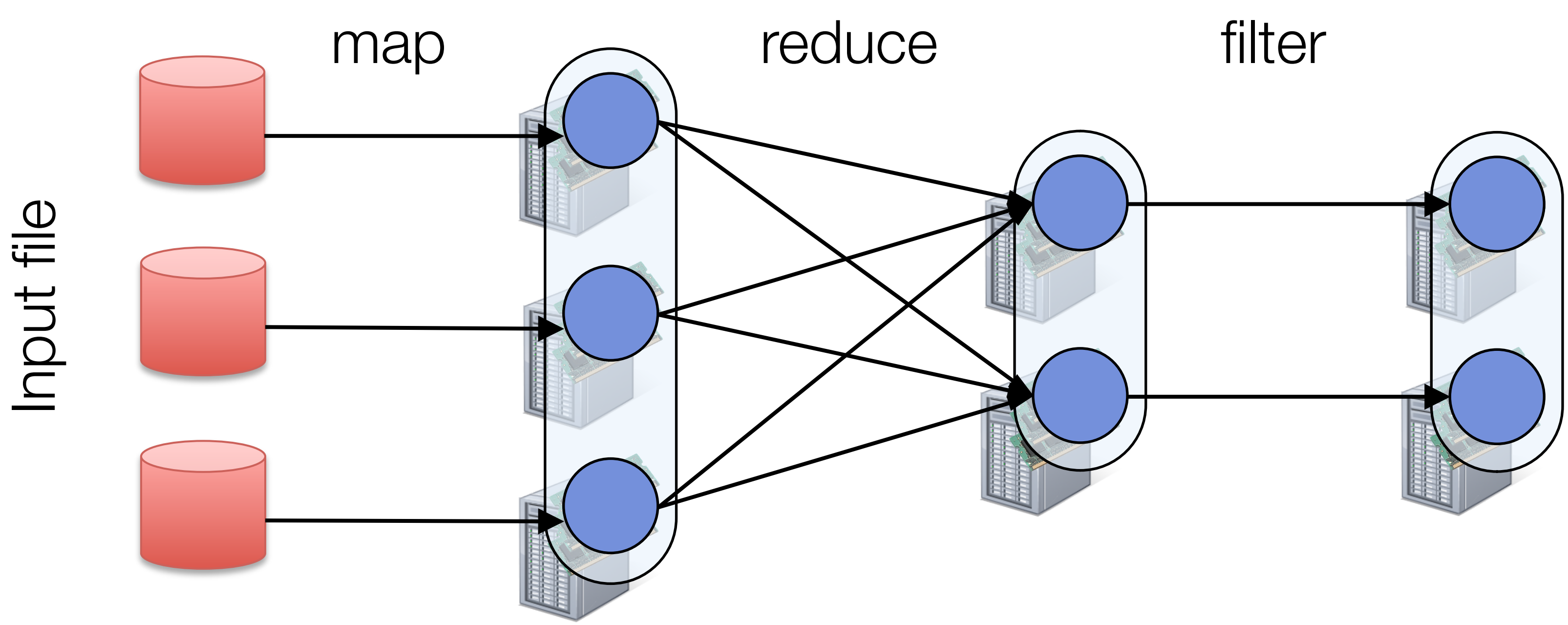


Result: full-text search of Wikipedia in 0.5 sec (vs 20 s for on-disk data)

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

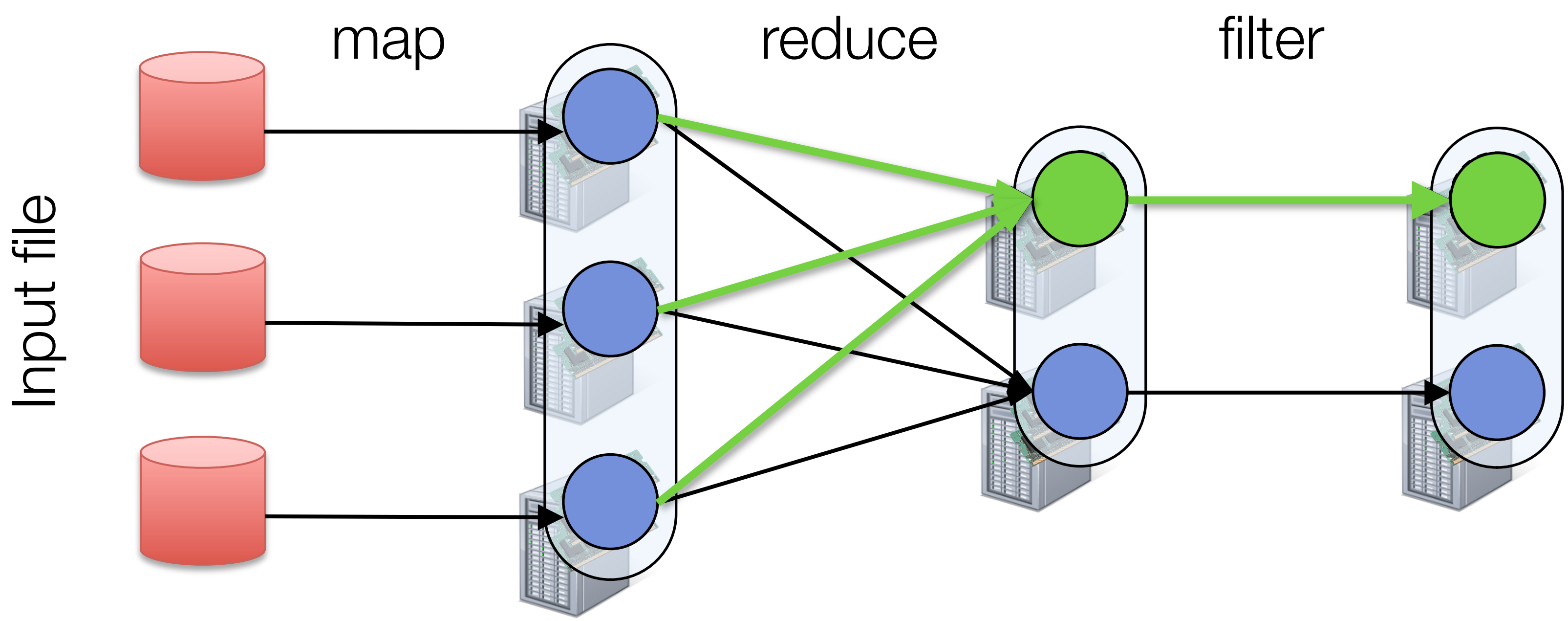
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Fault Tolerance

RDDs track *lineage* info to rebuild lost data

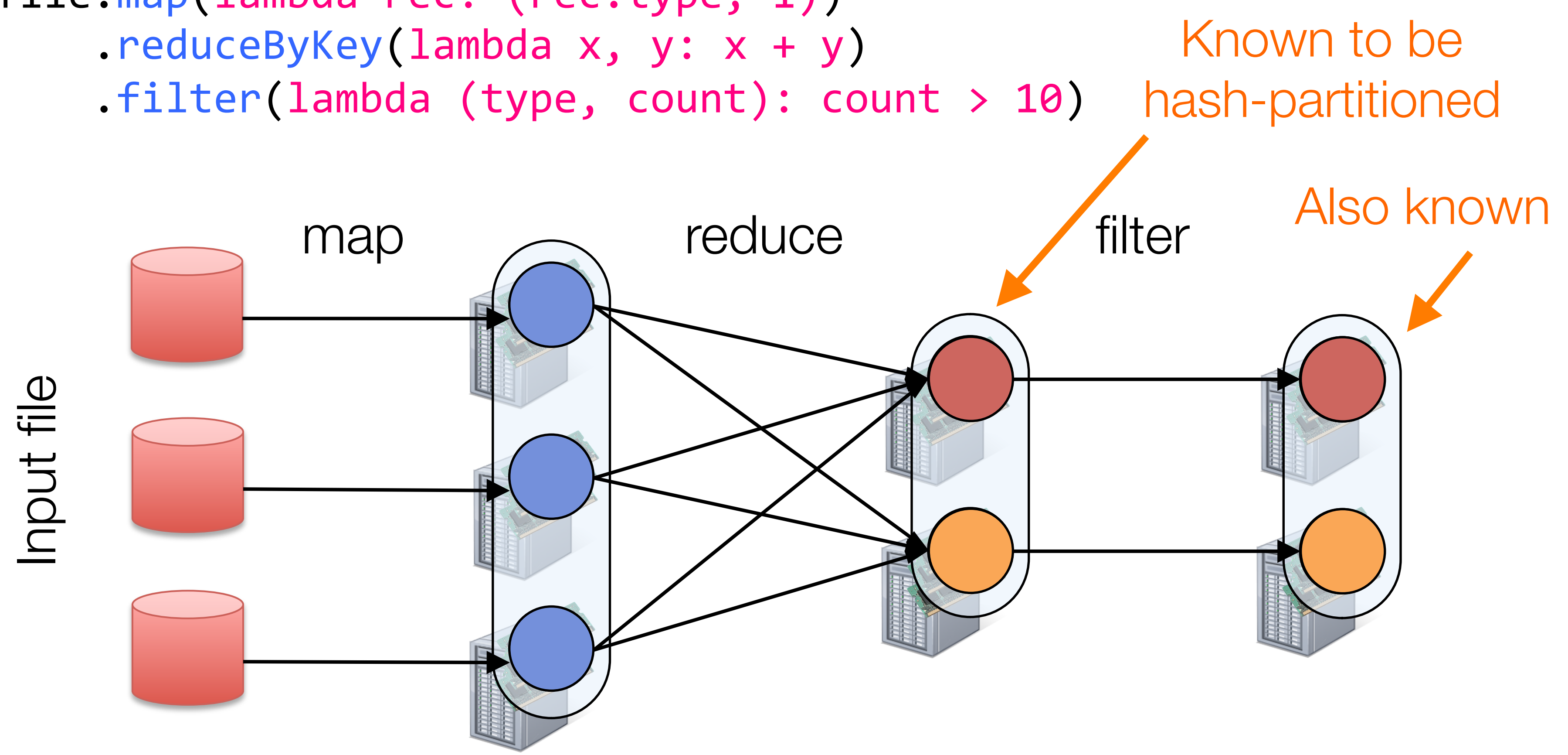
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Partitioning

RDDs know their partitioning functions

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

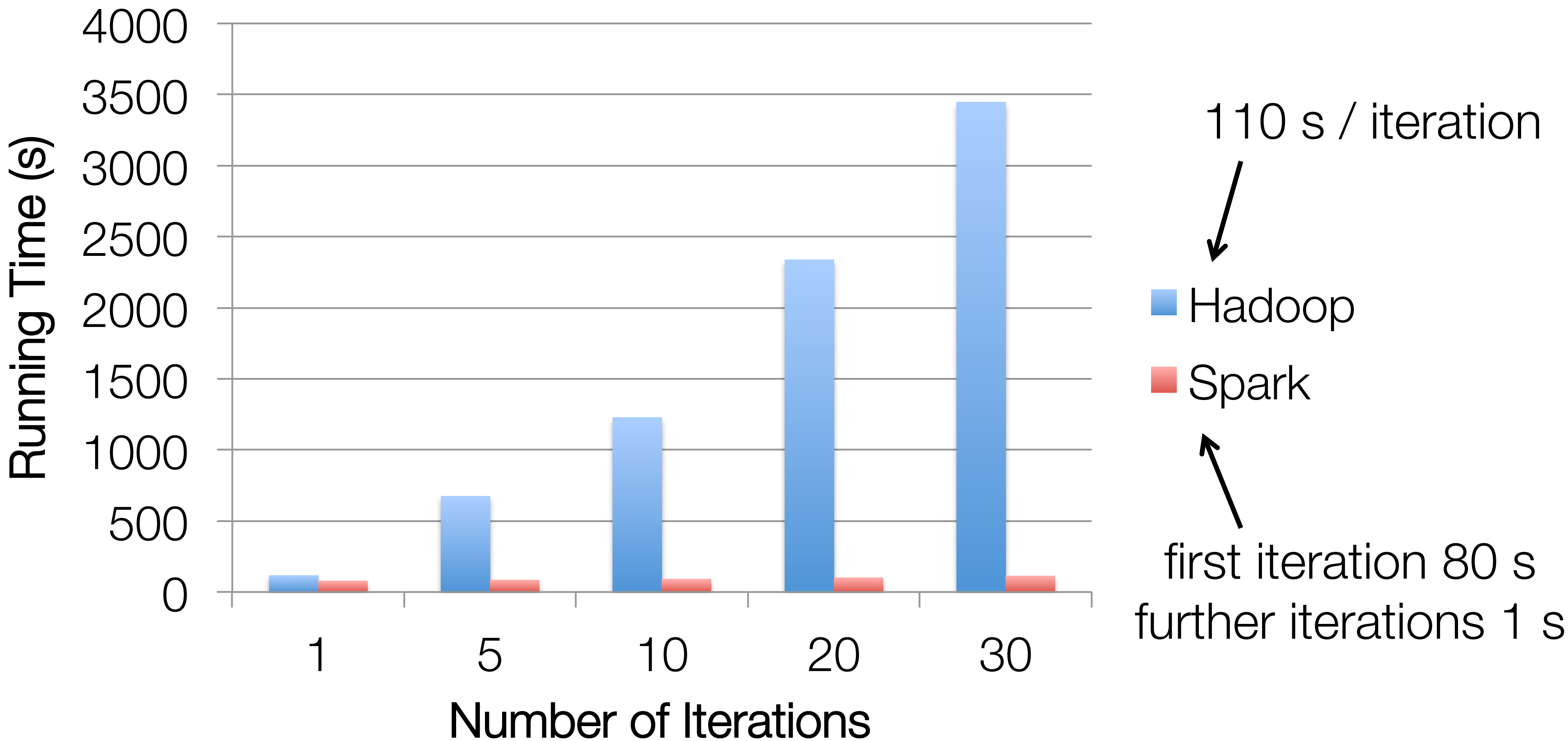
Numerical computing on Spark

Ongoing work

Logistic Regression

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

w = numpy.random.rand(D)

for i in range(iterations):
    gradient = data.map(lambda p:
        (1 / (1 + exp(-p.y * w.dot(p.x)))) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient

print “Final w: %s” % w
```

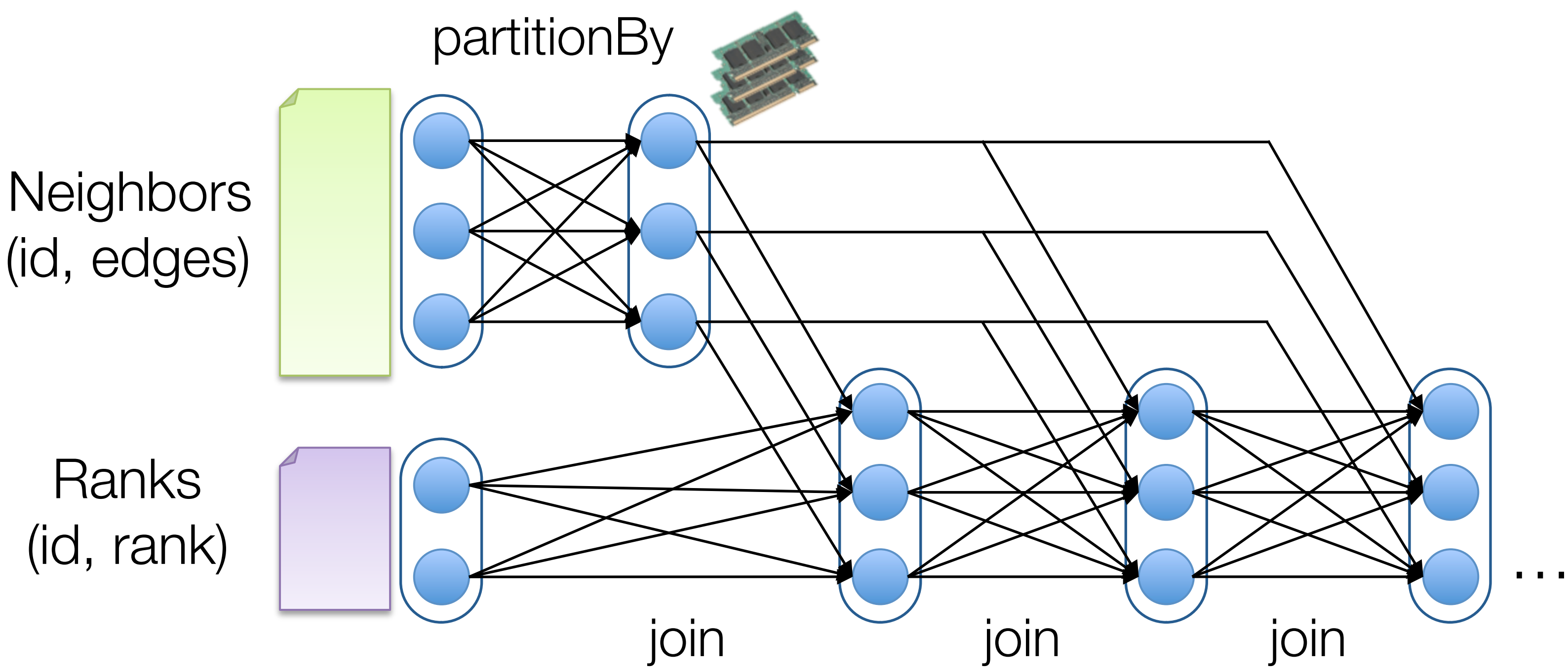
Logistic Regression Results



PageRank

Using cache(), keep neighbor lists in RAM

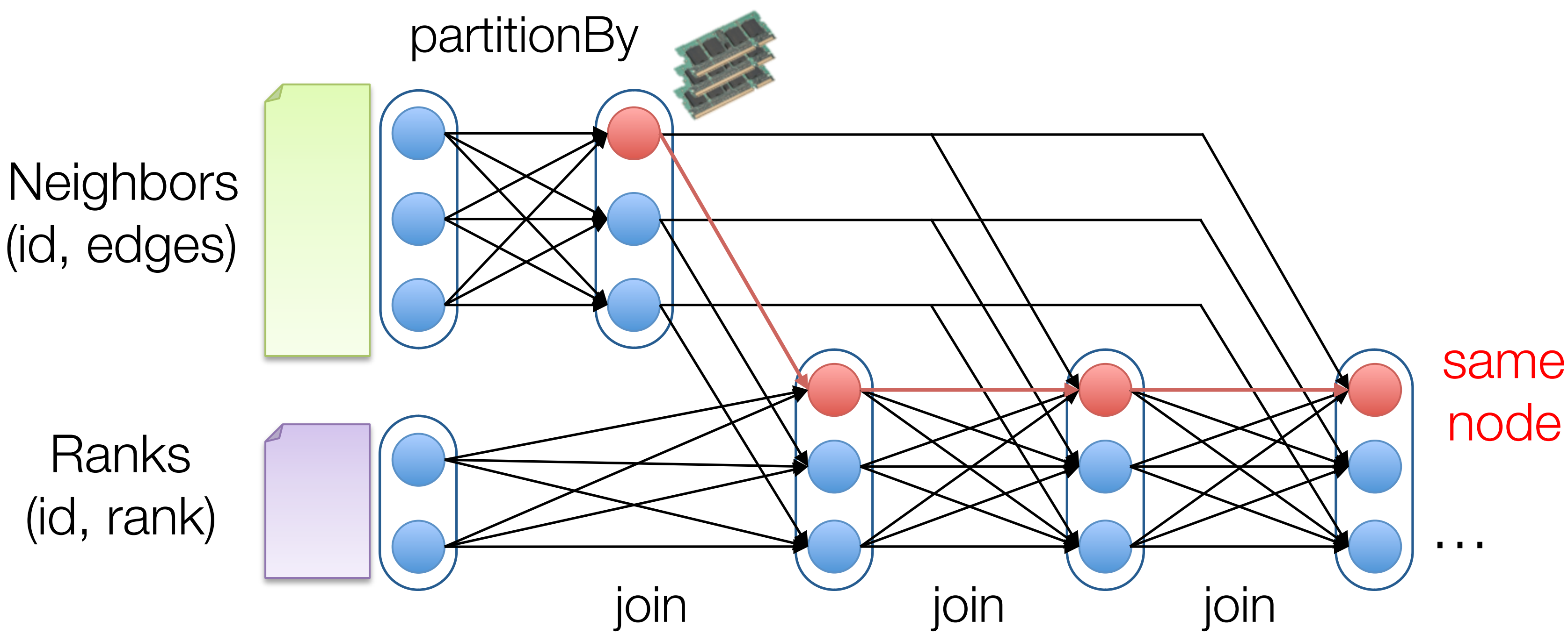
Using partitioning, avoid repeated hashing



PageRank

Using cache(), keep neighbor lists in RAM

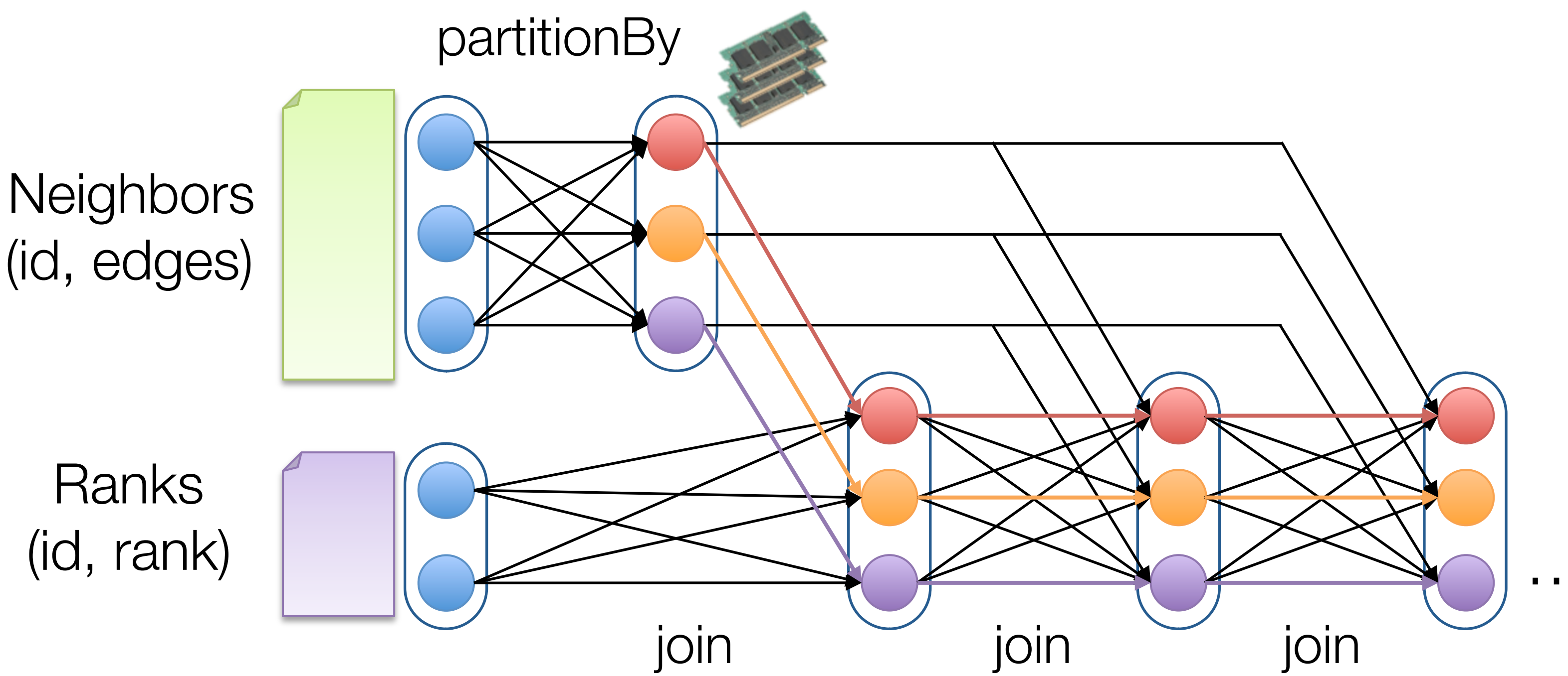
Using partitioning, avoid repeated hashing



PageRank

Using cache(), keep neighbor lists in RAM

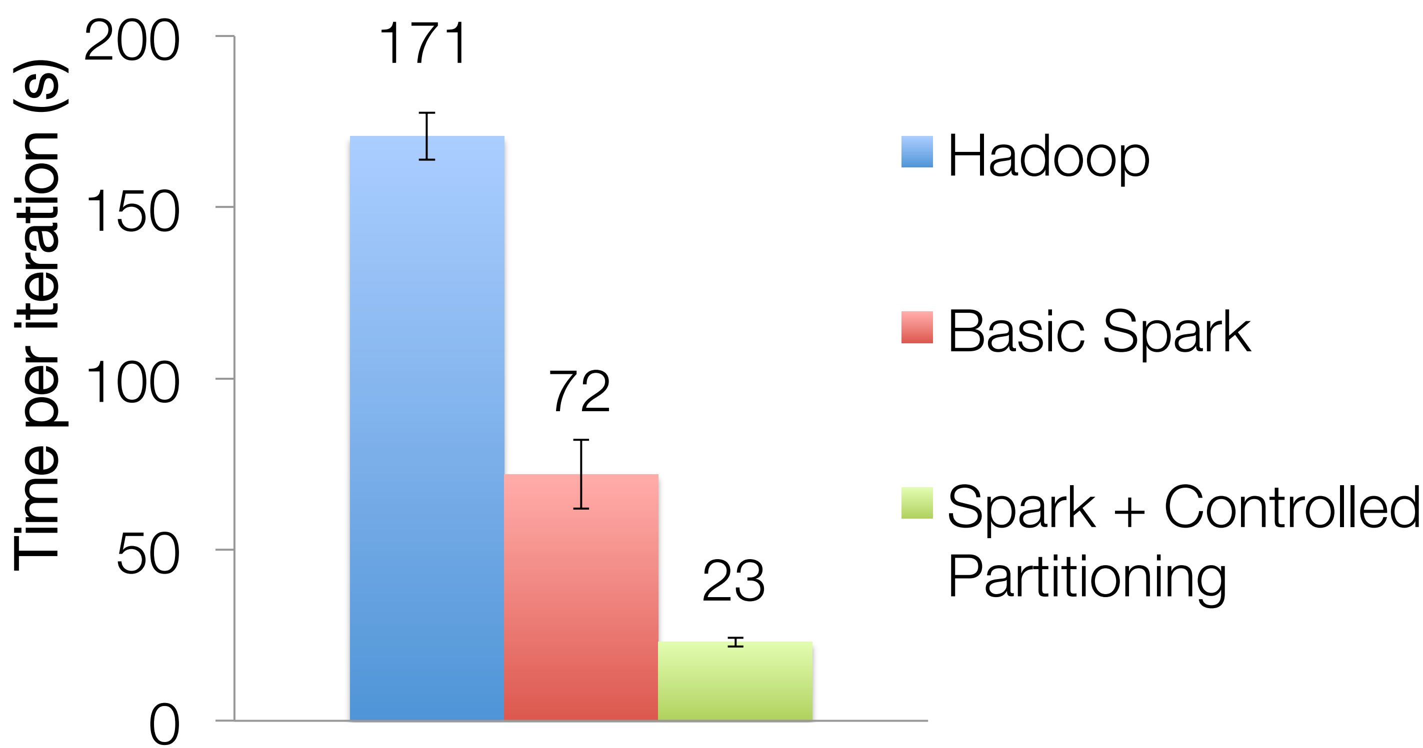
Using partitioning, avoid repeated hashing



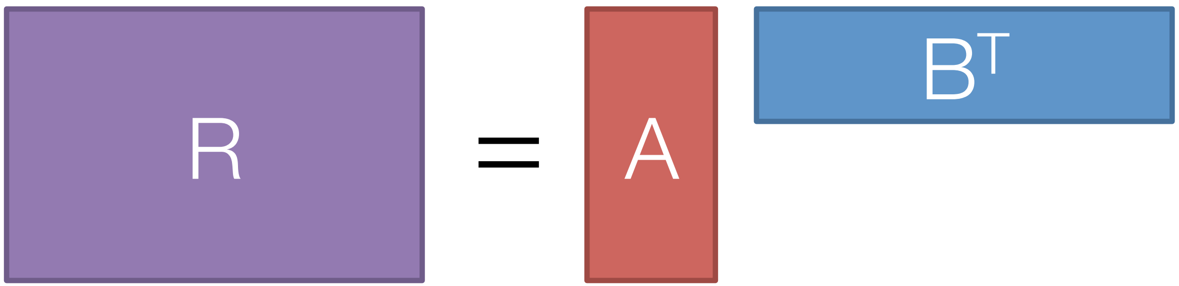
PageRank Code

```
# RDD of (id, neighbors) pairs
links = spark.textFile(...).map(parsePage)
            .partitionBy(128).cache()

ranks = links.mapValues(lambda v: 1.0)  # RDD of (id, rank)

for i in range(ITERATIONS):
    ranks = links.join(ranks).flatMap(
        lambda (id, (links, rank)):
            [(d, rank/links.size) for d in links]
    ).reduceByKey(lambda a, b: a + b)
```

PageRank Results

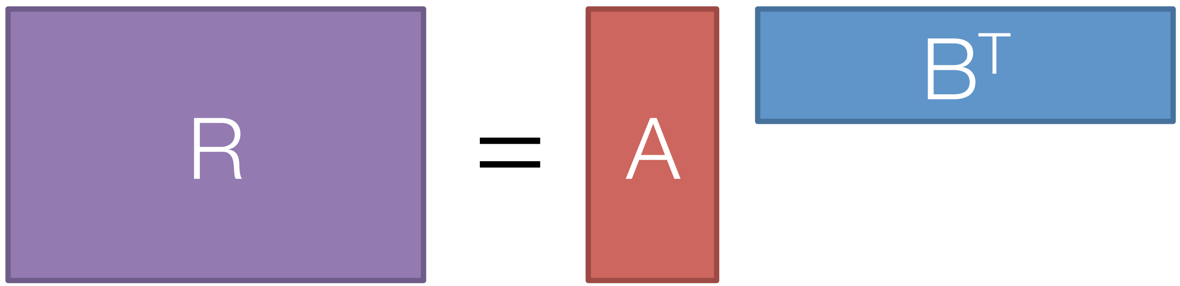


Alternating Least Squares



1. Start with random A_1, B_1
2. Solve for A_2 to minimize $||R - A_2 B_1^T||$
3. Solve for B_2 to minimize $||R - A_2 B_2^T||$
4. Repeat until convergence

ALS on Spark

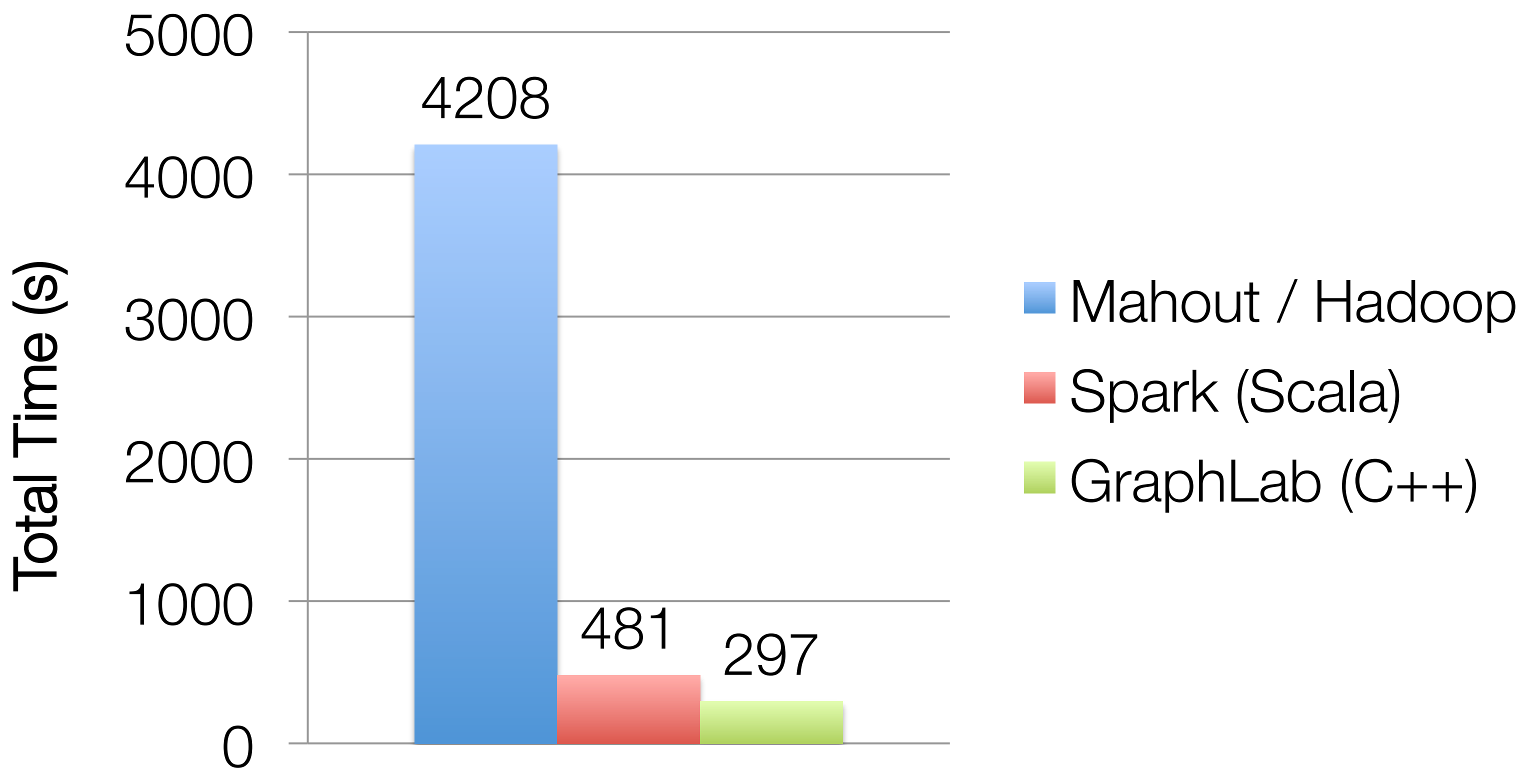


Cache 2 copies of R in memory, one partitioned by rows and one by columns

Keep A & B partitioned in corresponding way

Operate on blocks to lower communication

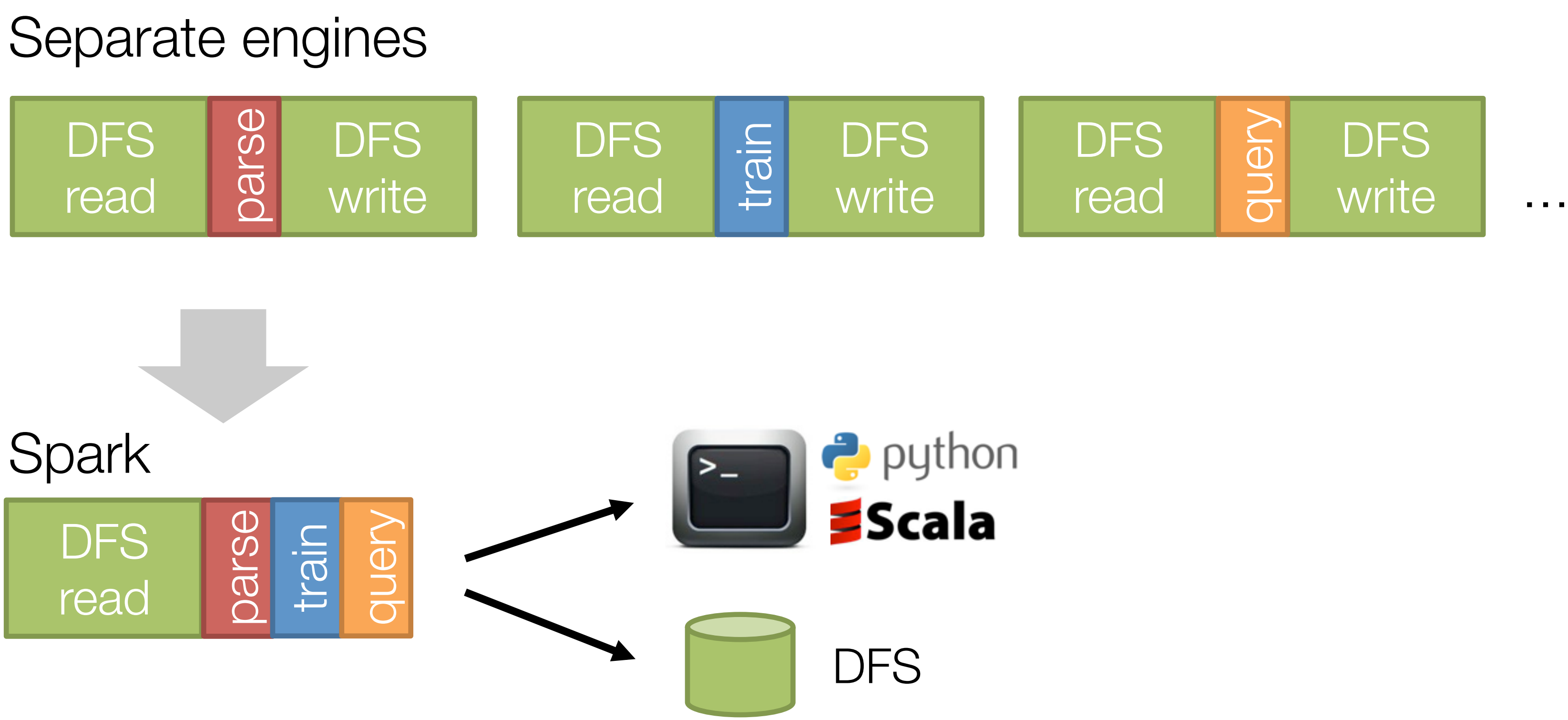
ALS Results



Total Time (s)
5000
4000
3000
2000
1000
0
4208
481
297
Mahout / Hadoop
Spark (Scala)
GraphLab (C++)

Benefit for Users

Same engine performs data extraction, model training and interactive queries



Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Numerical computing on Spark

Ongoing work

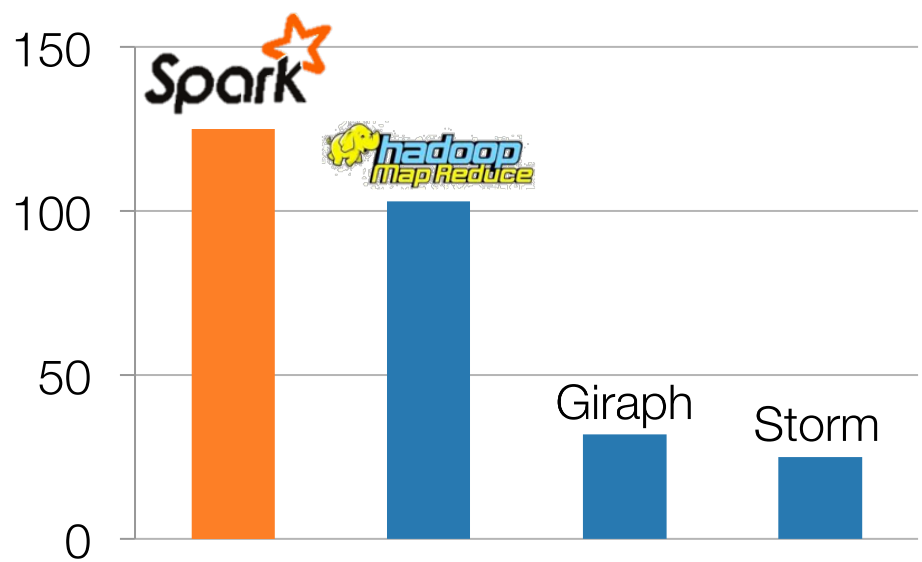
Spark Community

Most active open source community in big data

200+ developers, 50+ companies contributing

YAHOO! intel Adobe IBM
redhat amazon web services ebay
cloudera DATASTAX MAPR
阿里巴巴 Alibaba.com ClearStory DATA Now You See It webtrends
bizo CONVIVA DATABRICKS

Contributors in past year



Built-in ML Library: MLlib

classification: logistic regression, linear SVM, naïve Bayes, classification tree

regression: generalized linear models (GLMs), regression tree

collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

clustering: k-means||

decomposition: tall-skinny SVD, PCA

optimization: stochastic gradient descent, L-BFGS

Ongoing Work in MLlib

multiclass decision trees

stats library (e.g. stratified sampling, ScaRSR)

ADMM

LDA

40 contributors since project started Sept '13

SVD via ARPACK

Very mature Fortran77 package for computing eigenvalue decompositions

JNI interface available via netlib-java

Distributed using Spark distributed matrix-vector multiplies!

Convex Optimization

Distribute CVX by backing CVXPY with PySpark

Easy-to-express distributable convex programs

Need to know less math to optimize complicated objectives

```
from cvxpy import *

# Create two scalar optimization variables.
x = Variable()
y = Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = Minimize(square(x - y))

# Form and solve problem.
prob = Problem(obj, constraints)
prob.solve()  # Returns the optimal value.
print "status:", prob.status
print "optimal value", prob.value
print "optimal var", x.value, y.value
```

```
status: optimal
optimal value 0.999999989323
optimal var 0.999999998248 1.75244914951e-09
```

Research Projects

GraphX: graph computation via data flow

SparkR: R interface to Spark, and distributed matrix operations

ML pipelines: high-level machine learning APIs

Applications: neuroscience, traffic, genomics, general convex optimization

Spark and Research

Spark has all its roots in research, so we hope to keep incorporating new ideas!

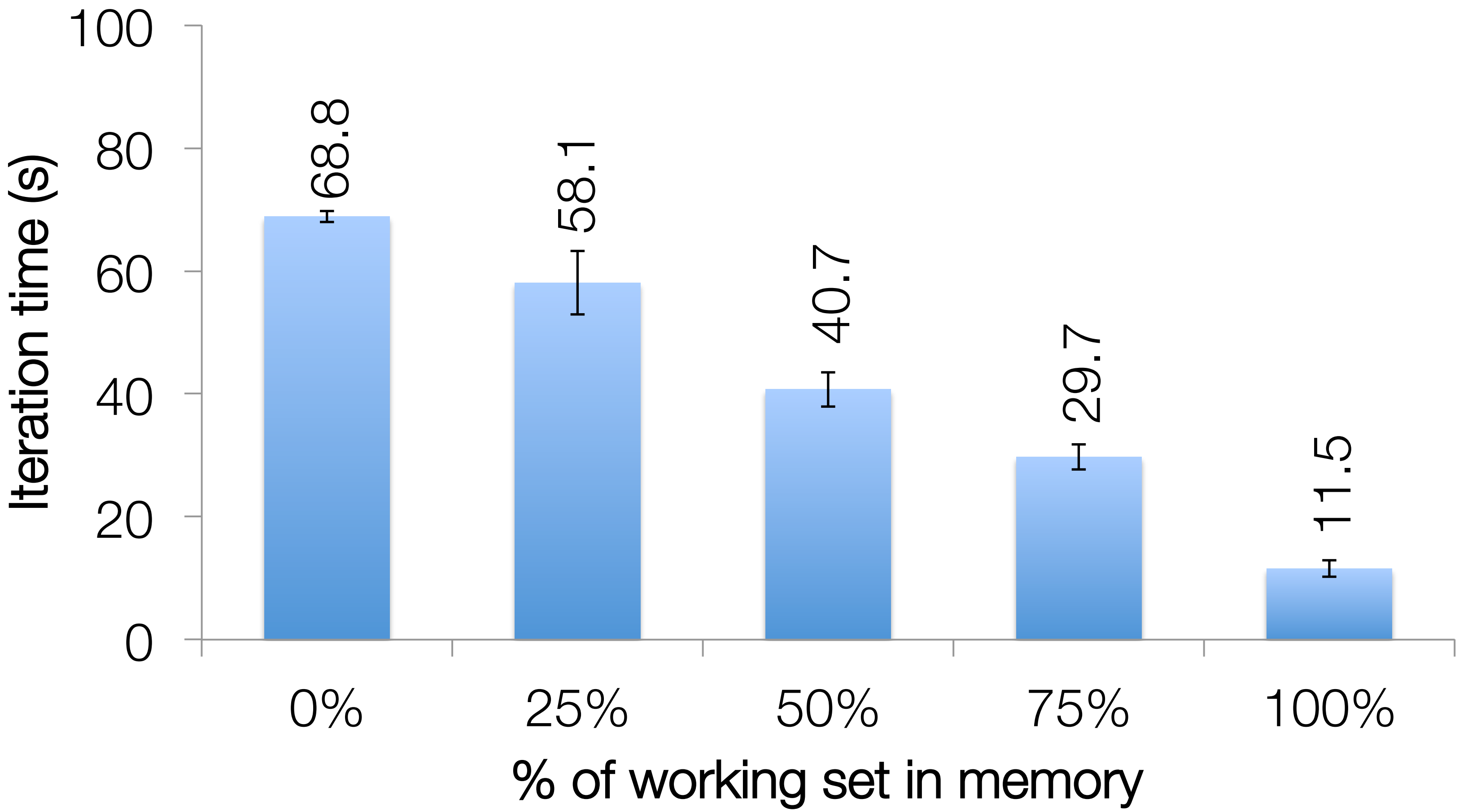
Conclusion

Data flow engines are becoming an important platform for numerical algorithms

While early models like MapReduce were inefficient, new ones like Spark close this gap

More info: spark.apache.org

Behavior with Less RAM



Spark in Scala and Java

```
// Scala:

val lines = sc.textFile(...)
lines.filter(x => x.contains(“ERROR”)).count()


// Java:

JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains(“error”);
  }
}).count();
```