Cloud Computing
Spark Internals and DataFrames

Minchen Yu
SDS@CUHK-SZ
Fall 2024

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF
DATA SCIENCE
數據科學學院

What's an RDD?

Resilient Distributed Dataset

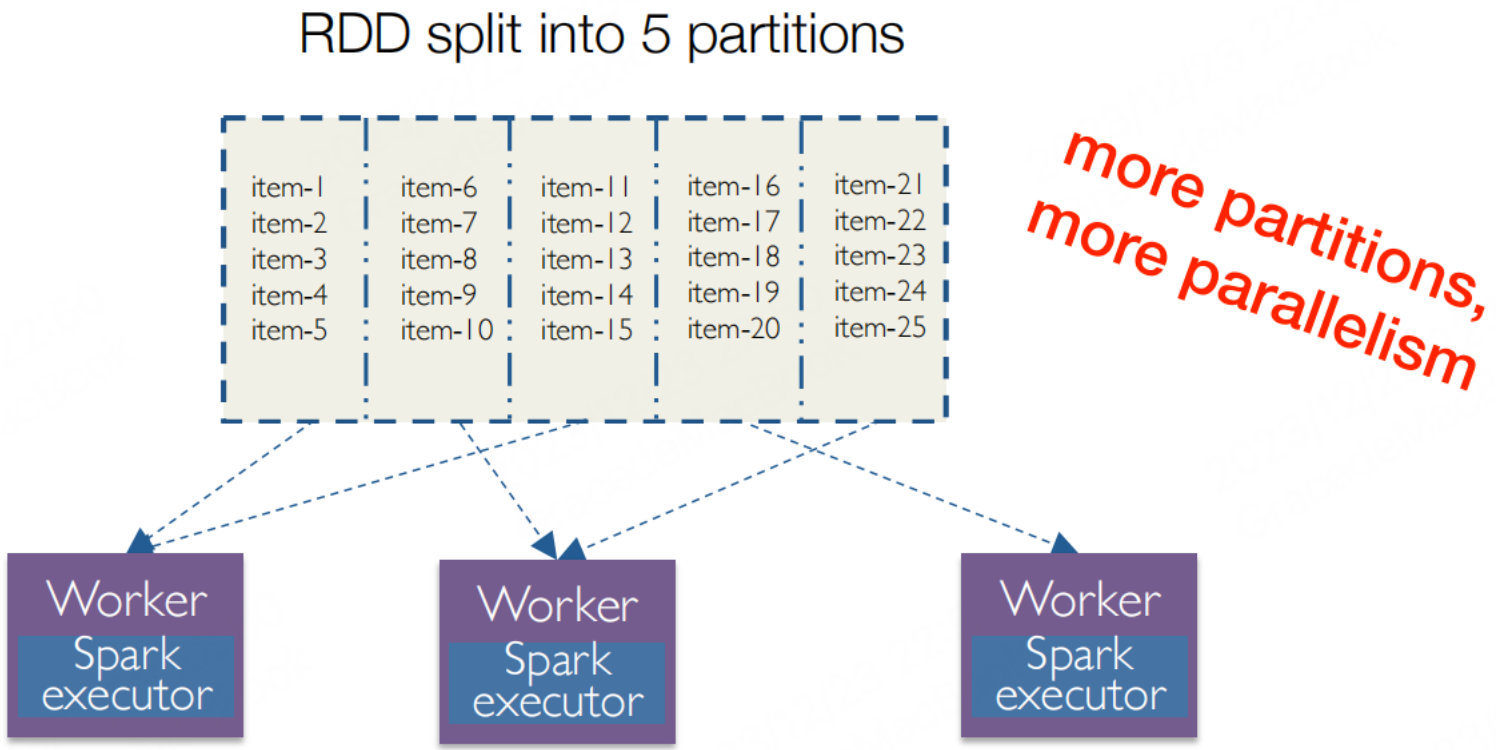
= immutable = partitioned

Developers define *transformations* on RDDs

Framework keeps track of *lineage*

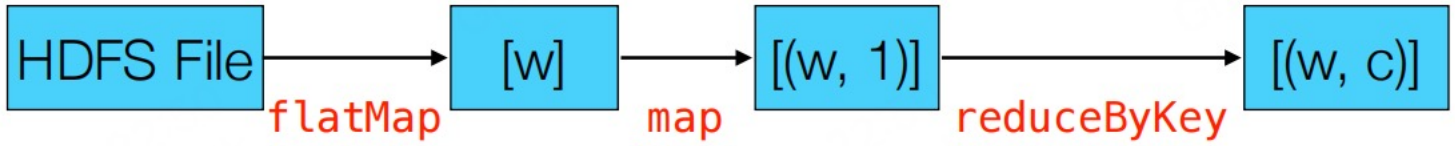
RDD is a distributed dataset

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

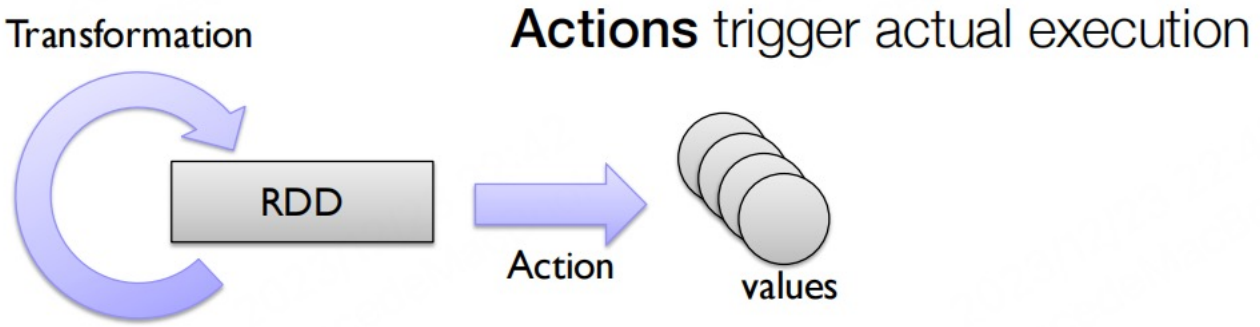


Spark WordCount

```
# 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://...")
```



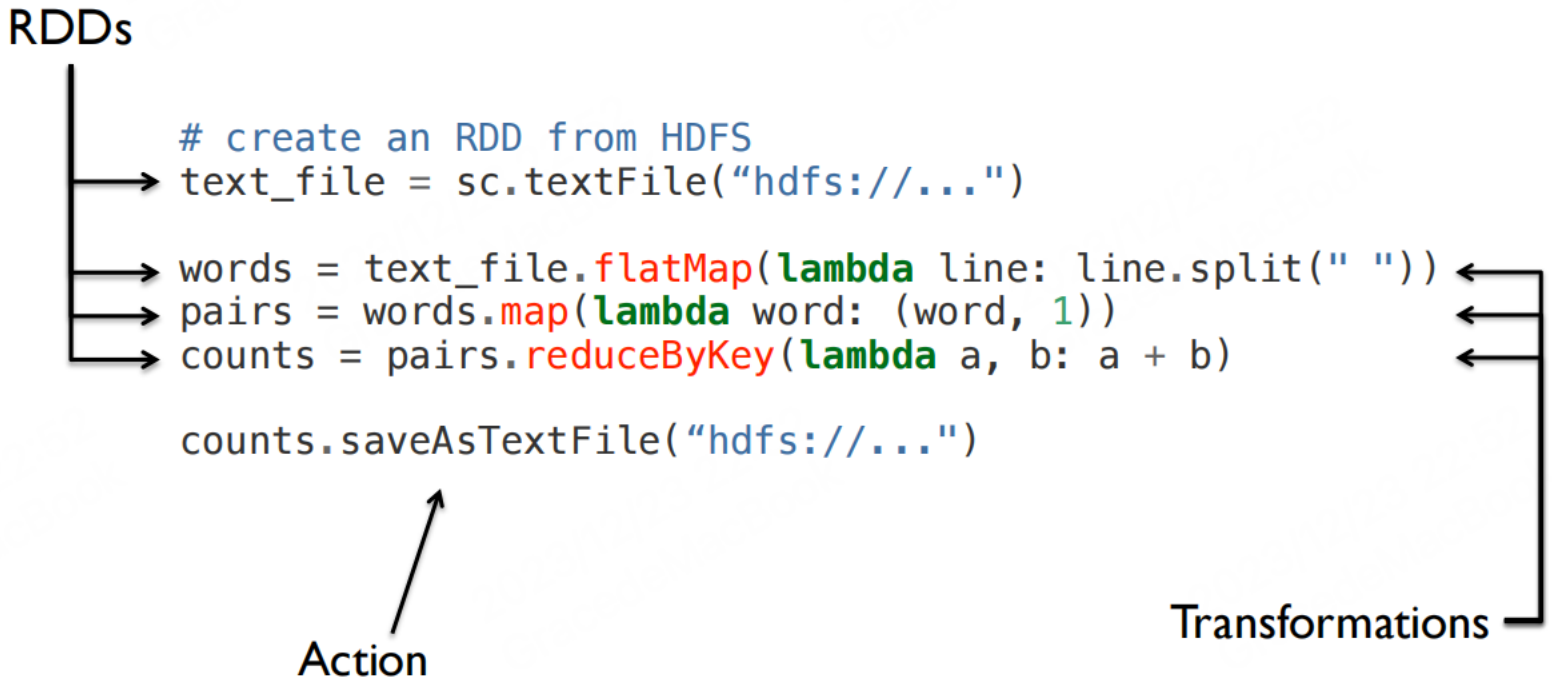
RDD lifecycle



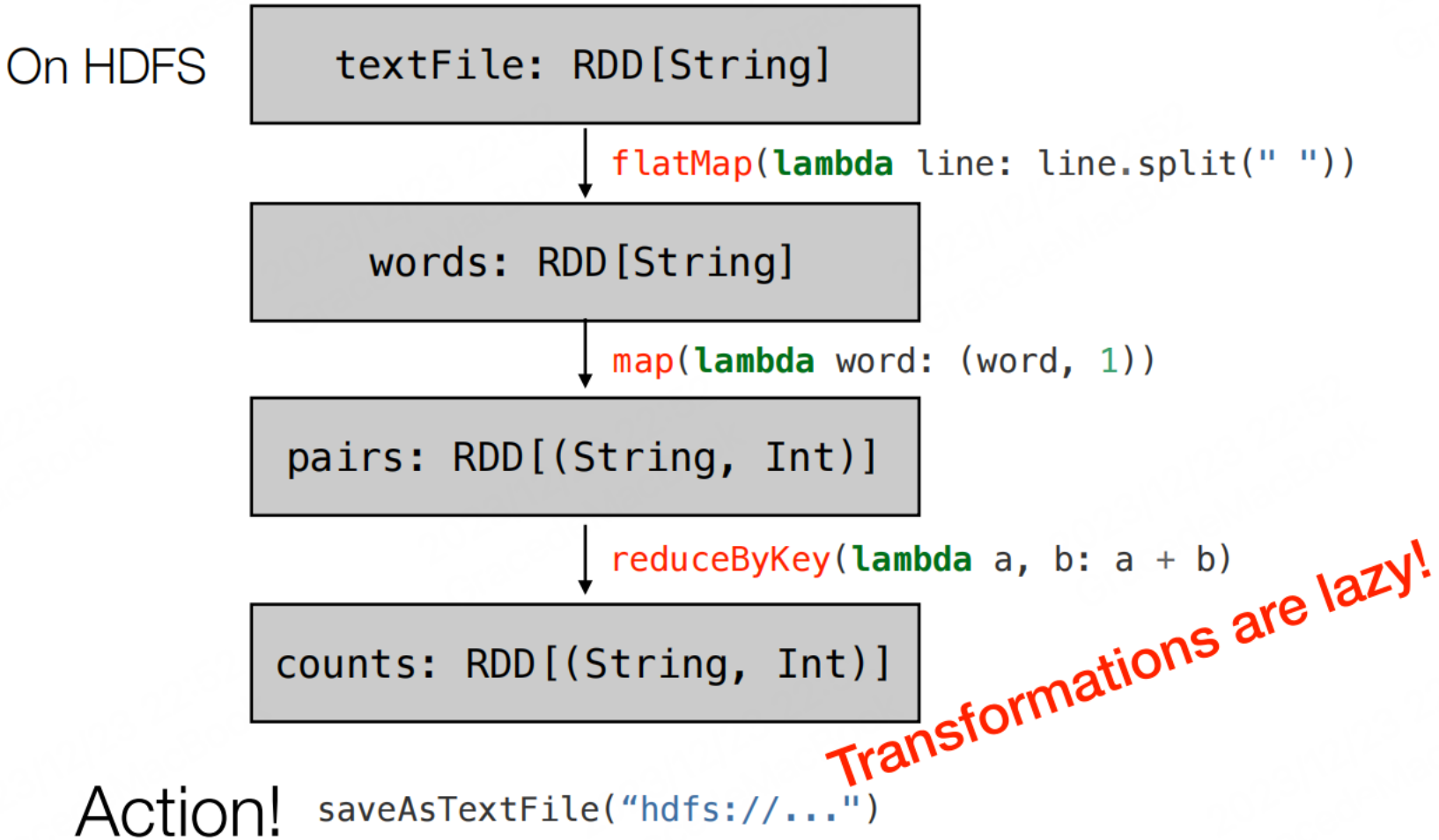
Transformations are *lazy*:

Framework keeps track of *lineage*

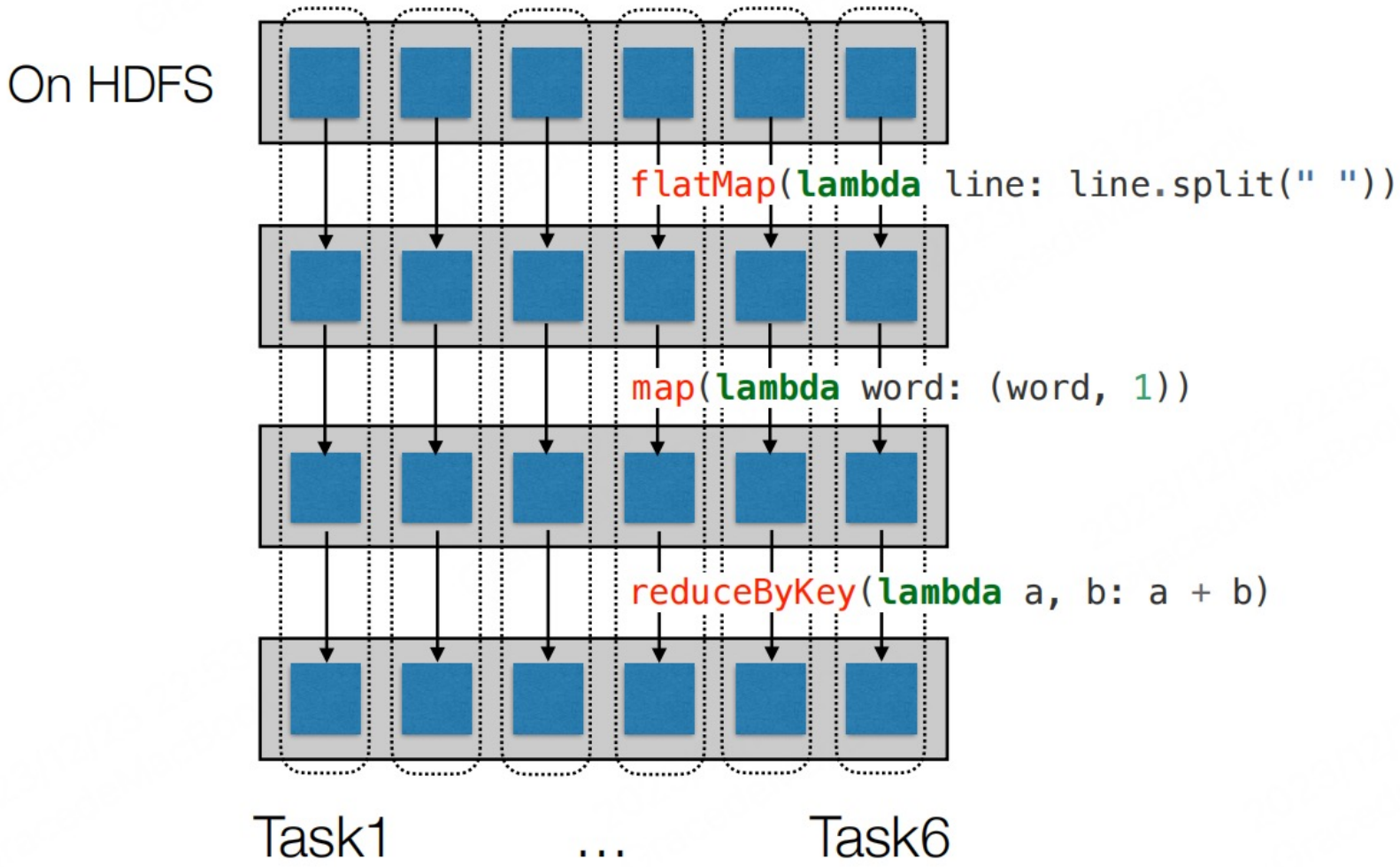
Spark WordCount



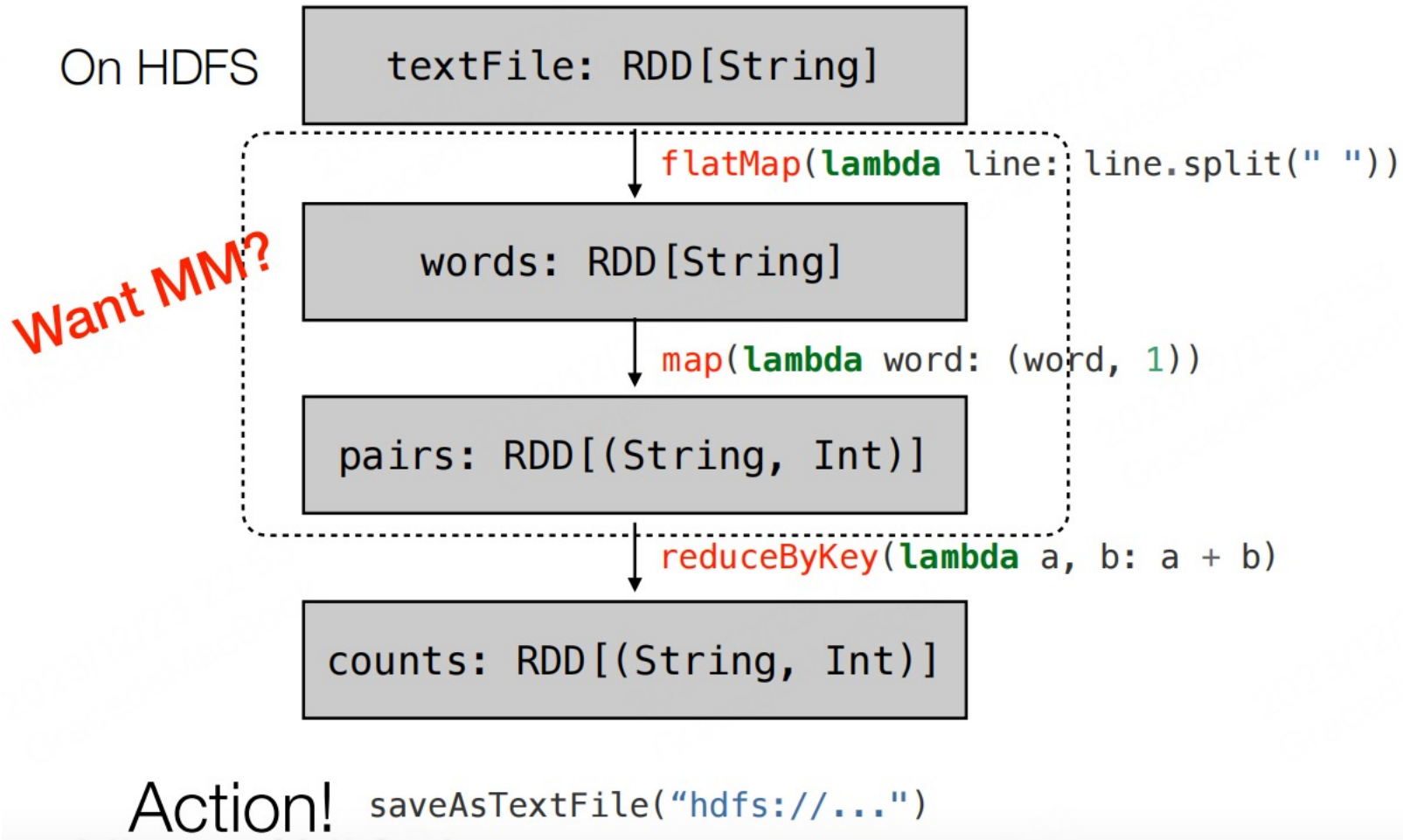
RDDs and lineage



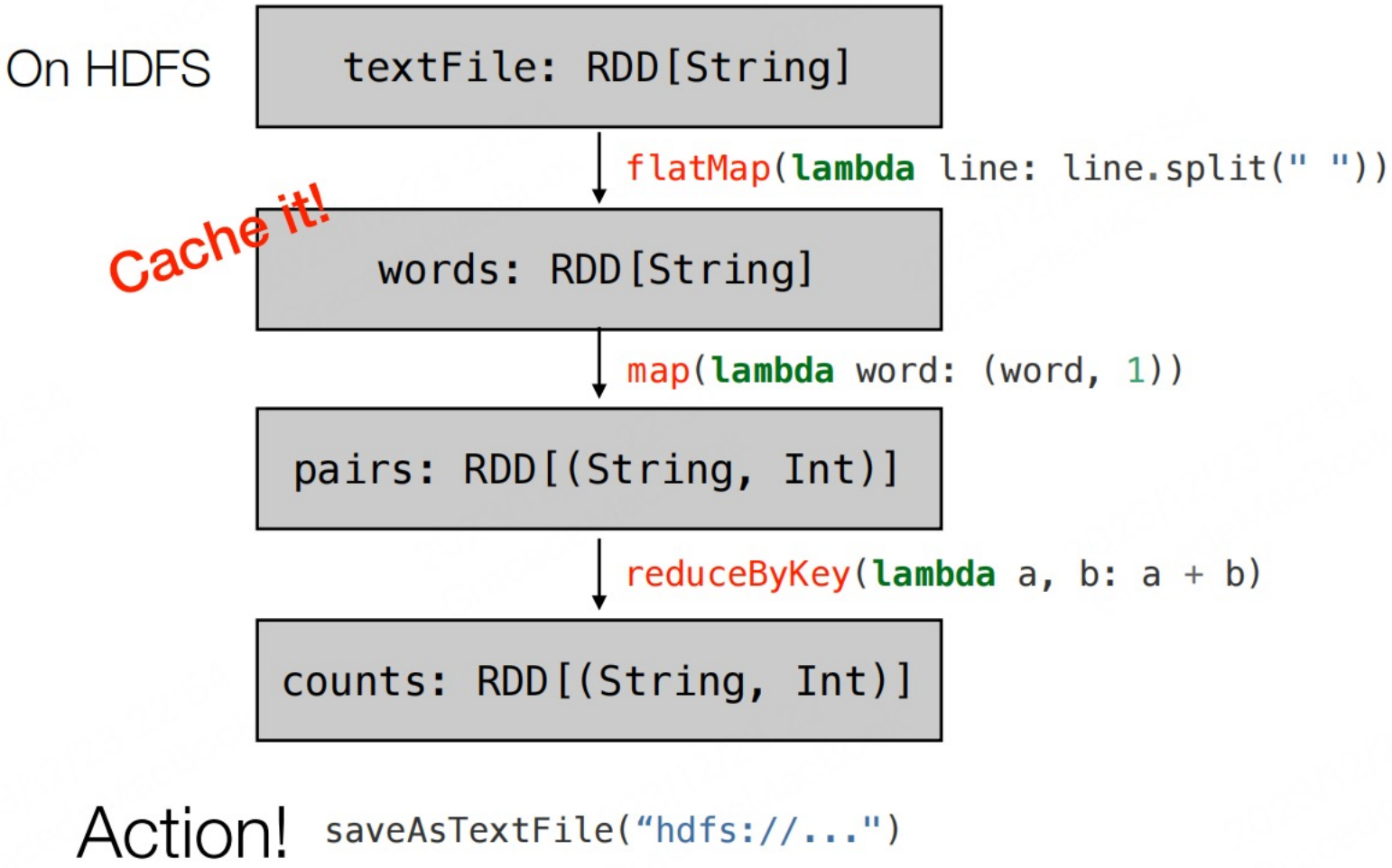
Partition-level view



RDDs and optimizations



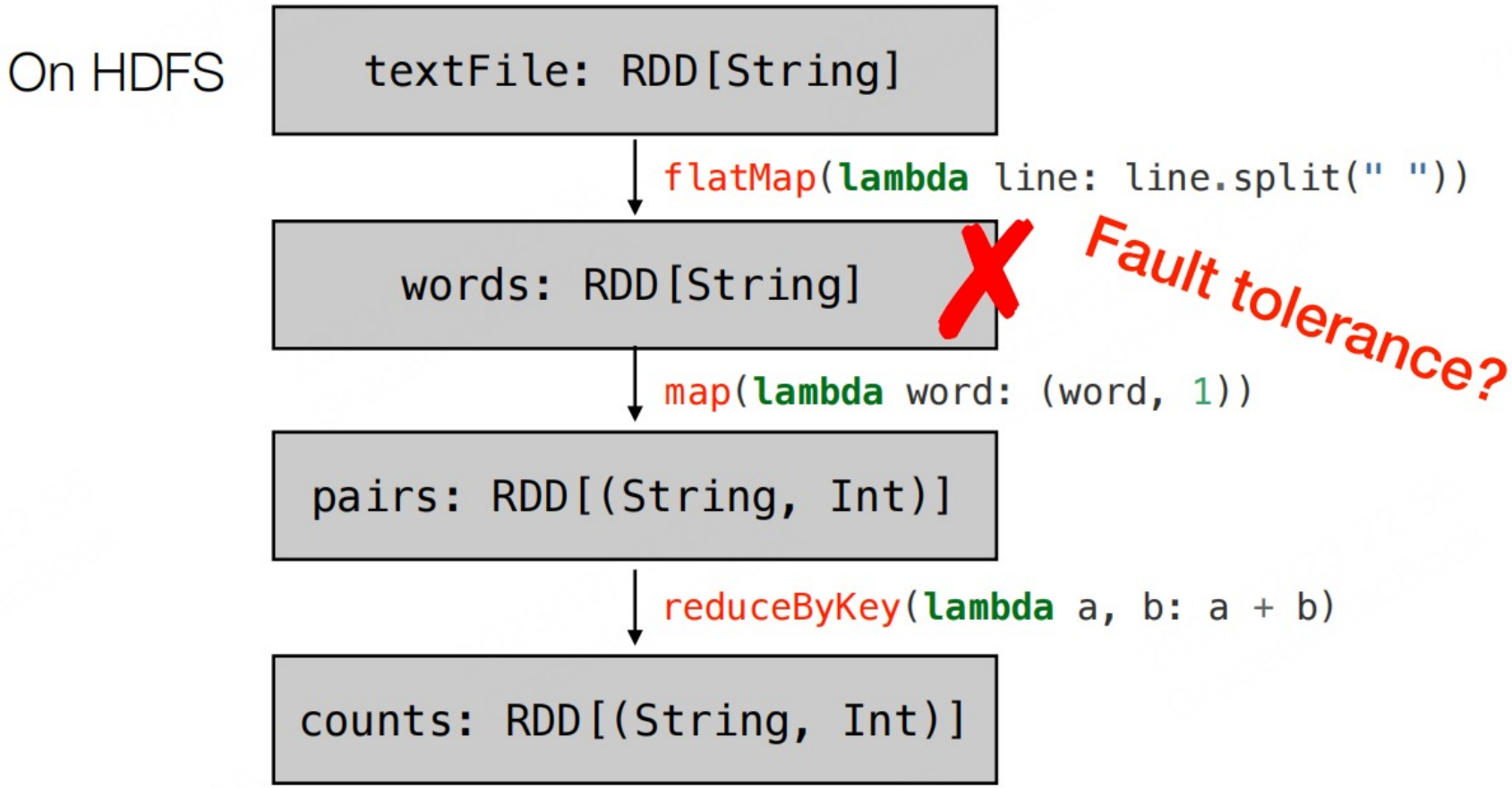
RDDs and caching



The RDDs are too large to be cached in memory?

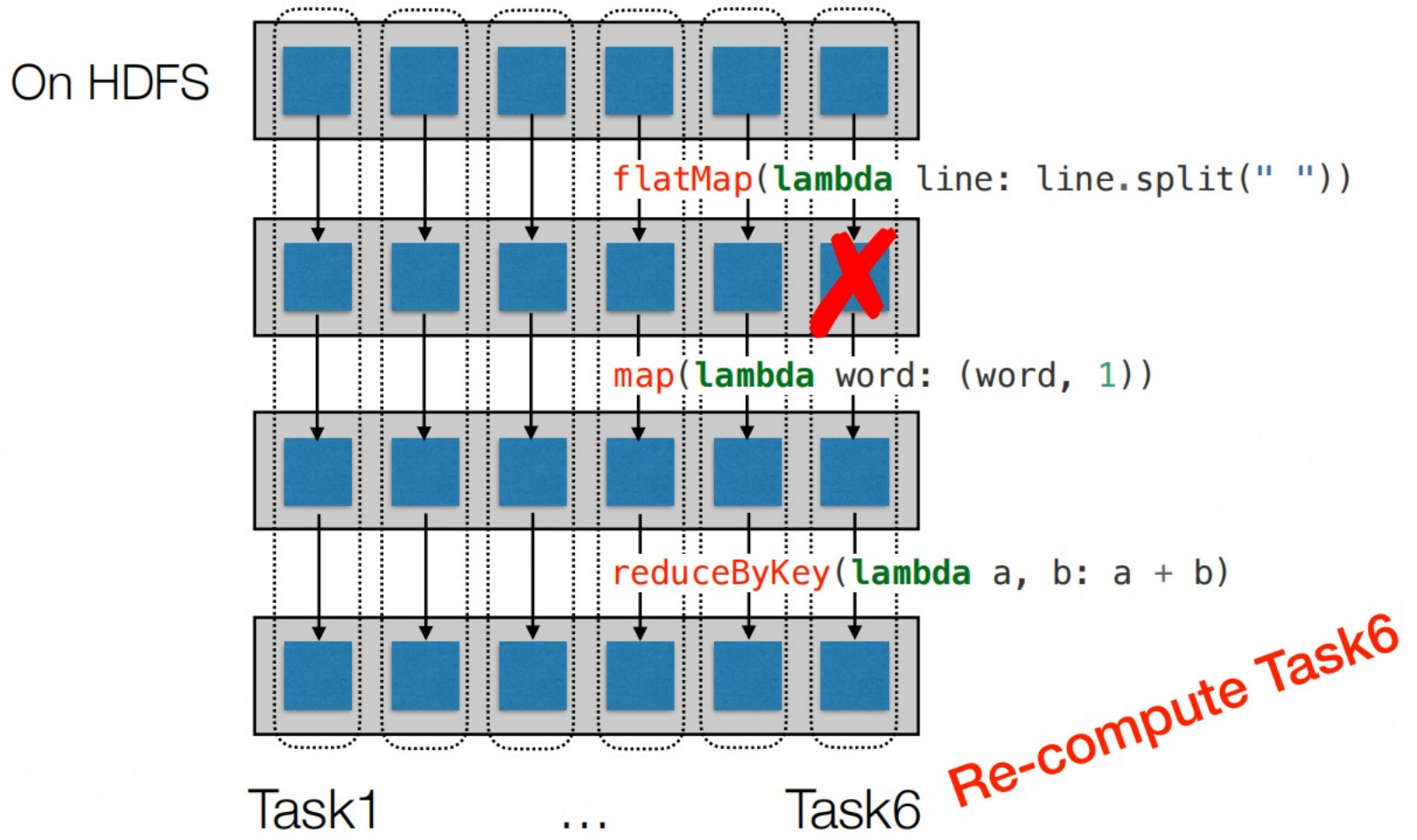
Cache the RDDs in *partial*, and spill the remaining partitions to disk

RDDs and fault tolerance

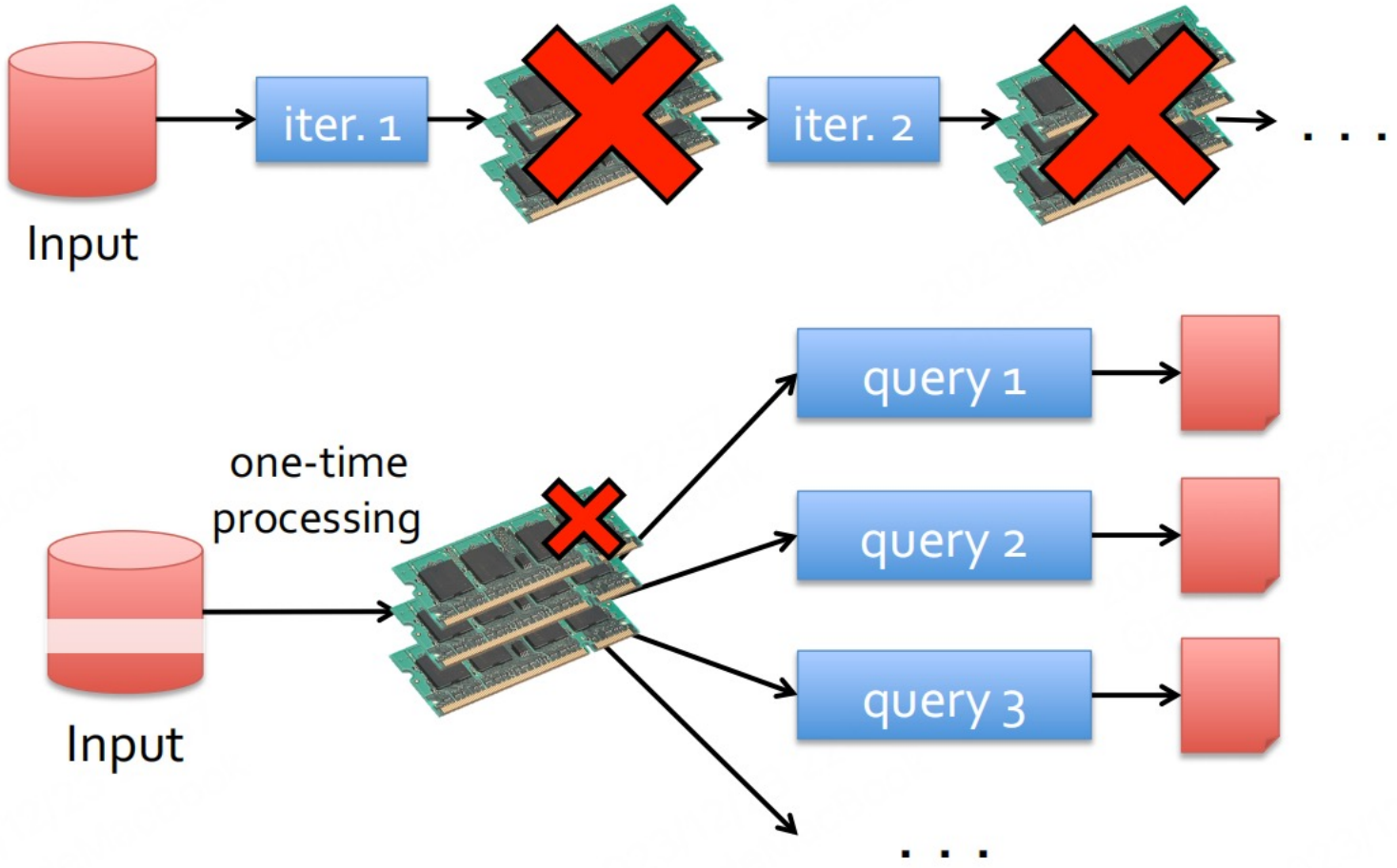


Replay the lineage to *recompute* lost RDD partitions

RDD recovery



RDD recovery



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

Replay the lineage to *recompute* lost RDD partitions

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

Recap: What's an RDD?

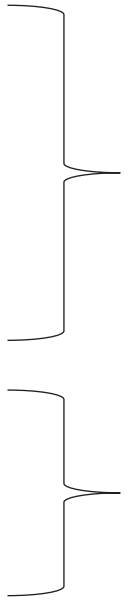
Set of partitions

List of dependencies on parent RDDs

Function to compute a partition given its parents

(Optional) partitioner (hash, range)

(Optional) preferred locations for each partition



text_file RDD

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

partitions = one per HDFS block

dependencies = none

compute = read corresponding block

preferredLocations = HDFS block location

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)

partitioner = none

join’ed RDD

```
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?

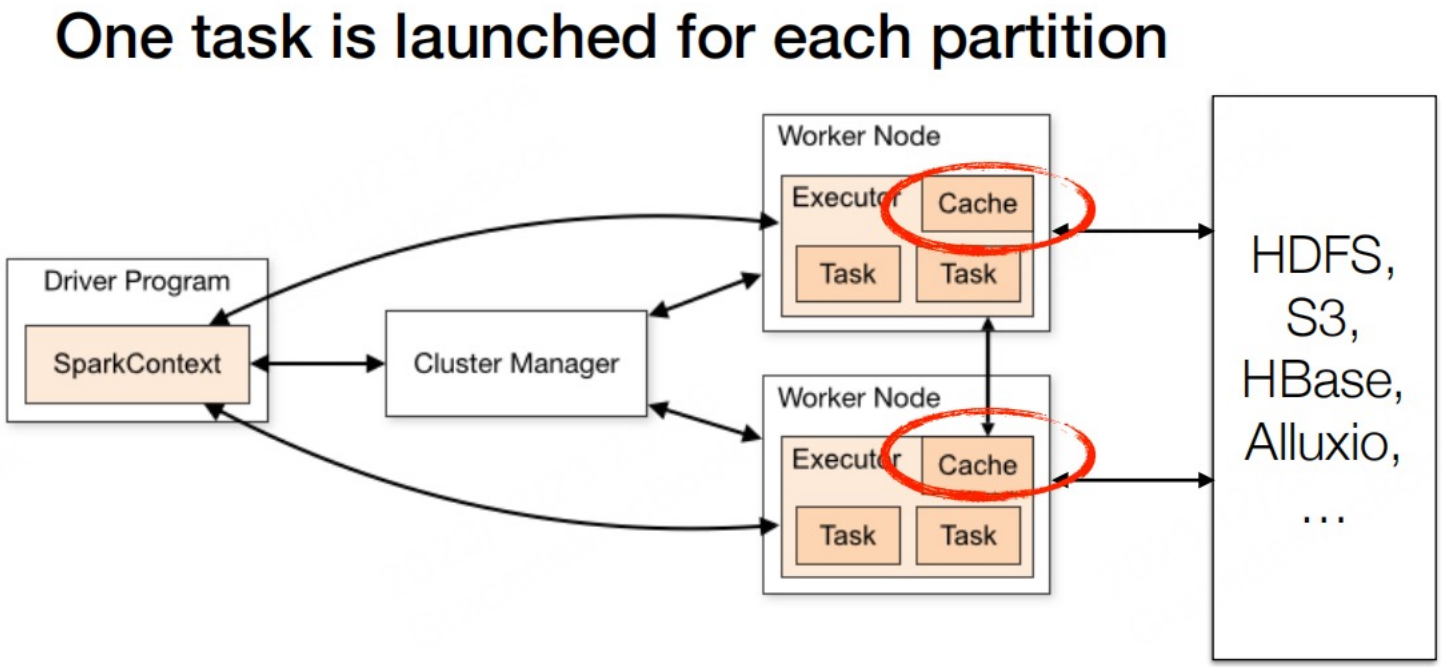
A: distributed collection of objects on disk

B: distributed collection of objects in memory

C: distributed collection of objects in Cassandra

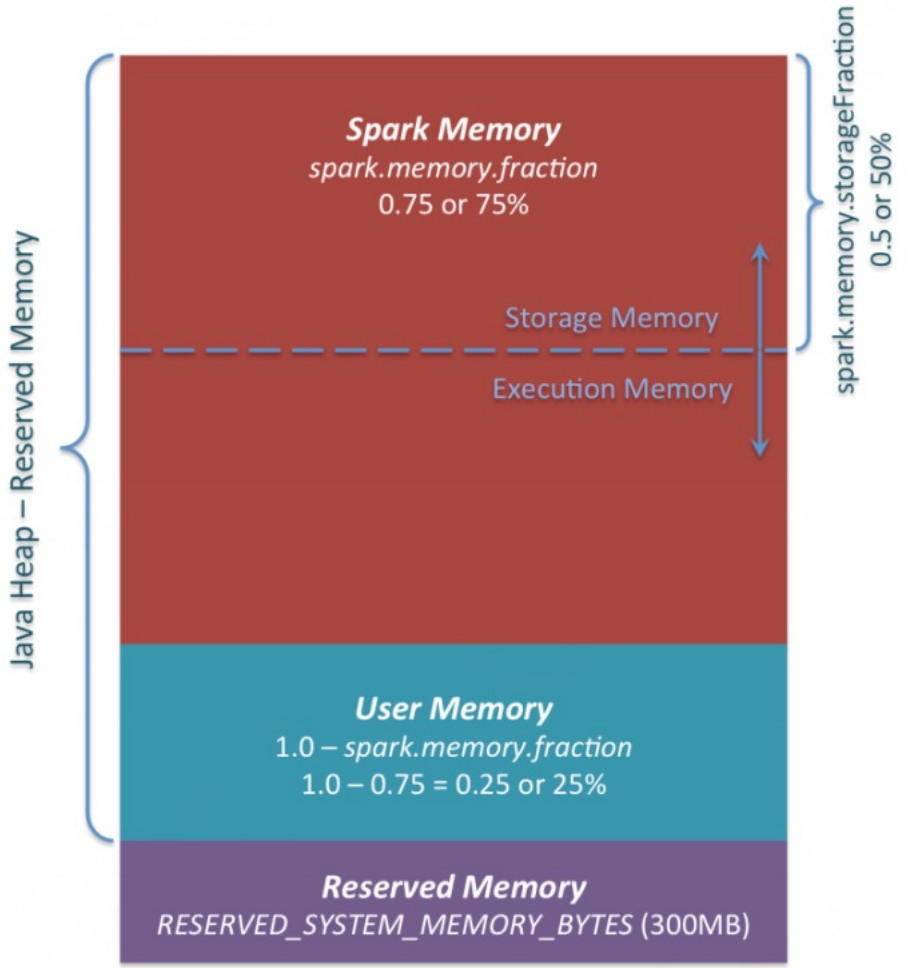
Answer: could be any of the above!

Spark architecture



How memory is managed in Spark?

Memory management



LRU cache for RDDs

moving boundary

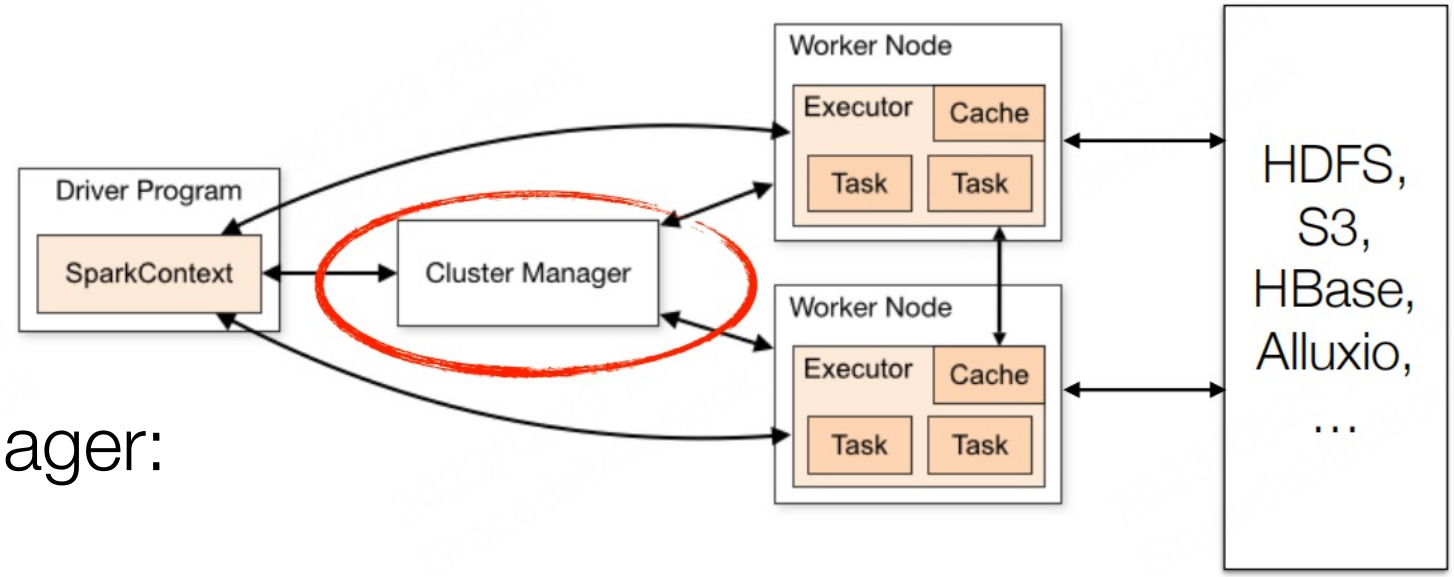
shuffle intermediate buffer

your data structures

reserved by Spark

Spark architecture

One task is launched for each partition



Cluster manager:

- ✓ standalone
- ✓ external: Mesos, YARN, Borg, Kubernetes, etc.

YARN

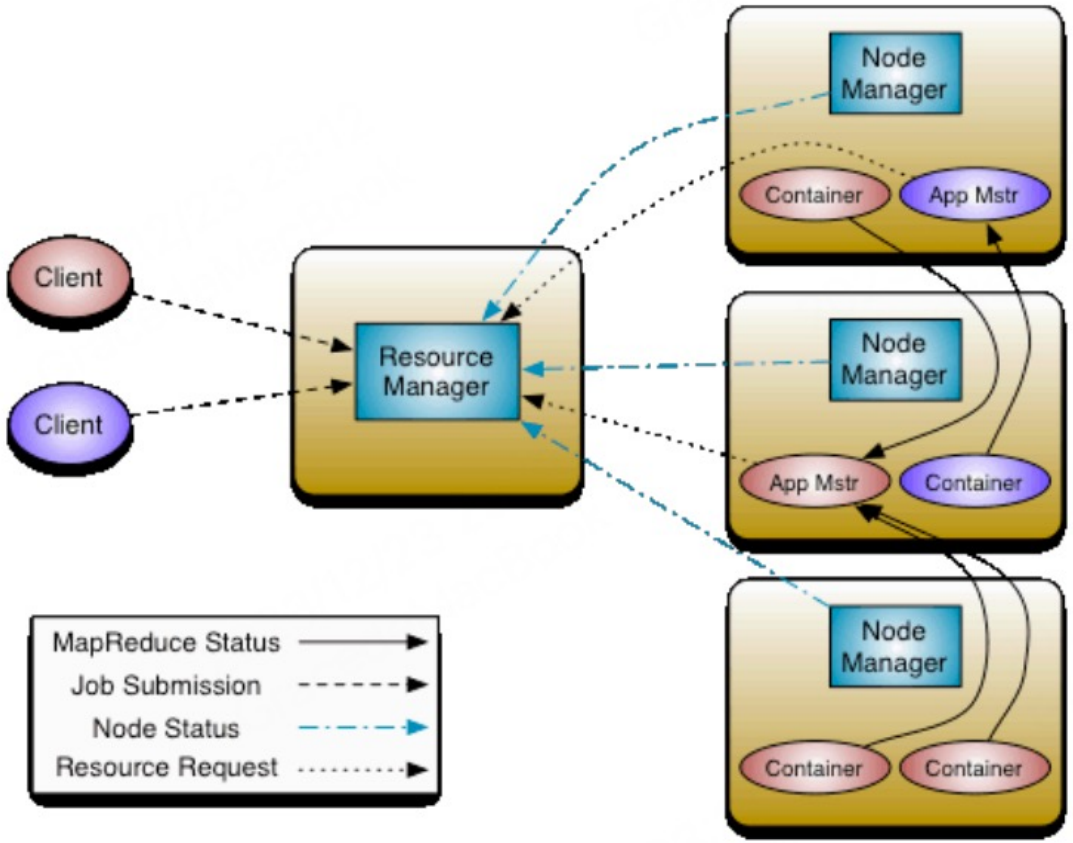
Hadoop's (original) limitations:

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

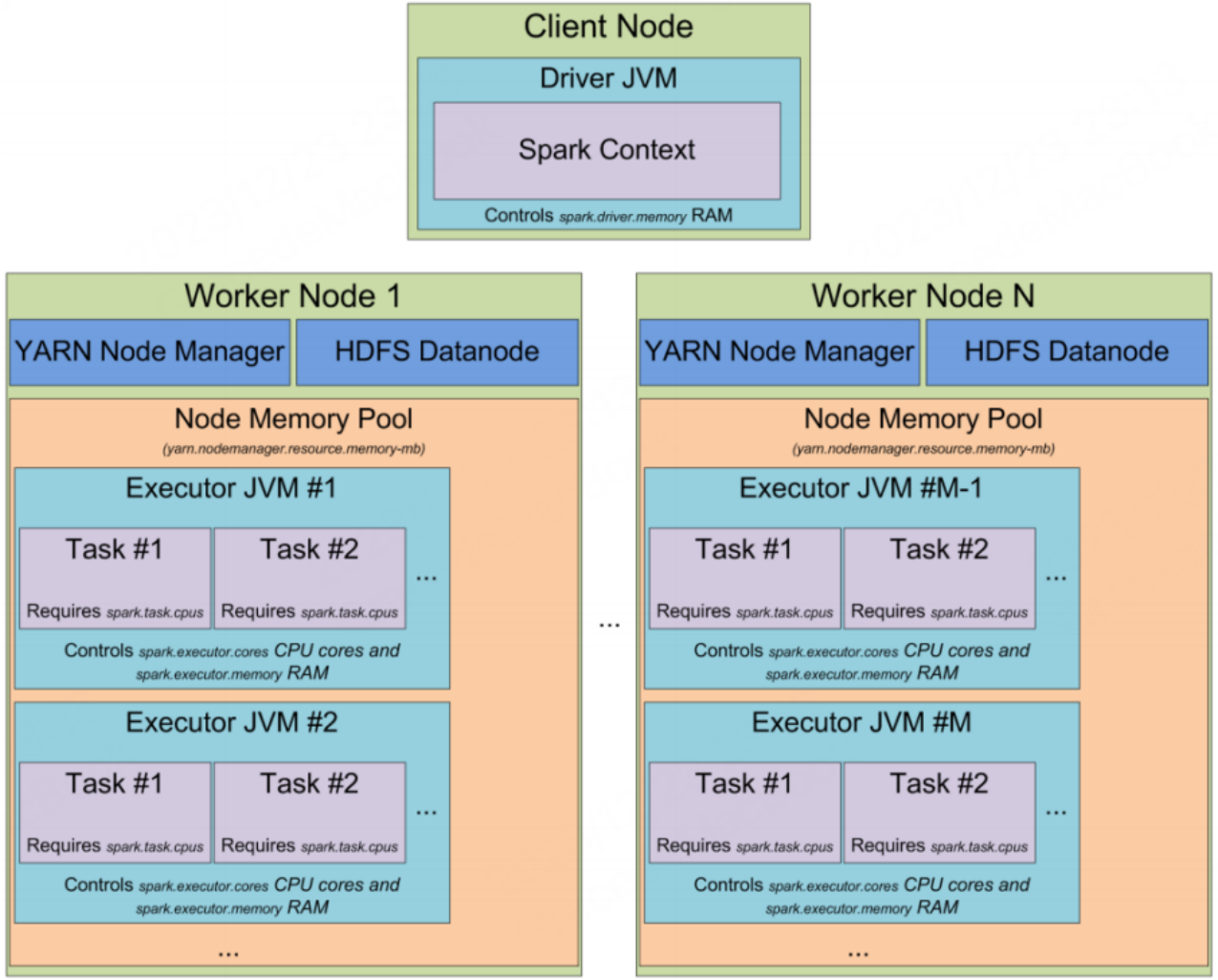
YARN = Yet-Another-Resource-Negotiator

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

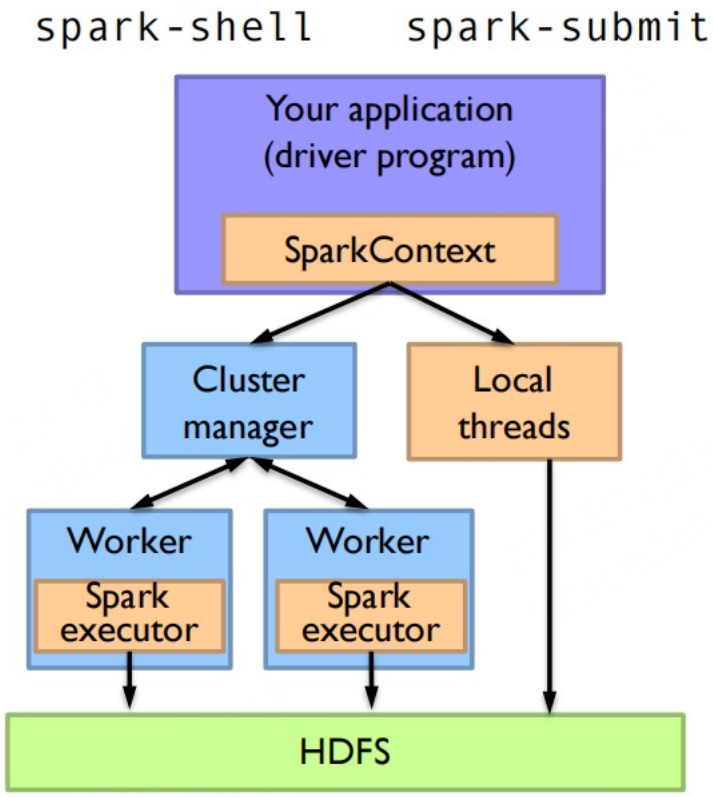
YARN


Client
Client
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



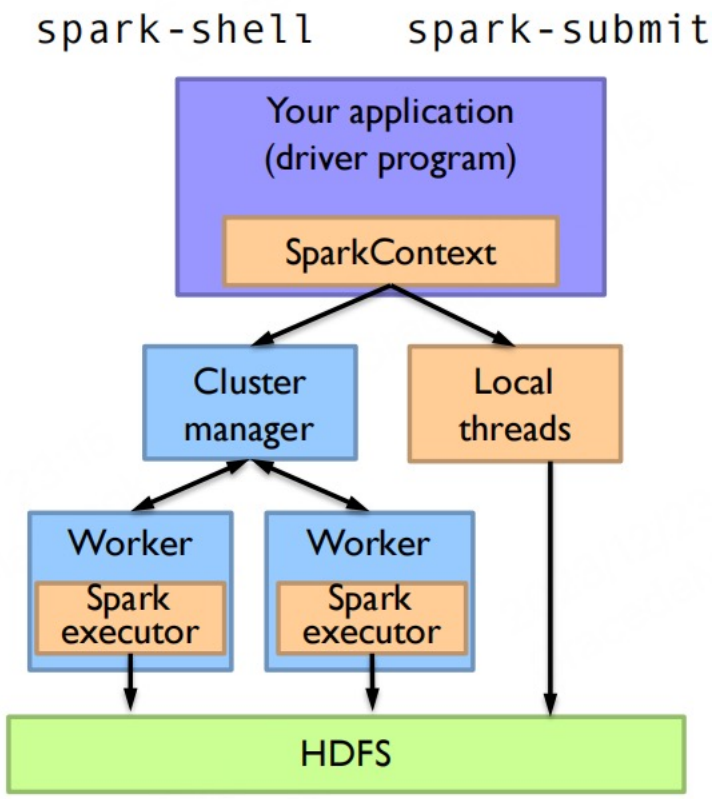
Multi-language:

- Scala, Java, Python, R

Spark context

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

Spark driver

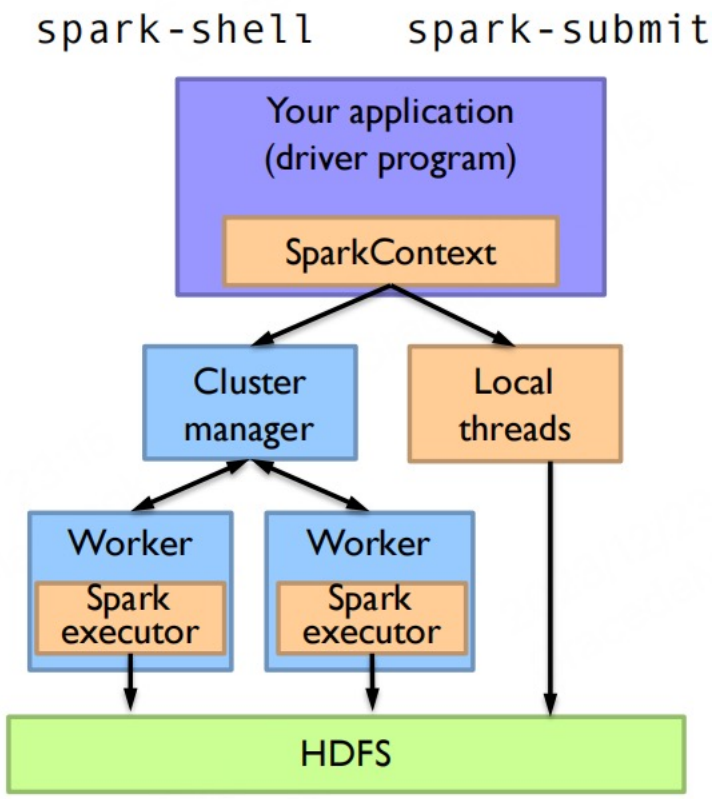



```
# create an RDD from HDFS
text_file = sc.textFile("hdfs://...")

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)

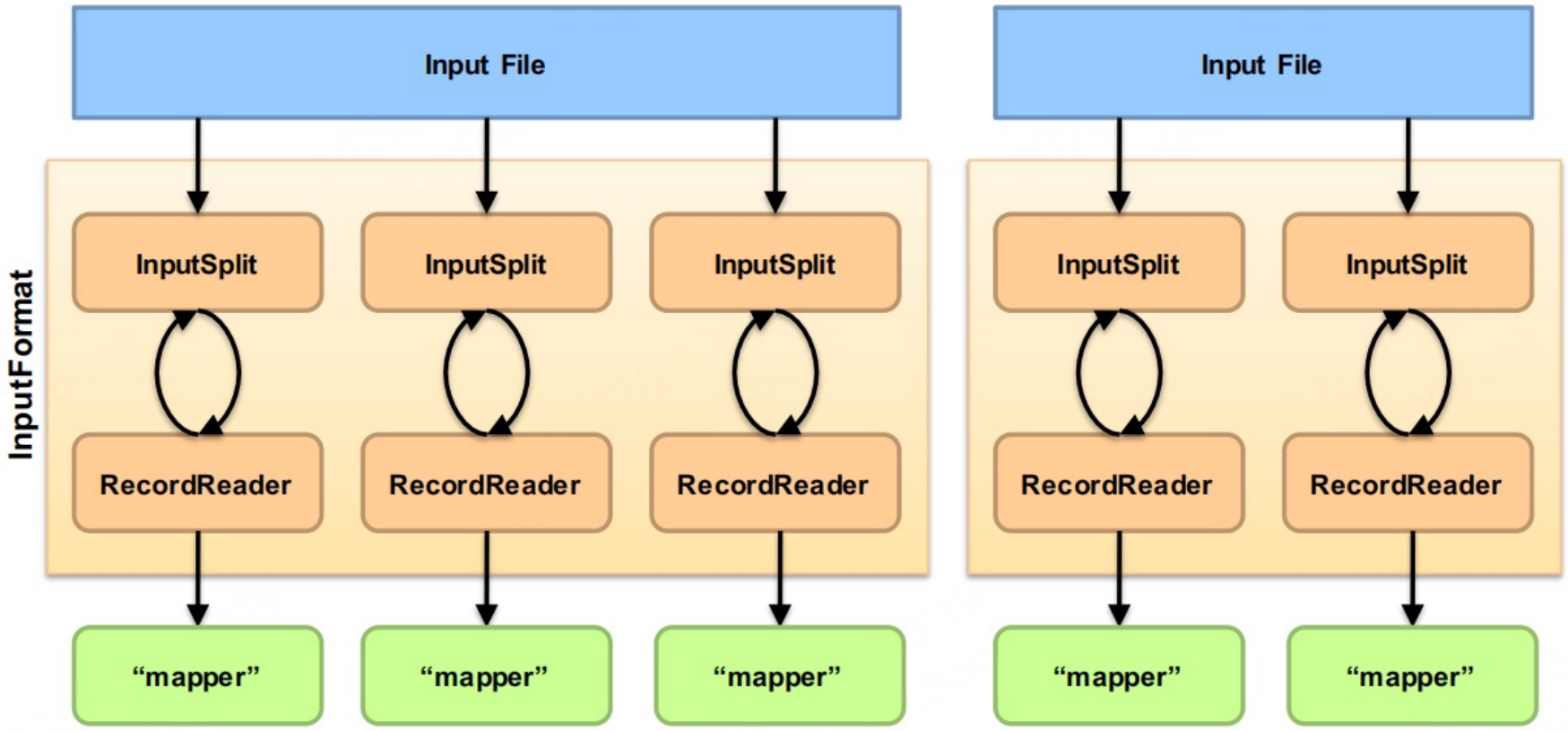
Spark driver



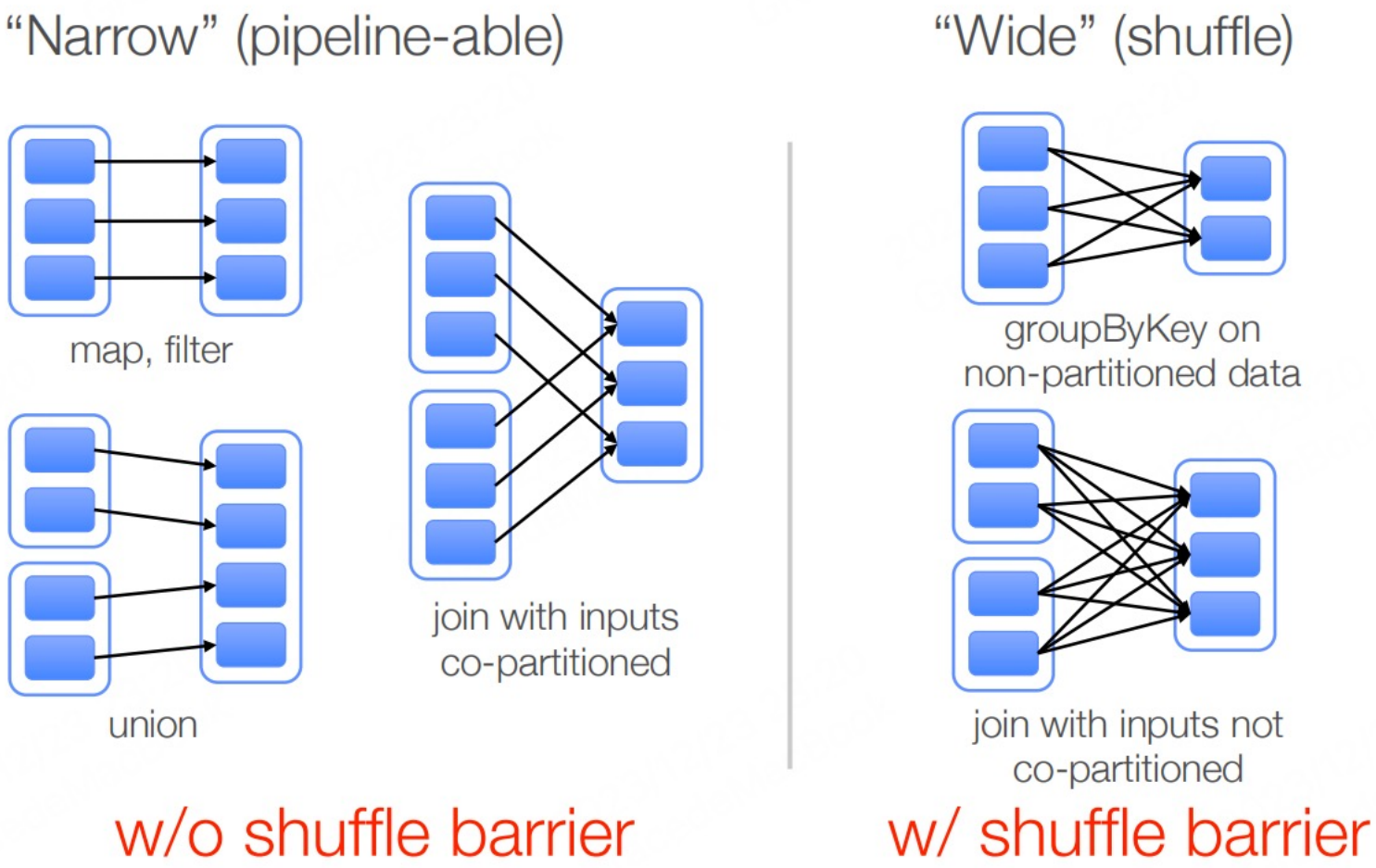

```
# create an RDD from HDFS
text_file = sc.textFile("hdfs://...")

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!

Starting points



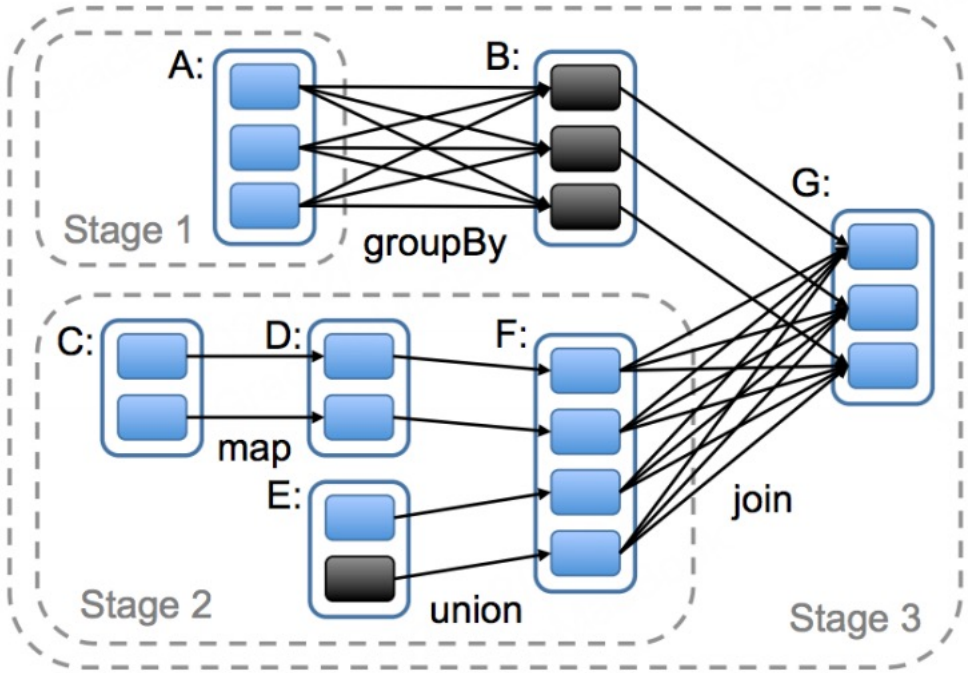
Physical operators



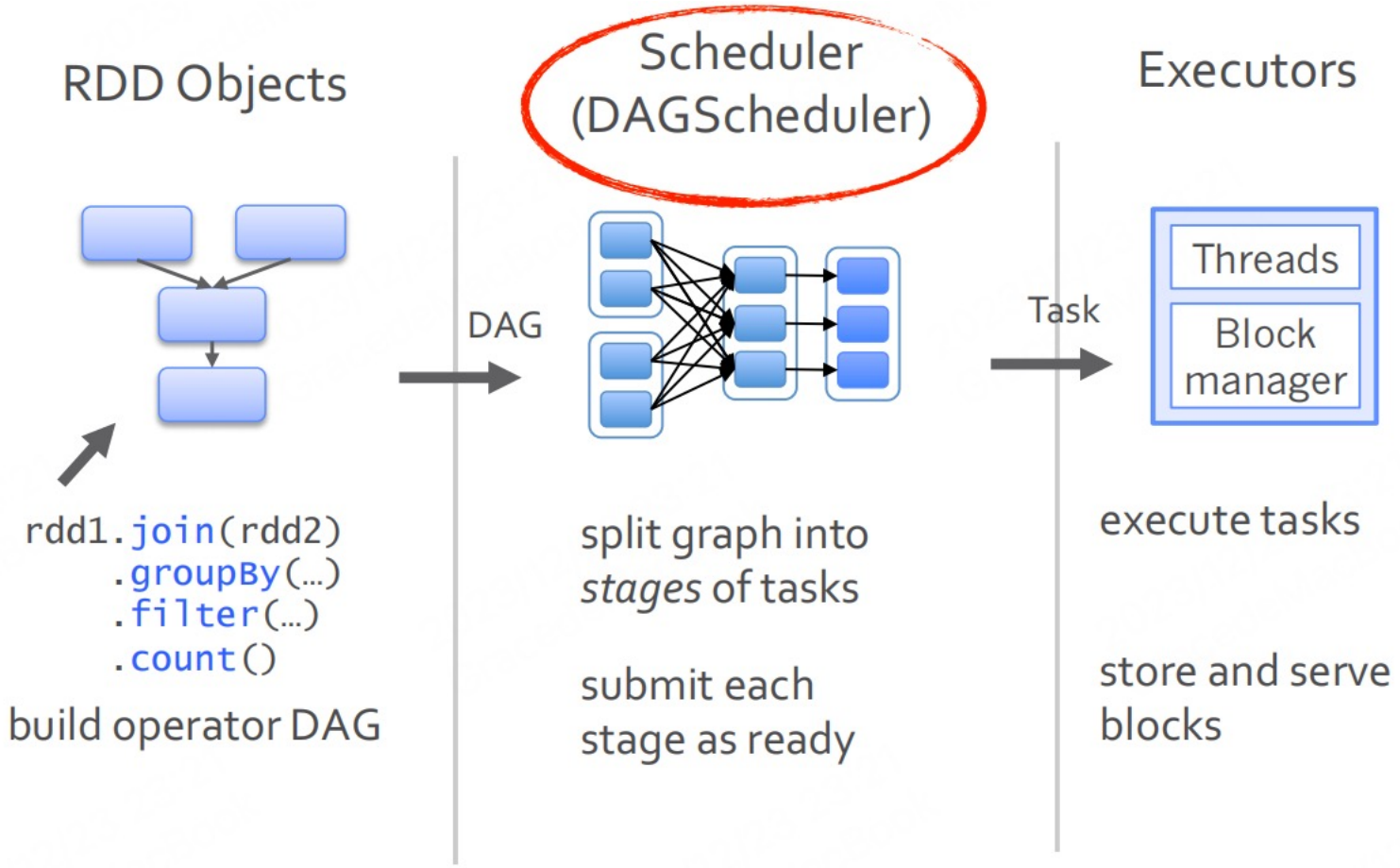
Execution plan

Directed Acyclic Graph (DAG)

- stage boundaries chartered by wide dependencies



Job scheduling



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

DAG Scheduler

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

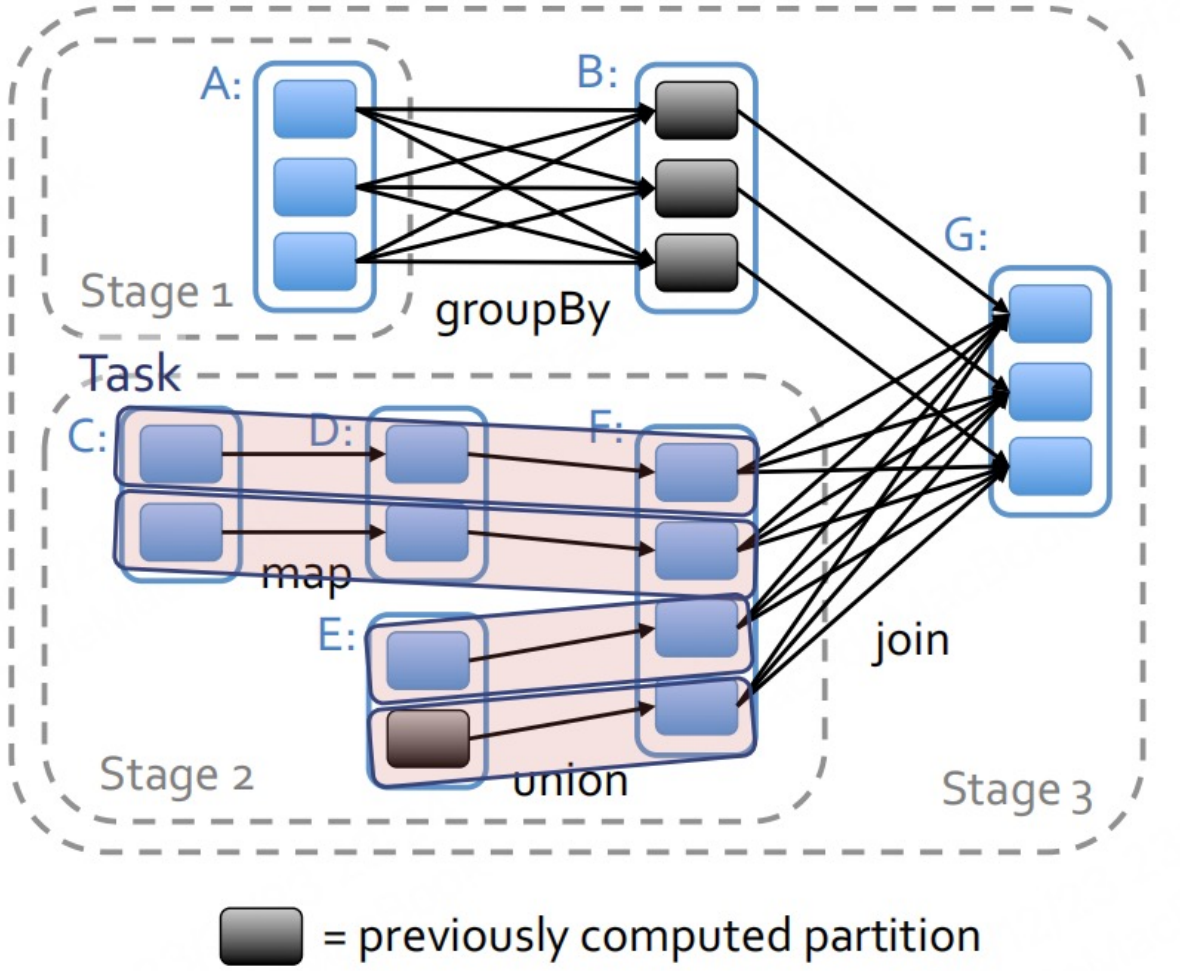
- build stages of tasks (depth-first search)
- submit tasks to lower-level scheduler, e.g., YARN, Mesos.
- lower-level scheduler schedules tasks based on locality
- resubmit failed stages, if any

Scheduler optimizations

Pipelines operations within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



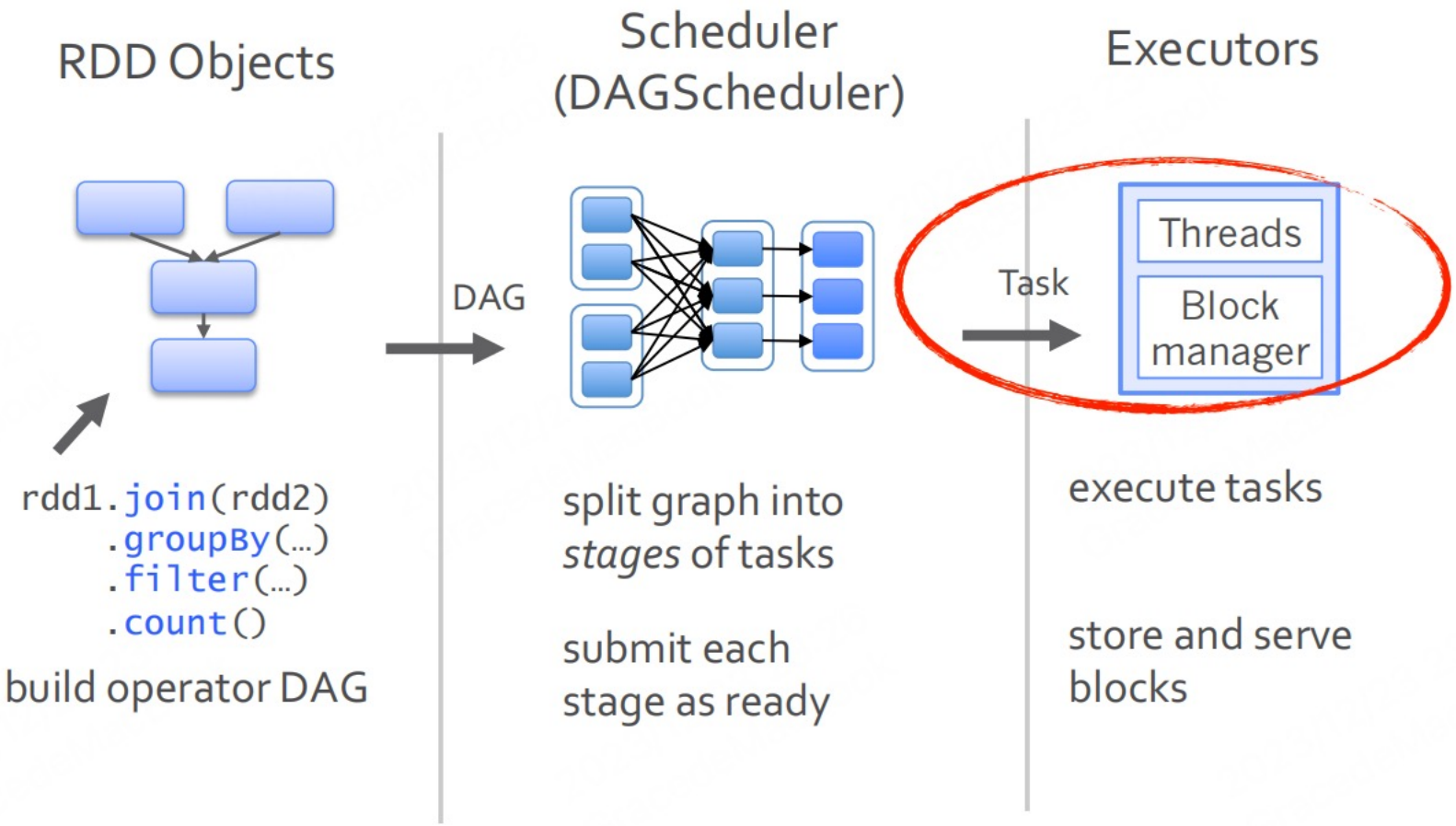
Task

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?



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

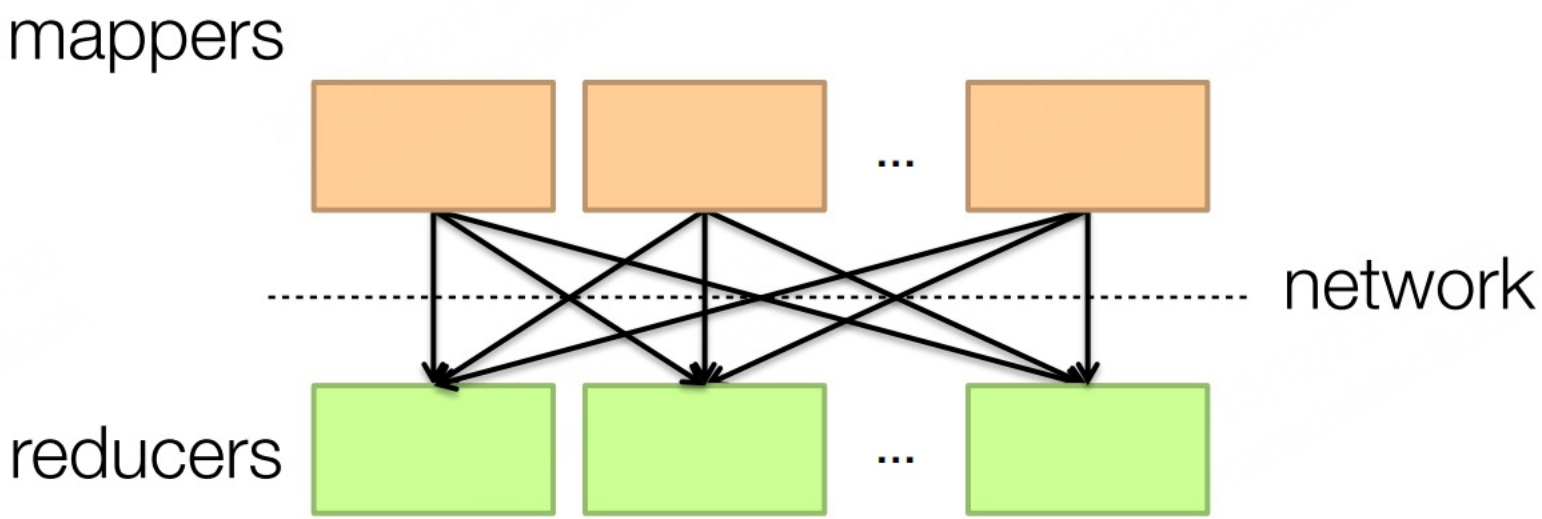
Why it works?

- 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

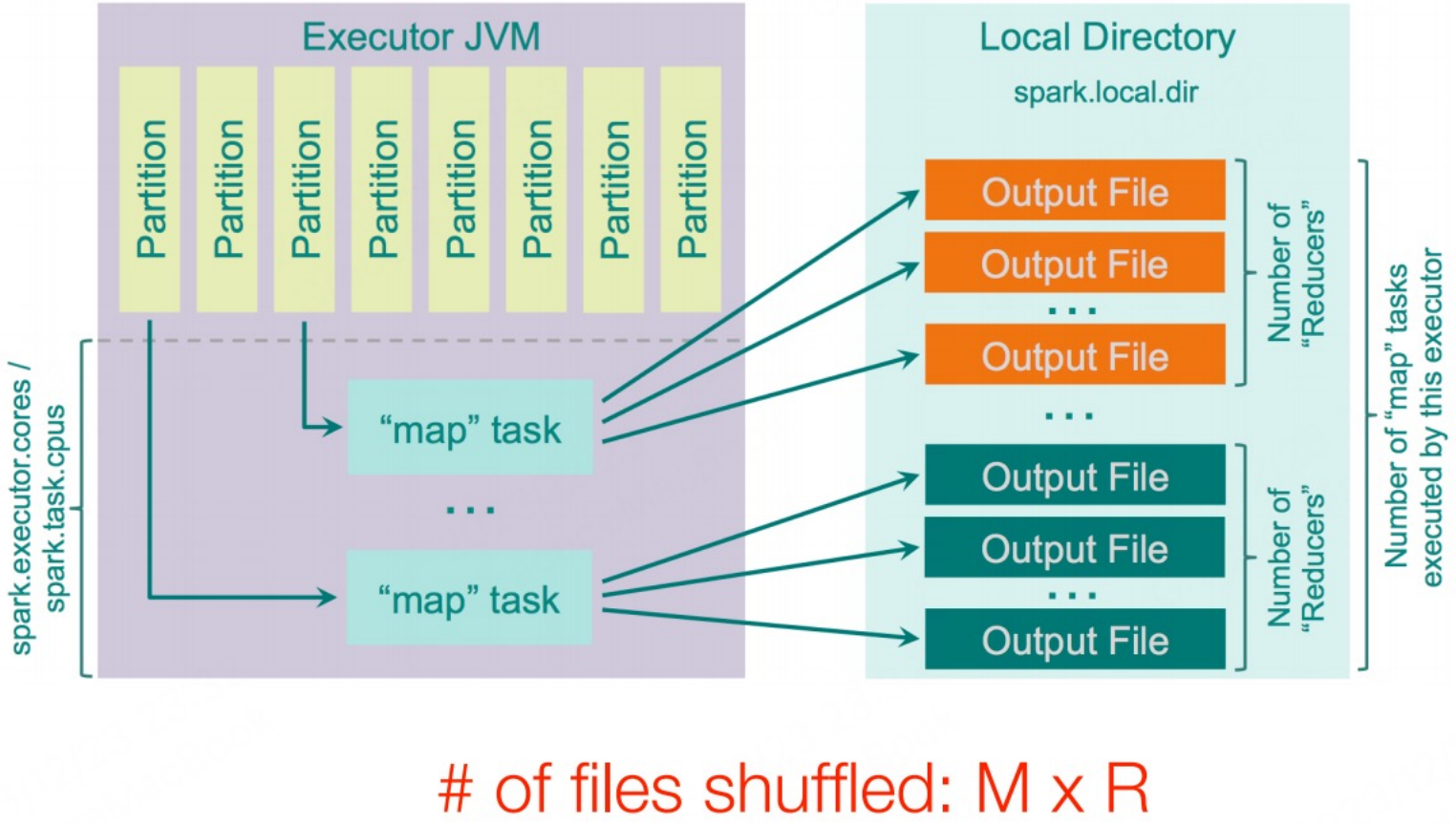
- **spark.shuffle.manager = hash**

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

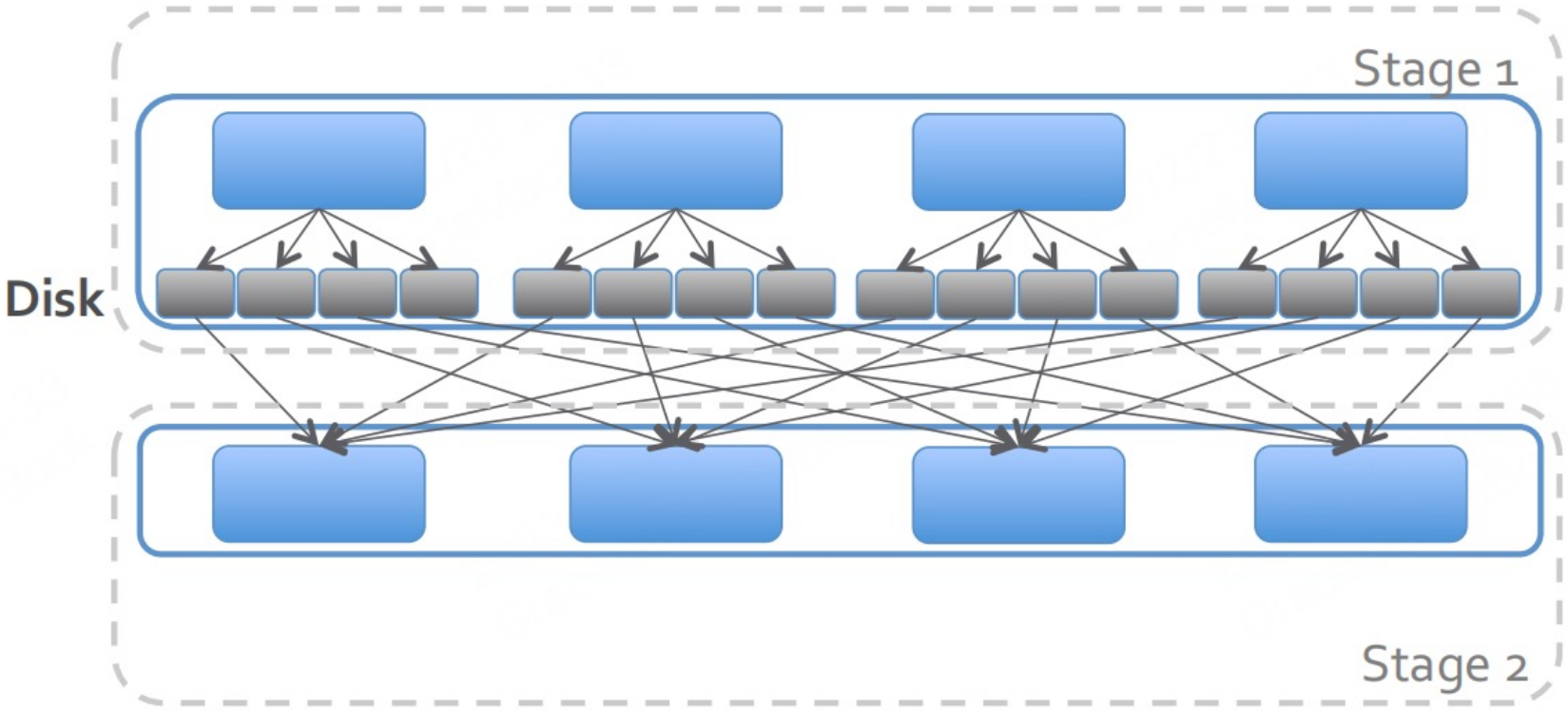
Could this be a problem?

Hash shuffle



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

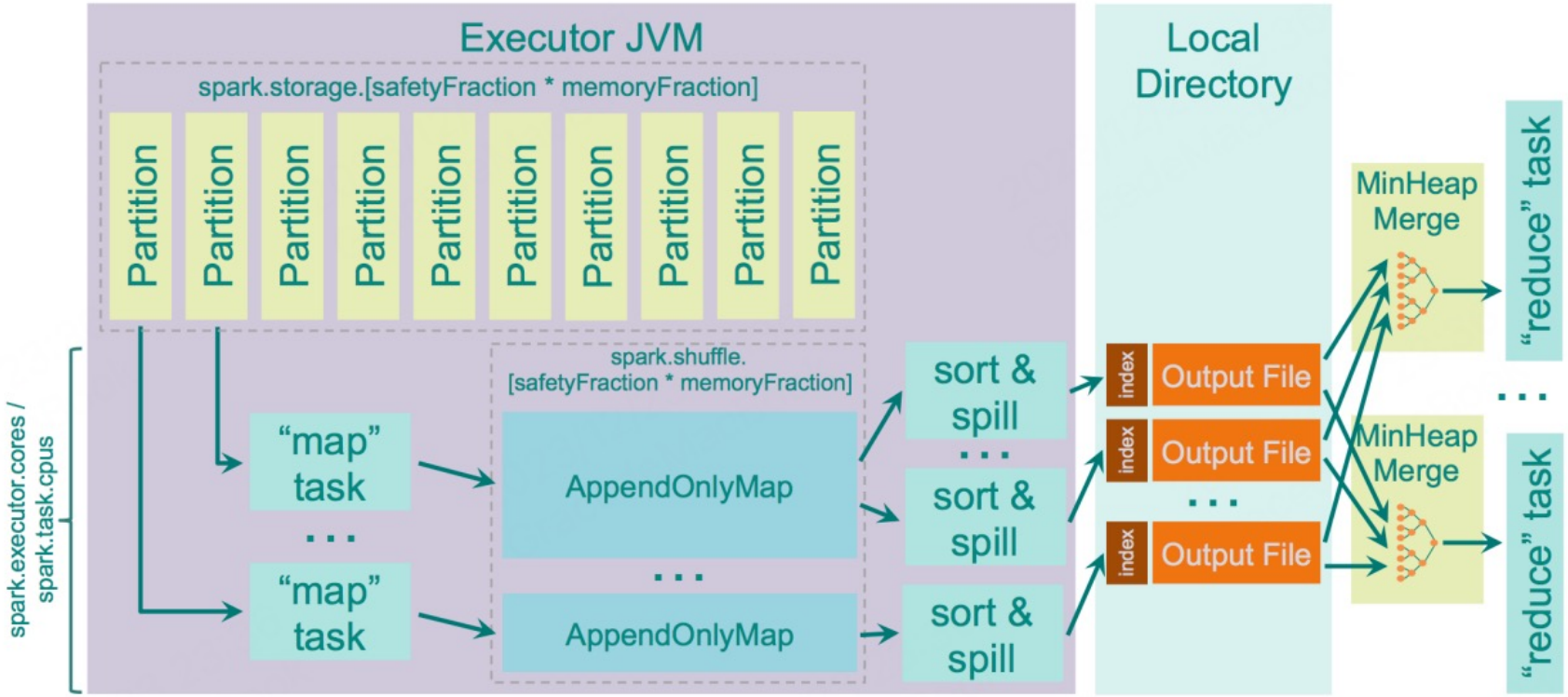
Default option since Spark 1.2.0

- **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)

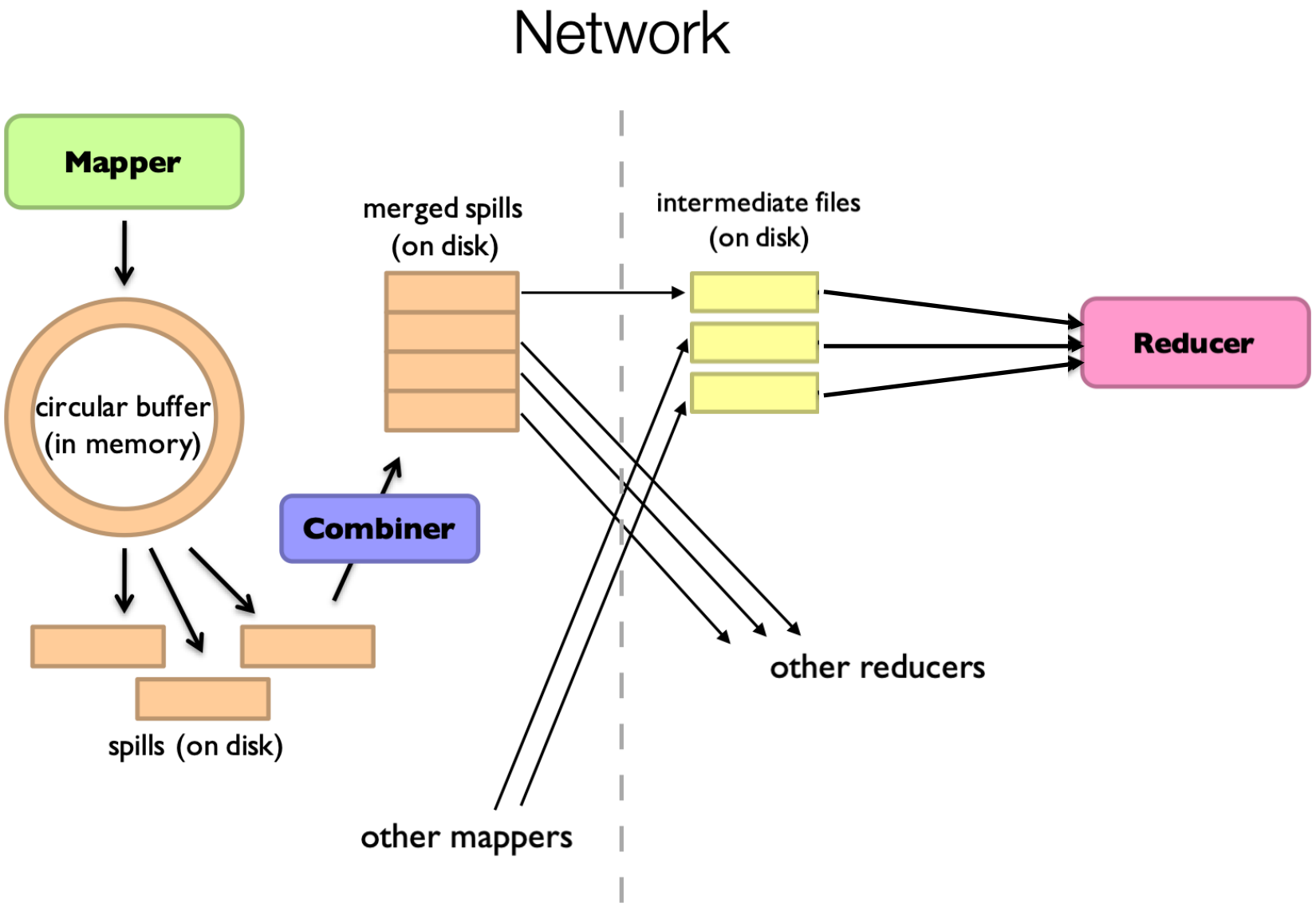
Cons:

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

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

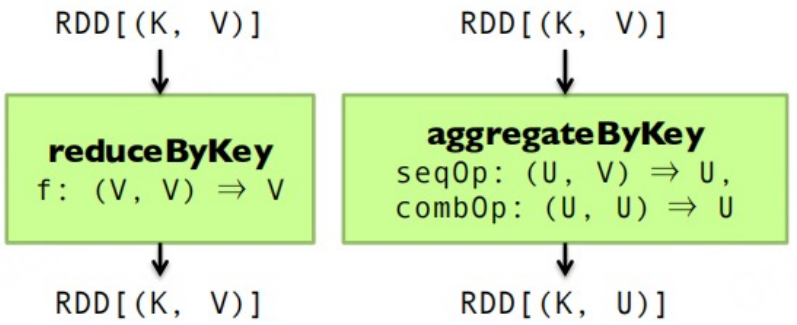
https://issues.apache.org/jira/browse/SPARK-7081

Remember this?



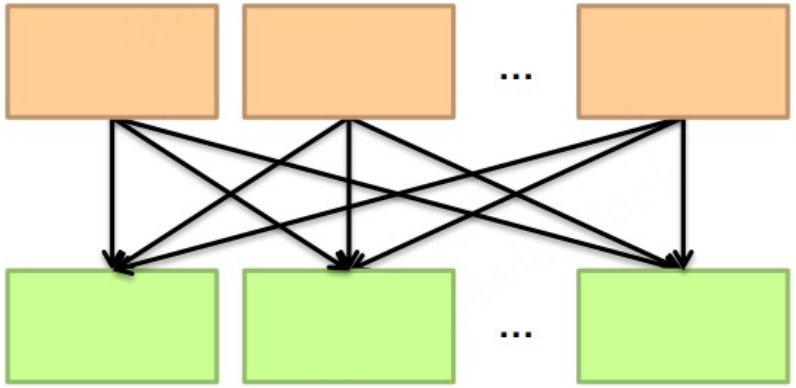
Where are the combiners in Spark?

Reduce-like operations

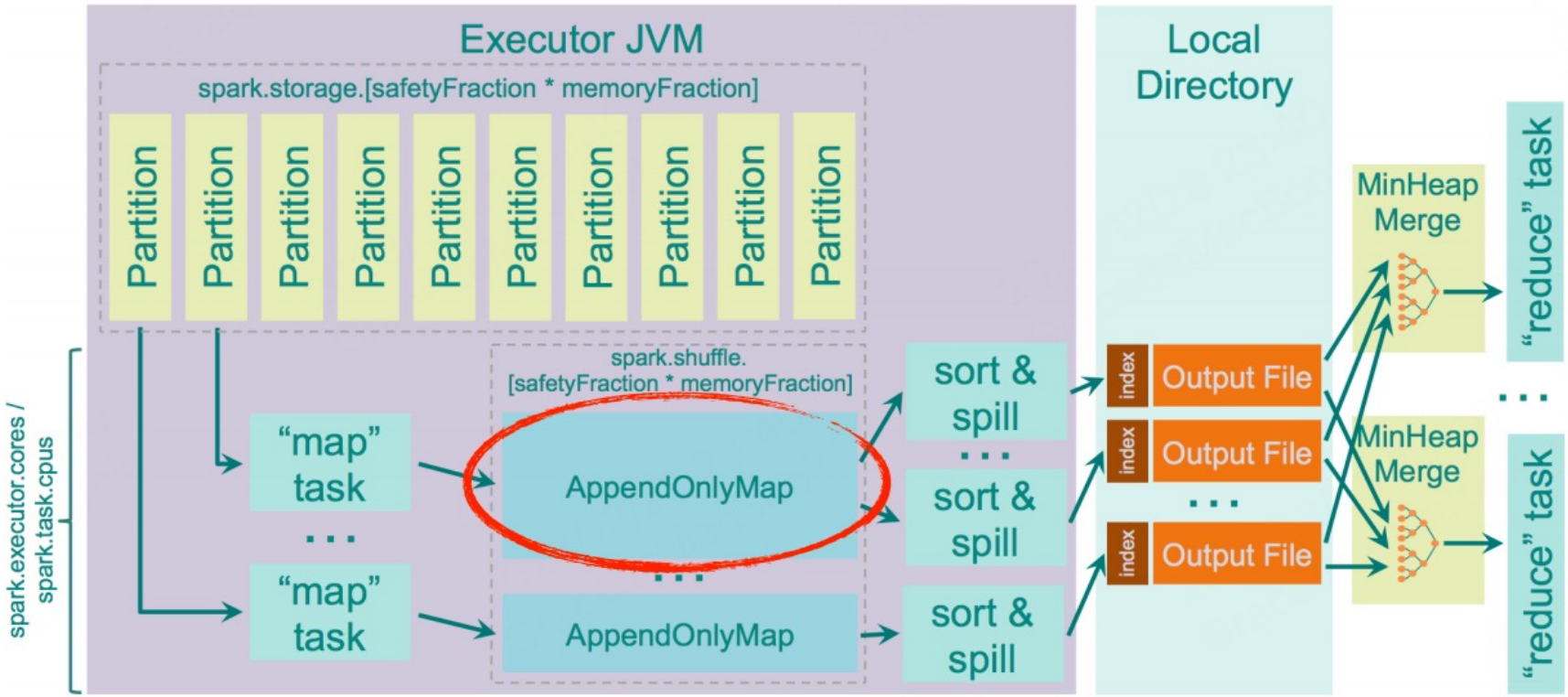


How can we optimize?

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?

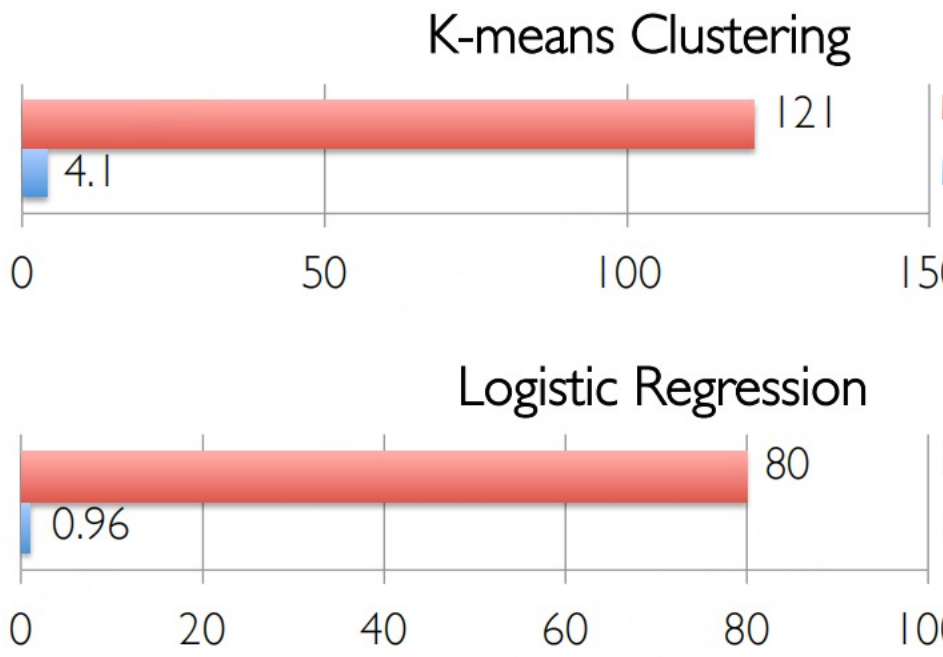
Use **combineByKey** method

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

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

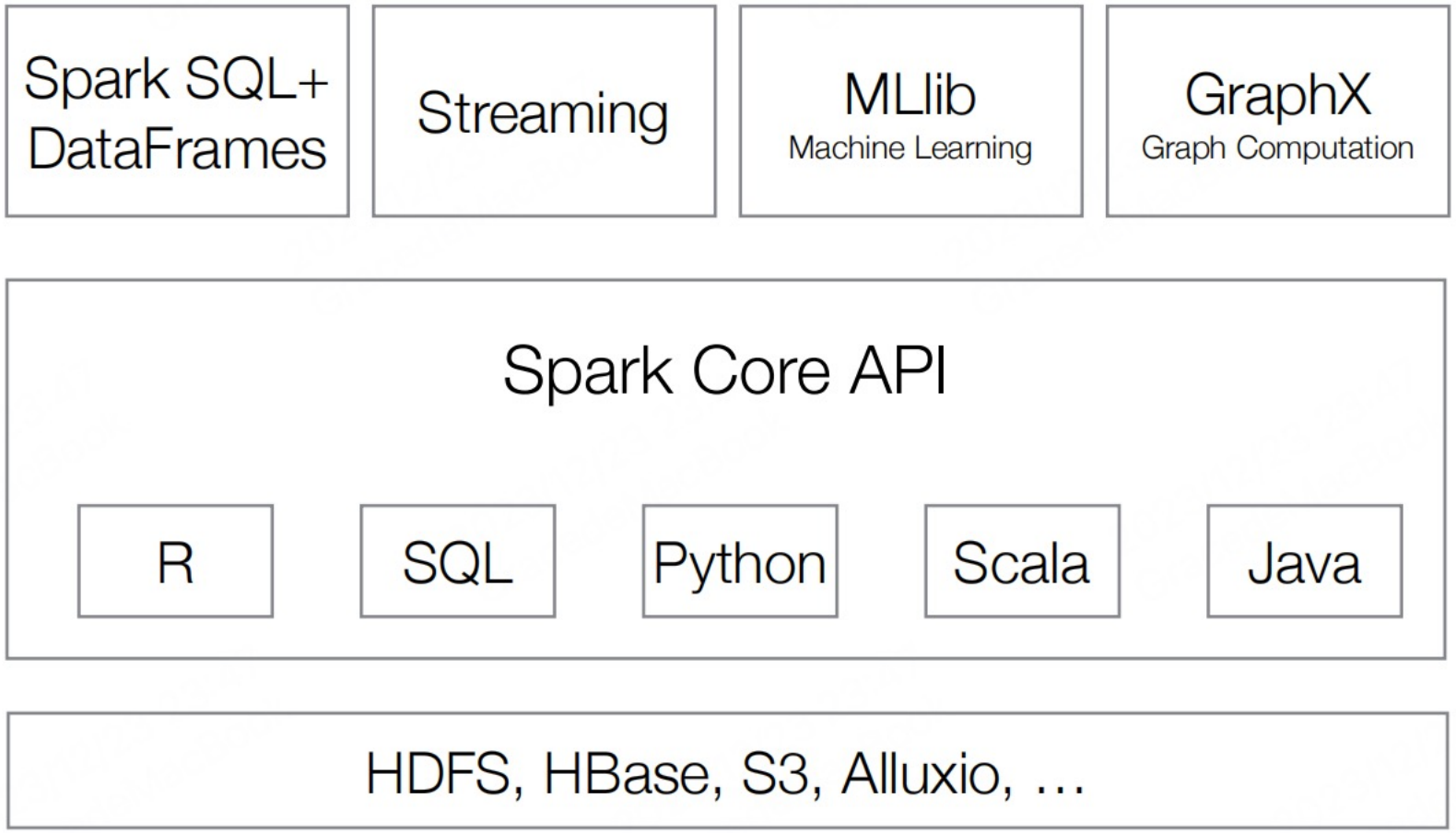
Lazy evaluation of the linage graph

- reduces wait states, better pipelining

Lower overhead for starting jobs

Less expensive shuffles

The Spark ecosystem



Spark SQL and DataFrames

RDDs are low-level

data.txt

```
Bob 23
Jimmy 32
Jason 27
Bob 26
Jason 75
Jimmy 24
Jason 45
```

```
# RDD Version:
data = sc.textFile("data.txt").map(lambda line: line.split(" "))
data.map(lambda x: (x[0], [int(x[1]), 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

Output:

```
[[u'Bob', 24], [u'Jimmy', 28], [u'Jason', 49]]
```

RDDs are low-level

data.txt

```
Bob 23
Jimmy 32
Jason 27
Bob 26
Jason 75
Jimmy 24
Jason 45
```

This?

```
# RDD Version:
data = sc.textFile("data.txt").map(lambda line: line.split(" "))
data.map(lambda x: (x[0], [int(x[1]), 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

Or this?

```
# DataFrame Version:
df.groupBy("name").avg("age").collect()
```

RDDs are low-level

They express the how of a solution better than the what

They’re slow on non-JVM languages like Python

They’re hard to be optimized by Spark (opaque data)

It’s too easy to build an inefficient RDD transformation chain

Inefficient transformation chain

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```

What's the problem?

join before filter

From RDDs to DataFrames

RDD with schema: distributed collection of data grouped into named columns

immutable once constructed

track linage info to efficiently recompute lost data

enable operations on collection of elements in parallel

```
# RDD Version:
data = sc.textFile("data.txt").map(lambda line: line.split(" "))
data.map(lambda x: (x[0], [int(x[1]), 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

```
# DataFrame Version:
df.groupBy("name").avg("age").collect()
```

DataFrames in Spark

DataFrames can be constructed

- by parallelizing existing collections (e.g., RDDs)
- by transforming an existing DataFrames
- from files in HDFS or any other storage system

Data sources

DataFrames APIs

DataFrame API provides a higher-level abstraction

- inspired by data frames in R and Python (Pandas)
- use a query language to manipulate data
- even supports SQL

```
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)

# Alternatively, using a Pandas-like syntax
young = users[users.age < 21]

# Increment everybody's age by 1
young.select(young["name"], young["age"] + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

```
# RDD Version:
data = sc.textFile("data.txt").map(lambda line: line.split(" "))
data.map(lambda x: (x[0], [int(x[1]), 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()


# DataFrame Version:
df.groupBy("name").avg("age").collect()


# SQL Version:
df.registerTempTable("people")
sqlContext.sql("SELECT name, avg(age) FROM people GROUP BY
name").show()
```

DataFrames and Spark SQL

DataFrames are fundamentally tied to Spark SQL

- The DataFrames API provides a programmatic interface—a domain-specific language (DSL)—for interacting with data
- Spark SQL provides a SQL-like interface
- Anything you can do in Spark SQL, you can do in DataFrames, and vice versa

Spark SQL

Spark SQL allows you to manipulate distributed data w/ SQL queries:

- SQLContex: sql provides a rich subset of SQL92
- HiveContex: hiveql provides a richer Hive's SQL dialect

Spark SQL

You issue SQL queries through SQLContext (or HiveContext),

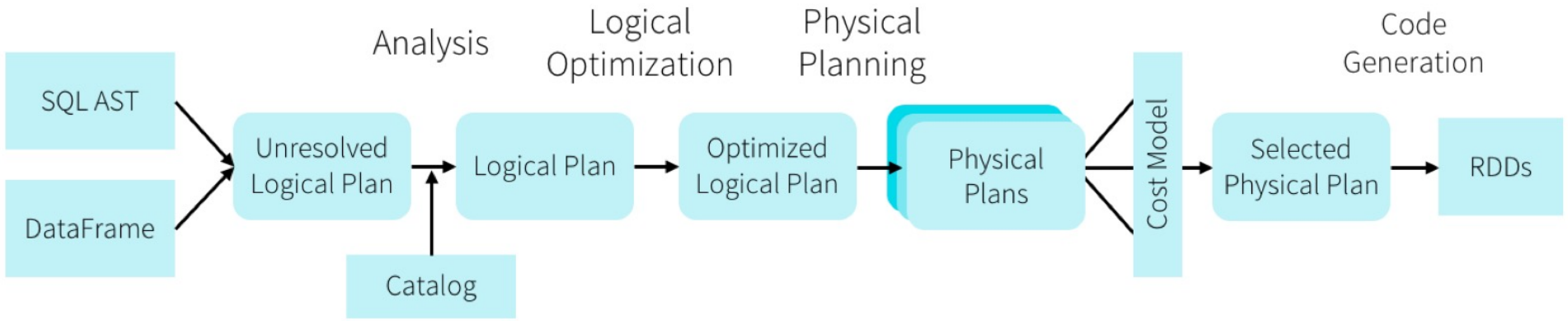
using the sql() method

- The sql() method returns a DataFrame
- DataFrame methods mix with SQL queries seamlessly

DataFrame is more than
“writing less code”

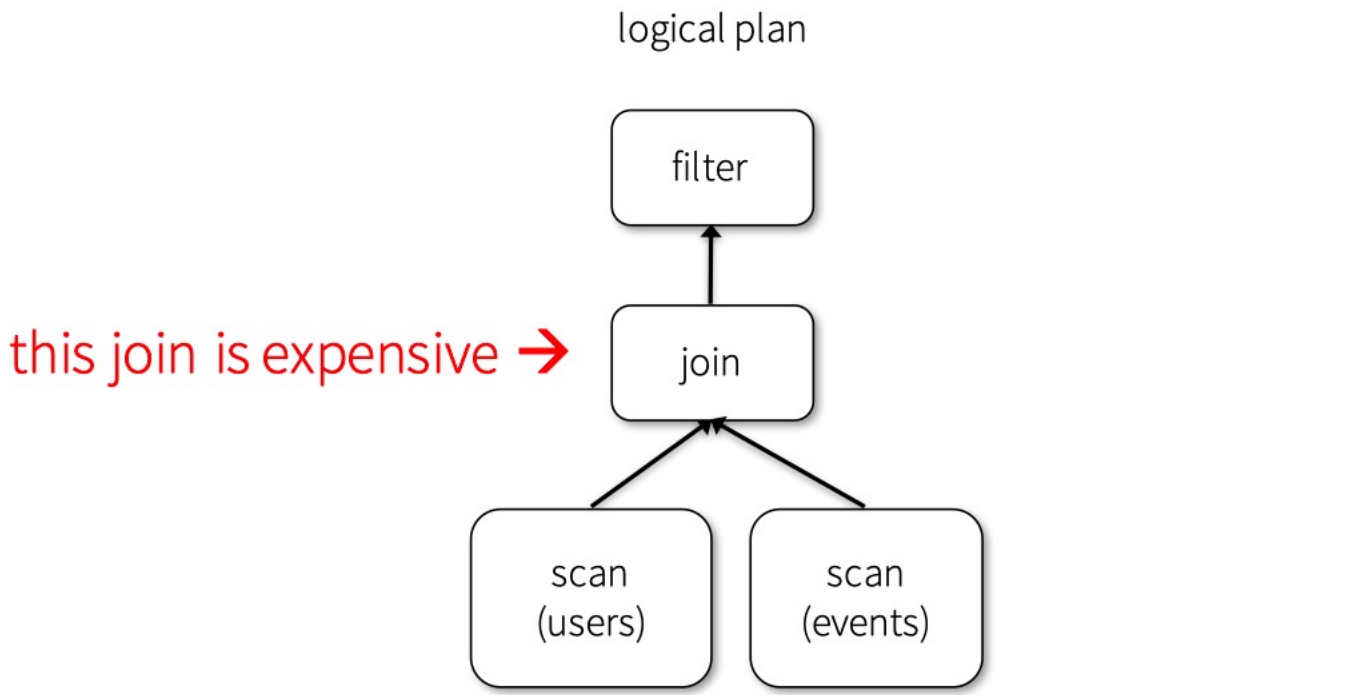
DataFrame queries are optimized

Execution is lazy, allowing it to be optimized by Catalyst



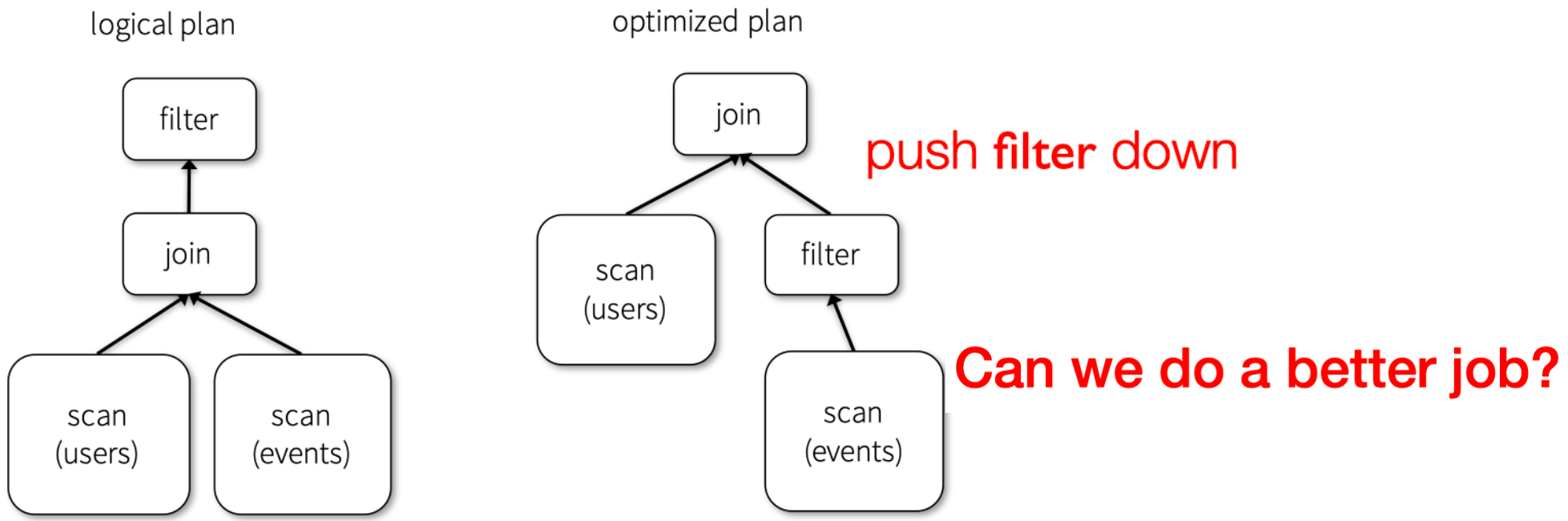
Plan optimization & execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



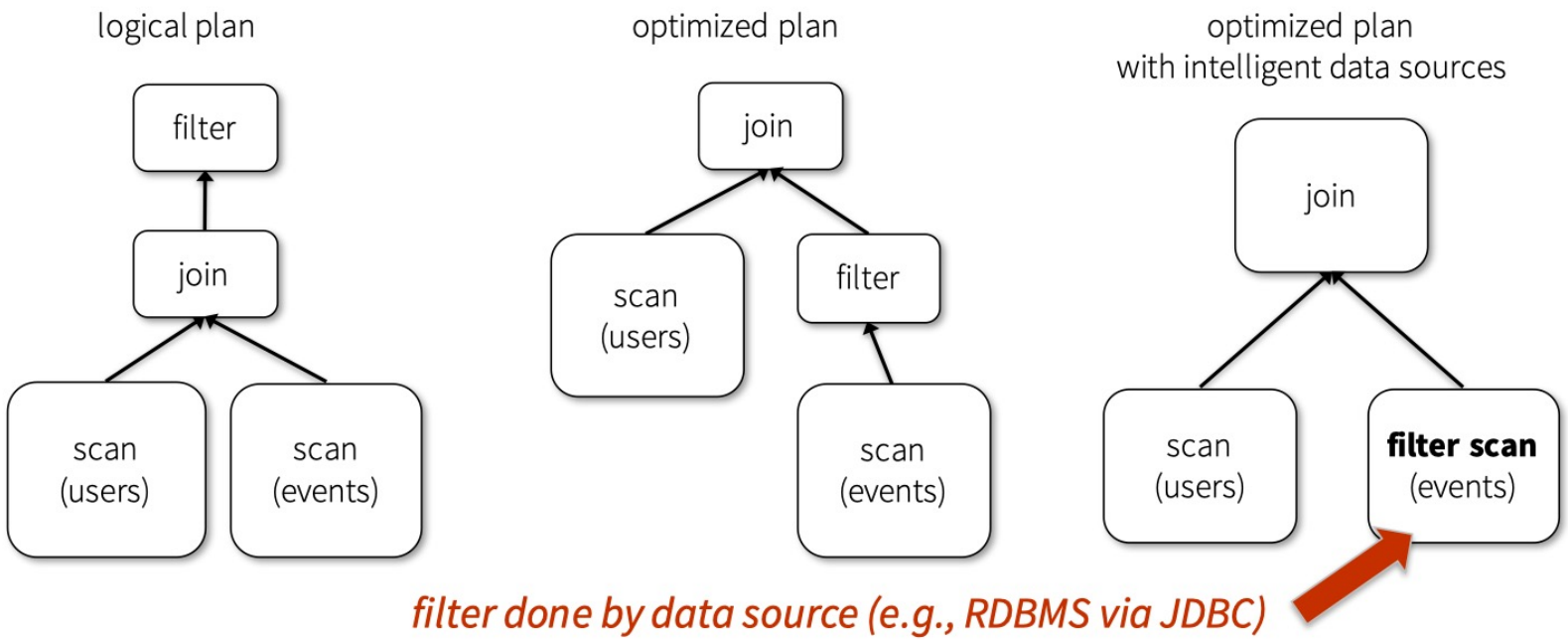
Plan optimization & execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```

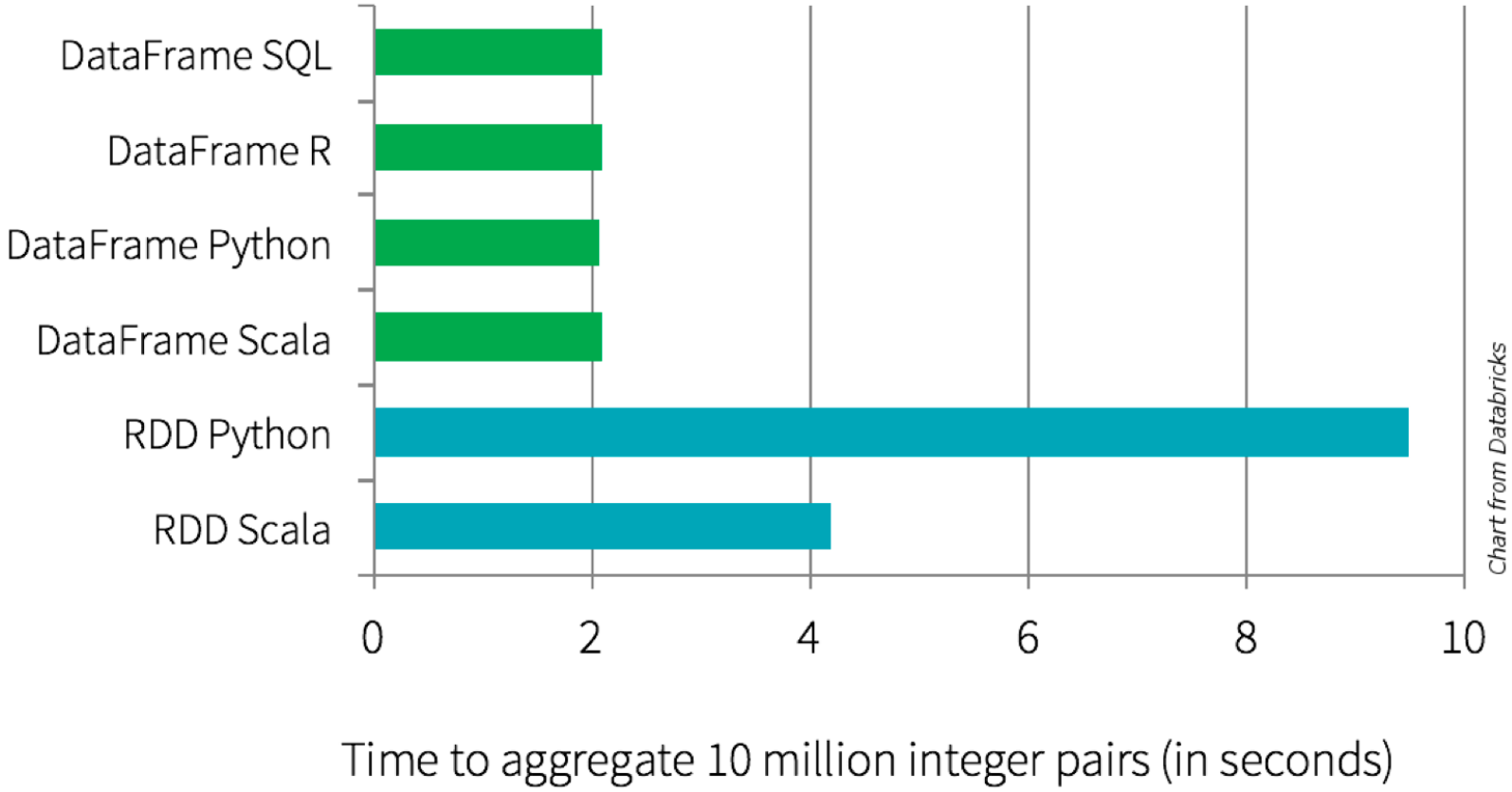


Plan optimization & execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



DataFrames are faster



Source from Databricks

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

Credits

- Some slides are adapted from course slides of COMP 4651 in HKUST