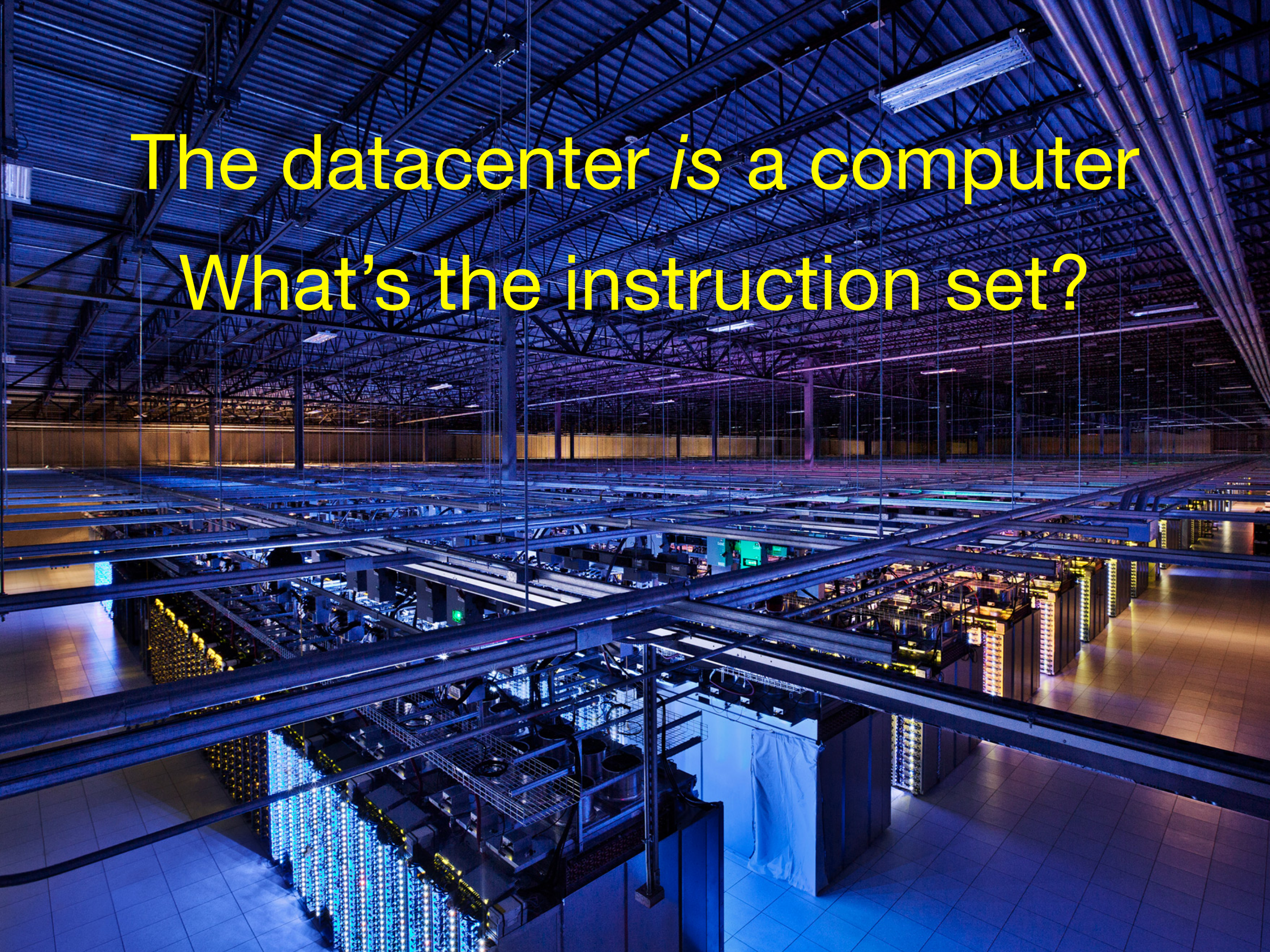
Advanced Cloud Computing

Spark Internals

Wei Wang
CSE@HKUST
Spring 2025

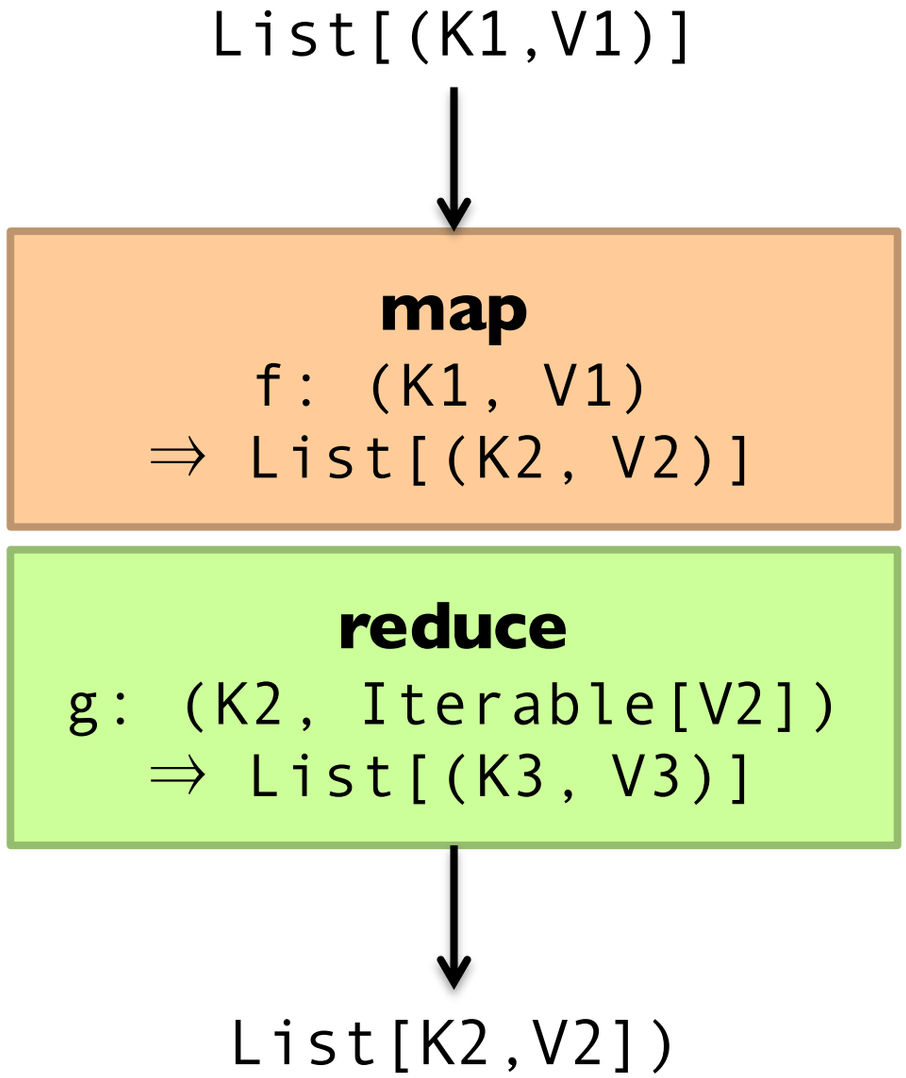
THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系



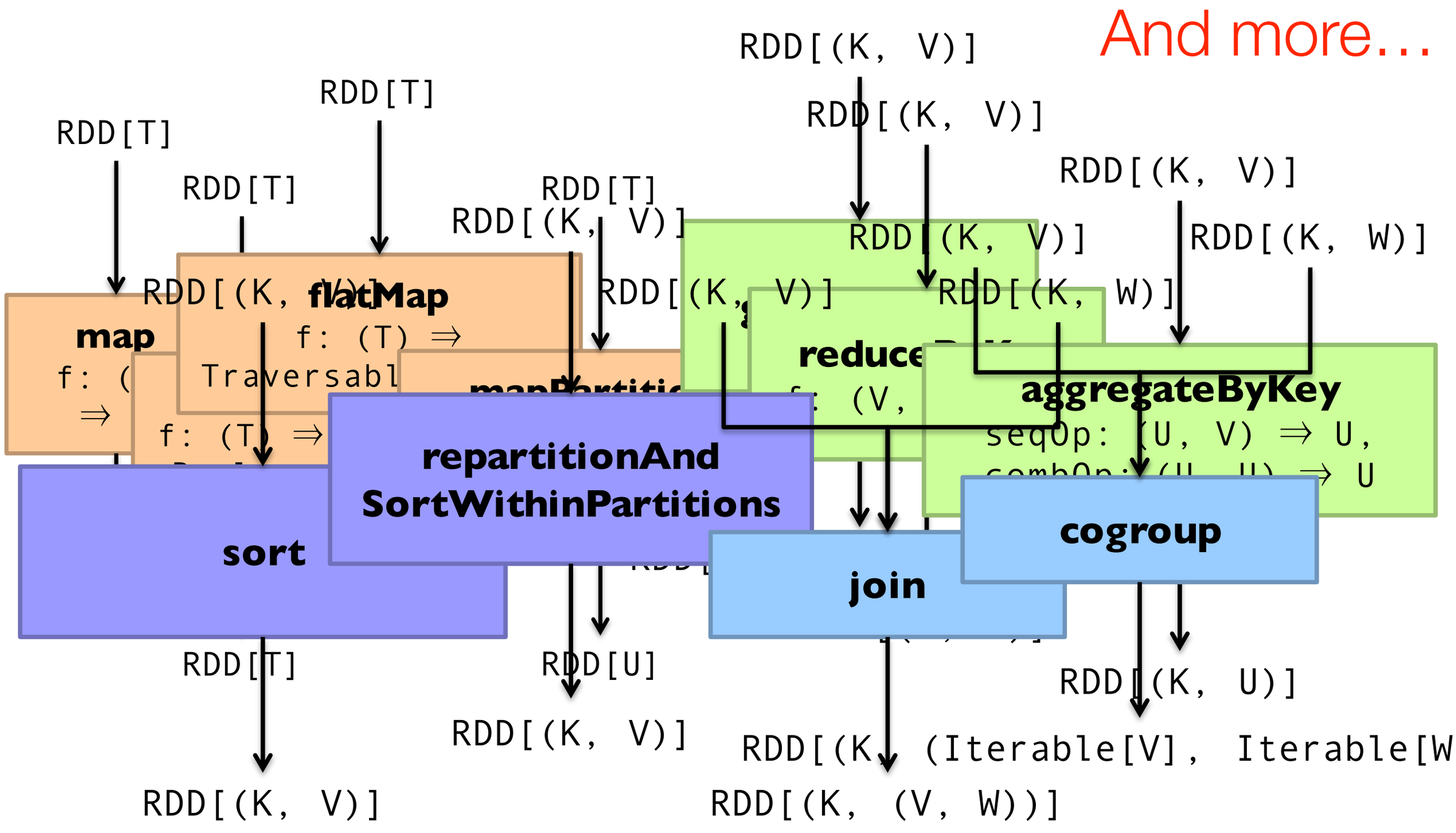
The datacenter *is* a computer

What's the instruction set?

MapReduce


List[(K1,V1)]
map
f: (K1, V1)
⇒ List[(K2, V2)]
reduce
g: (K2, Iterable[V2])
⇒ List[(K3, V3)]
List[K2,V2])

Spark


And more...
RDD[T]
RDD[T]
RDD[T]
RDD[T]
RDD[(K, V)]
RDD[(K, V)]
RDD[(K, V)]
RDD[(K, V)]
RDD[(K, W)]
RDD[(K, W)]
map
f: (
⇒
flatMap
f: (T) ⇒
Traversabl
f: (T) ⇒
reduce
f: (V,
aggregateByKey
seqOp: (U, V) ⇒ U,
repartitionAnd
SortWithinPartitions
sort
join
cogroup
RDD[T]
RDD[U]
RDD[(K, V)]
RDD[(K, U)]
RDD[(K, (Iterable[V], Iterable[W]
RDD[(K, V)]
RDD[(K, (V, W))]

What's an RDD?

Resilient Distributed Dataset

= immutable = partitioned

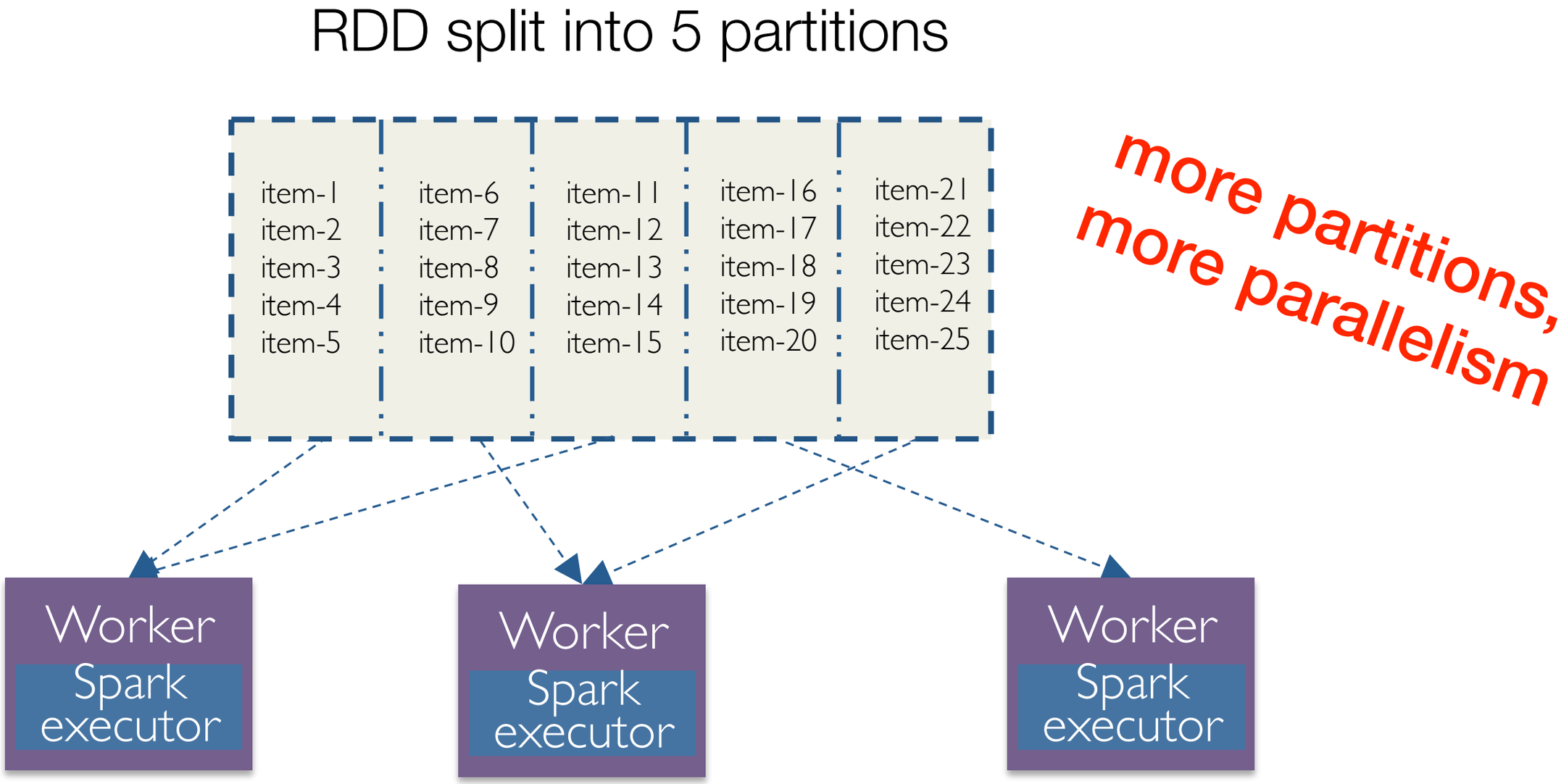
Wait, but how do you actually do anything?

Developers define *transformations* on RDDs

Framework keeps track of *lineage*

RDD is a distributed dataset

Programmer specifies number of **partitions** for an RDD

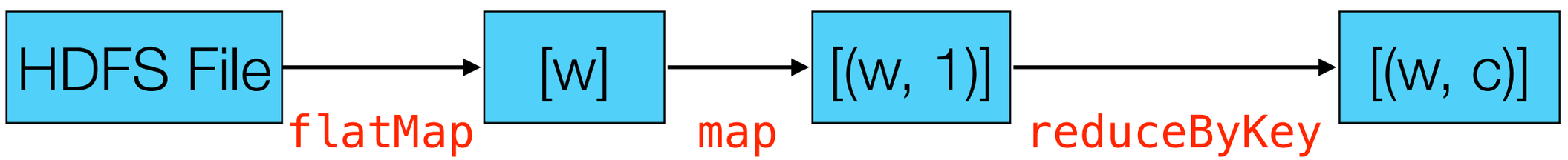


Spark WordCount

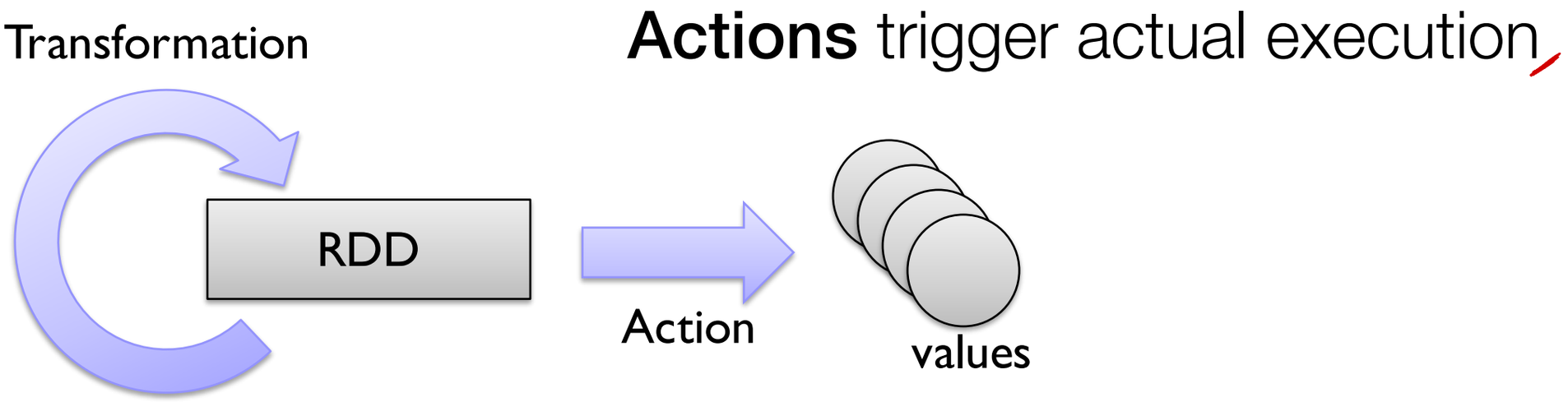
```
# create an RDD from HDFS
text_file = sc.textFile(“hdfs://...")

text_file.flatMap(lambda line: line.split(" ")) \
         .map(lambda word: (word, 1)) \
         .reduceByKey(lambda a, b: a + b) \
         .saveAsTextFile(“hdfs://...")
```

default partition

one to map



RDD lifecycle



Transformations are *lazy*:
Framework keeps track of *lineage*

Spark WordCount

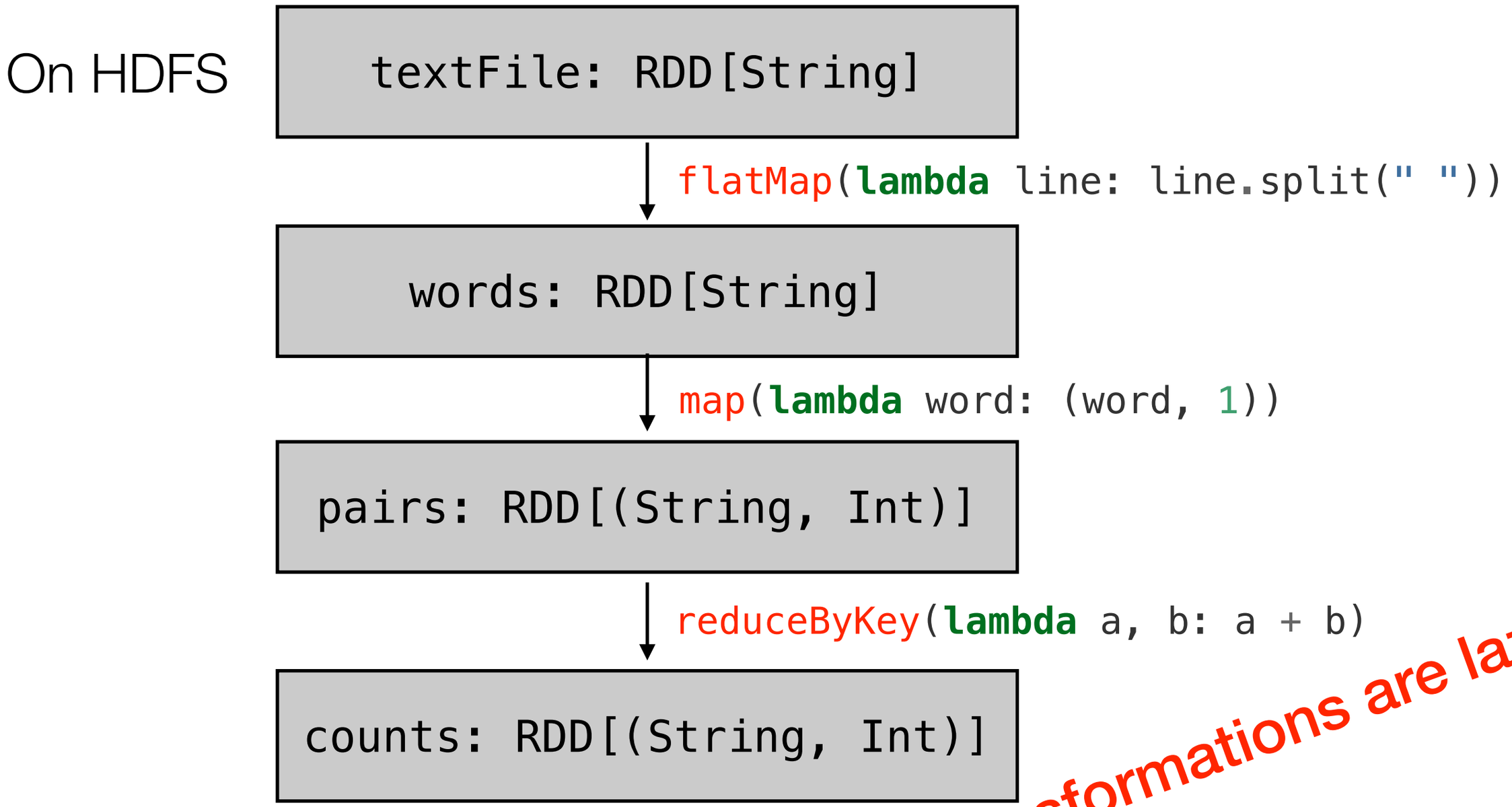
RDDs

```
# create an RDD from HDFS
text_file = sc.textFile(“hdfs://...")

words = text_file.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile(“hdfs://...")
```

Action

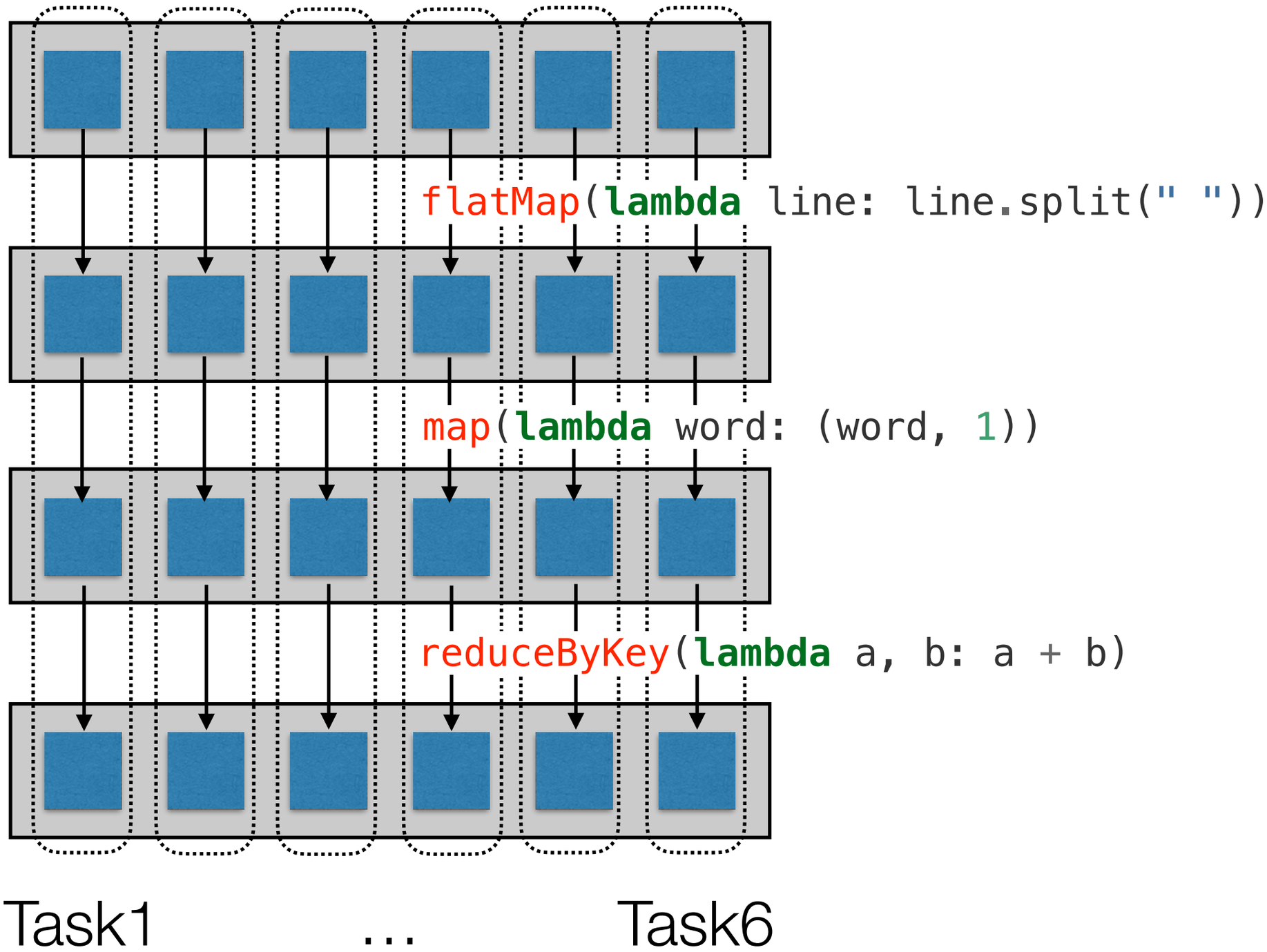
Transformations

RDDs and lineage

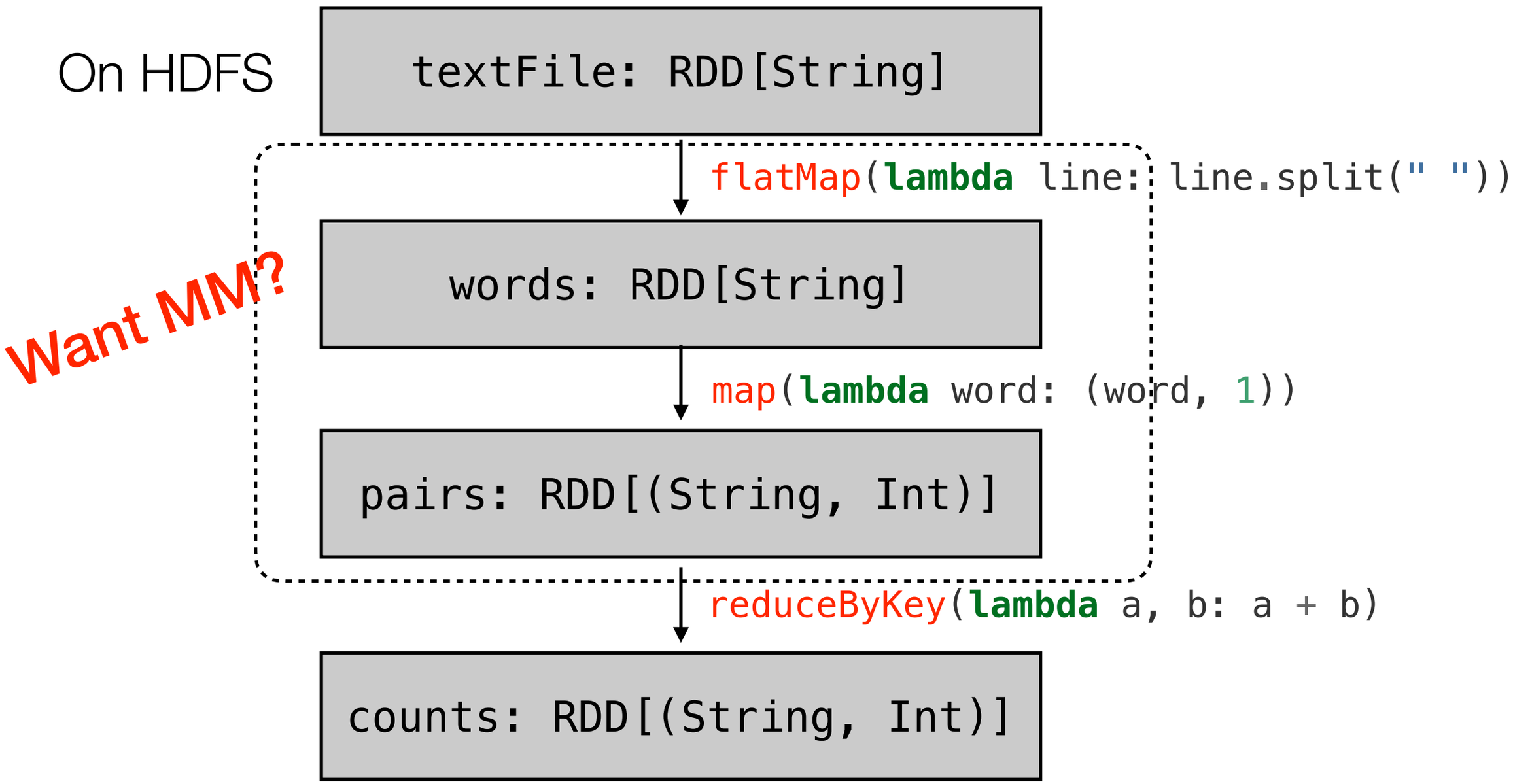


Partition-level view

On HDFS

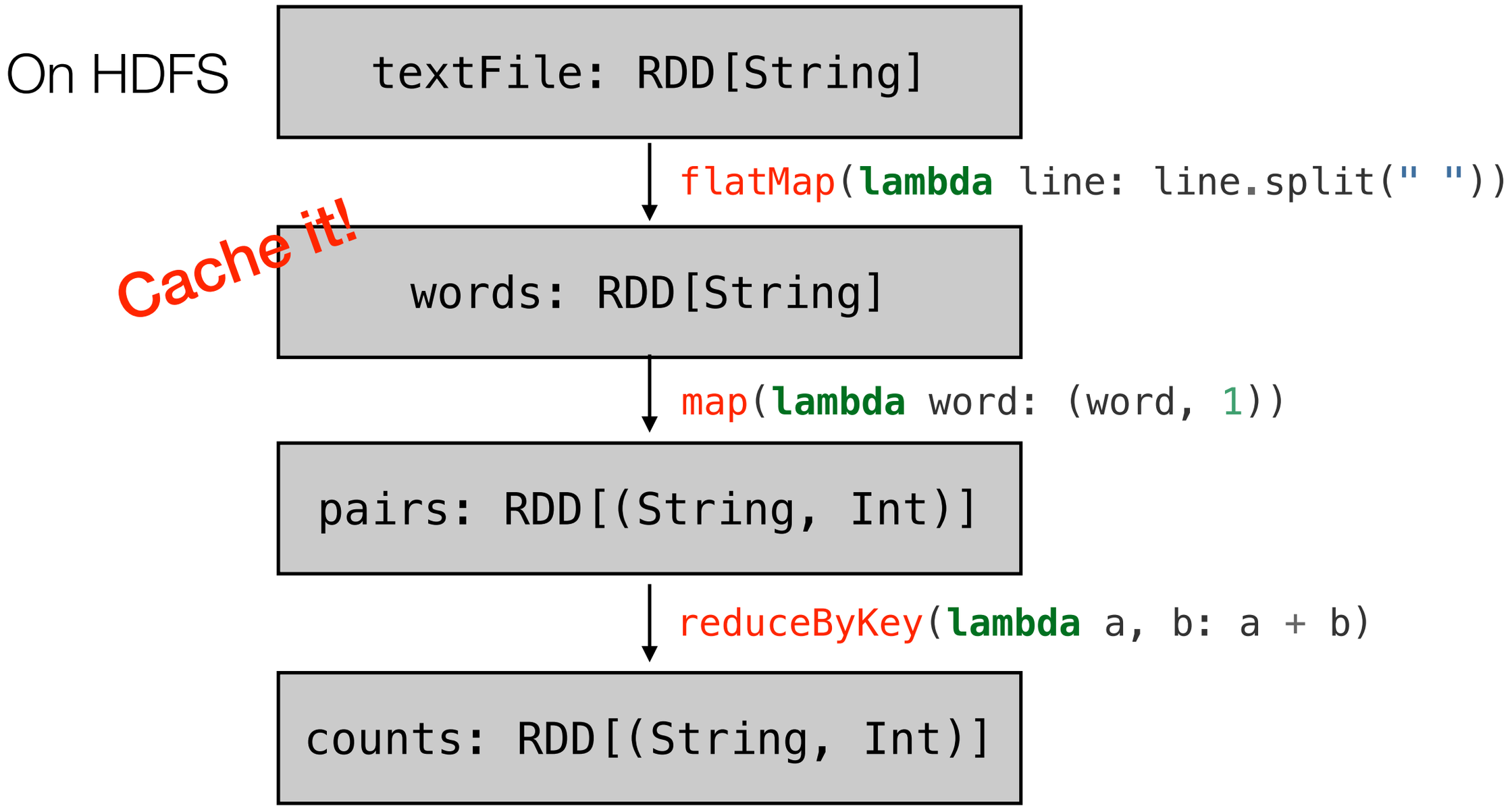


RDDs and optimizations



Action! `saveAsTextFile("hdfs://...")`

RDDs and caching

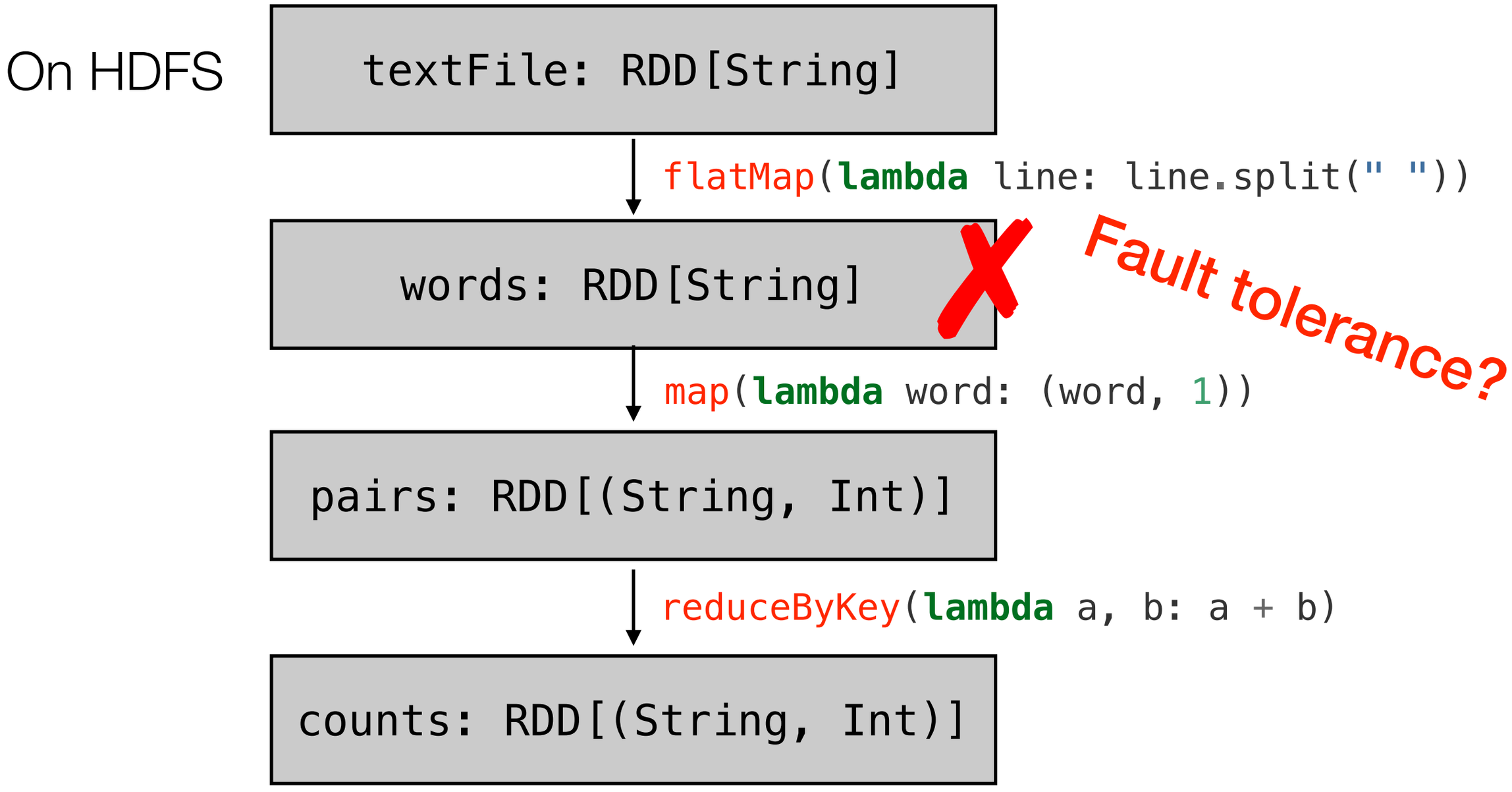


Action! `saveAsTextFile("hdfs://...")`

The RDDs are too large to be cached in memory?

Cache the RDDs in *partial*, and spill the remaining partitions to disk

RDDs and fault tolerance

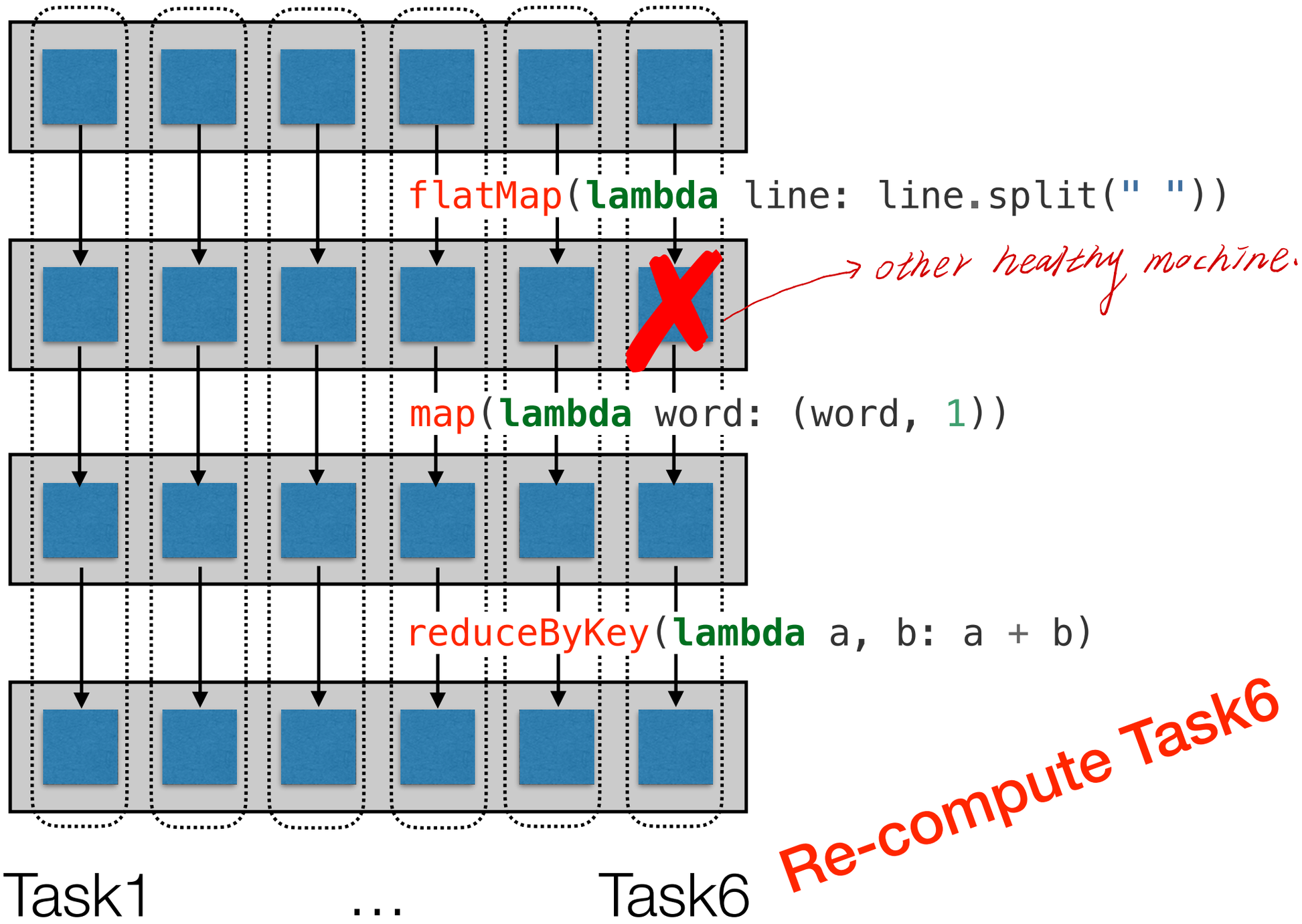


Action! `saveAsTextFile("hdfs://...")`

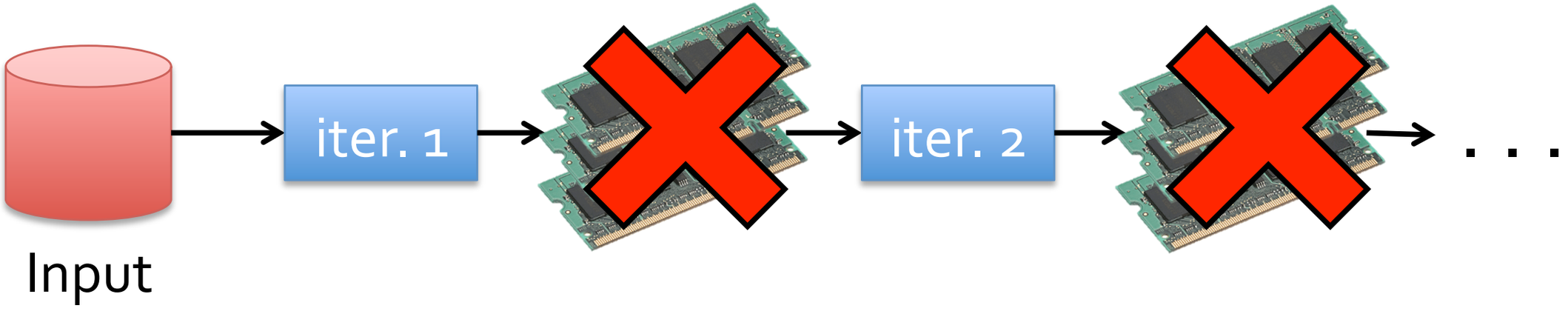
Replay the lineage to *re-compute* lost RDD partitions

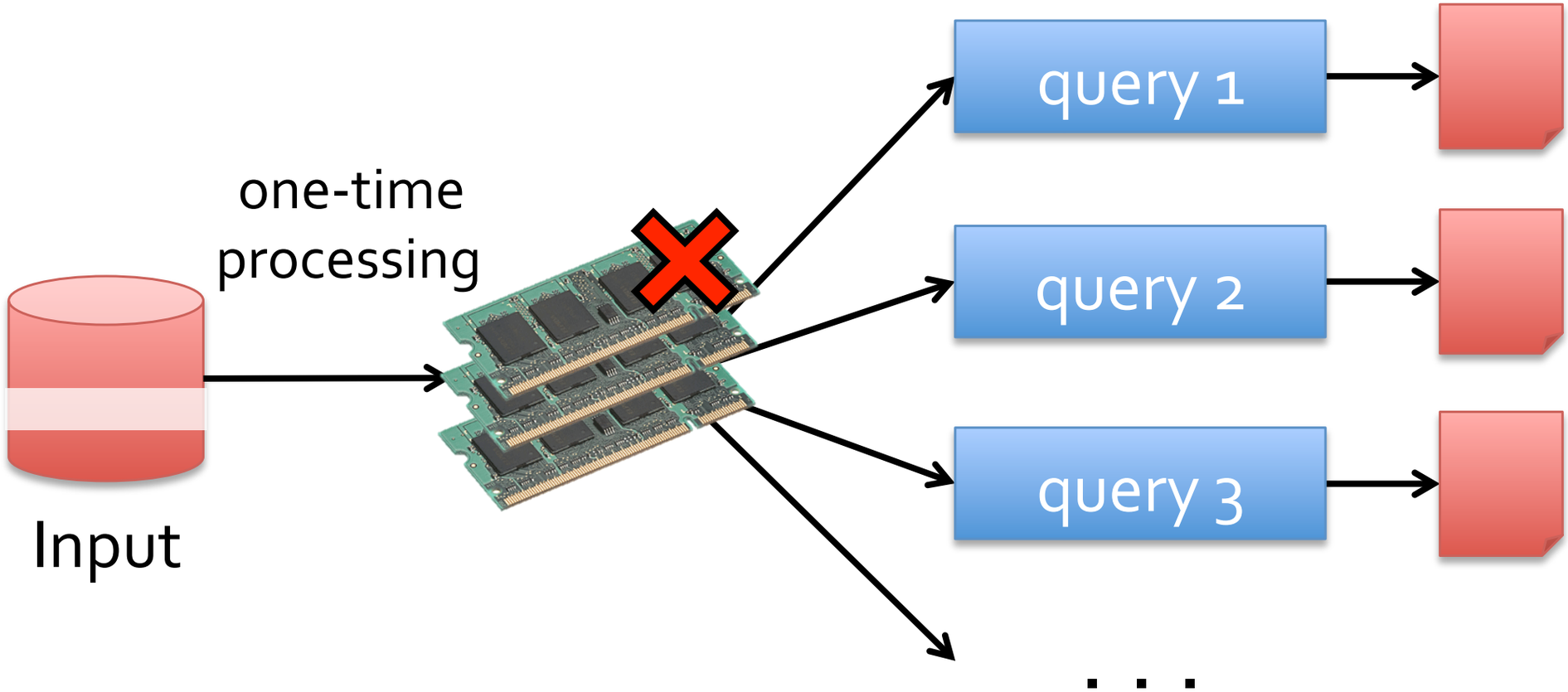
RDD recovery

On HDFS



RDD recovery





Zaharia, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” Talk in NSDI 2012.

Replay the lineage to *re-compute* lost RDD partitions

Does this work for ***any*** transformations (e.g., deterministic, randomized)?

if encounter

Recap: What's an RDD?

Set of partitions

List of dependencies on parent RDDs

Function to compute a partition given its parents

lineage

(Optional) *partitioner* (hash, range)

(Optional) *preferred locations* for each partition

optimized execution

text_file RDD

```
text_file = sc.textFile(“hdfs://...")
```

lineage.

partitions = one per HDFS block

dependencies = none

compute = read corresponding block

preferredLocations = HDFS block location

× don't need to shuffle.

partitioner = none

`words` RDD

```
text_file = sc.textFile(“hdfs://...”)

words = text_file.flatMap(lambda line: line.split(" "))
```

partitions = same as parent RDD (`text_file`)

dependencies = “one-to-one” on parent

compute = compute parent and apply `flatMap`

preferredLocations = none (ask parent)

follow parent.

partitioner = none

join'ed RDD

created from python collection.

```
rdd1 = sc.parallelize([(“foo”, 1), (“bar”, 2), (“baz”, 3)])
rdd2 = sc.parallelize([(“foo”, 4), (“bar”, 5), (“bar”, 6)])
joinedRDD = rdd1.join(rdd2)
```

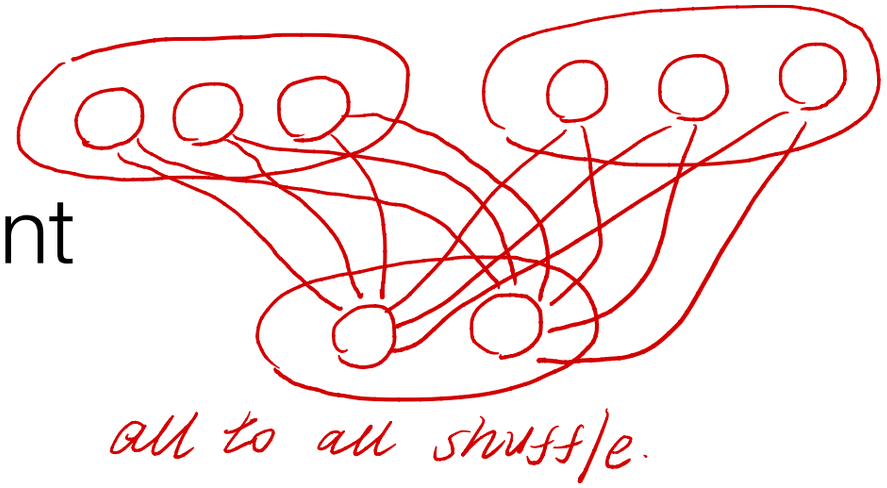
partitions = one per reduce task

dependencies = “shuffle” on each parent

compute = read and join shuffled data

preferredLocations = none

partitioner = HashPartitioner(numTasks)



Quiz: What is an RDD?

A: distributed collection of objects on disk

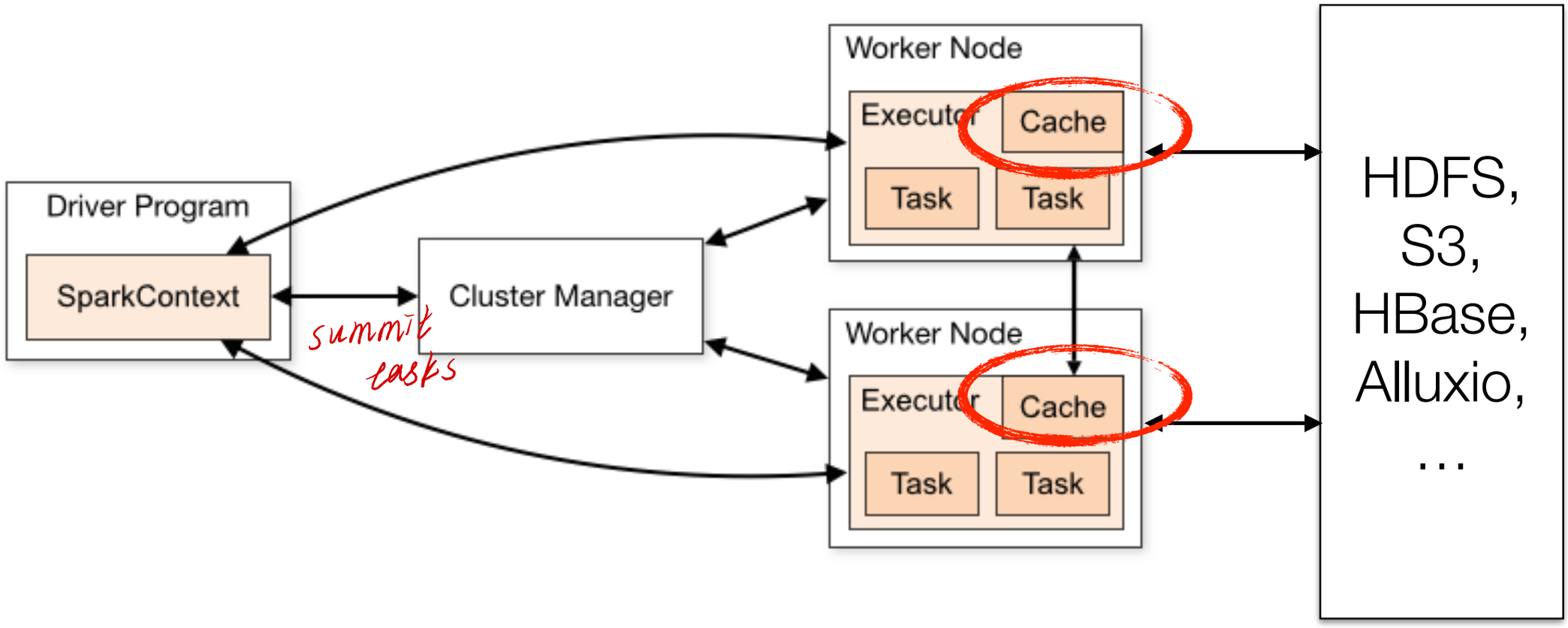
B: distributed collection of objects in memory

C: distributed collection of objects in Cassandra

Answer: could be any of the above!

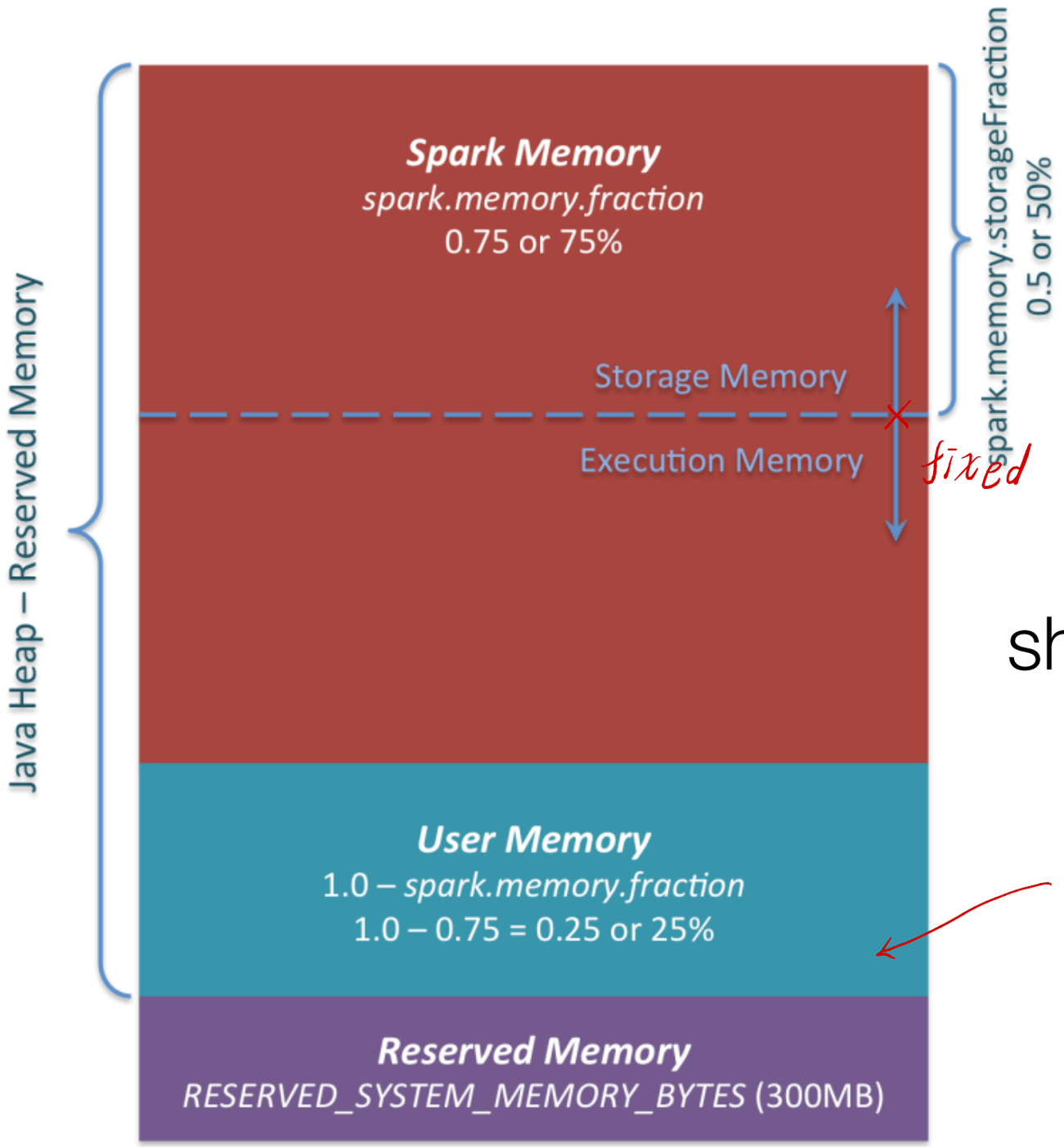
Spark architecture

One task is launched for each partition



How memory is managed in Spark?

Memory management



LRU cache for RDDs

moving boundary

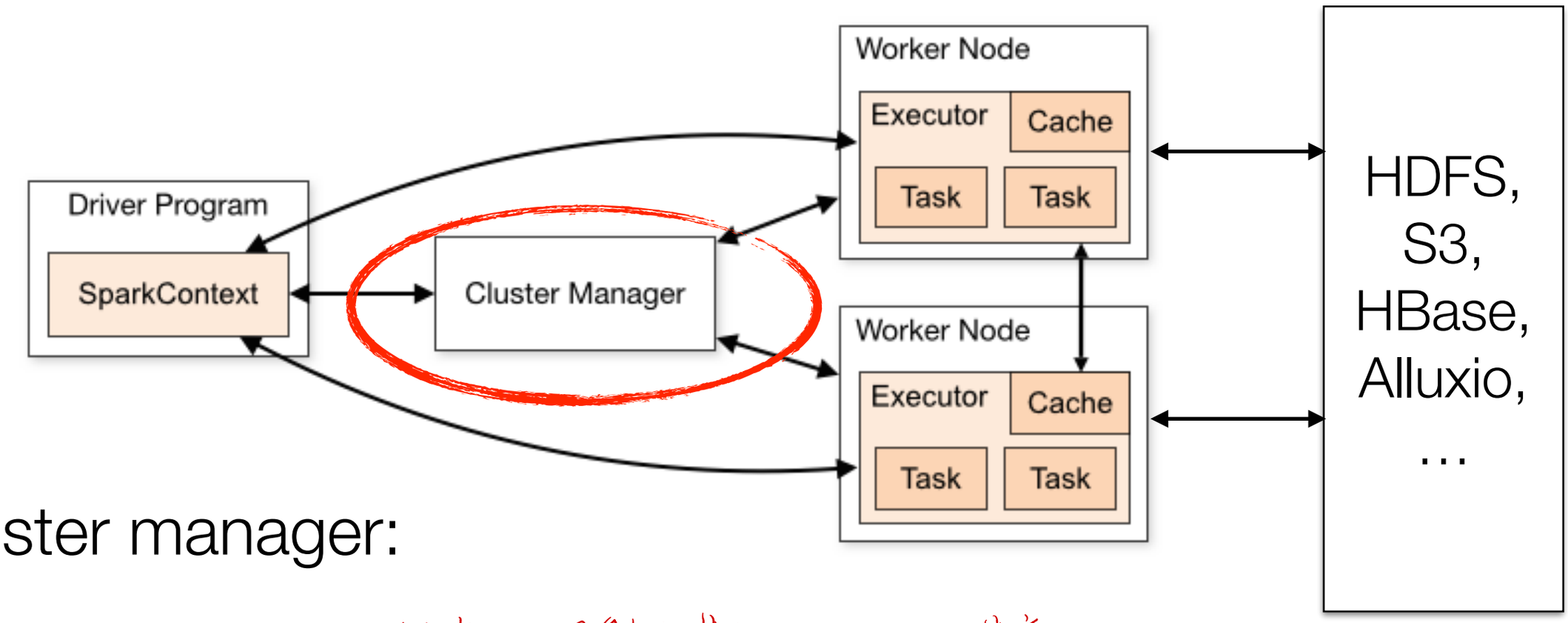
shuffle intermediate buffer

your data structures

reserved by Spark

Spark architecture

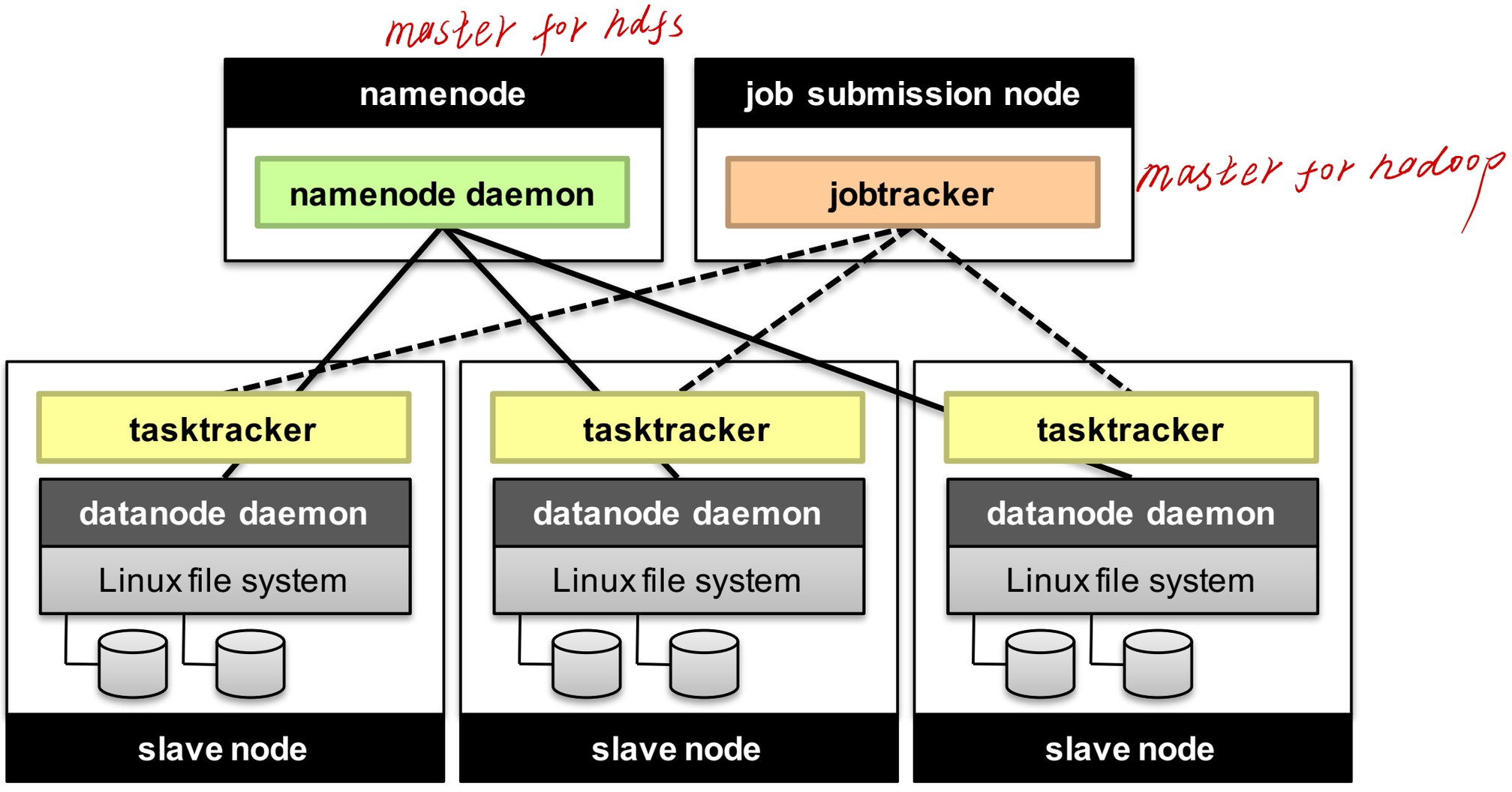
One task is launched for each partition



Cluster manager:

- standalone 分生b的，不能与其它framework共享。
- external: Mesos, YARN, Borg, Kubernetes, etc.

Compare with Hadoop



YARN

Hadoop's (original) limitations:

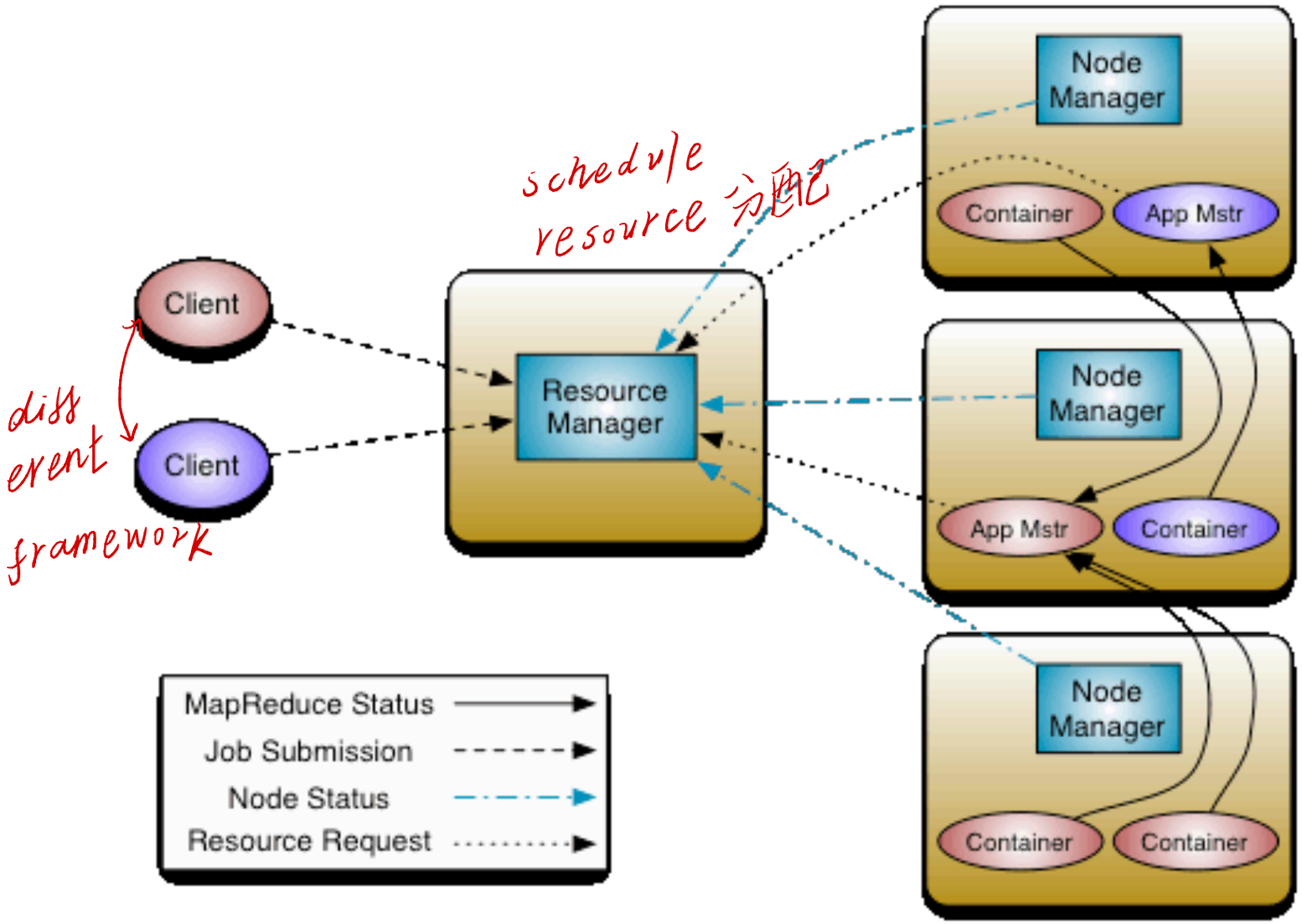
- can only run MapReduce
- what if we want to run other distributed frameworks?

cluster resource management.

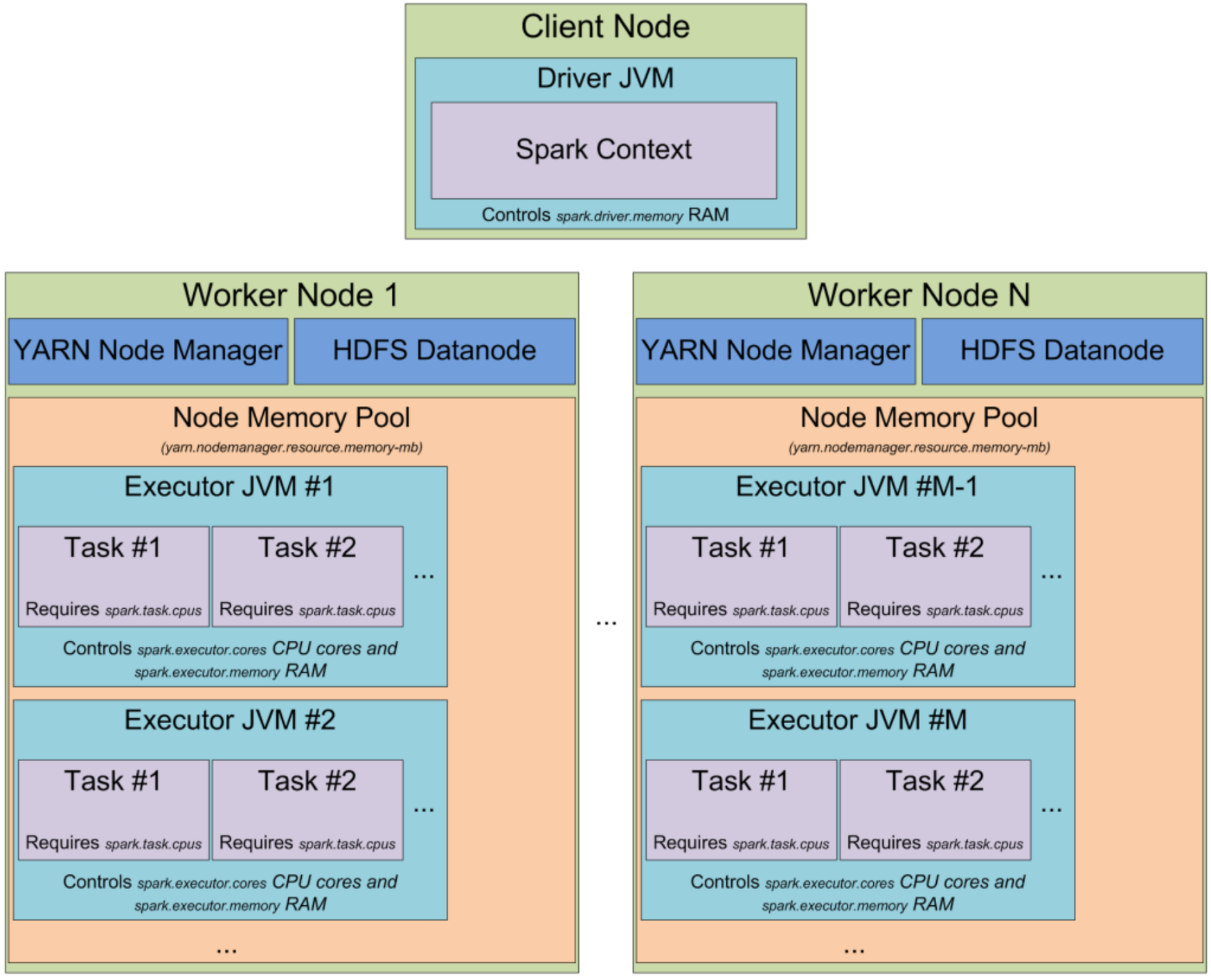
YARN = Yet-Another-Resource-Negotiator

- provides API to develop any generic distribution apps
- handles scheduling and resource request
- MapReduce (MR2) is one such app in YARN, so is Spark

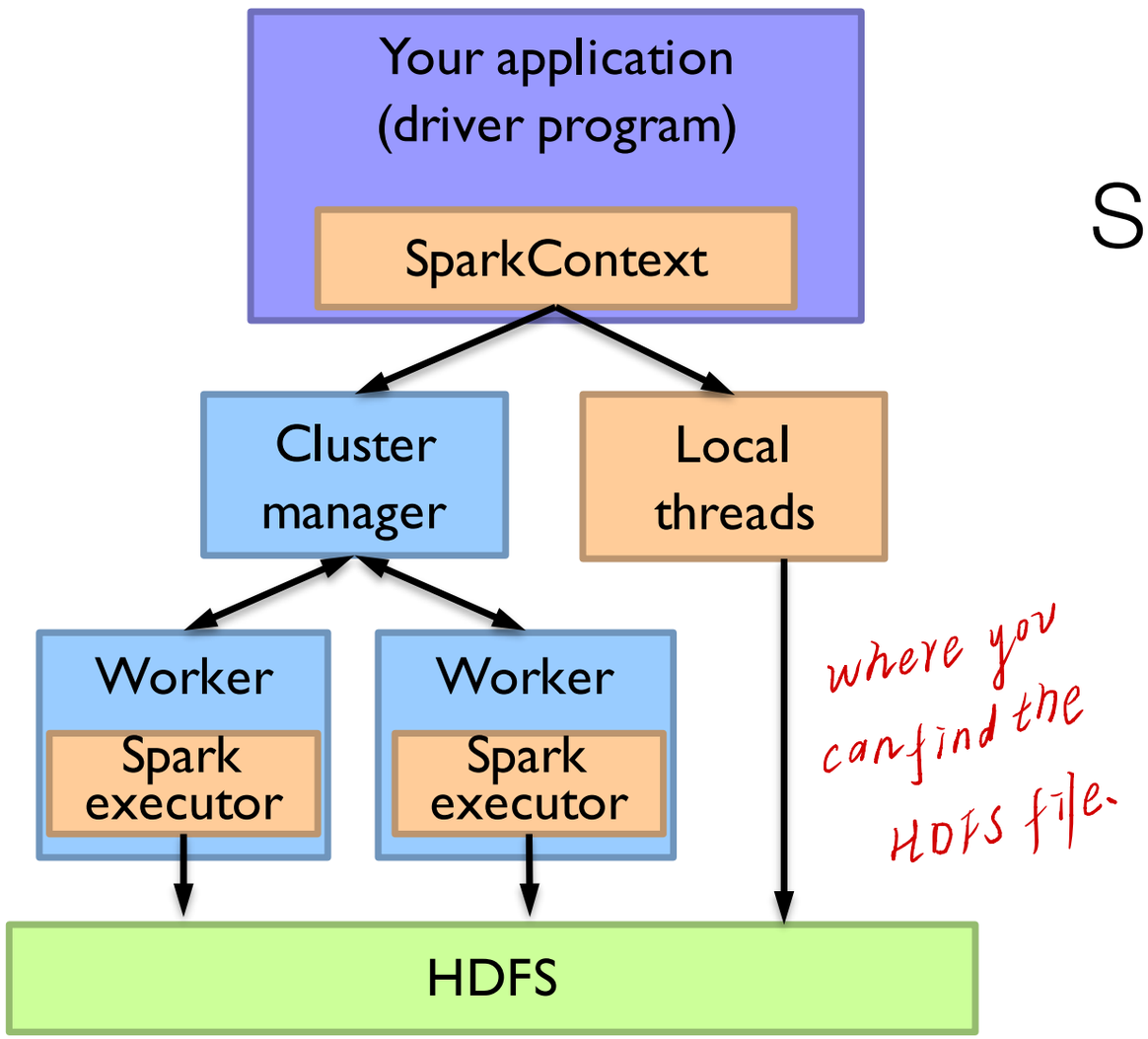
YARN


schedule
resource 资源
Client
Client
diff
erent
framework
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

Spark on YARN



Spark programs



Multi-language:

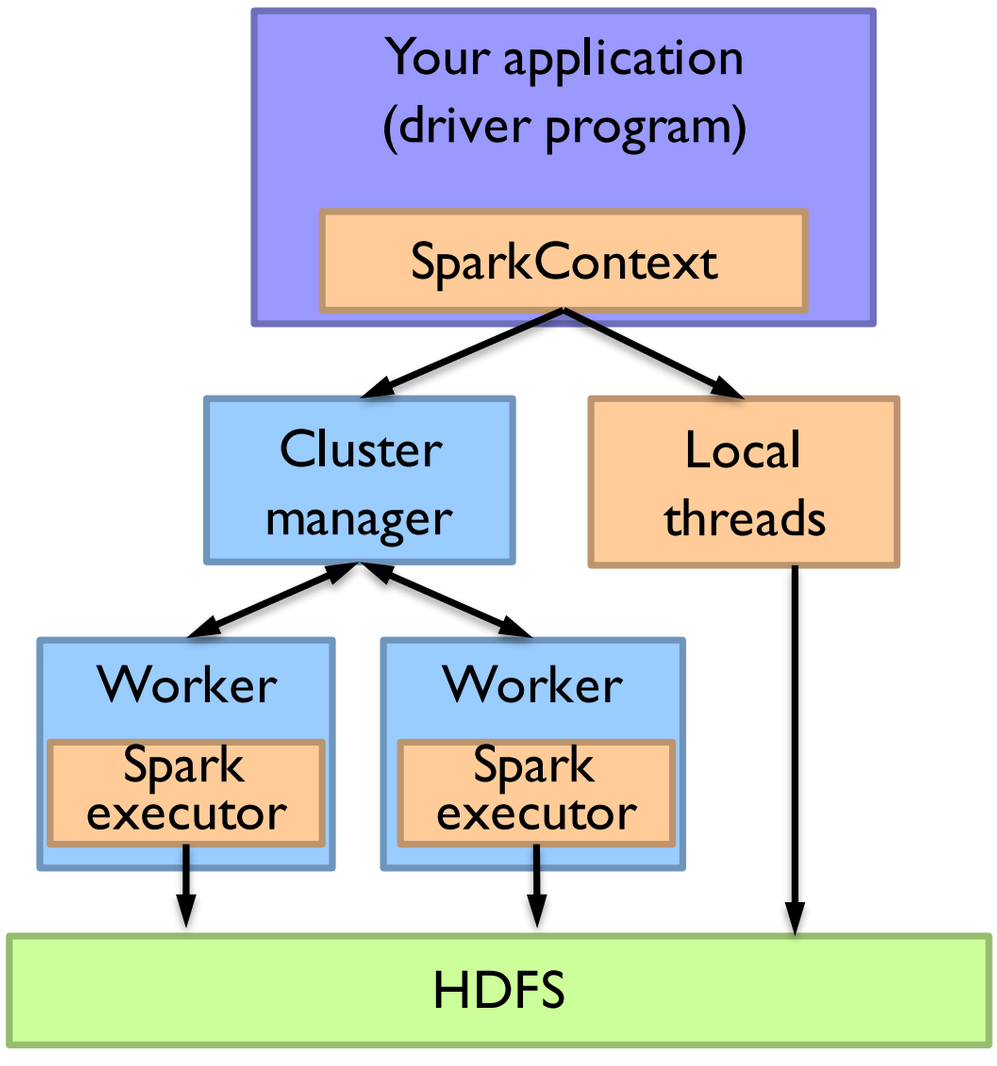
- Scala, Java, Python, R

Spark context

- tells the framework where to find the cluster
- used to create RDDs

Spark driver

`spark-shell` `spark-submit`



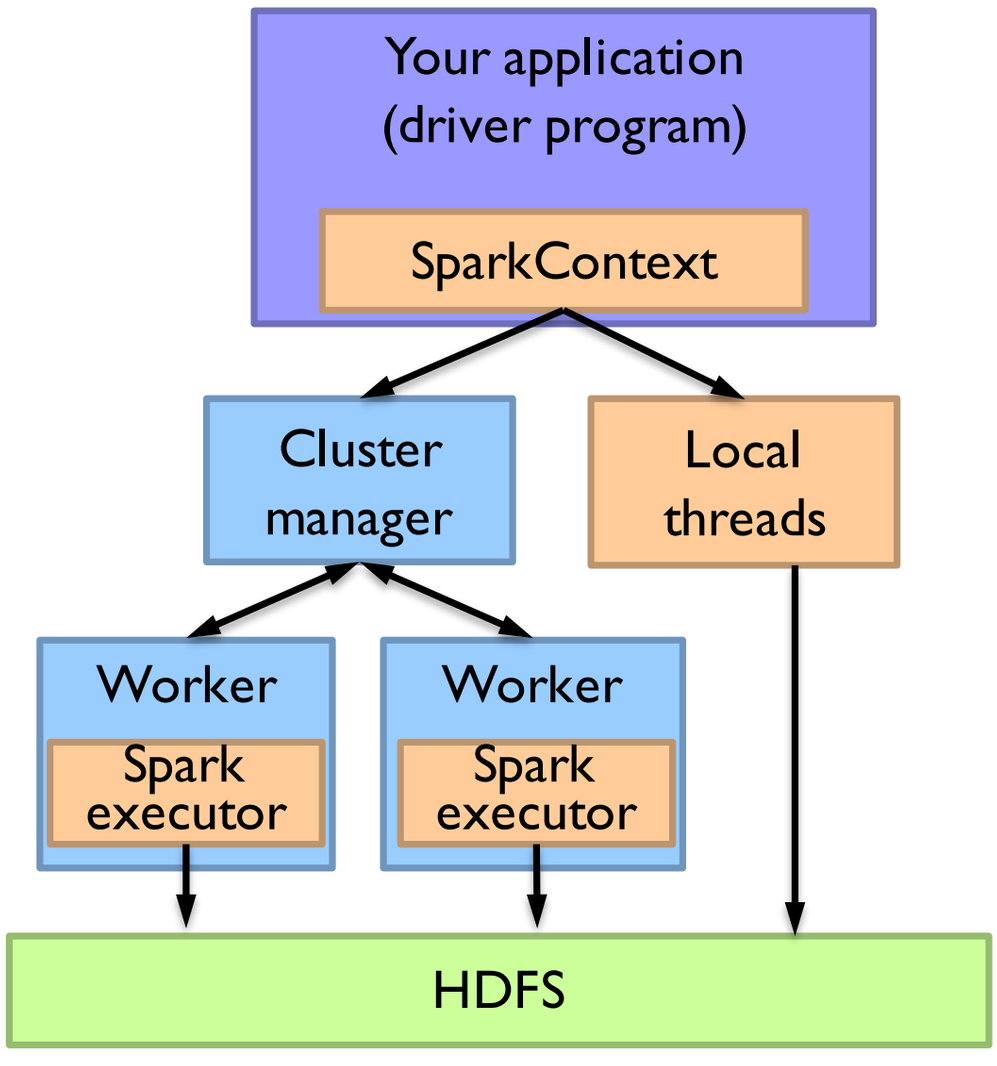

```
# create an RDD from HDFS
text_file = sc.textFile(“hdfs://...")

text_file.flatMap(lambda line: line.split(" ")) \
         .map(lambda word: (word, 1)) \
         .reduceByKey(lambda a, b: a + b) \
         .saveAsTextFile(“hdfs://...")
```

What’s happening to the functions?

physical operators (upcoming)

Spark driver

`spark-shell` `spark-submit`



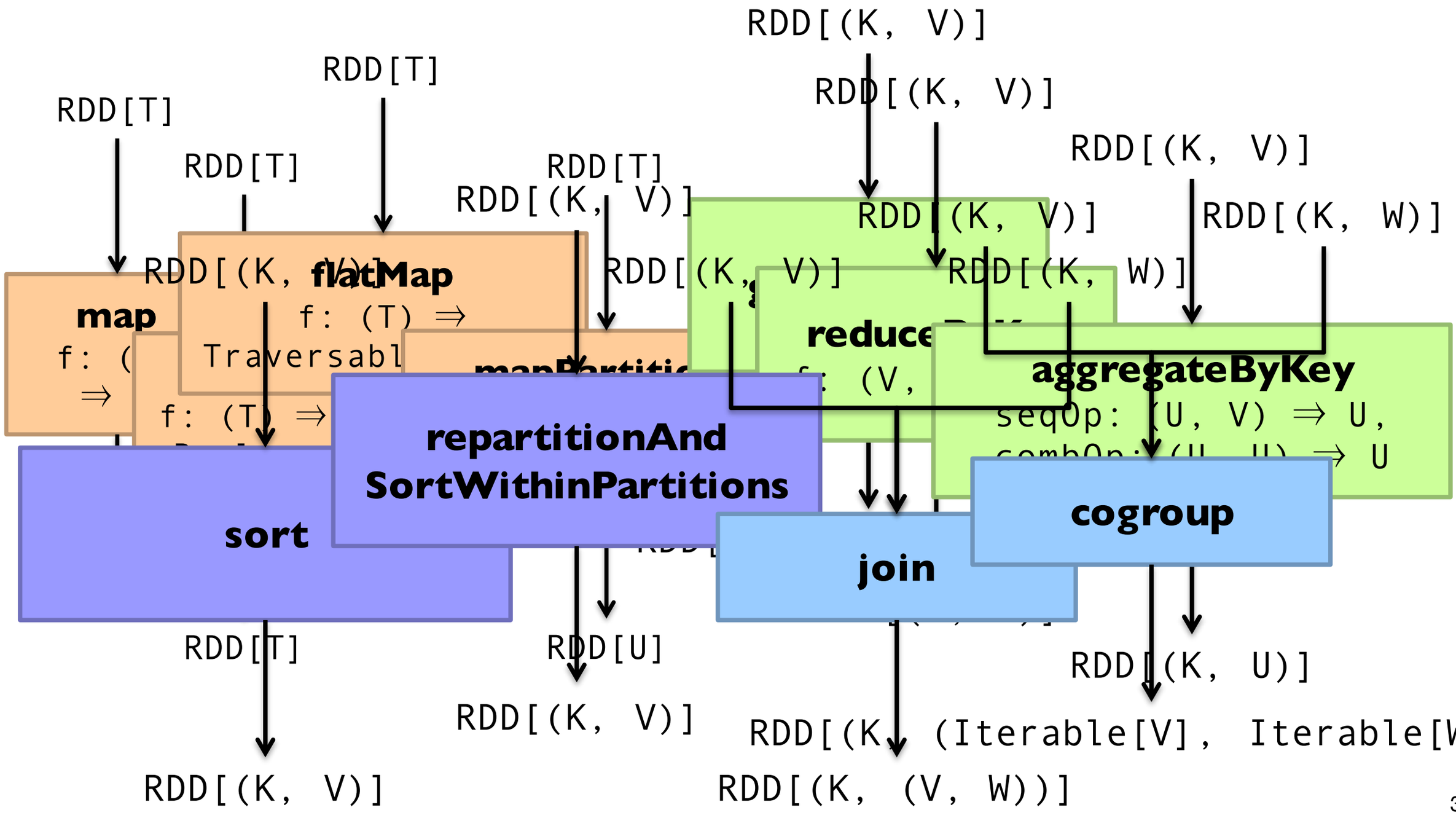

```
# create an RDD from HDFS
text_file = sc.textFile(“hdfs://...")

text_file.flatMap(lambda line: line.split(" ")) \
         .map(lambda word: (word, 1)) \
         .reduceByKey(lambda a, b: a + b) \
         .saveAsTextFile(“hdfs://...")
```

Beware of the collection action!

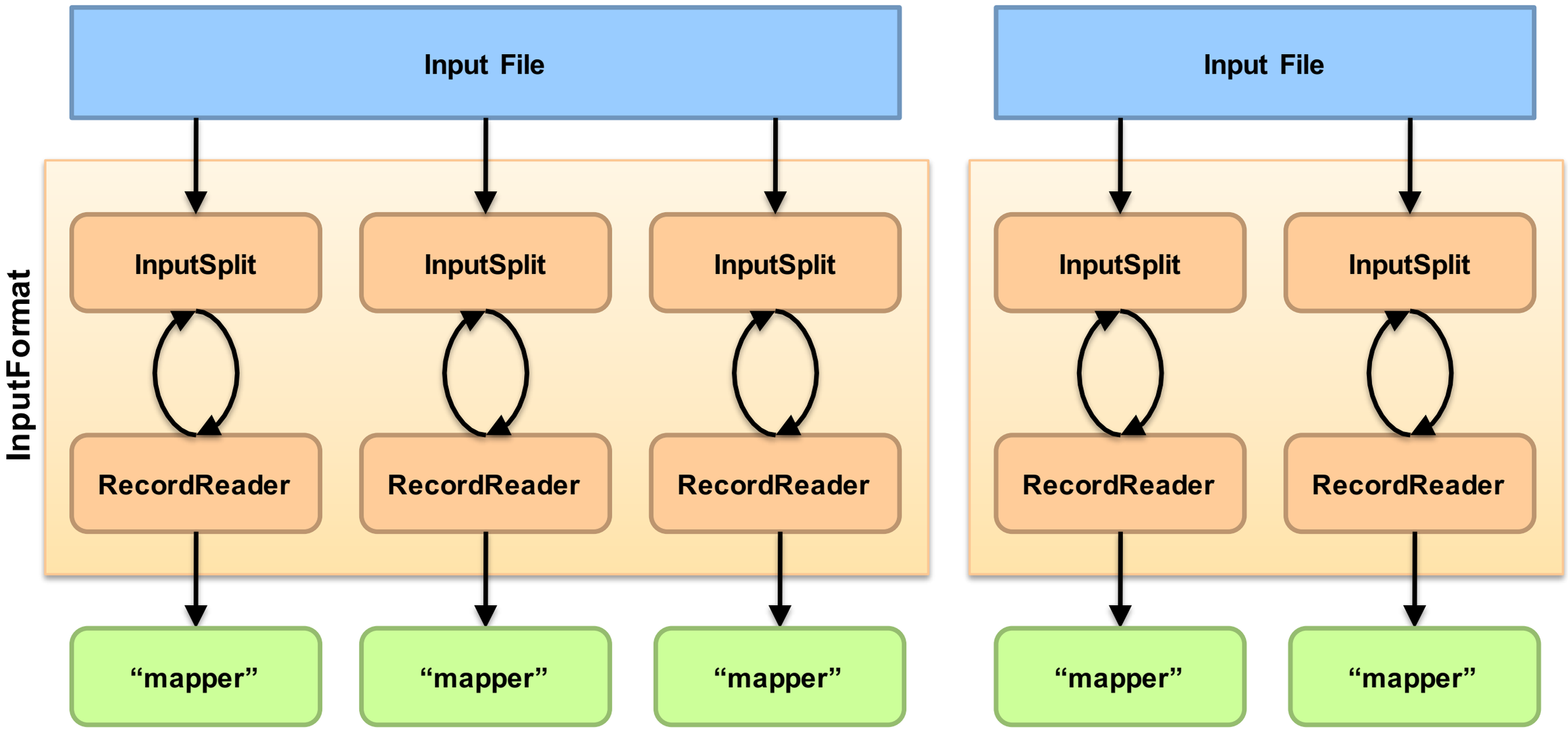
may overflow

Spark Python APIs

http://spark.apache.org/docs/2.0.0/api/python/pyspark.html



Starting points

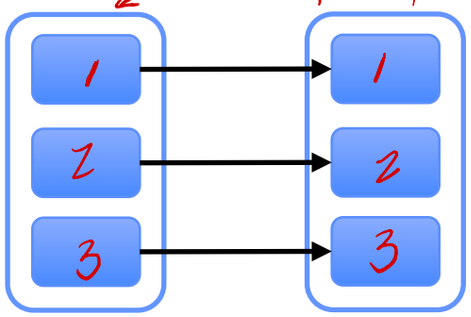


Physical operators

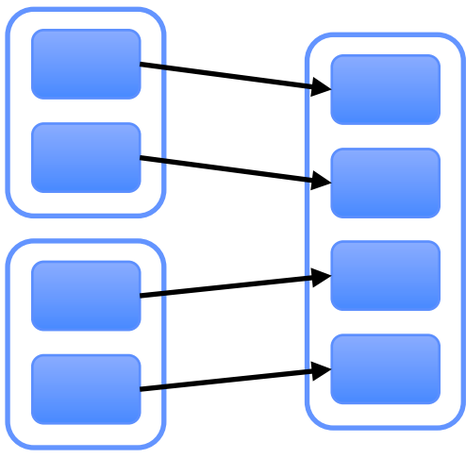
narrow dependency

"Narrow" (pipeline-able)

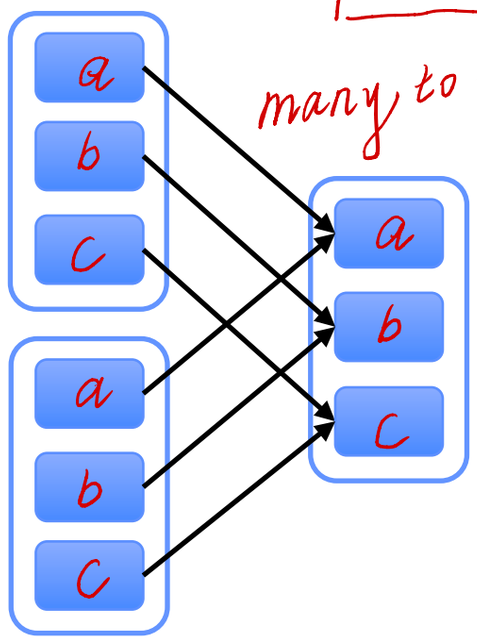
depend on a RDD



map, filter

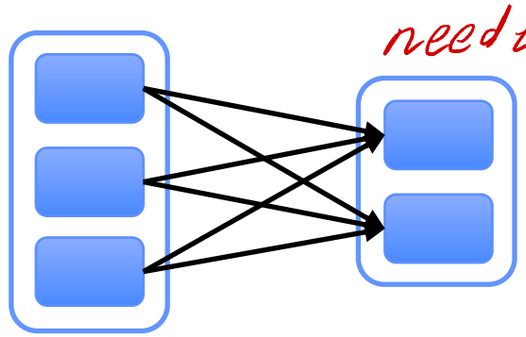


union

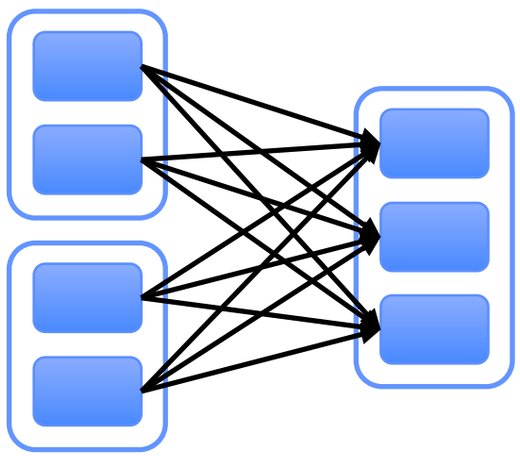


join with inputs
co-partitioned

"Wide" (shuffle)



groupByKey on
non-partitioned data



join with inputs not
co-partitioned

property: don't need to wait to other parent RDDs fully completed to compute next RDD

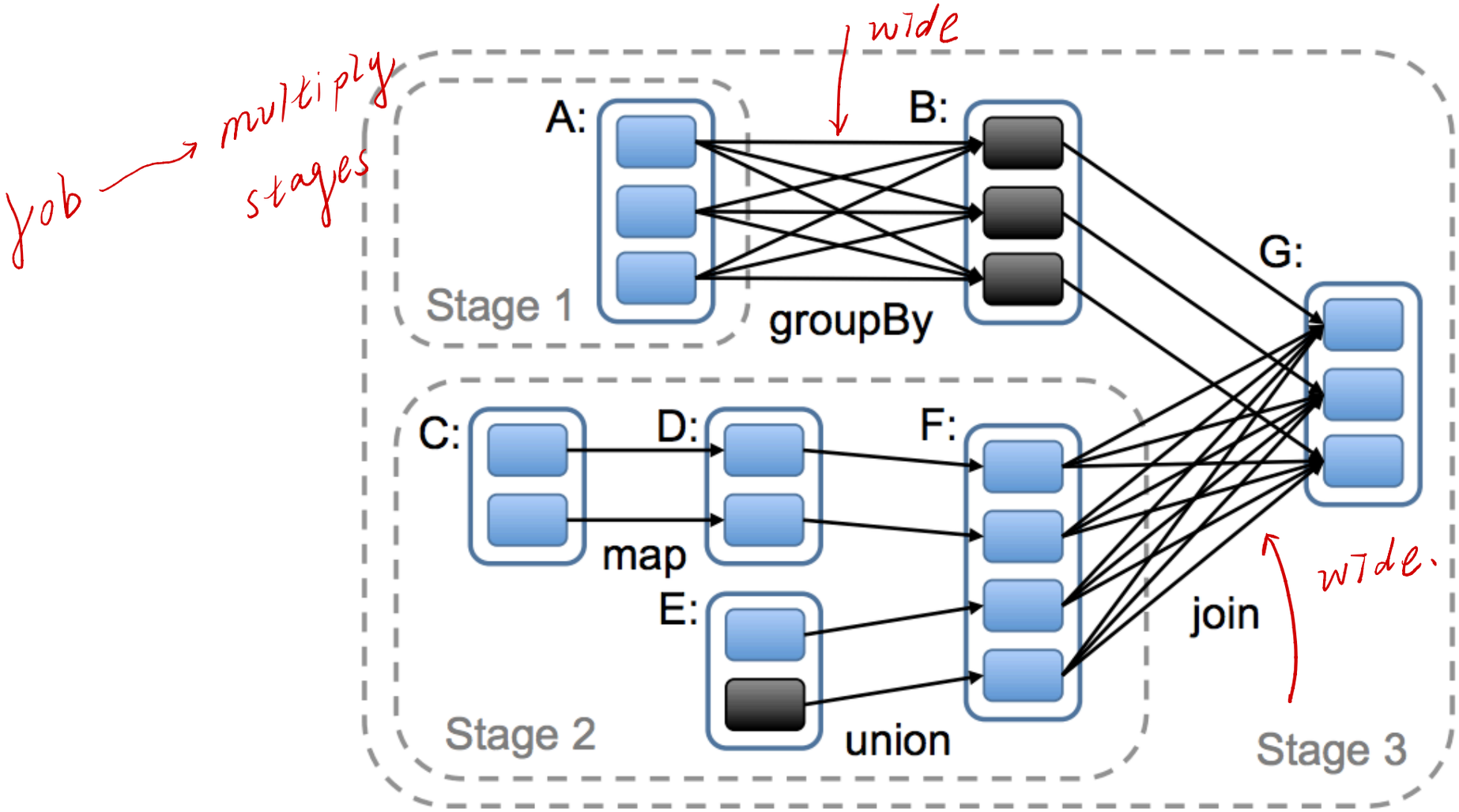
w/o shuffle barrier

w/ shuffle barrier

Execution plan

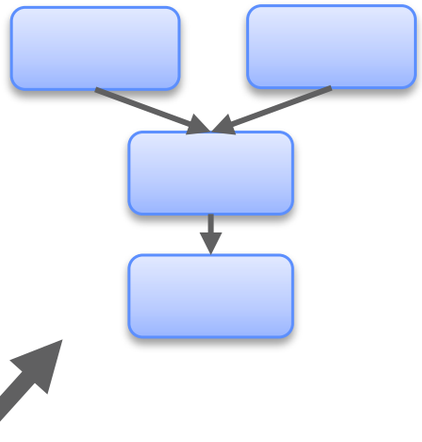
Directed Acyclic Graph (DAG)

- stage boundaries chartered by wide dependencies



Job scheduling

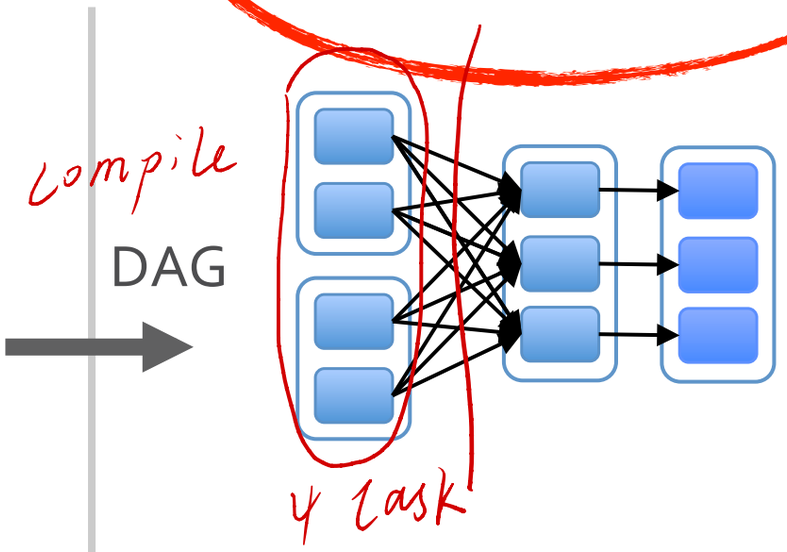
RDD Objects



```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

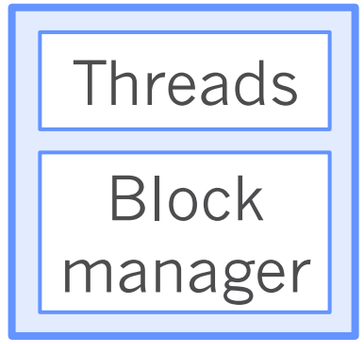
build operator DAG

Scheduler (DAGScheduler)



Executors

Task



execute tasks

store and serve blocks

DAG Scheduler

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

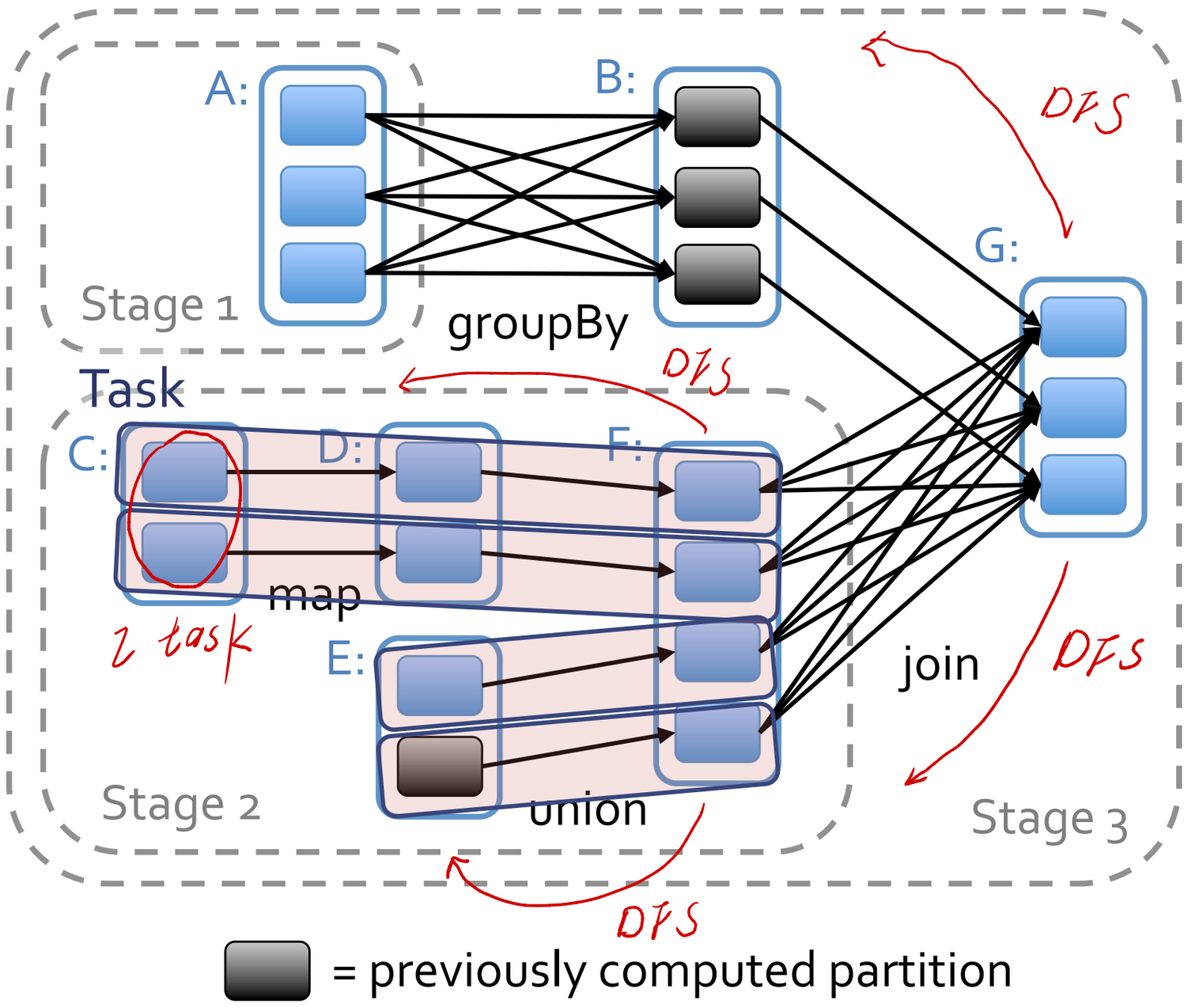
- build stages of tasks (depth-first search)
- submit tasks to lower-level scheduler, e.g., YARN, Mesos.
- lower-level scheduler schedules tasks based on locality
- resubmit failed stages, if any

Scheduler optimizations

Pipelines operations within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



Task has multiply operations.

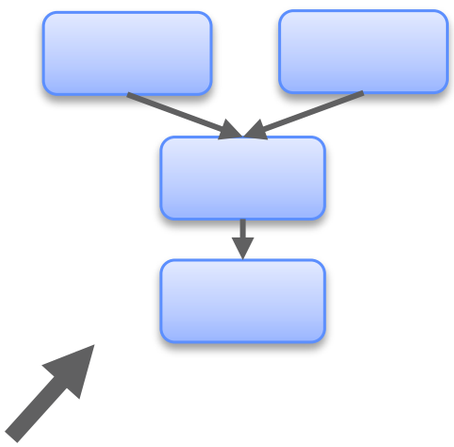
Unit of work to execute in an executor thread

Unlike MapReduce, there is no “map” vs “reduce” task

Each task either partitions its output for “shuffle,” or send the output back to the driver

How to assign tasks to executors?

RDD Objects

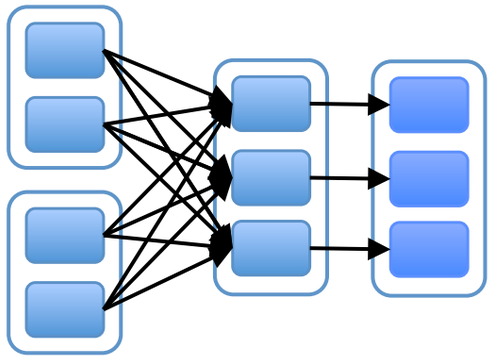


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

build operator DAG

Scheduler (DAGScheduler)

DAG

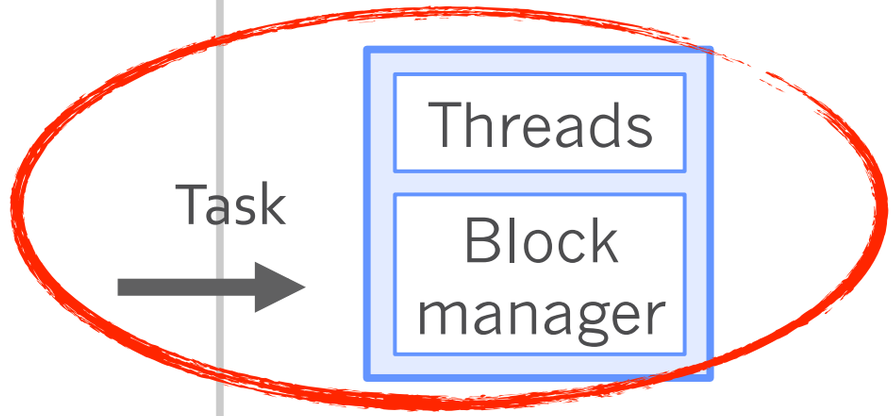


split graph into *stages* of tasks

submit each stage as ready

Executors

Task



execute tasks

store and serve blocks

Xin, “Advanced Spark Training,” in Spark Summit 2014

Move computation close to data!

Task scheduling

Spark assigns tasks to machines based on **data locality**

- if a task needs to process a partition that is available in memory on a node, send the task to that node

Different levels of locality are used

- same machine, but on disk
- in the memory of another machine on the same rack
- …

Task scheduling

Spark adopts **delay scheduling**

- wait a bit (3 secs by default) for a free executor before downgrading to the next locality level

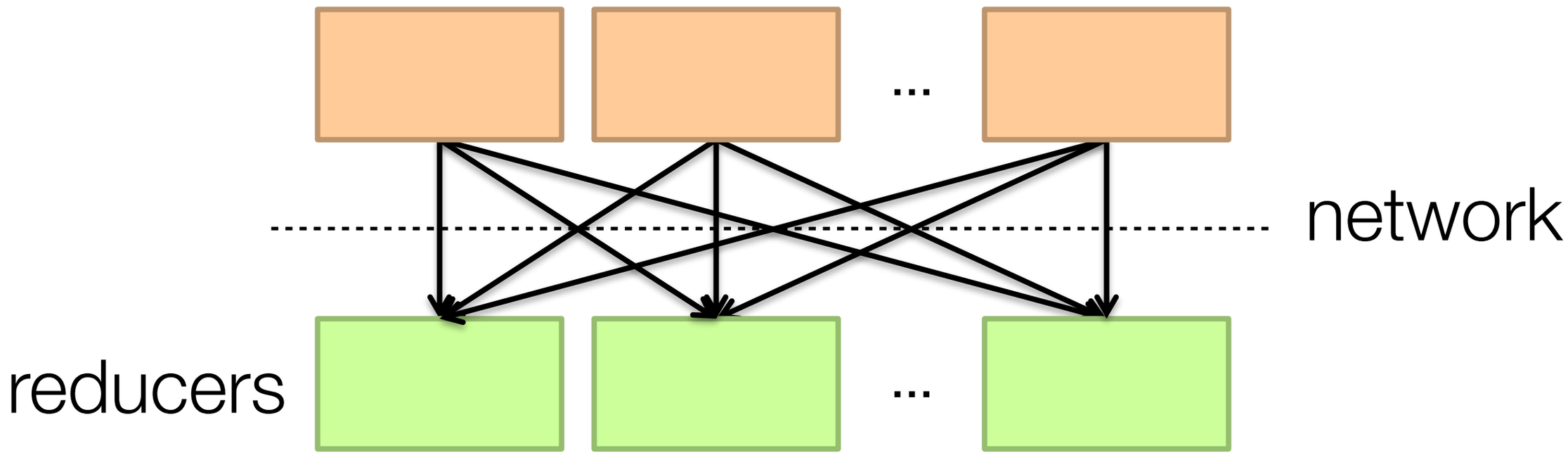
Why it works?

- tasks are short-running (sub-second)
- tasks are many

It shouldn't take long to find an available executor at the most desirable locality level!

Now that tasks have completed, what’s next?

Can’t avoid synchronization



How shuffle is implemented in Spark?

Hash shuffle

Default option prior to Spark 1.2.0

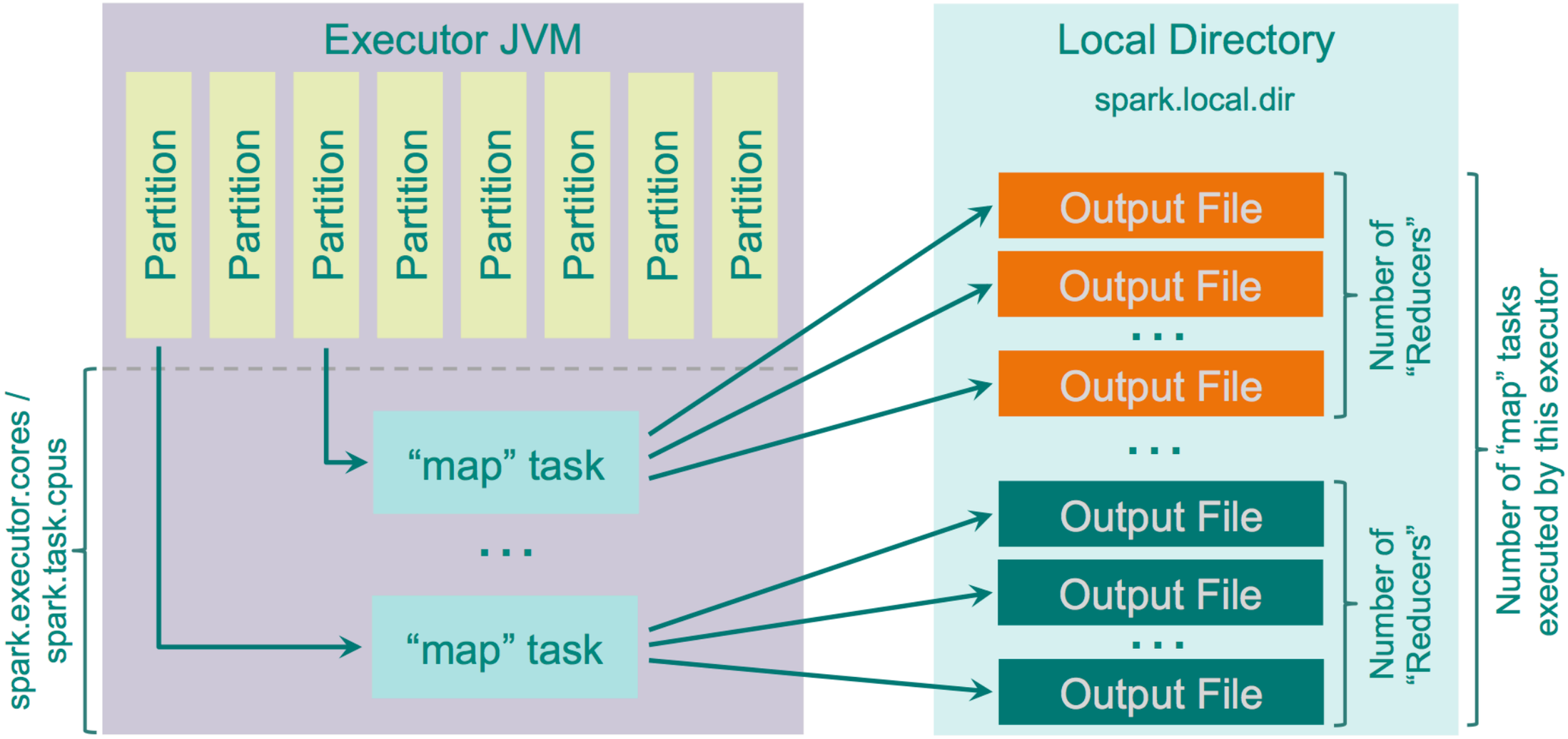
- **spark.shuffle.manager = hash**

Each mapper task creates separate file for each separate reducer

- M x R total files shuffled
- e.g., 46k mappers and 46k reducers generate 2b files on a Yahoo! cluster

Could this be a problem?

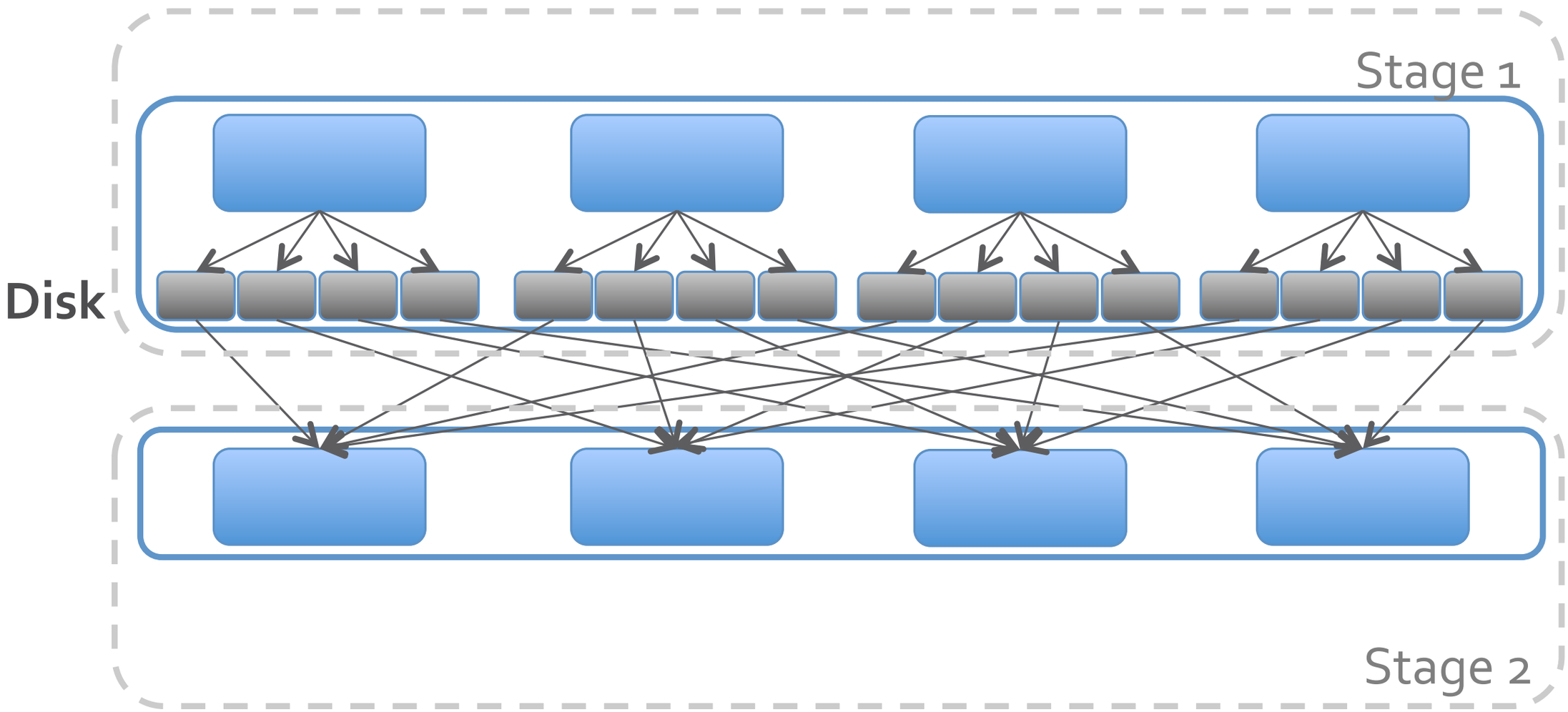
Hash shuffle



# of files shuffled: M x R

Source: http://0x0fff.com/spark-architecture-shuffle/

Hash shuffle



Hash shuffle

Pros:

- fast — no sorting is required, no hash table maintained
- no memory overhead for sorting the data
- no I/O overhead — data is written once and read once

Cons:

- doesn't scale well with a large number of M and R
- writing big amount of files results in I/O skew towards random I/O (up to 100x slower)

Sort shuffle

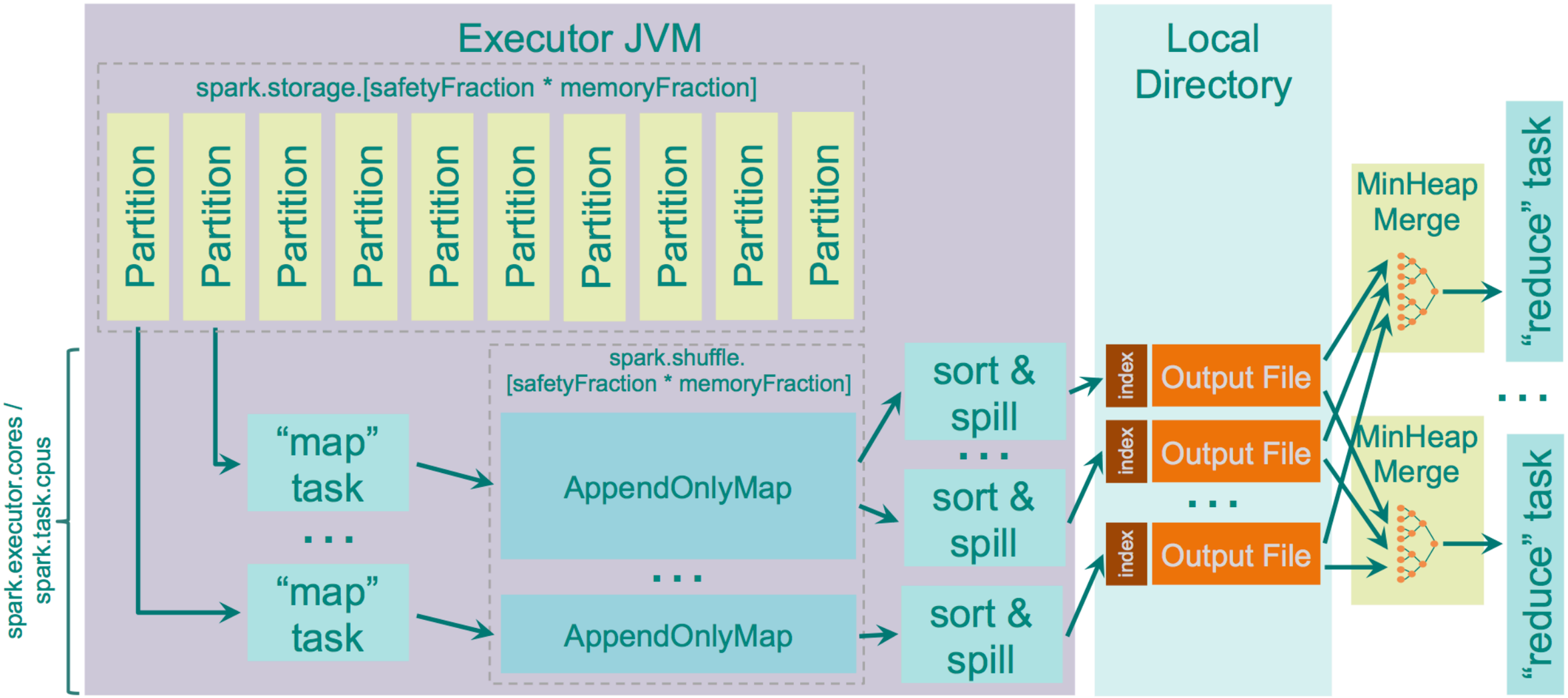
Default shuffle option since Spark 1.2.0

- **spark.shuffle.manager = sort**

Similar to Hadoop MapReduce, each mapper outputs a single file ordered (and indexed) by “reducer” id

- before **fread**, use **fseek** to locate the chunk of the data related to “reducer *x*”
- sort data on the “reduce” side using TimSort

Sort shuffle



Sort shuffle

Pros:

- smaller amount of files created on “map” side
- mostly sequential reads/writes (random I/O occasionally)

Cons:

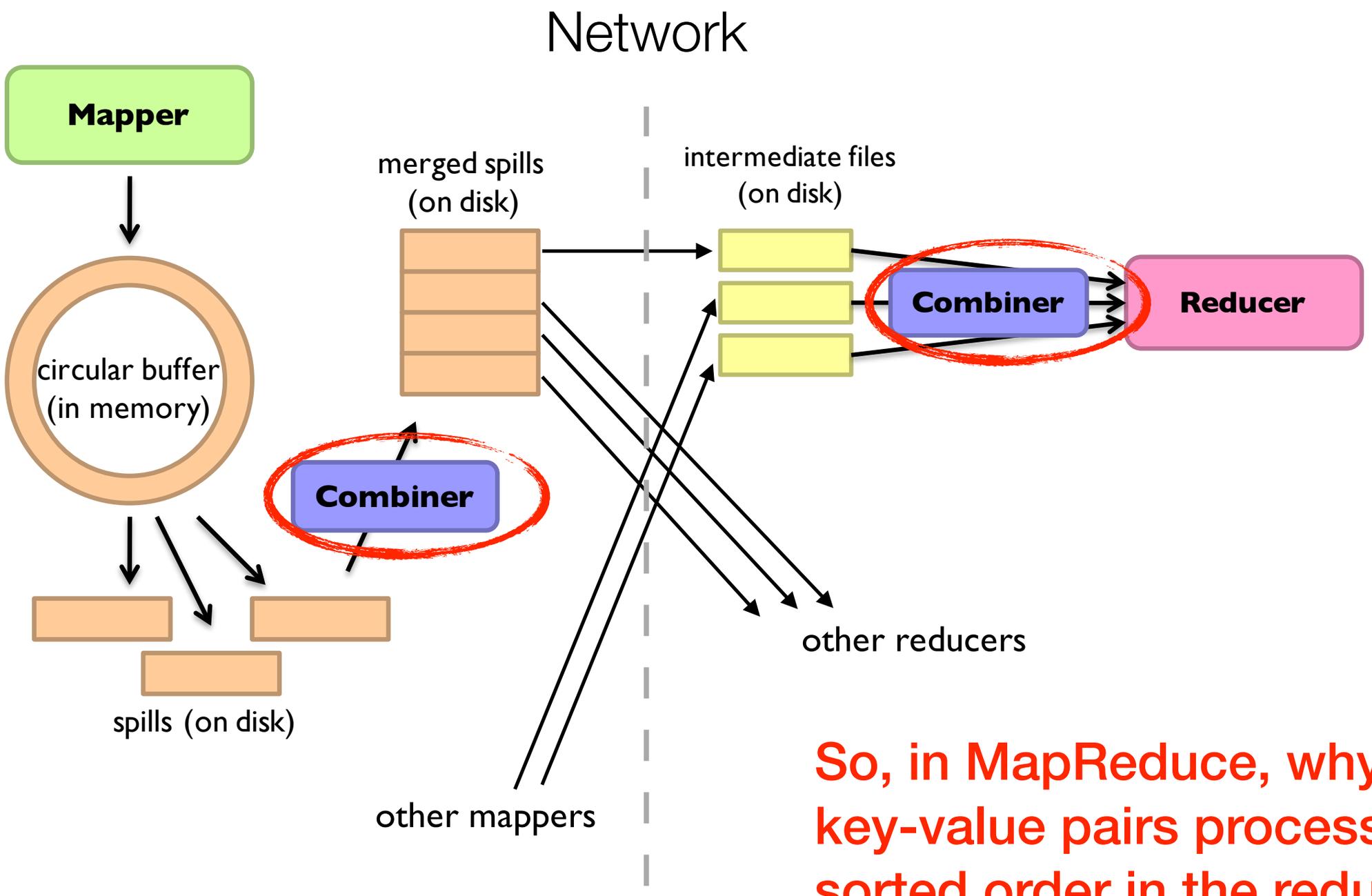
- sorting is slower than hashing
- hash shuffle might work better on SSD drives

random I/O is quick.

Starting Spark 1.4.0, a more advanced, yet complicated, shuffle option, called **Tungsten sort**, has been introduced

https://issues.apache.org/jira/browse/SPARK-7081

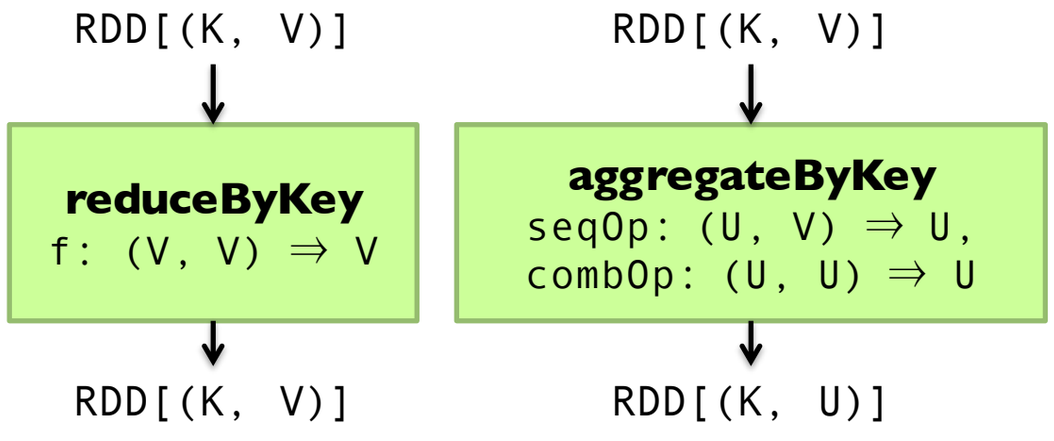
Remember this?



So, in MapReduce, why are key-value pairs processed in sorted order in the reducer?

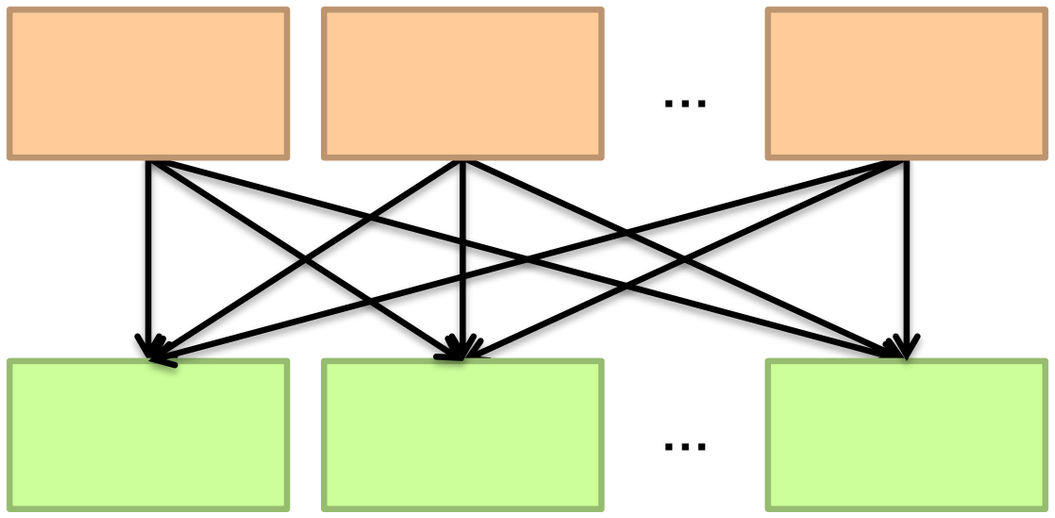
Where are the combiners in Spark?

Reduce-like operations

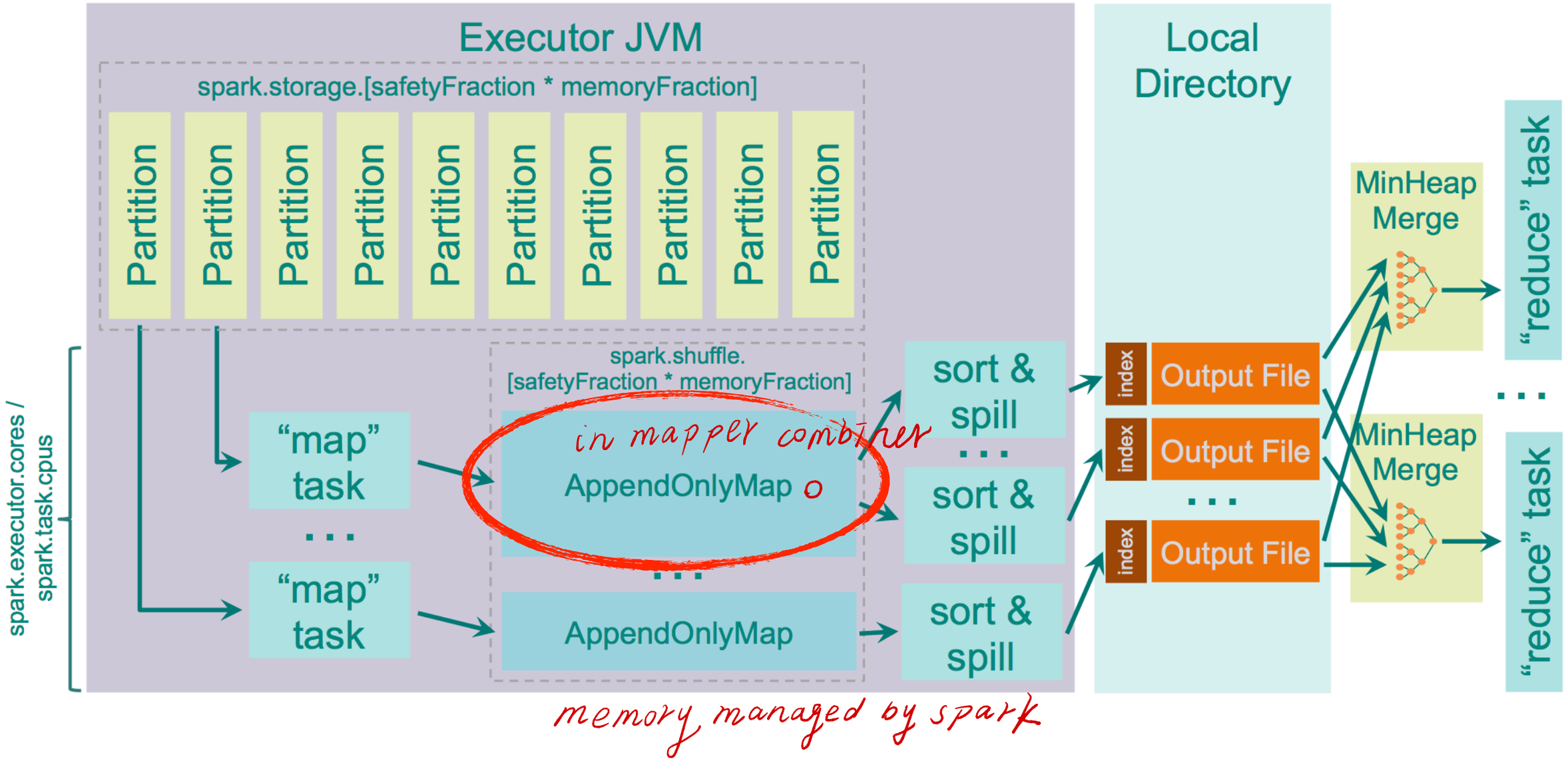


How can we optimize?

What happened to combiners?



Sort shuffle



~~Apply "combiner" logic *in place* to the Hash table~~

In-place “combiner”

Store map output data to **AppendOnlyMap**

- each new value added for existing key is getting through (default) “combine” logic with existing value
- output of “combine” is stored as the new value

Wanna customize your own “combiner” logic?

Use **combineByKey** method

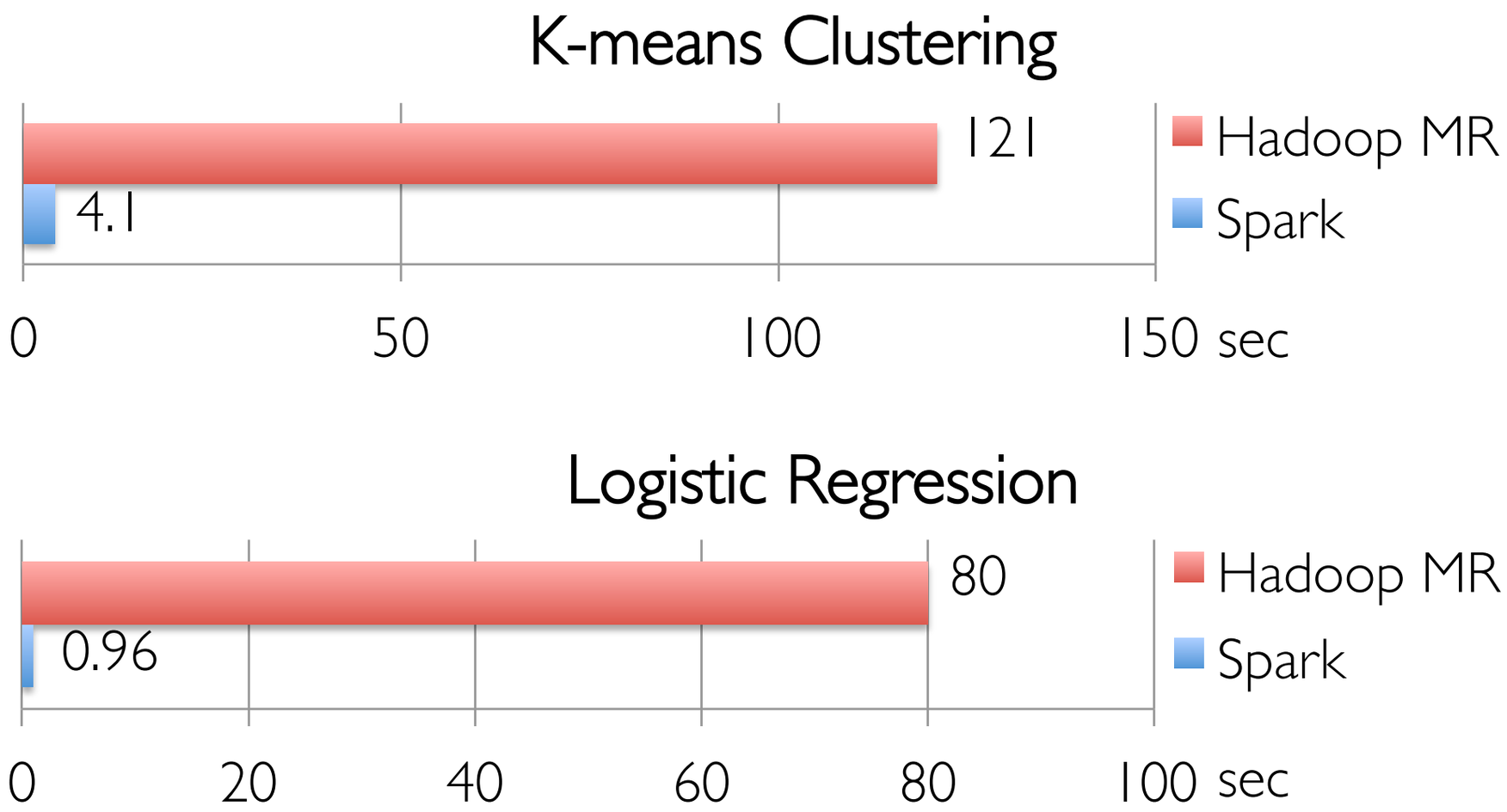
Spark vs. MapReduce

	Hadoop MapReduce	Spark
Storage	Disk only	In memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc.
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, R and Python

get partial outputs

In-memory makes a big difference

Two iterative ML algorithms



Sources from Databricks

Spark #wins

High-level programming with rich operators

RDD abstraction supports optimizations

- pipelining, caching, etc.

Multi-language support

- Scala, Java, Python, R

Spark #wins

Generalized patterns

- unified engine for many use cases

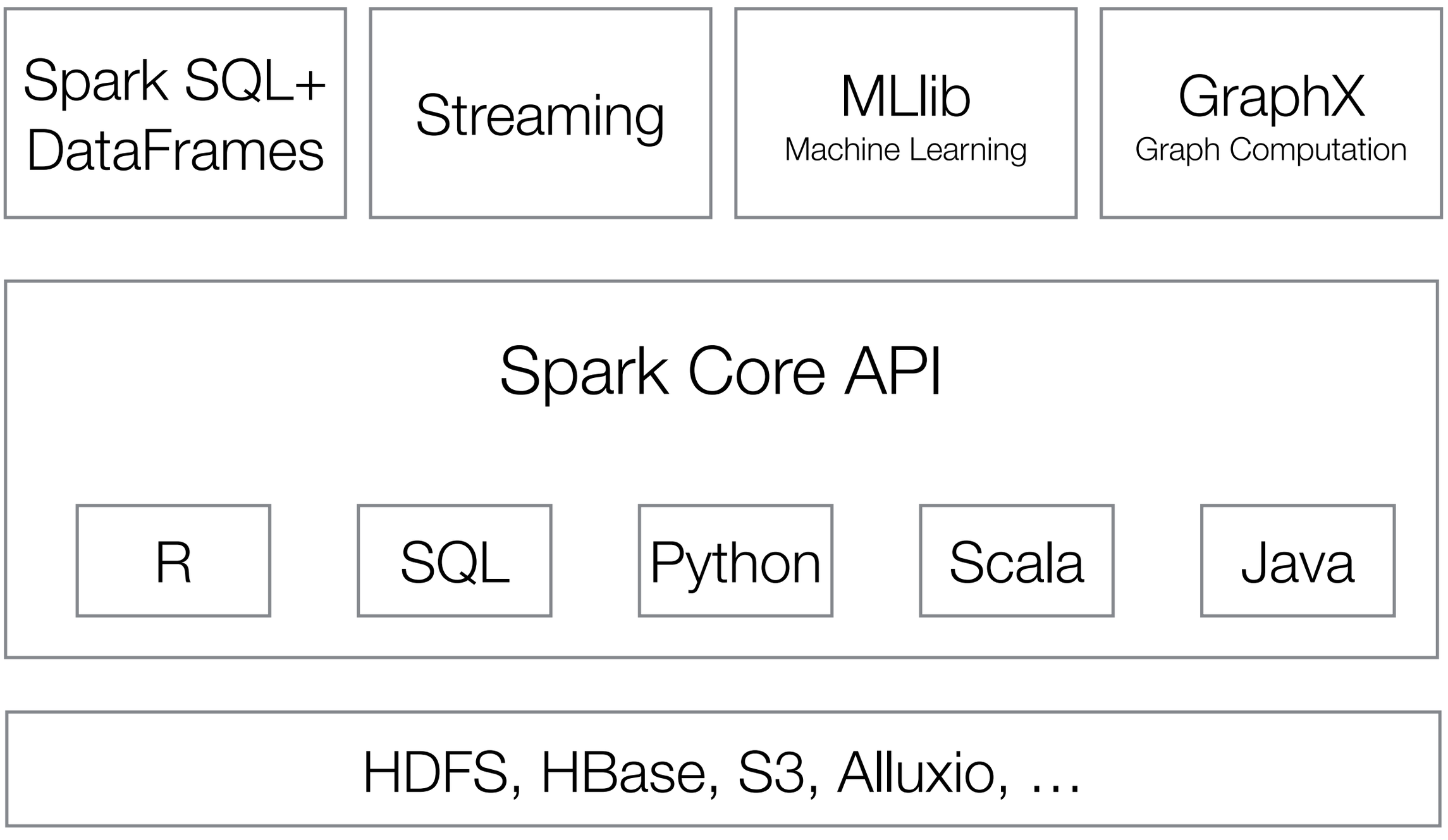
Lazy evaluation of the linage graph

- reduces wait states, better pipelining

Lower overhead for starting jobs

Less expensive shuffles

The Spark ecosystem



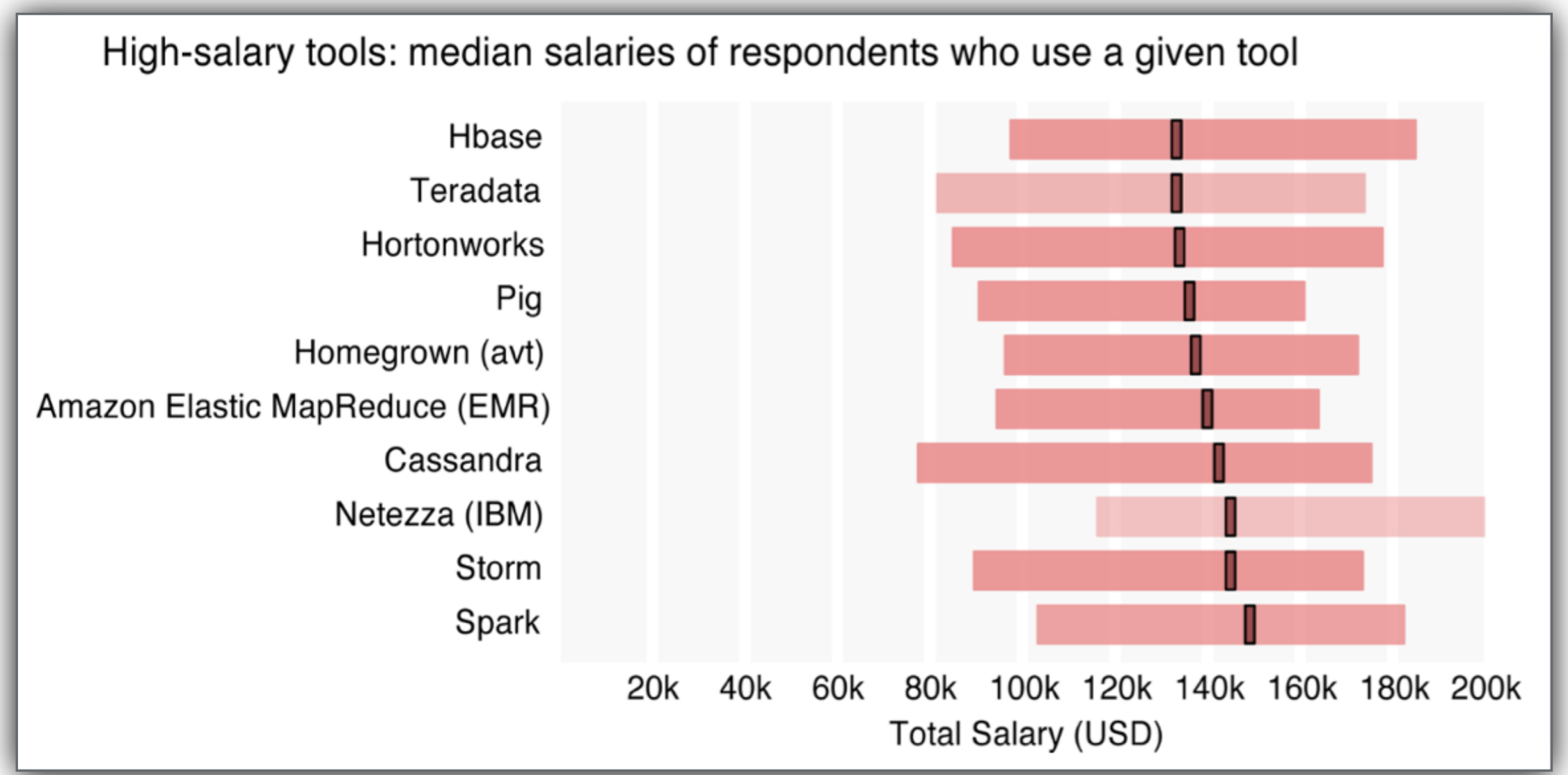
Spark research papers

Spark: Cluster Computing with Working Sets
Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
USENIX HotCloud 2010

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
USENIX NSDI 2012

There is one more thing…

Spark expertise tops big data salaries



Over 800 respondents across 53 countries and 41 U.S. states

http://www.oreilly.com/data/free/2014-data-science-salary-survey.csp

Credits

Slides are adapted from Prof. Jimmy Lin’s slides at the University of Waterloo