Apache Spark

28-01-2016
Jeroen Schot - jeroen.schot@surfsara.nl
Mathijs Kattenberg - mathijs.kattenberg@surfsara.nl
Machiel Jansen - machiel.jansen@surfsara.nl

A Data-Parallel Approach

Restrict the programming interface so that the system can do more automatically. Use ideas from functional programming:

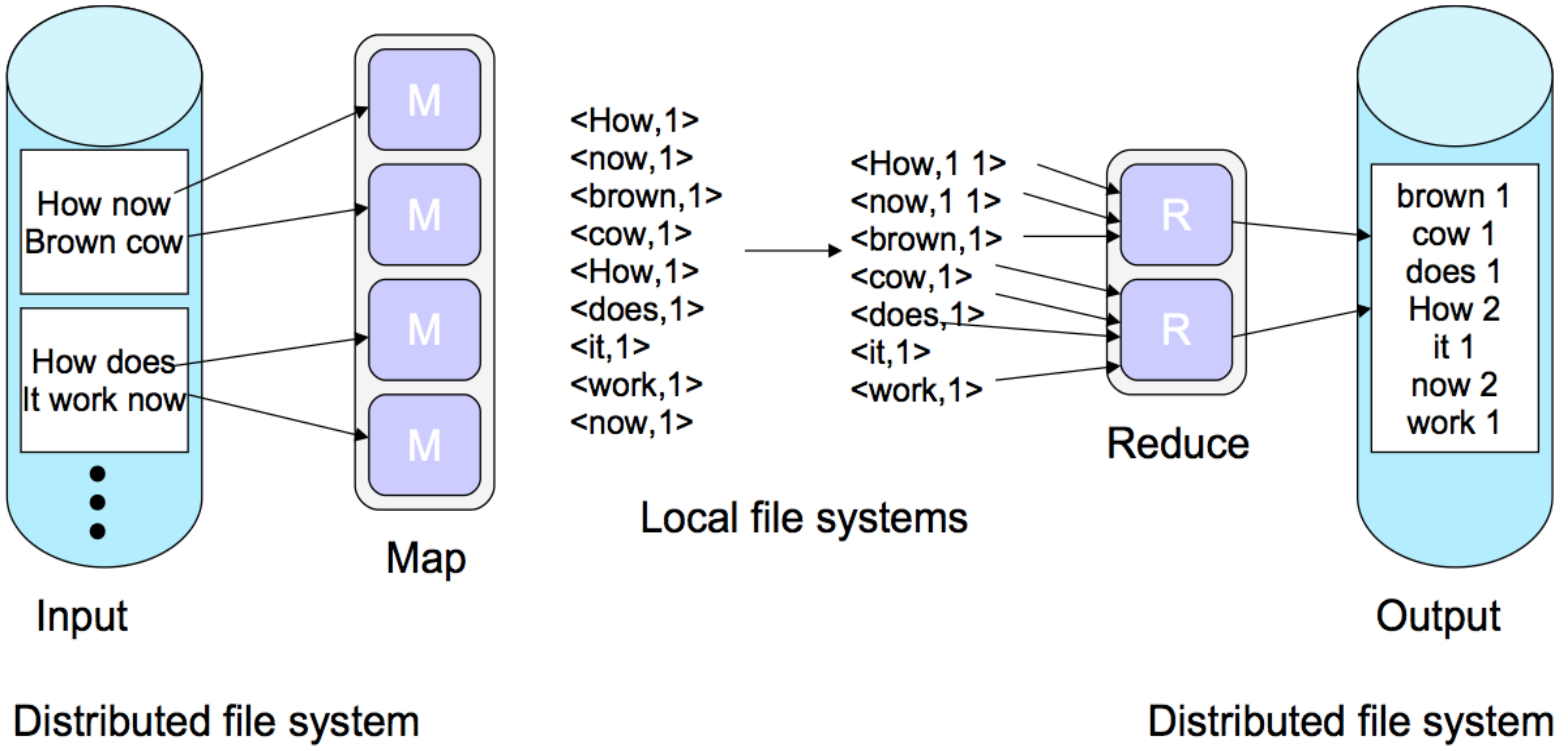
“Here is a function, apply it to all of the data”

- I do not care where it runs (the system should handle that)
- Feel free to run it twice on different nodes (no side effects!)

MapReduce Programming Model

Map function: $(K_1, V_1) \longrightarrow list(K_2, V_2)$

Reduce function: $(K_2, list(V_2)) \longrightarrow list(K_3, V_3)$



Problems with MapReduce

- Difficulty to convert problem to MR algorithm:
  MR not expressive enough?

- Performance issues due to disk I/O between every job:
  Unsuited for iterative algorithms or interactive use

Higher Level Frameworks

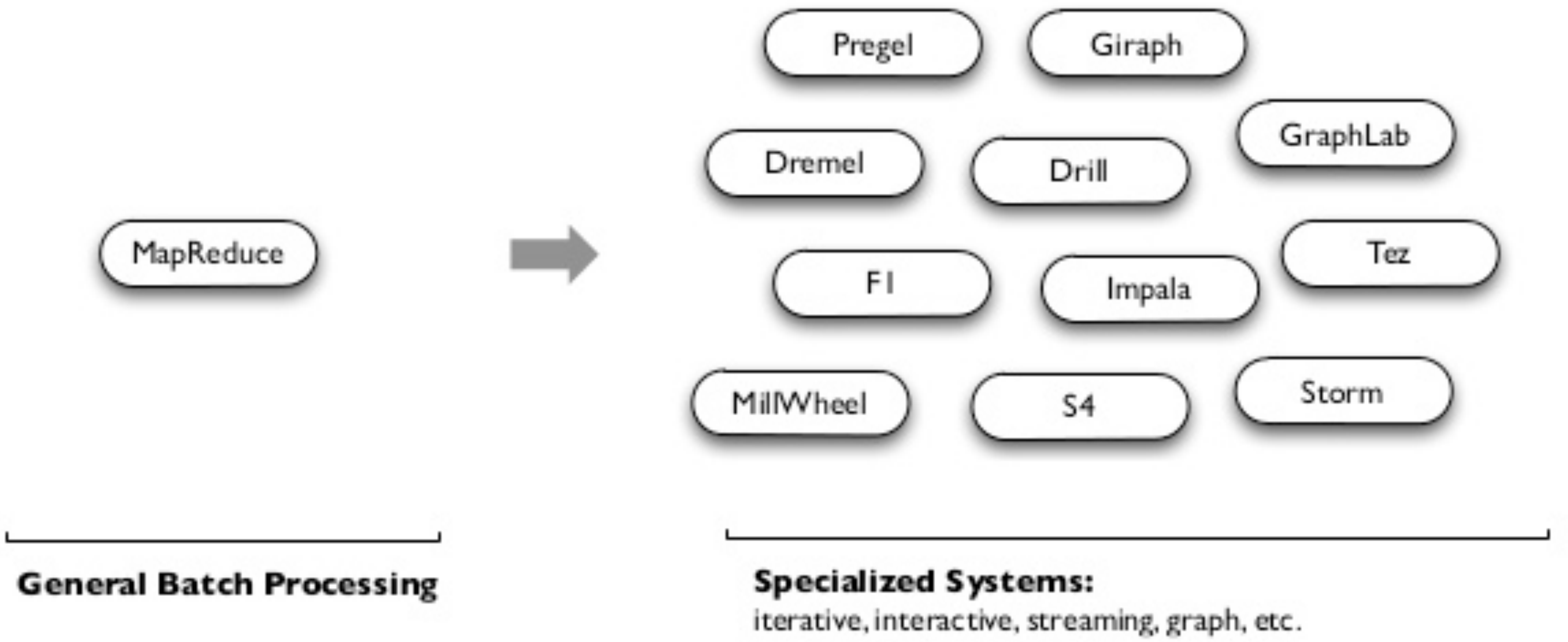
HIVE

cascading

APACHE GIRAPH

SURF SARA

Specialized systems



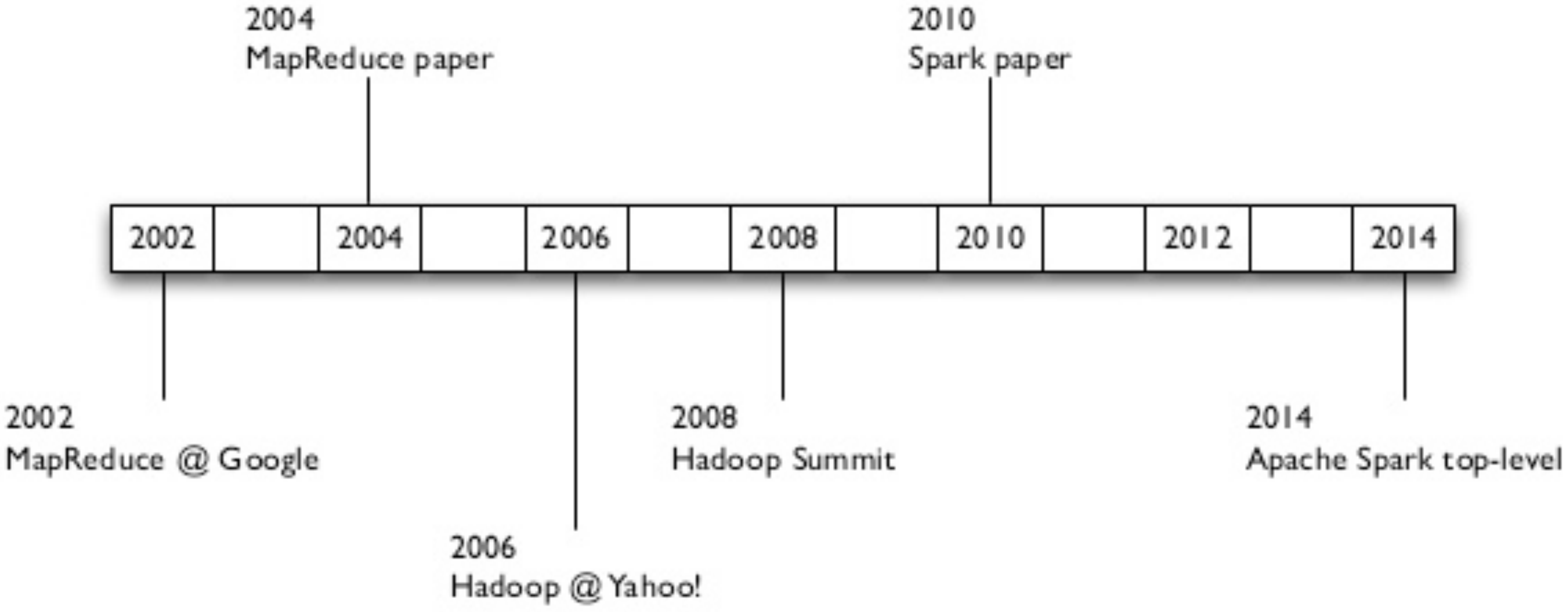
http://www.slideshare.net/rxin/stanford-cs347-guest-lecture-apache-spark

SURF SARA

Solved?

- Performance issues solved only partially
- How about workflows that need multiple components?

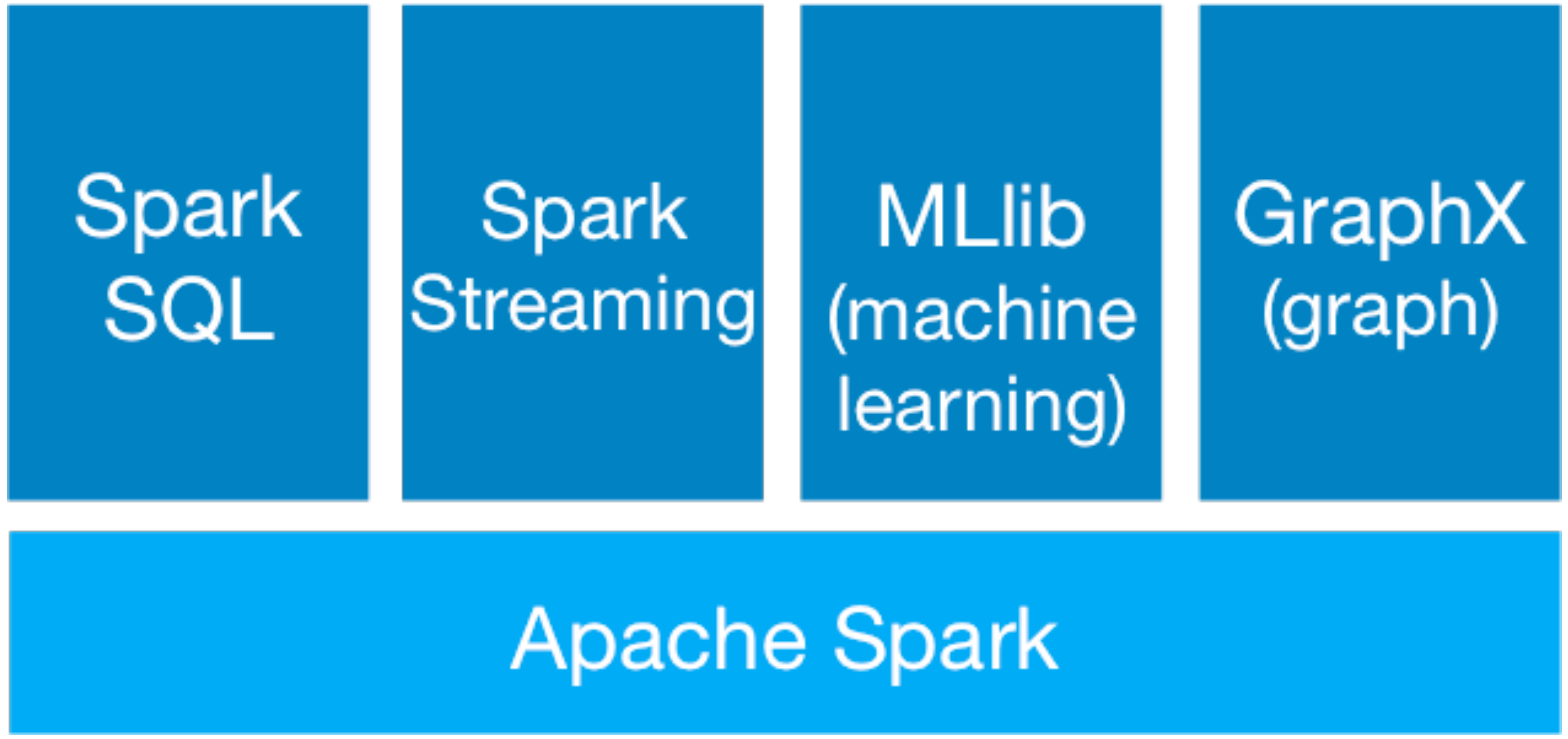
Enter Spark


2004
MapReduce paper
2010
Spark paper
2002
2004
2006
2008
2010
2012
2014
2002
MapReduce @ Google
2006
Hadoop @ Yahoo!
2008
Hadoop Summit
2014
Apache Spark top-level

Spark’s approach

- General purpose processing framework for DAG’s
- Fast data sharing
- Idiomatic API (if you know Scala)

Spark ecosystem



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

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets

https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

RDD properties

- Collection of objects/elements
- Spread over many machines
- Built through parallel transformations
- Immutable

RDD origins

There are two ways to create a RDD from scratch

Parallelised collections:
distribute existing single-machine collections (List, HashMap)

Hadoop datasets:
files from HDFS-compatible filesystem (Hadoop InputFormat)

Operations on RDDs

Transformations:

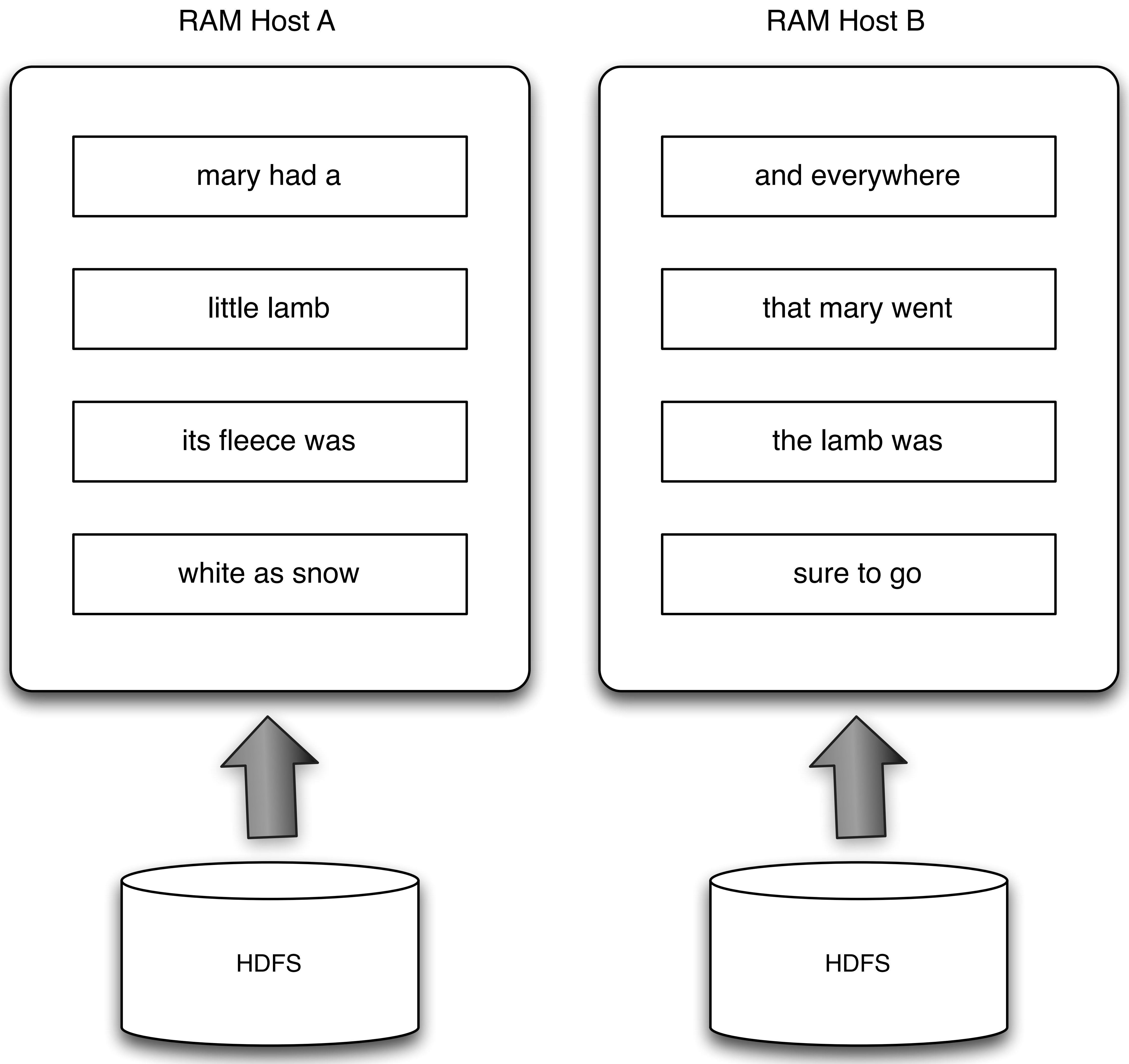
- Lazily computed
- Create new RDD
- Example: ‘map’

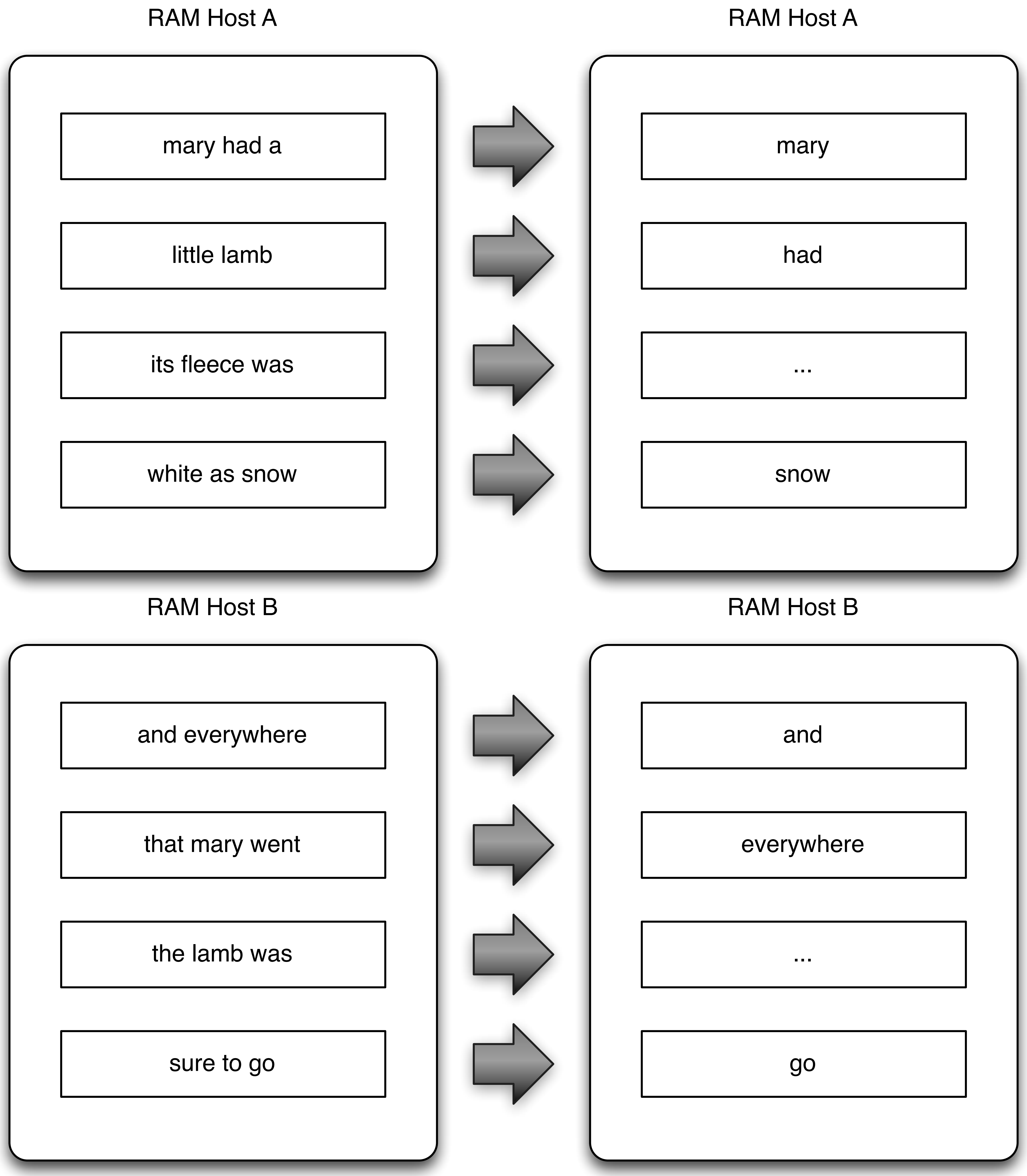
Actions:

- Triggers computation
- Example: ‘count’, ‘saveAsTextFile’

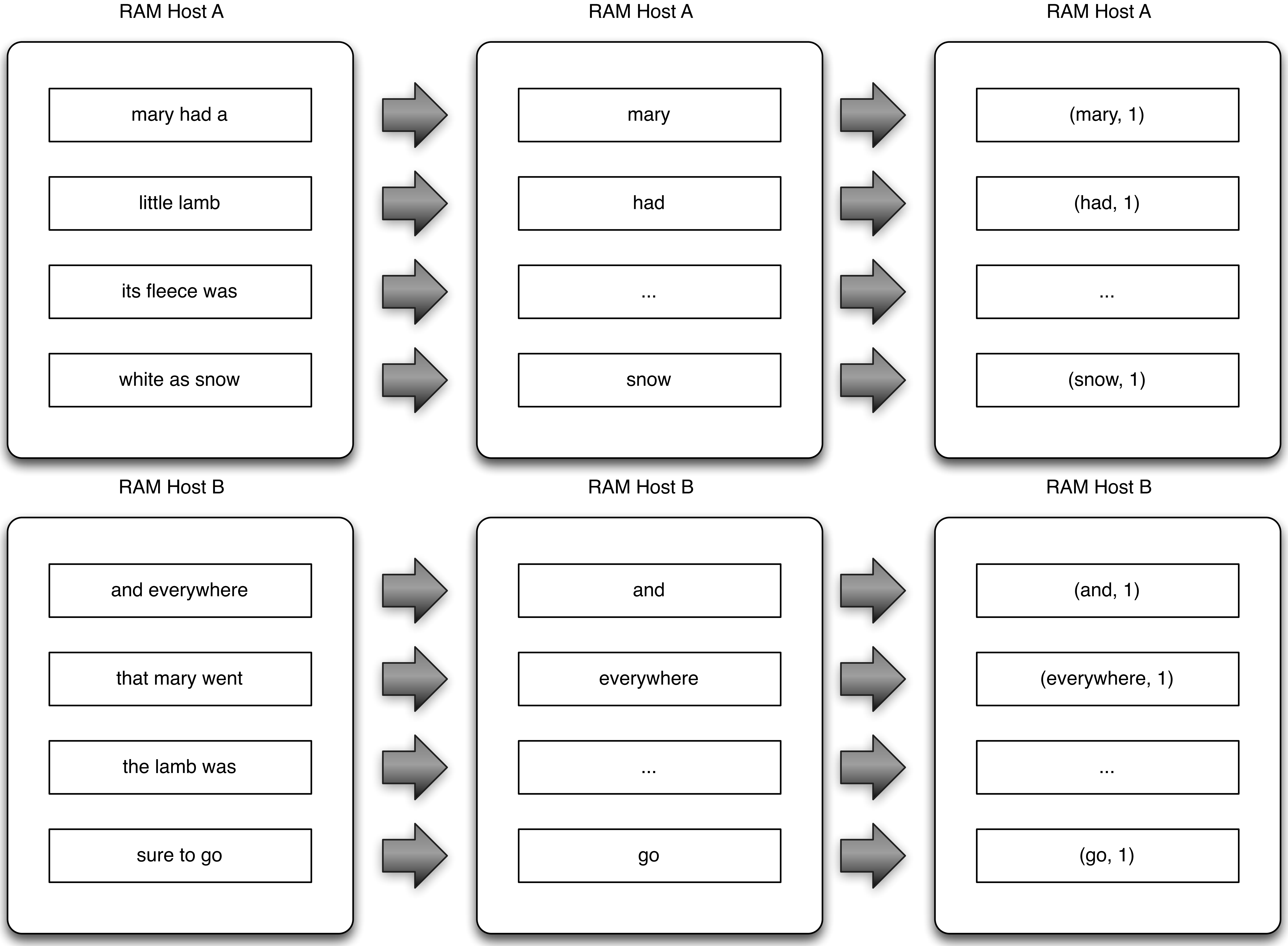
SURF SARA

An RDD from HDFS



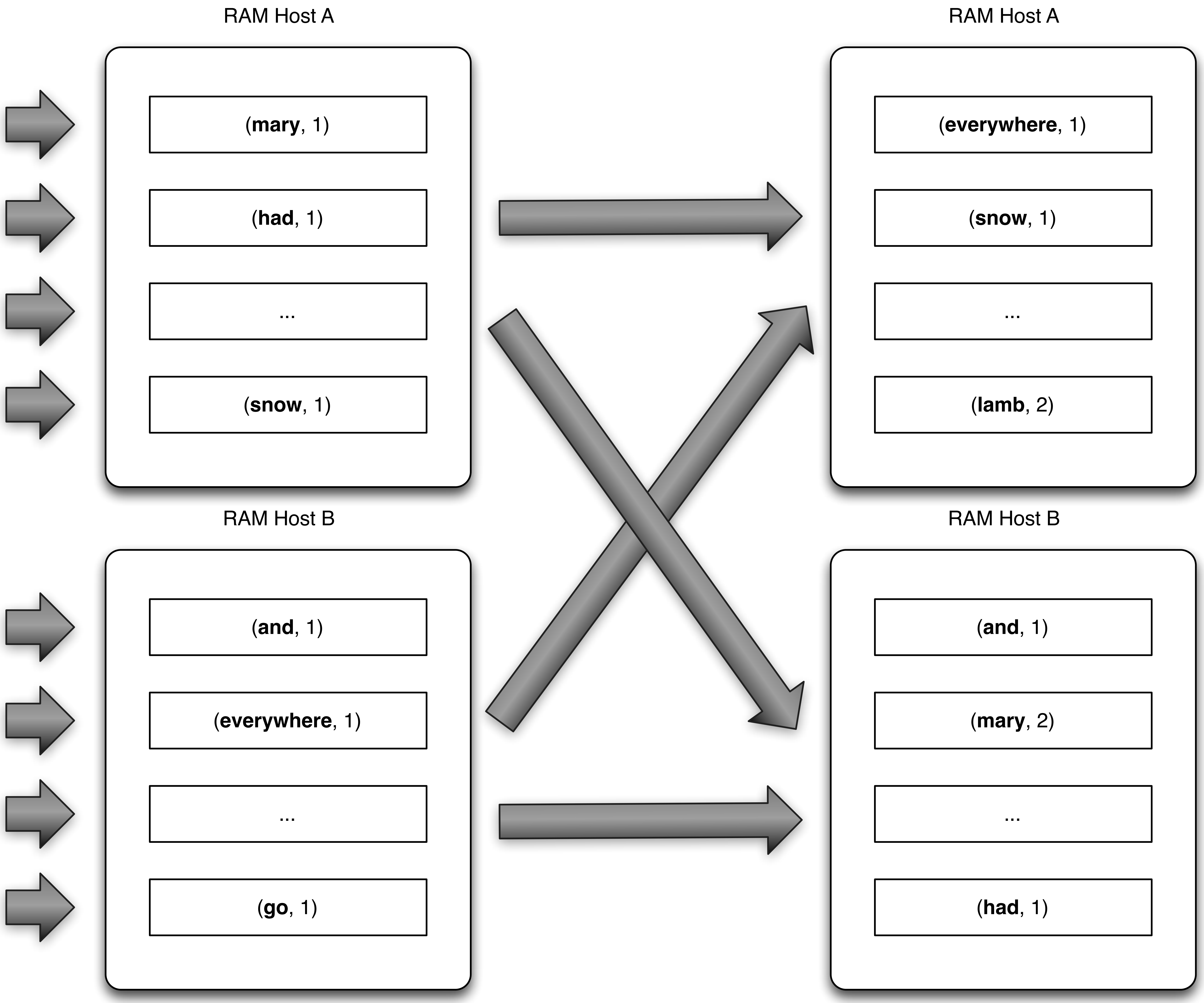



```
rdd.flatMap(lambda s: s.split(" "))
```



`rdd.flatMap(lambda s: s.split(" "))` `rdd.map(lambda w: (w, 1))`

SURF SARA



rdd.reduceByKey(lambda x, y: x + y)

Transformations

RDD's are created from other RDD's using transformations:

map(f) => pass every element through function *f*

reduceByKey(f) => aggregate values with same key using *f*

Transformations

RDD’s are created from other RDD’s using transformations:

map(f) => pass every element through function *f*

reduceByKey(f) => aggregate values with same key using *f*

filter(f) => select elements for which function *f* is true

flatMap(f) => similar to map, but one-to-many

join(r) => joined dataset with RDD *r*

union(r) => union with RDD *r*

sample, intersection, distinct, groupByKey, sortByKey, cartesian…

Actions

Transformations give no output (no side-effects) and don’t result in any real work (laziness)

Results from RDD’s via actions:

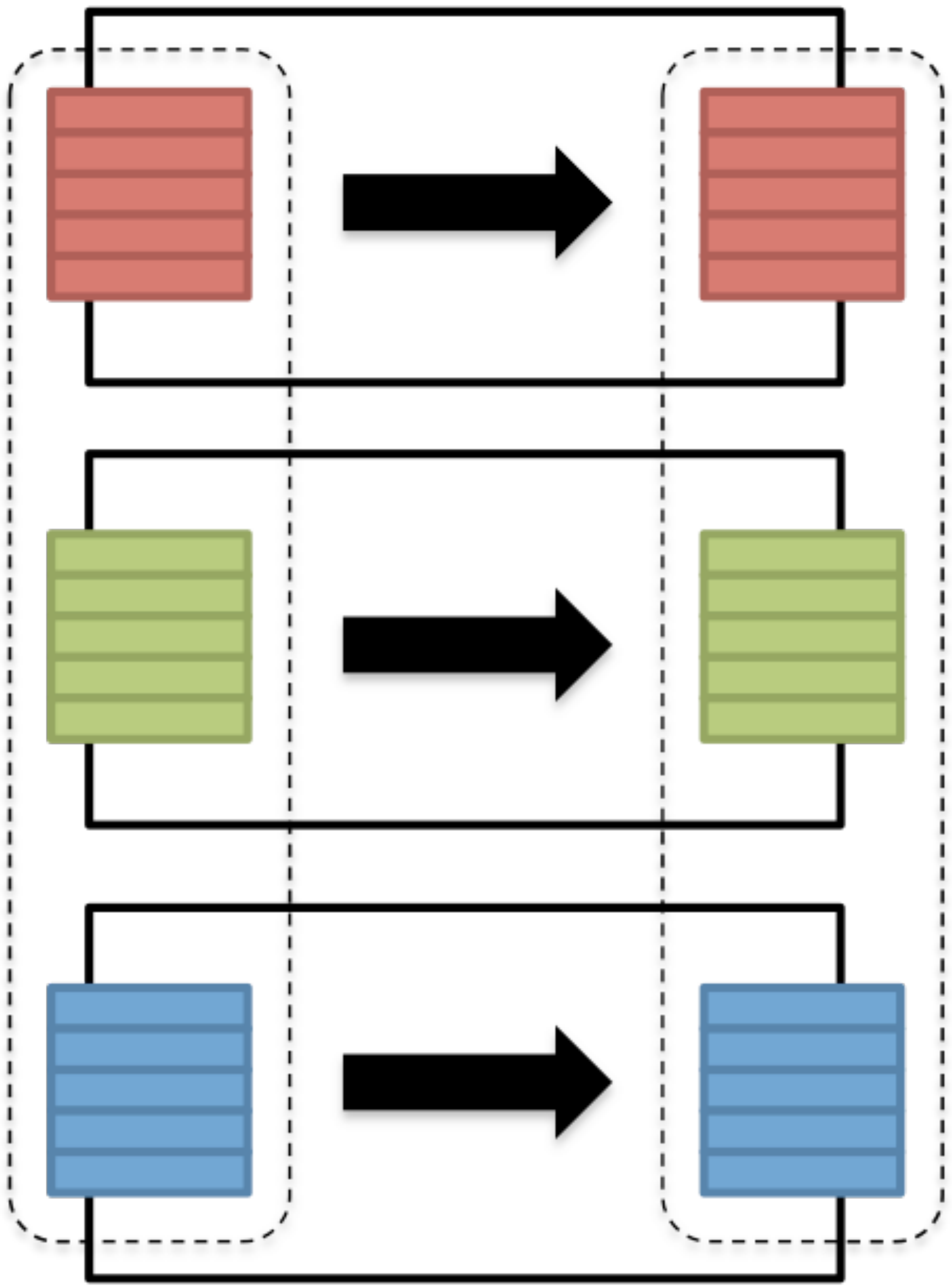
- count() => return the number of elements
- take(n) => select the first *n* elements
- saveAsTextFile(file) => store dataset as file

Lineage, laziness & persistence

- Spark stores lineage information for every RDD partition
- Intermediate RDDs are computed only when needed
- By default RDDs are not retained in memory
  — use the cache/persist methods on ‘hot’ RDDs

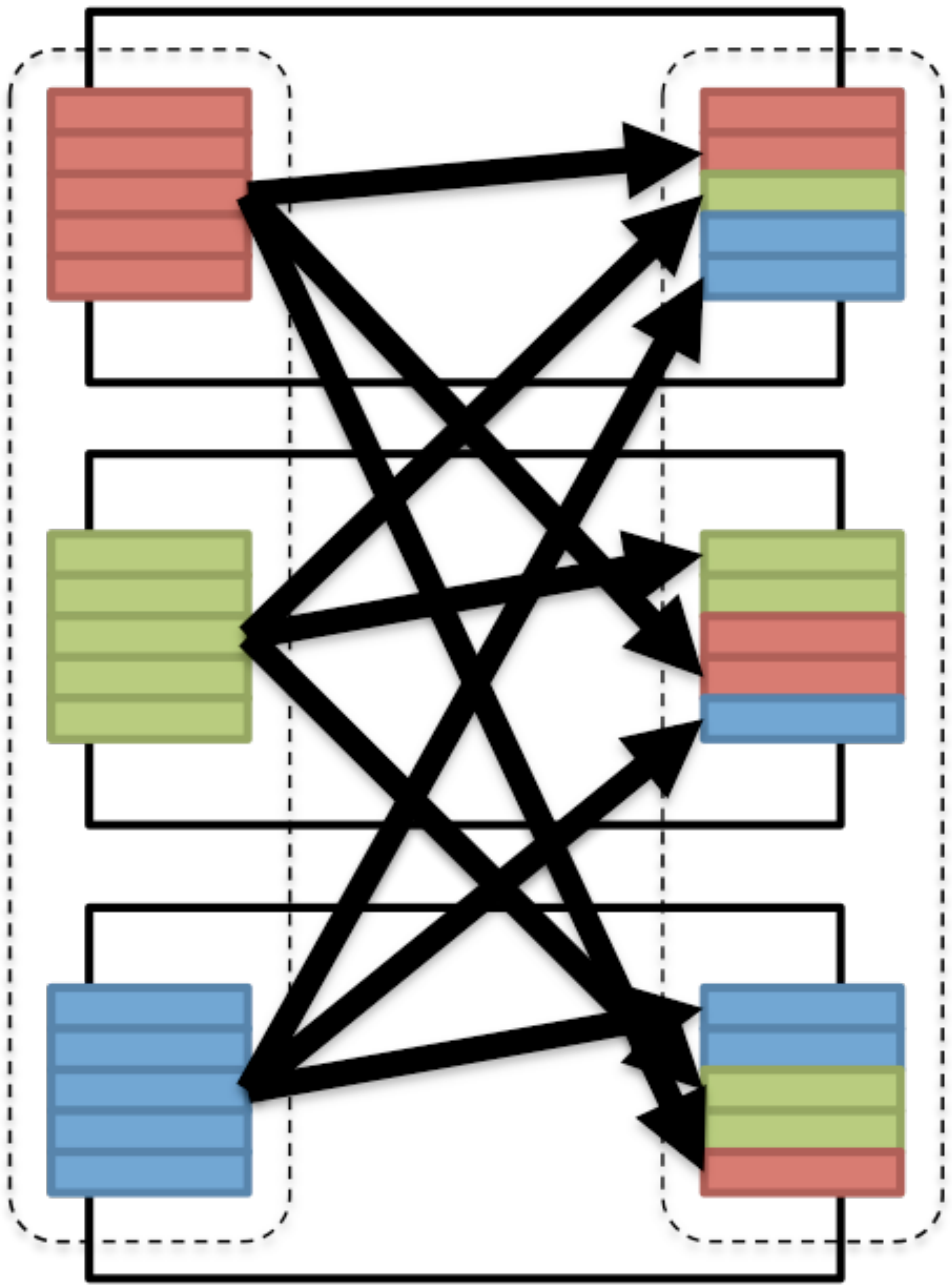
Narrow transformation

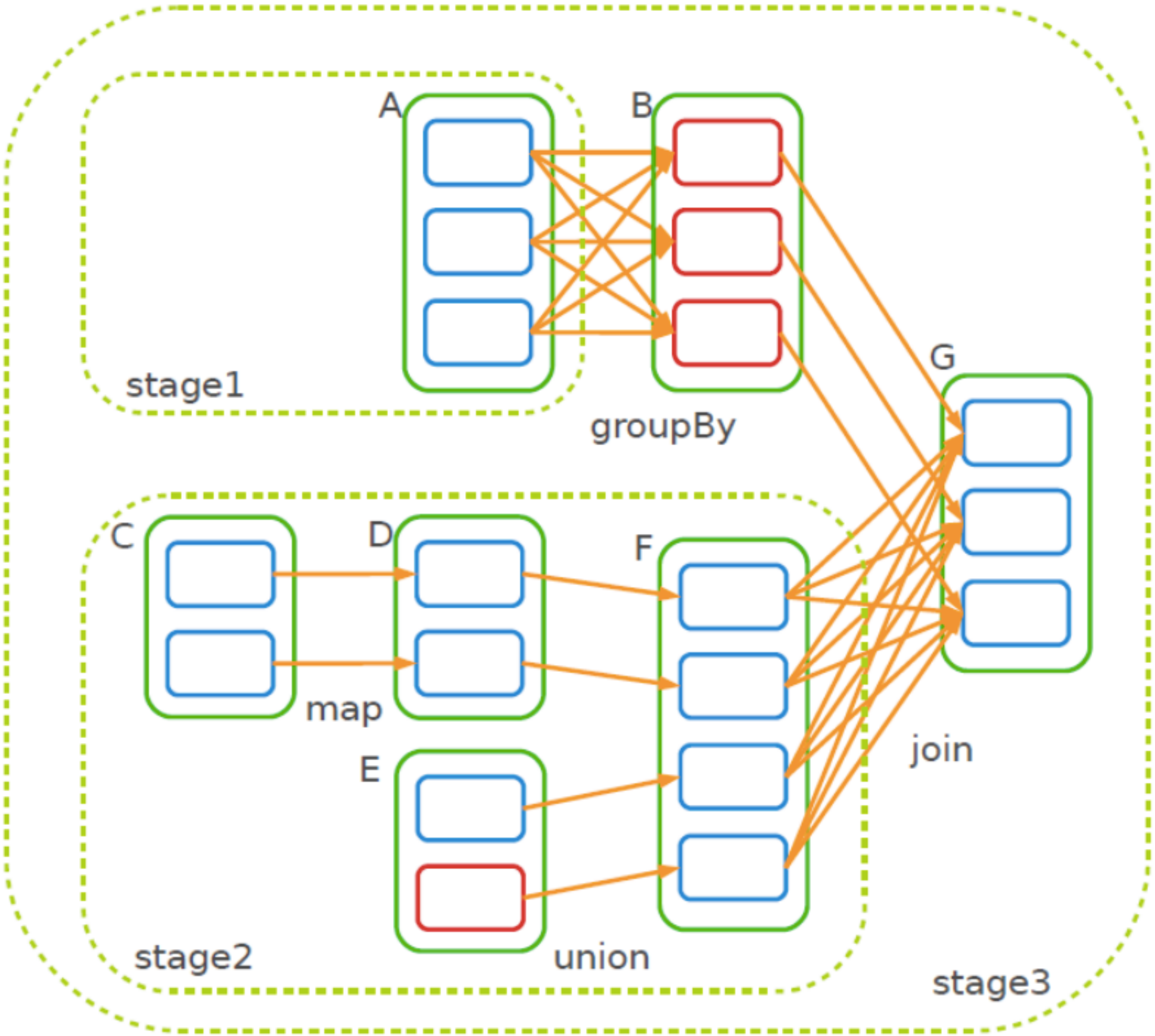
- Input and output stays in same partition
- No data movement is needed



Wide transformation

- Input from other partitions are required
- Data shuffling is needed before processing




A
B
G
stage1
groupBy
C
D
F
map
join
E
stage2
union
stage3

PairRDDs

RDDs of (key, value) tuples are ‘special’

A number of transformations only for PairRDDs:

- reduceByKey, groupByKey
- join, cogroup

Spark: a general framework

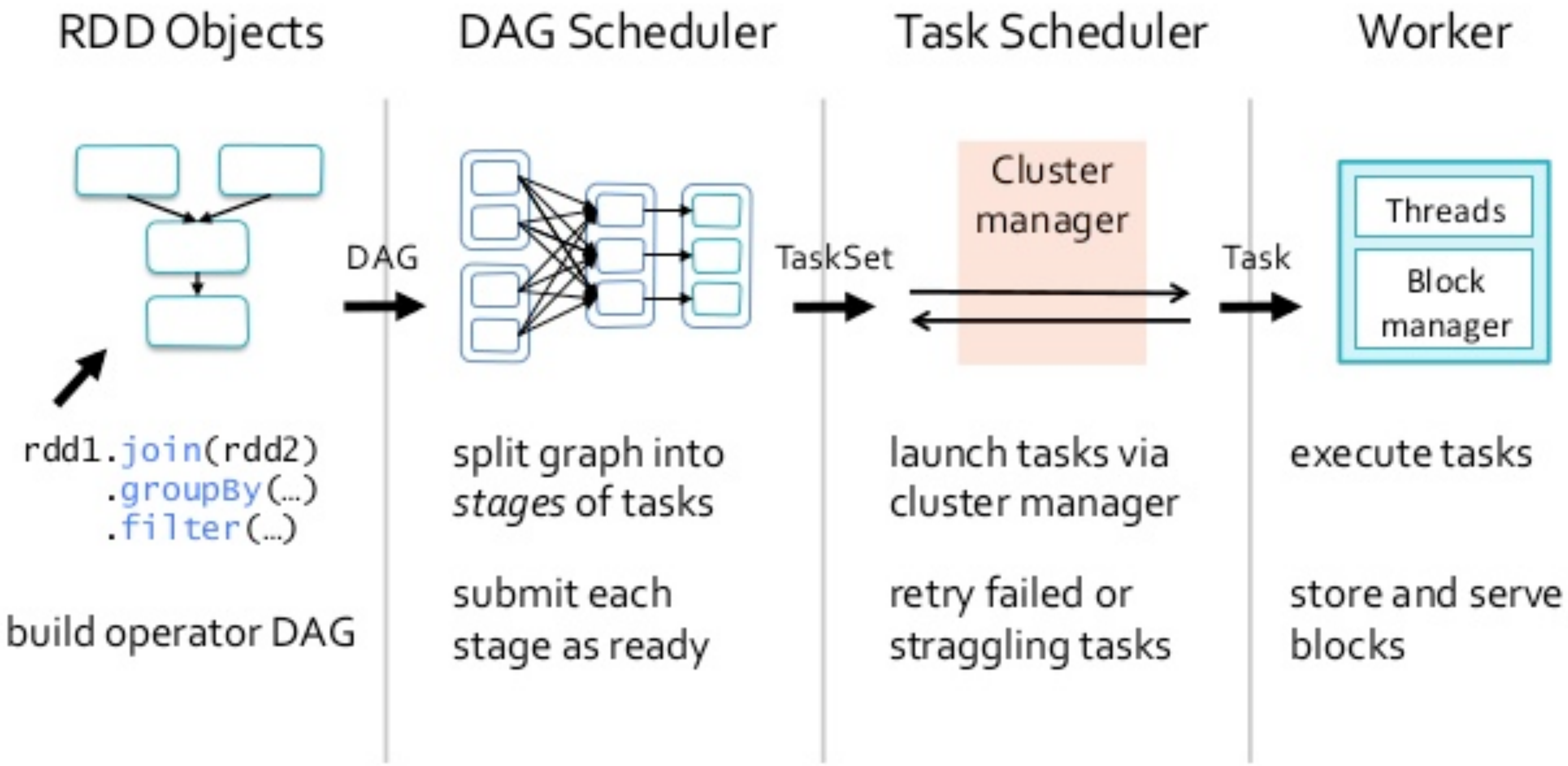
Spark aims to generalize MapReduce to support new applications with a more efficient engine, and simpler for the end users.

Write programs in terms of distributed datasets and operations on them

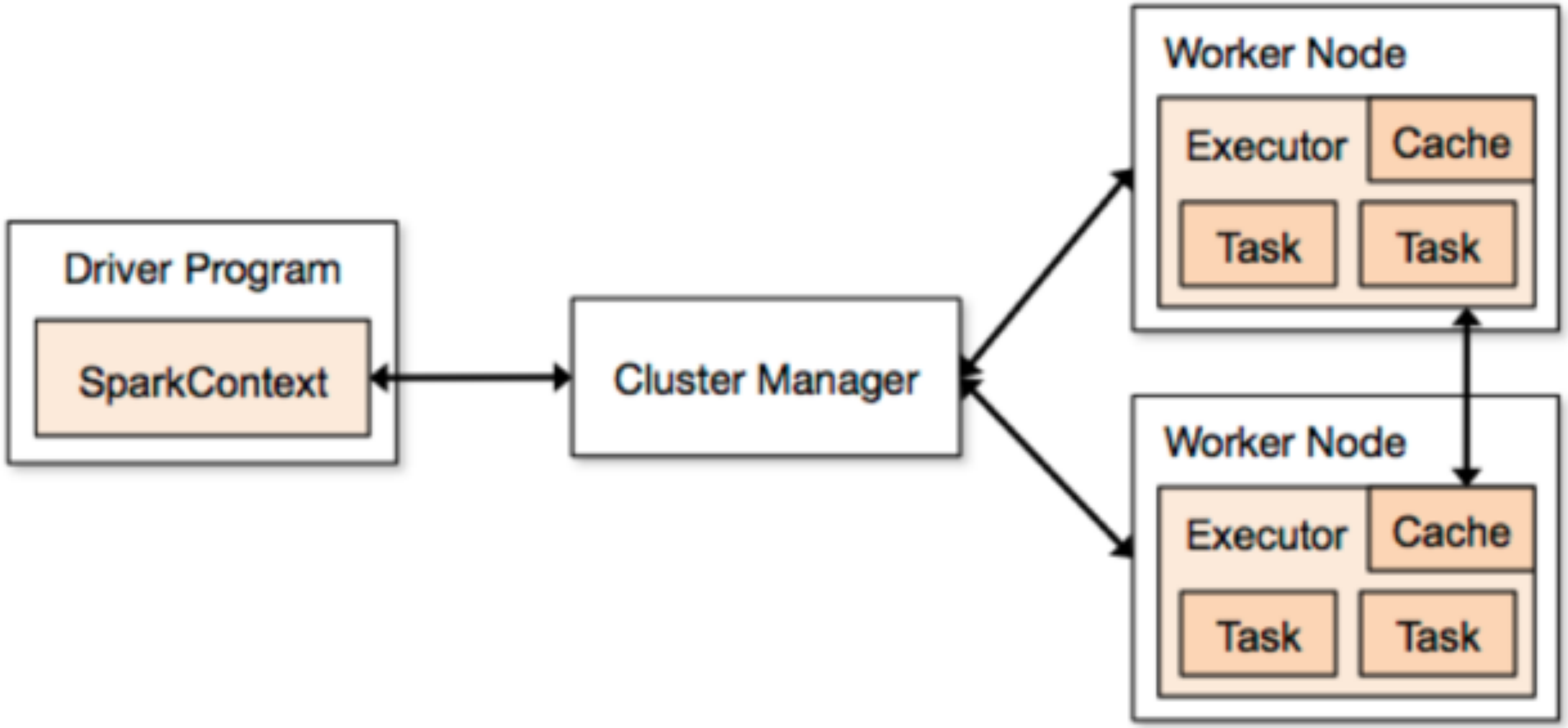
Accessible from multiple programming languages:

- Scala
- Java
- Python
- R (only via dataframes)

Execution Process



An Executing Application



Shared variables

- In general: avoid!
- When needed: read-only
- Two helpful types:
  broadcast variables, accumulators

Broadcast variables

- Wrapper around an object
- Copy send once to every worker
- Use case: lookup-table
- Should fit in the main memory of a single worker
- Can only be used read-only

SURF SARA

Accumulators

- Special variable to which workers can only “add”
- Only the driver can read
- Similar to MapReduce counters

RDD limitations

- Reading structured data sources (schema)
- Tuple juggling

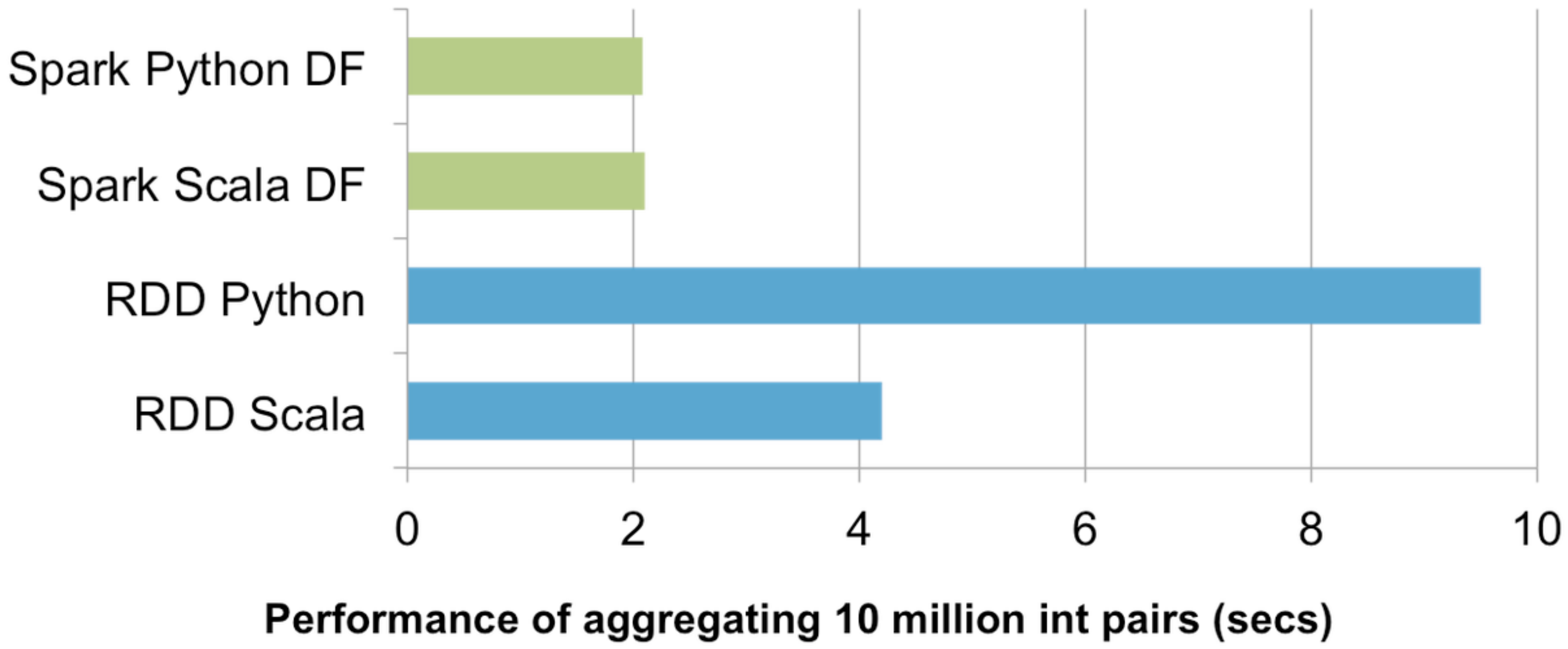
  ([a, b, c]) => (a, [a, b, c]) => (c, [a, b, c]) etc

- Flexibility hinders optimiser

SparkSQL & DataFrames

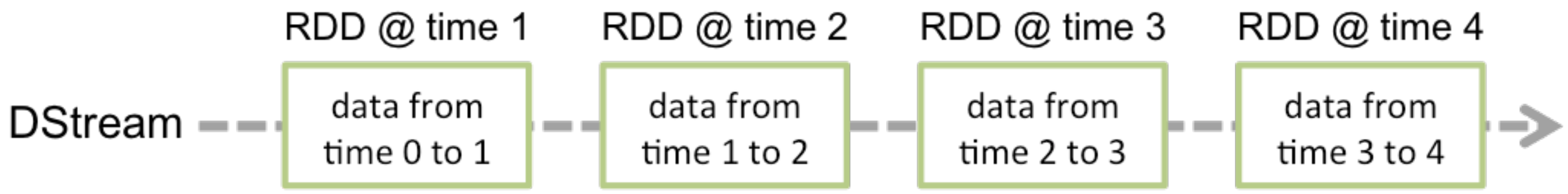
- Inspiration from SQL & Pandas
- Columnar data representation
- Automatically reading data in Avro, CSV, JSON, .. format
- Easy conversion from/to RDD's

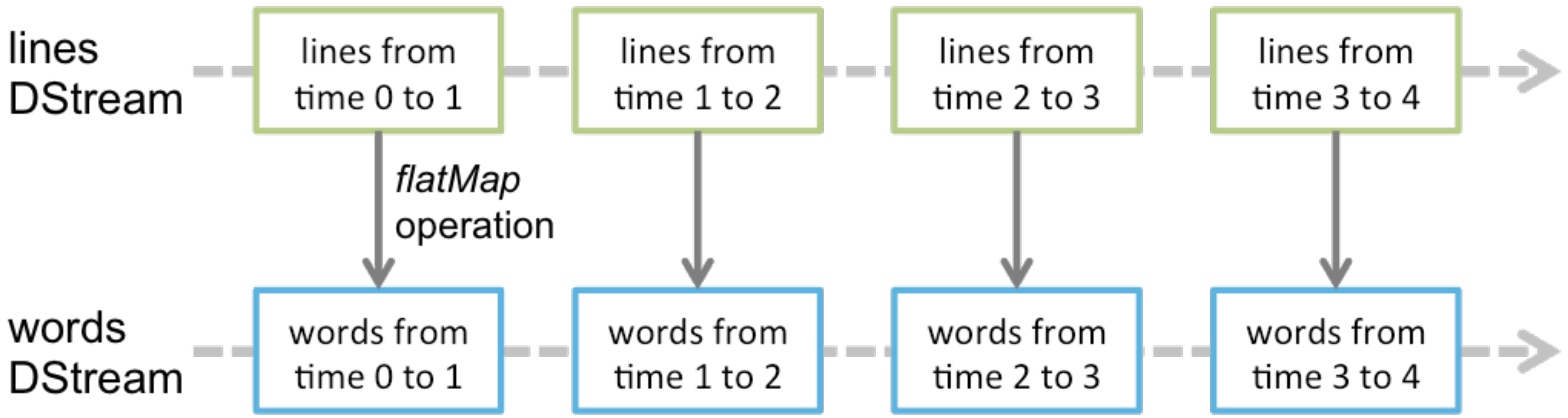
DataFrame performance



https://databricks.com/blog/2015/04/24/recent-performance-improvements-in-apache-spark-sql-python-dataframes-and-more.html

Discretized Streams





http://spark.apache.org/docs/latest/streaming-programming-guide.html

SURF SARA

Spark Streaming

Spark uses microbatches to get close to real-time performance
Intervals for batch creation can be set

http://spark.apache.org/docs/latest/streaming-programming-guide.html

SURF SARA

Streaming data sources

- Kafka
- Flume
- HDFS/S3
- Kinesis
- Twitter
- TCP socket
- Pluggable interface, write your own

Machine Learning Library (MLlib)

Common machine learning algorithms on top of Spark:

- classification: SVM, naive Bayes
- regression: logistic regression, decision trees, isotonic regression
- clustering: K-means, PIC, LDA
- collaborative filtering: alternating least squares
- dimensionality reduction: SVD, PCA

Deployment

- Stand-alone cluster
- On cluster scheduler (YARN / Mesos)
- Local, single machine
  (easy way to get started: docker-stacks)

Usage

- Interactive shell:
  - spark-shell (Scala)
  - pyspark (Python)
- Notebook
- Standalone application
  - spark-submit <jar> / <py>

Distributed data store

amazon web services™ S3

cassandra

hadoop HDFS

SURF SARA

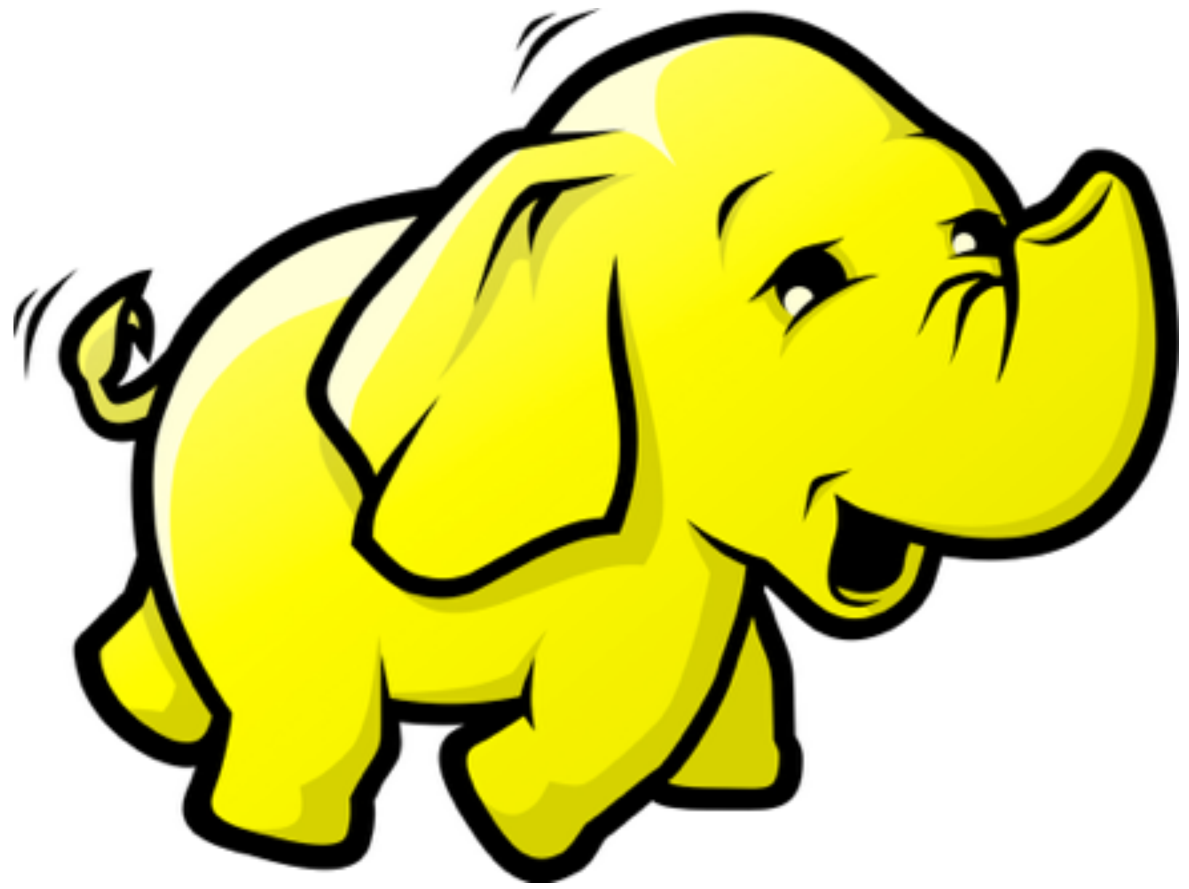
Summary

- Spark replaces MapReduce
- RDDs enable fast distributed data processing
- Learn Scala

Intermezzo

There are only two hard things in Computer Science:
cache invalidation and naming things.

-- Phil Karlton



https://pixelastic.github.io/pokemonorbigdata/