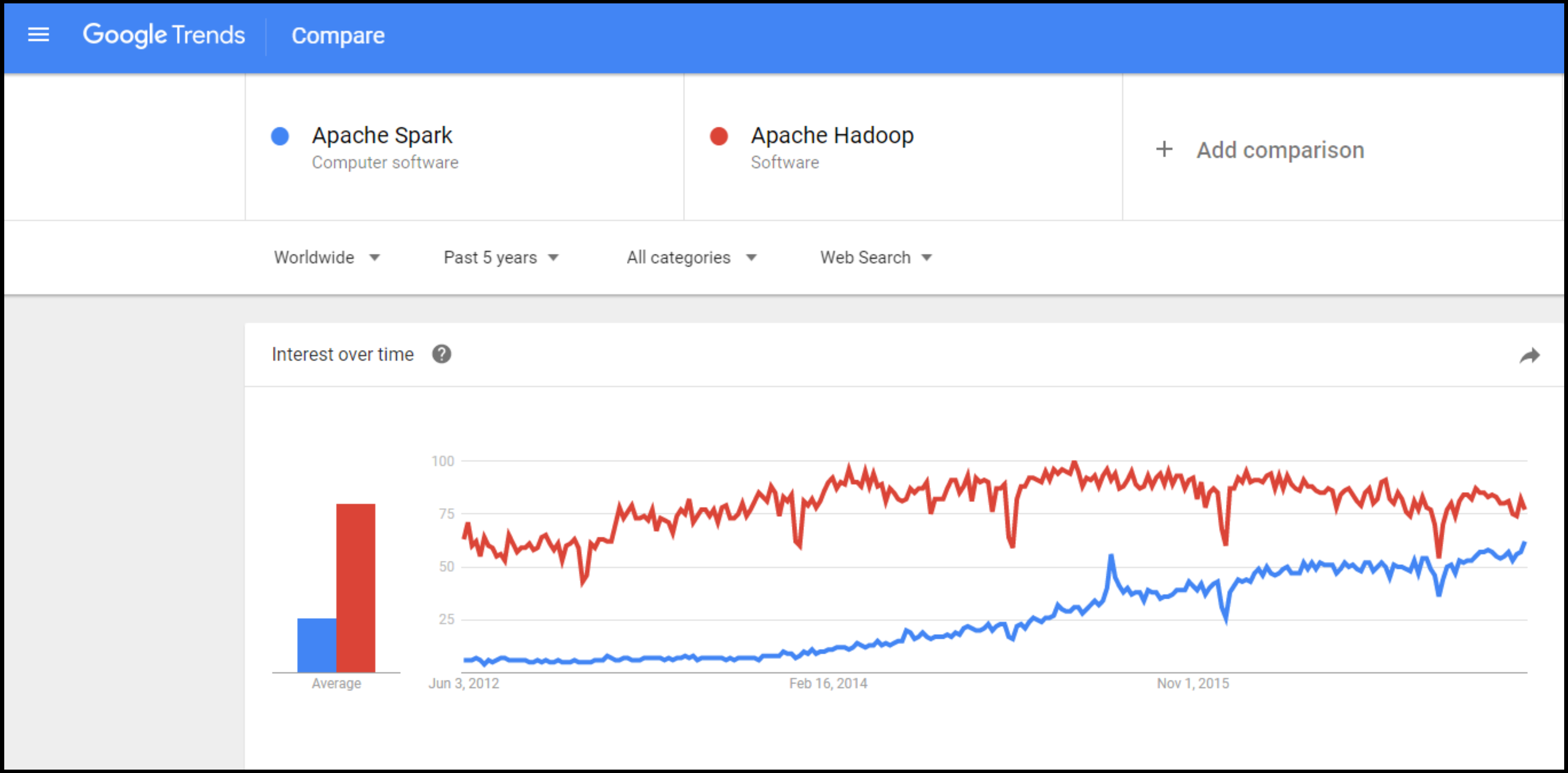
CC5212-1

PROCESAMIENTO MASIVO DE DATOS
OTOÑO 2017

Lecture 8: Apache Spark (Core)

Aidan Hogan
aidhog@gmail.com

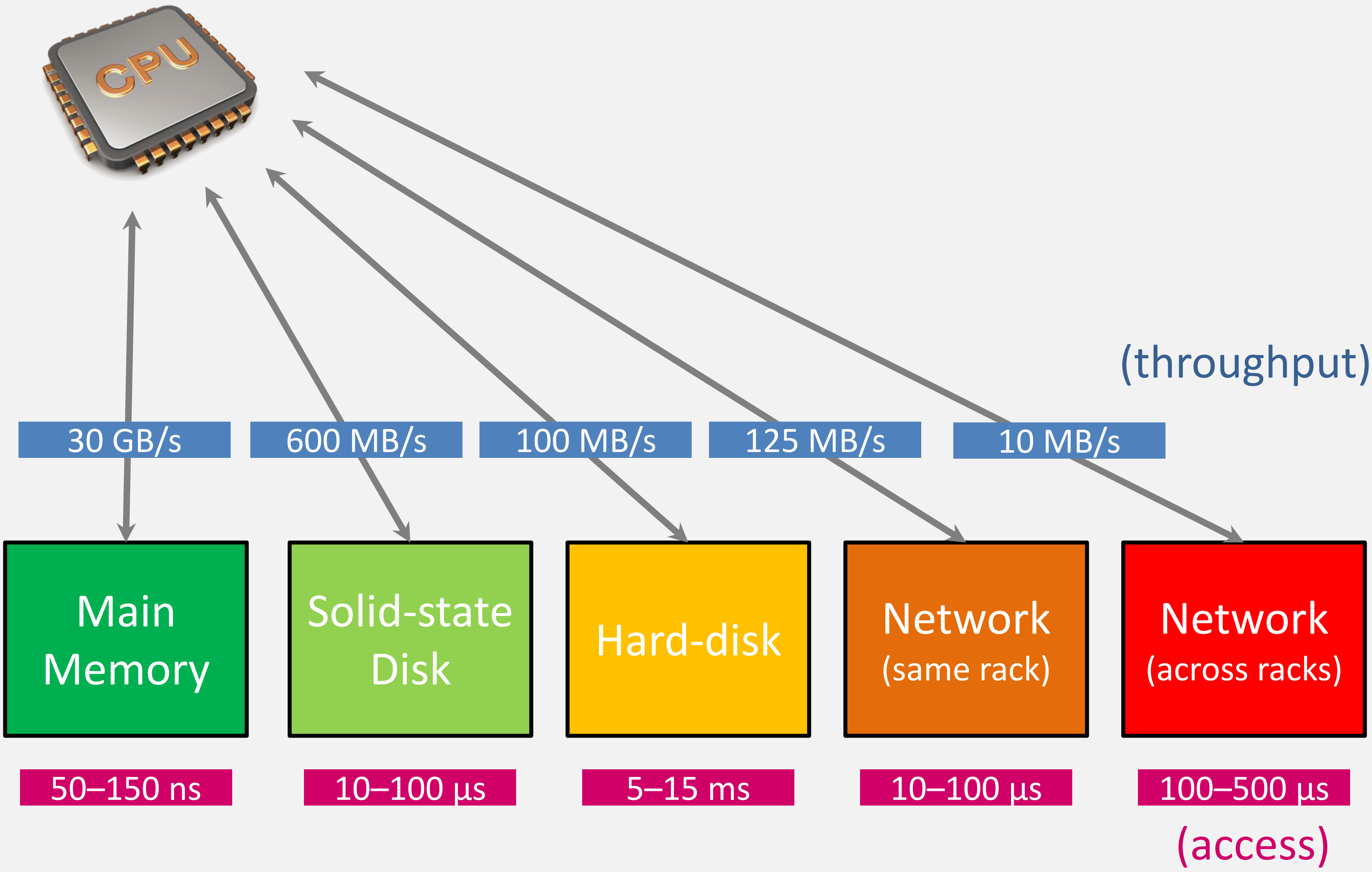
Spark vs. Hadoop



What is the main weakness of Hadoop?

Let's see …

Data Transport Costs



MapReduce/Hadoop

1. Input

2. Map

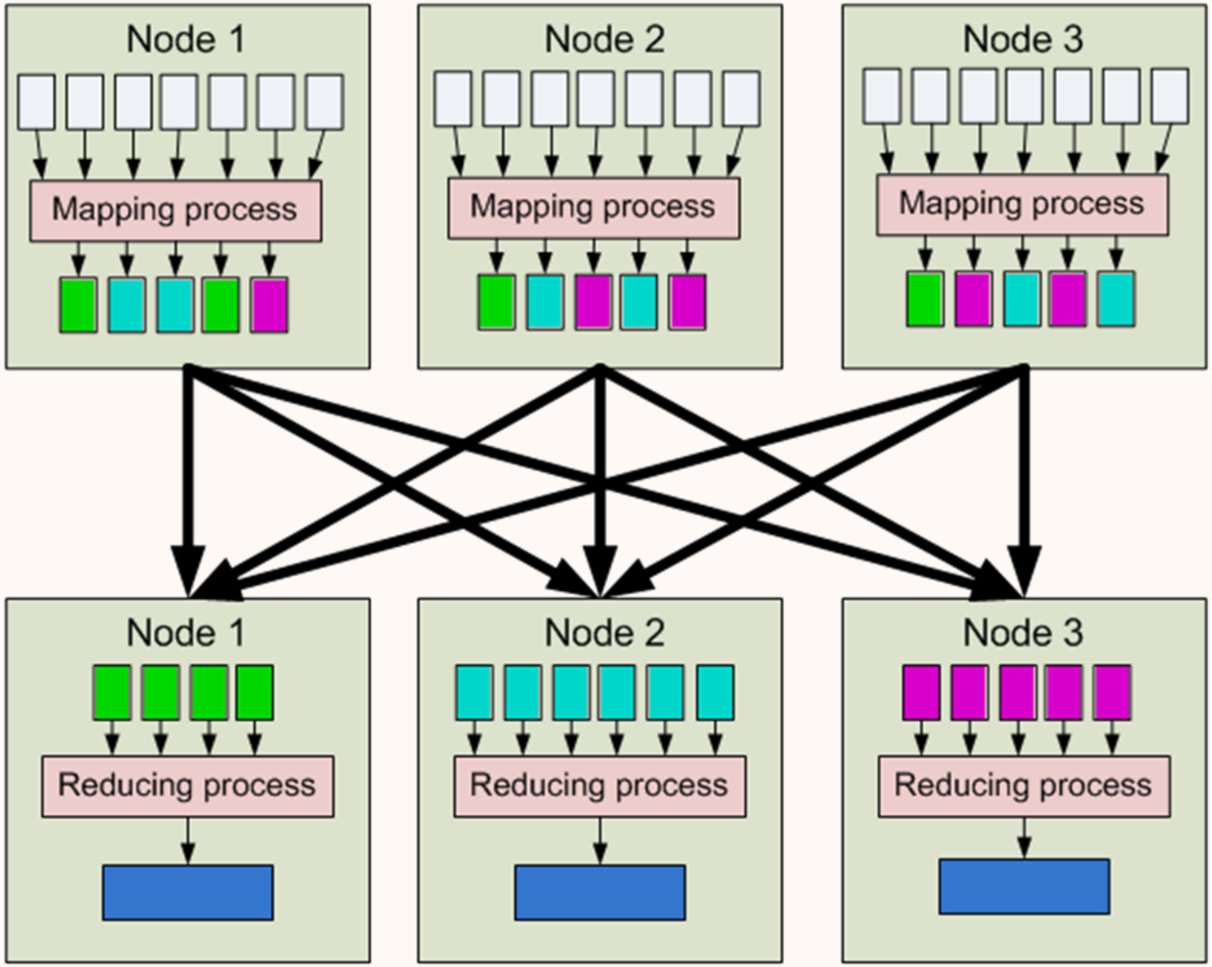
3. Partition [Sort]

4. Shuffle

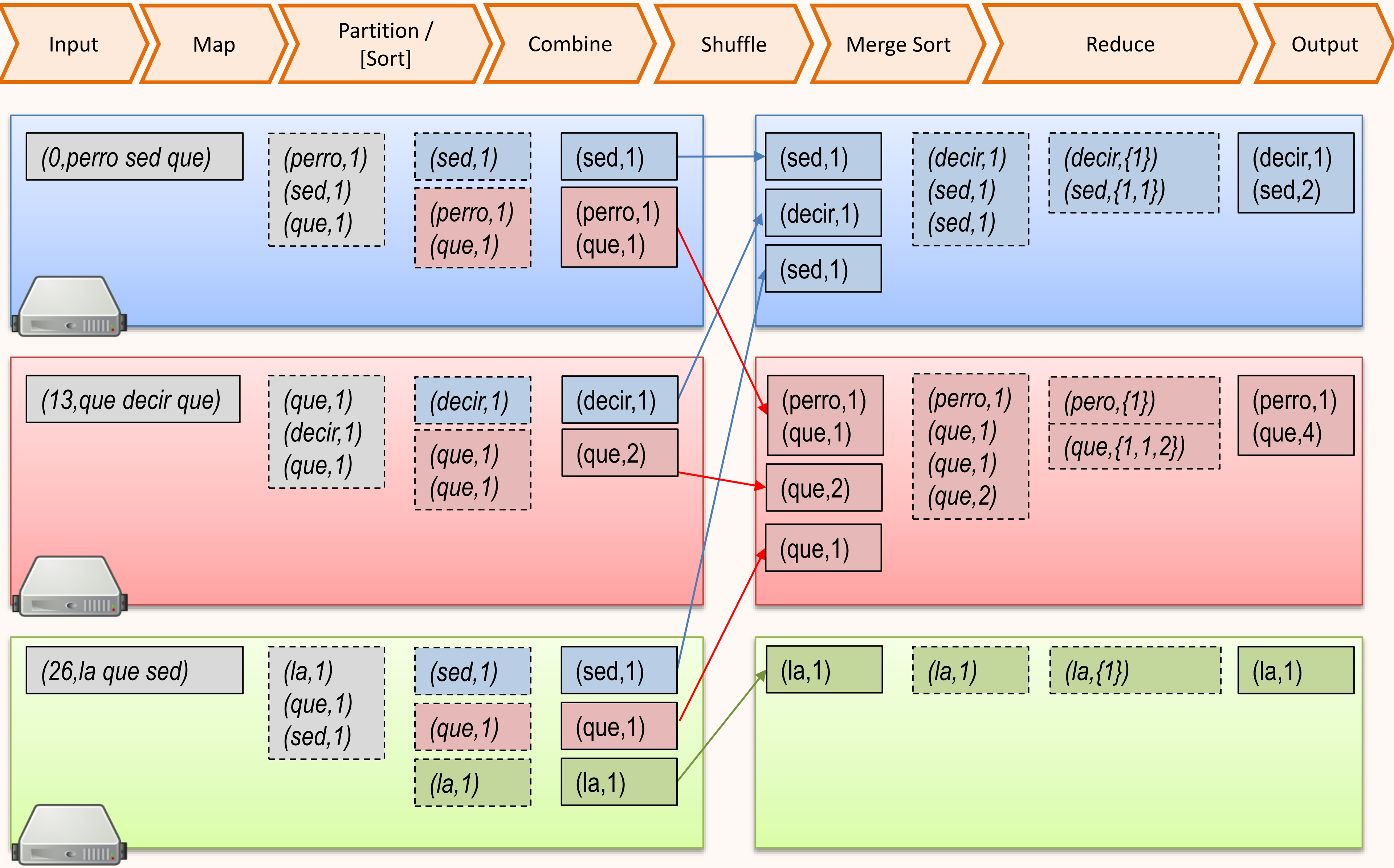
5. Merge Sort

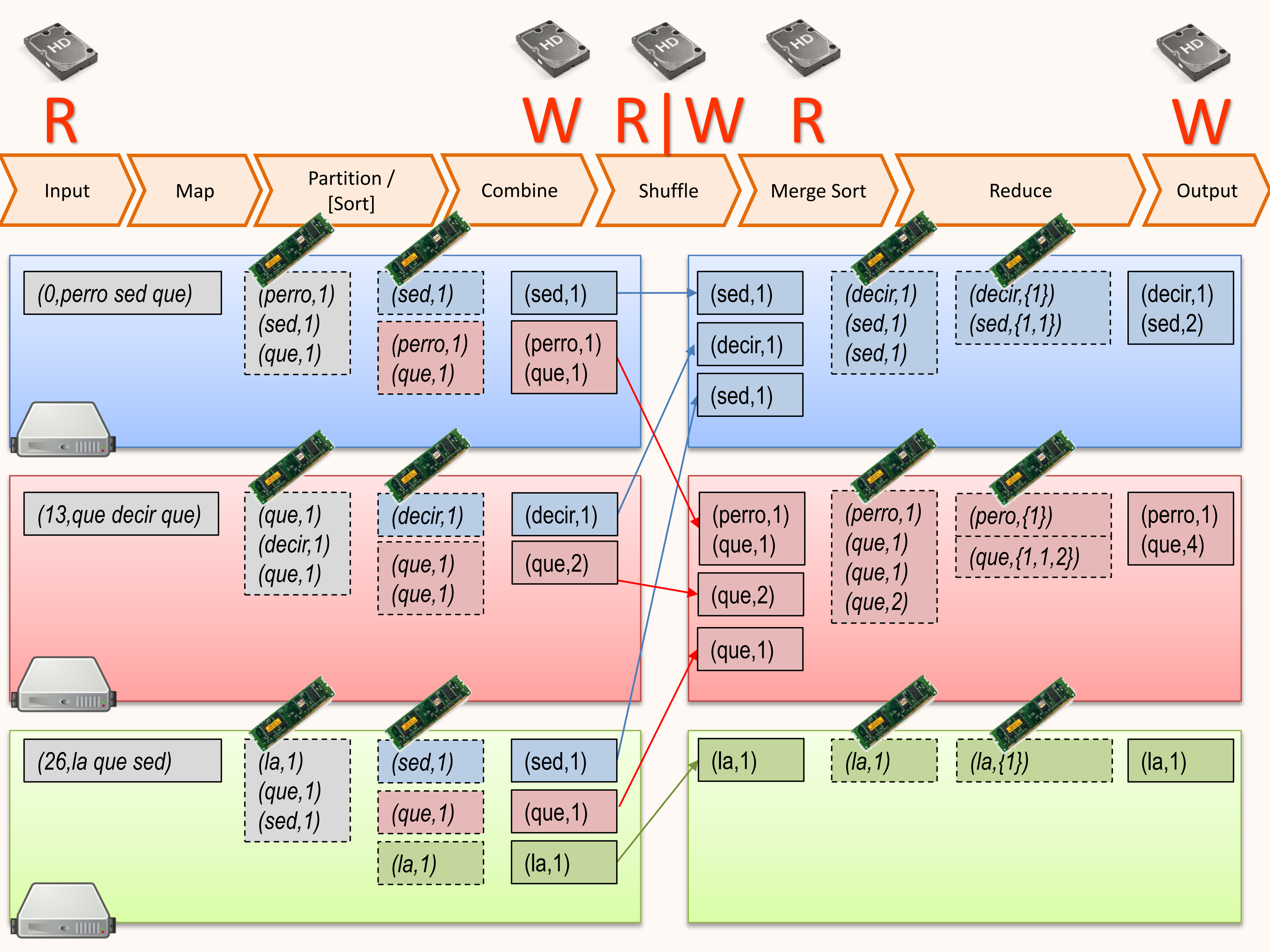
6. Reduce

7. Output

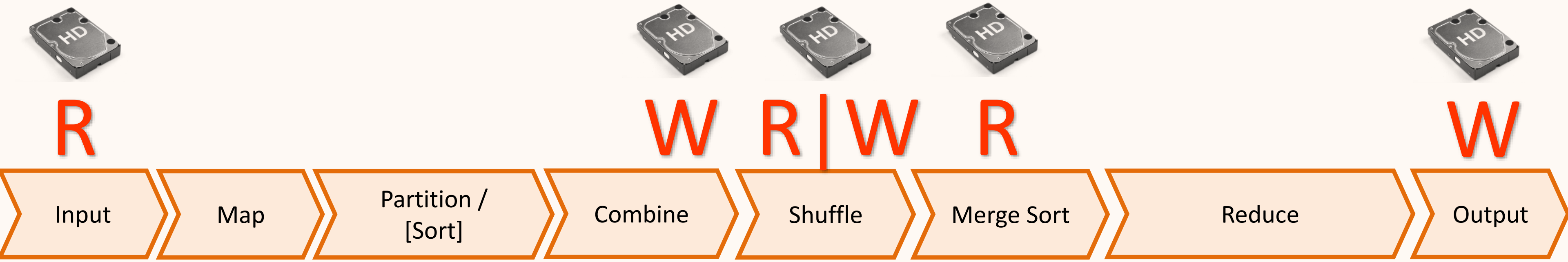


MapReduce/Hadoop

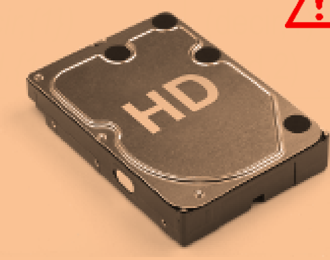


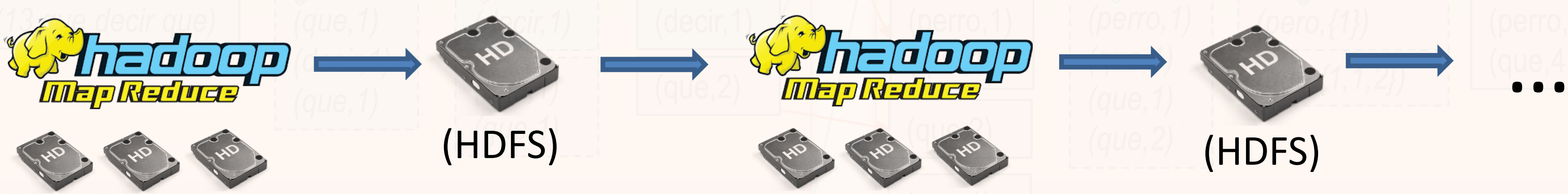


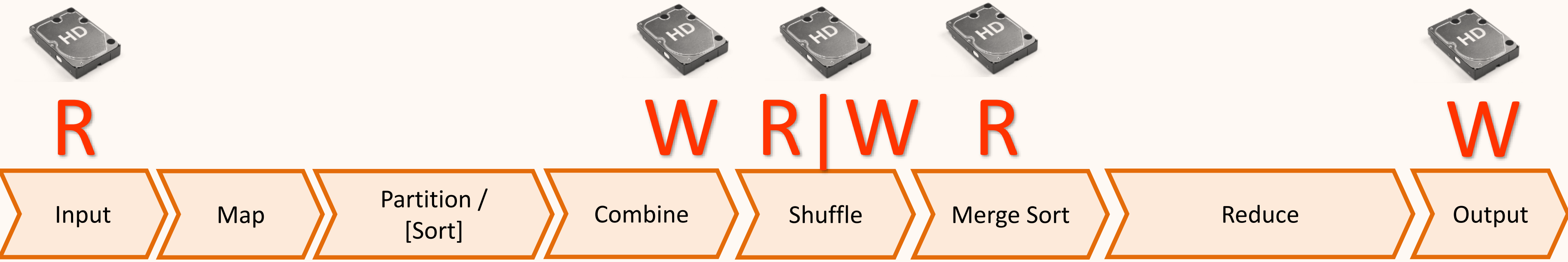
R
W
R|W
R
W
Input
Map
Partition / [Sort]
Combine
Shuffle
Merge Sort
Reduce
Output
(0,perro sed que)
(perro,1) (sed,1) (que,1)
(sed,1)
(perro,1) (que,1)
(sed,1)
(perro,1) (que,1)
(13,que decir que)
(que,1) (decir,1) (que,1)
(decir,1)
(que,1) (que,1)
(decir,1)
(que,2)
(26,la que sed)
(la,1) (que,1) (sed,1)
(sed,1)
(que,1)
(la,1)
(sed,1)
(que,1)
(la,1)
(sed,1)
(decir,1)
(sed,1)
(decir,1) (sed,1) (sed,1)
(decir,{1}) (sed,{1,1})
(decir,1) (sed,2)
(perro,1) (que,1)
(que,2)
(que,1)
(perro,1) (que,1) (que,1) (que,2)
(pero,{1})
(que,{1,1,2})
(perro,1) (que,4)
(la,1)
(la,1)
(la,{1})
(la,1)



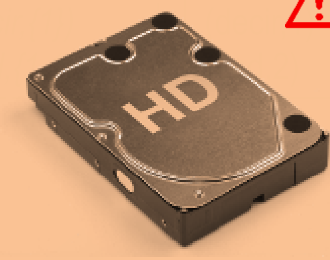
MapReduce/Hadoop always coordinates
between phases (Map → Shuffle → Reduce)
and between high-level tasks (Count → Order)
using the hard-disk.







MapReduce/Hadoop always coordinates
between phases (Map → Shuffle → Reduce)
and between high-level tasks (Count → Order)
using the hard-disk.



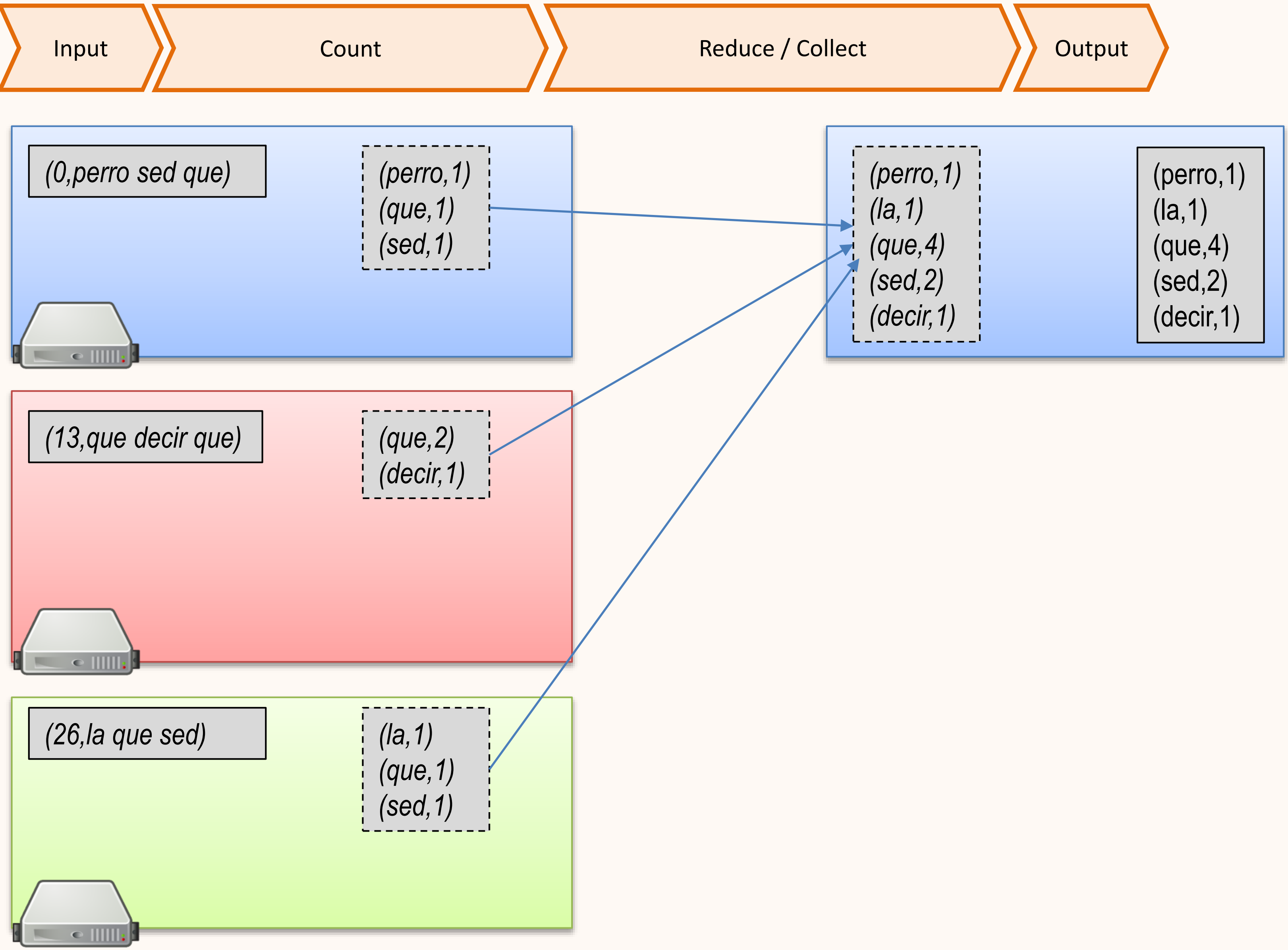
We saw this already counting words …

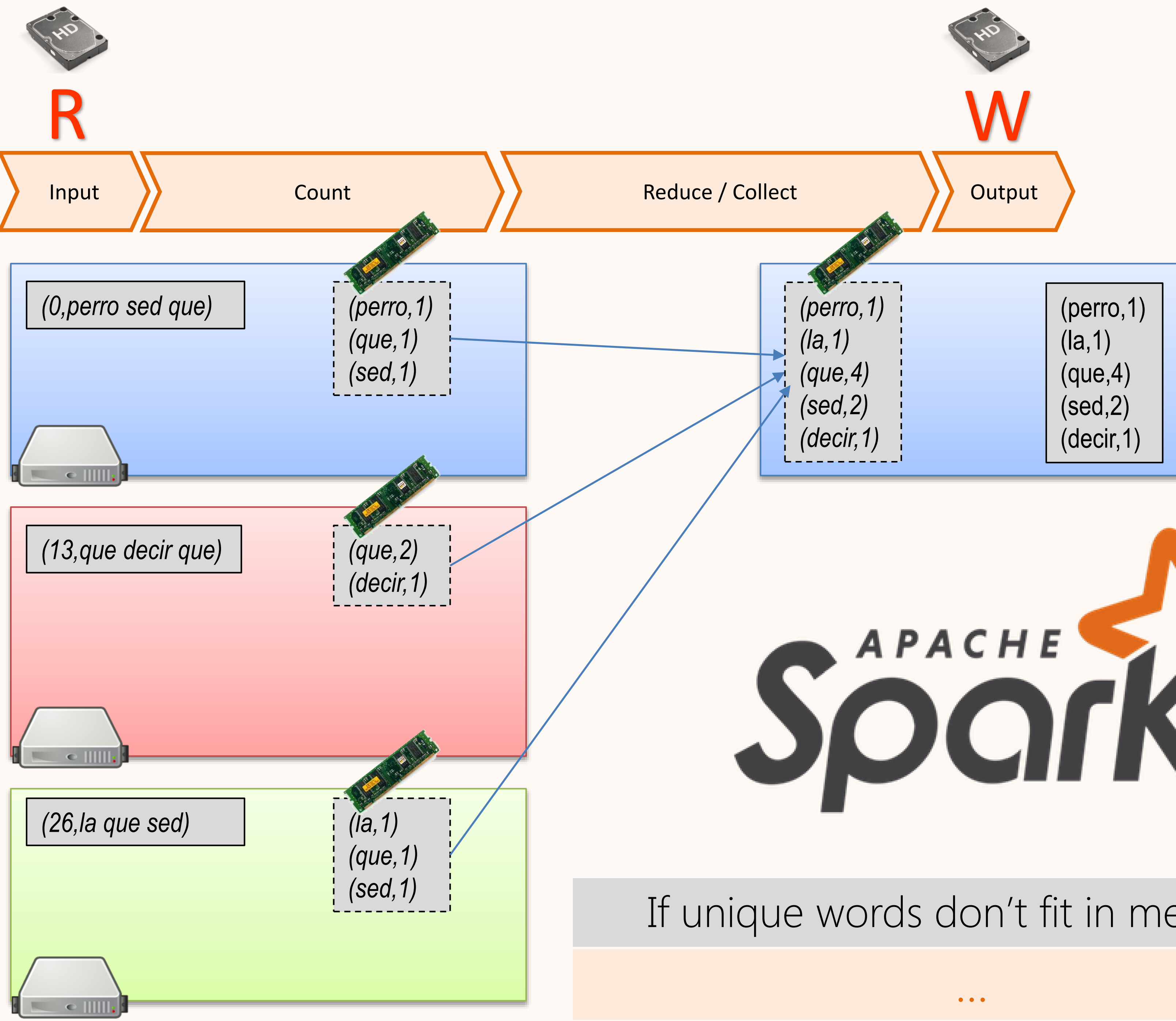
- In memory on one machine: seconds
- On disk on one machine: minutes
- Over MapReduce: minutes

Any alternative to these options?

- In memory on multiple machines: ???

Simple case: Unique words fit in memory



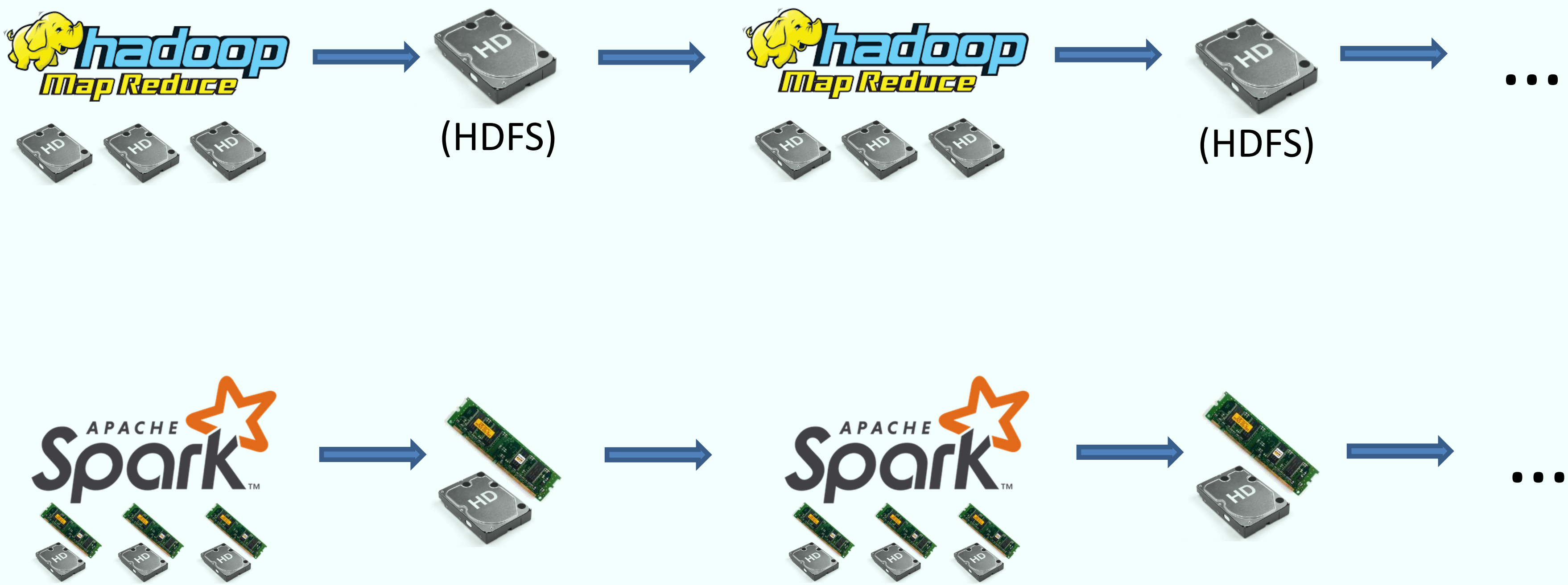


If unique words don't fit in memory?

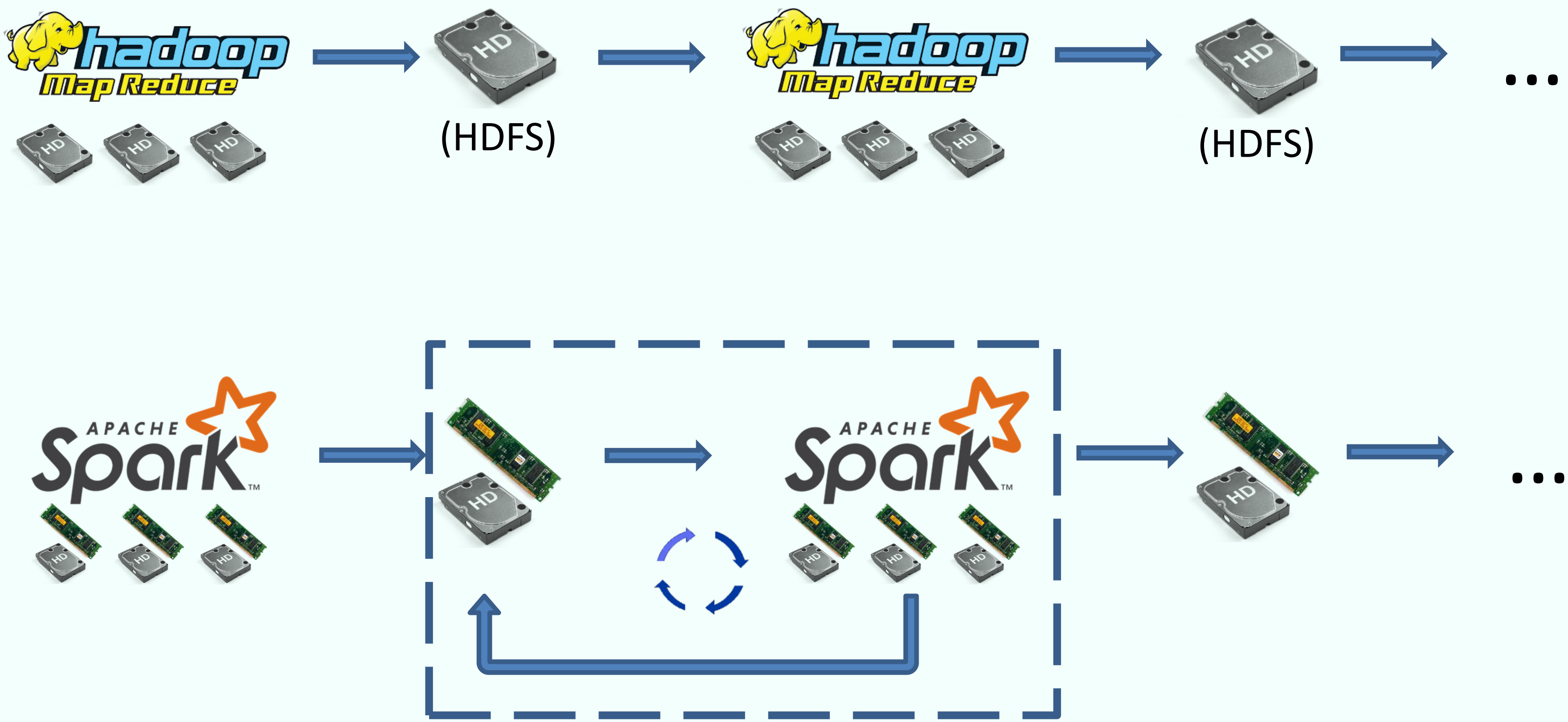
…

APACHE SPARK

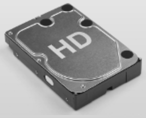
Main idea: Program with main memory



Main idea: Program (recursively) with main memory



Spark storage: Resilient Distributed Dataset

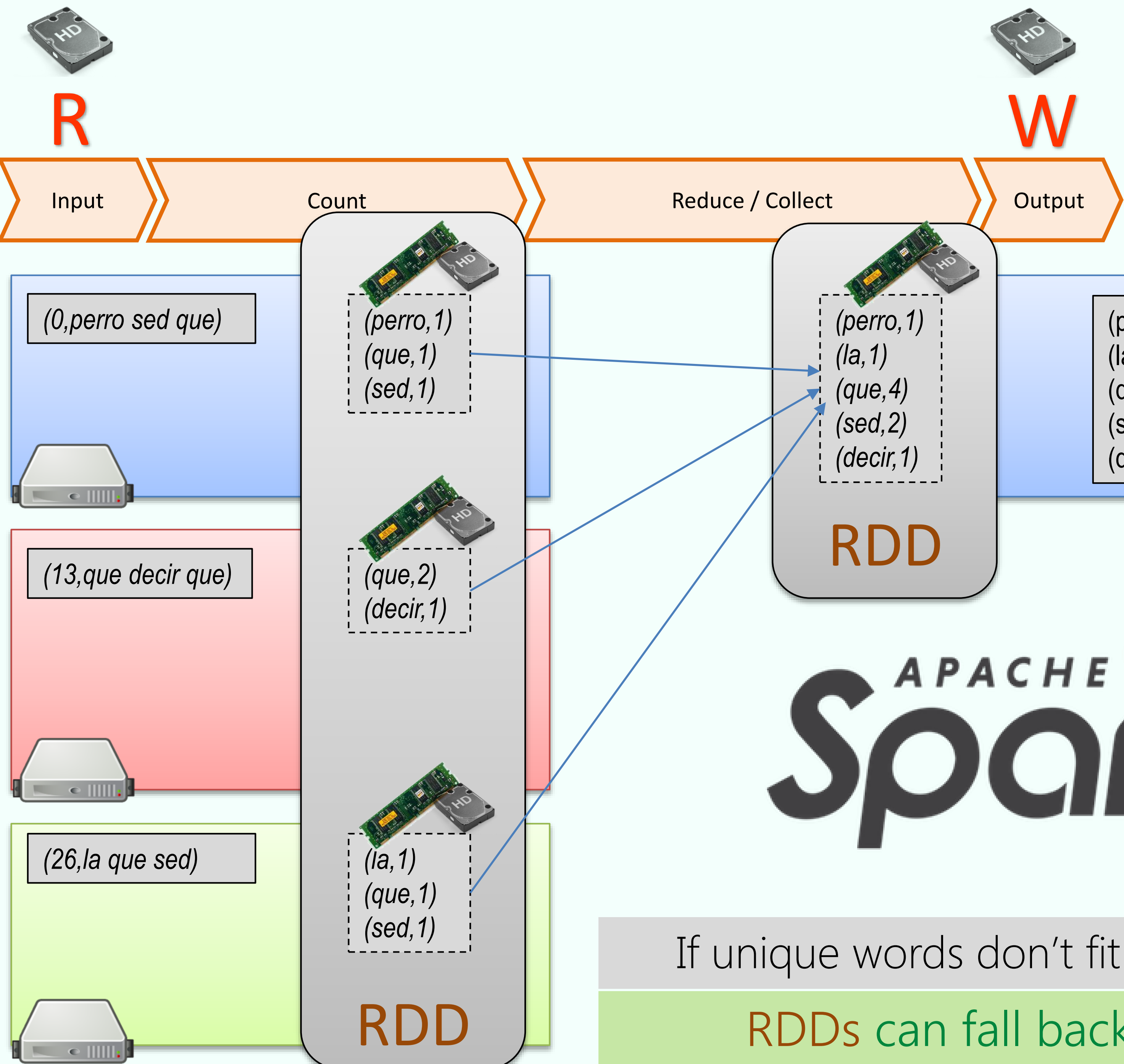


(HDFS)

HDFS

RDD

Like HDFS, RDD abstracts distribution, fault-tolerance, etc., ...
... but RDD can also abstract hard-disk / main-memory



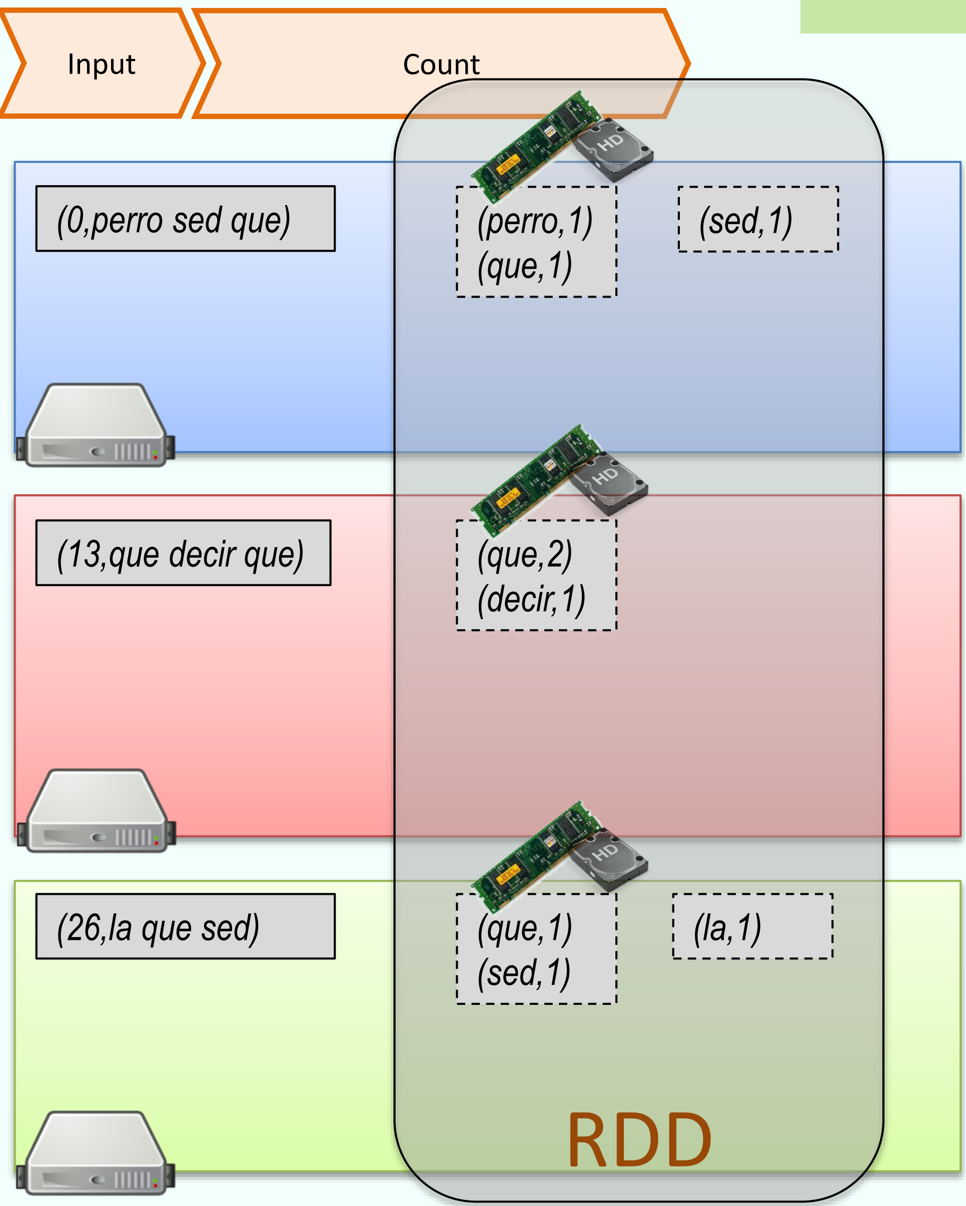
If unique words don't fit in memory?

RDDs can fall back to disk

Spark storage: Resilient Distributed Dataset

- Resilient: Fault-tolerant
- Distributed: Partitioned
- Dataset: Umm, a set of data

RDDs can have multiple virtual partitions on one machine



Types of RDDs in Spark

- `HadoopRDD`
- `FilteredRDD`
- `MappedRDD`
- `PairRDD`
- `ShuffledRDD`
- `UnionRDD`
- `PythonRDD`
- `DoubleRDD`
- `JdbcRDD`
- `JsonRDD`
- `SchemaRDD`
- `VertexRDD`
- `EdgeRDD`
- `CassandraRDD`
- `GeoRDD`
- `EsSpark`

Specific types of RDDs permit specific operations

`PairRDD` of particular importance for M/R style operators

APACHE SPARK: EXAMPLE

Spark: Products by Hour

customer412	1L_Leche	2014-03-31T08:47:57Z	$900
customer412	Nescafe	2014-03-31T08:47:57Z	$2.000
customer412	Nescafe	2014-03-31T08:47:57Z	$2.000
customer413	400g_Zanahoria	2014-03-31T08:48:03Z	$1.240
customer413	El_Mercurio	2014-03-31T08:48:03Z	$3.000
customer413	Gillette_Mach3	2014-03-31T08:48:03Z	$3.000
customer413	Santo_Domingo	2014-03-31T08:48:03Z	$2.450
customer413	Nescafe	2014-03-31T08:48:03Z	$2.000
customer414	Rosas	2014-03-31T08:48:24Z	$7.000
customer414	400g_Zanahoria	2014-03-31T08:48:24Z	$9.230
customer414	Nescafe	2014-03-31T08:48:24Z	$2.000
customer415	1L_Leche	2014-03-31T08:48:35Z	$900
customer415	300g_Frutillas	2014-03-31T08:48:35Z	$830
…			

$(\text{customer,item,time,price})$

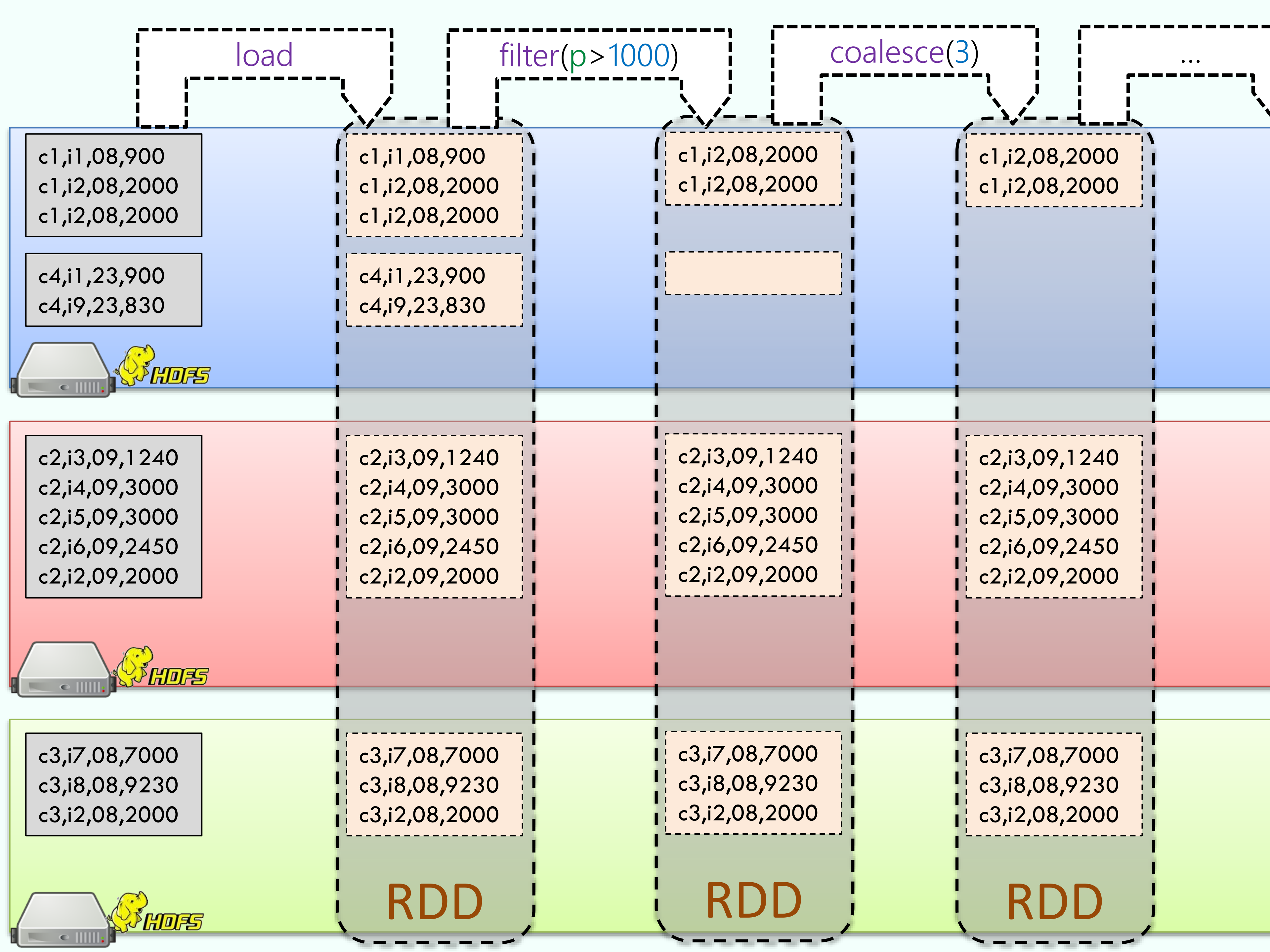
Number of customers buying each premium item per hour of the day

Spark: Products by Hour

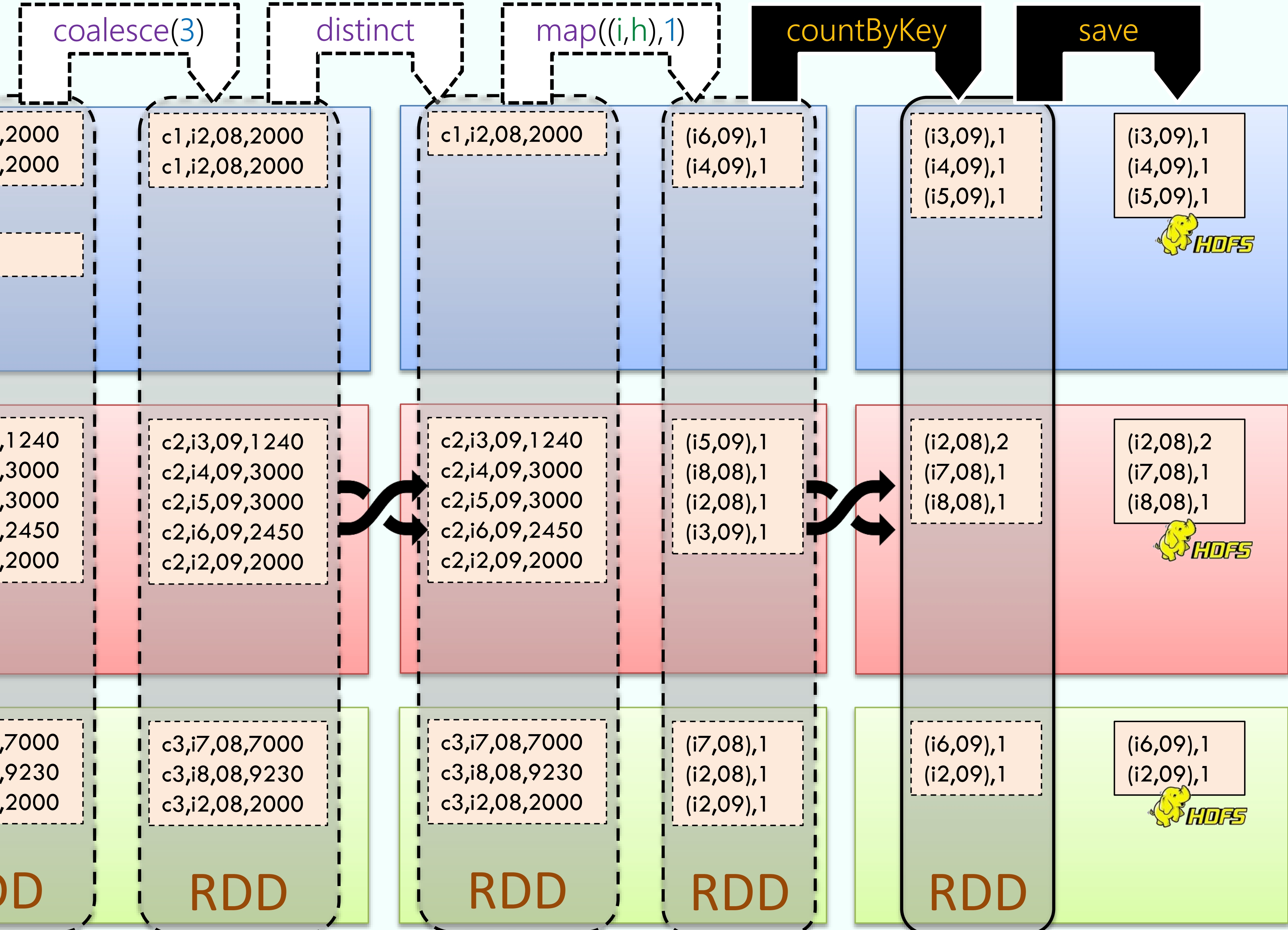
c1	i1	08	900
c1	i2	08	2000
c1	i2	08	2000
c2	i3	09	1240
c2	i4	09	3000
c2	i5	09	3000
c2	i6	09	2450
c2	i2	09	2000
c3	i7	08	7000
c3	i8	08	9230
c3	i2	08	2000
c4	i1	23	900
c4	i9	23	830
…			

(customer,item,hour,price)

Number of customers buying each premium item per hour of the day



load
filter(p>1000)
coalesce(3)
...
c1,i1,08,900
c1,i2,08,2000
c1,i2,08,2000
c4,i1,23,900
c4,i9,23,830
HDFS
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
HDFS
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
HDFS
c1,i1,08,900
c1,i2,08,2000
c1,i2,08,2000
c4,i1,23,900
c4,i9,23,830
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD
c1,i2,08,2000
c1,i2,08,2000
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD
c1,i2,08,2000
c1,i2,08,2000
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD


coalesce(3)
distinct
map((i,h),1)
countByKey
save
c1,i2,08,2000
c1,i2,08,2000
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD
c1,i2,08,2000
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD
(i6,09),1
(i4,09),1
(i5,09),1
(i8,08),1
(i2,08),1
(i3,09),1
(i7,08),1
(i2,08),1
(i2,09),1
RDD
(i3,09),1
(i4,09),1
(i5,09),1
(i2,08),2
(i7,08),1
(i8,08),1
(i6,09),1
(i2,09),1
RDD
(i3,09),1
(i4,09),1
(i5,09),1
HDFS
(i2,08),2
(i7,08),1
(i8,08),1
HDFS
(i6,09),1
(i2,09),1
HDFS

APACHE SPARK: TRANSFORMATIONS & ACTIONS

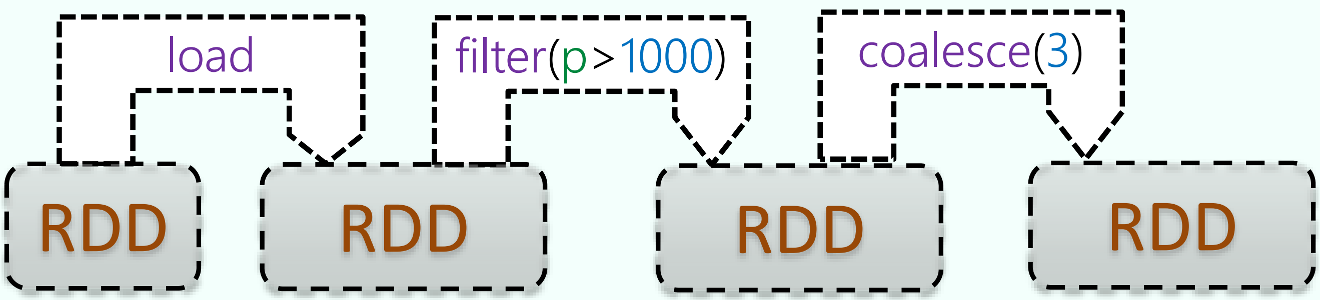
Spark: Transformations vs. Actions

Transformations are run lazily …

… they result in "virtual" RDDs

… they are only run to complete an action

… for example:



… are not run immediately

Transformations

R.transform()

f = function argument
R = RDD argument
. = simple argument

`map(f)`	`intersection(R)`	`cogroup(R)`
`flatMap(f)`	`distinct()`	`cartesian(R)`
`filter(f)`	`groupByKey()`	`pipe(.)`
`mapPartitions(f)`	`reduceByKey(f)`	`coalesce(.)`
`mapPartitionsWithIndex(f)`	`aggregateByKey(f)`	`repartition(.)`
`sample(.,.,.)`	`sortByKey()`	...
`union(R)`	`join(R)`	...

Any guesses why some are underlined?

They require a shuffle

https://spark.apache.org/docs/latest/programming-guide.html#transformations

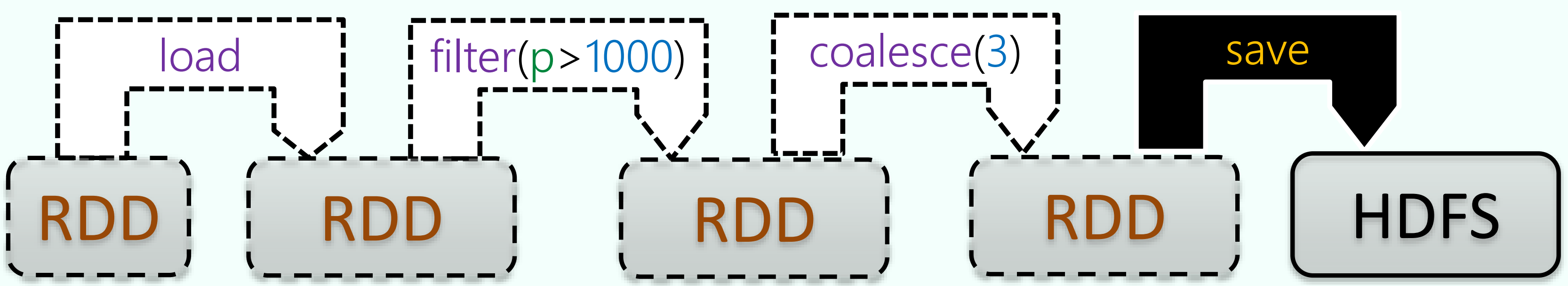
Spark: Transformations vs. Actions

Actions are where things execute ...

... they result in "materialised" RDDs

... all ancestor transformations are run

... for example:



... all steps are now run

... but intermediate RDDs are <u>not</u> kept

Actions

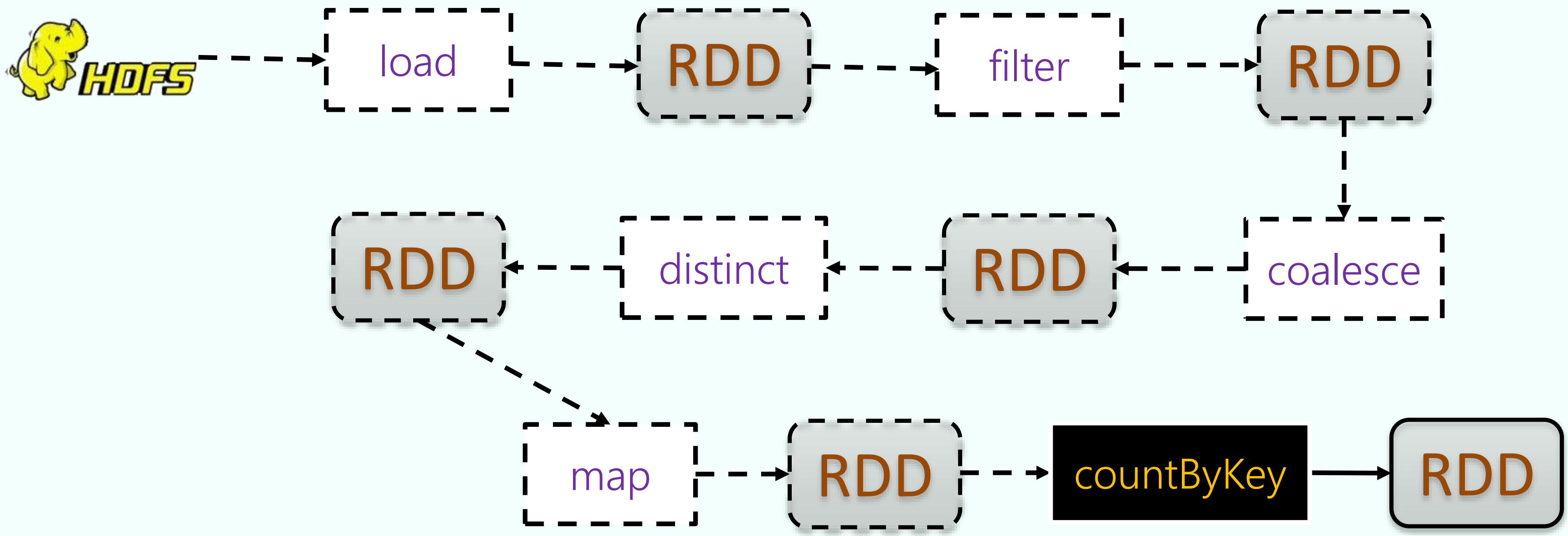
R.action()

f = function argument
. = simple argument

```
reduce(ƒ)              saveToCassandra()
collect()              saveAsTextFile(.)
count()                saveAsSequenceFile(.)
first()                saveAsObjectFile(.)
take(.)                countByKey()
takeSample(.,.)        foreach(ƒ)
takeOrdered(.)         ...
```

https://spark.apache.org/docs/latest/programming-guide.html#actions

Spark: Directed Acyclic Graph (DAG)



Spark: Products by Hour

receipts.txt

c1	i1	08	900
c1	i2	08	2000
c1	i2	08	2000
c2	i3	09	1240
c2	i4	09	3000
c2	i5	09	3000
c2	i6	09	2450
c2	i2	09	2000
c3	i7	08	7000
c3	i8	08	9230
c3	i2	08	2000
c4	i1	23	900
c4	i9	23	830
…			

(customer,item,hour,price)

customer.txt

c1	female	40
c2	male	24
c3	female	73
c4	female	21
…		

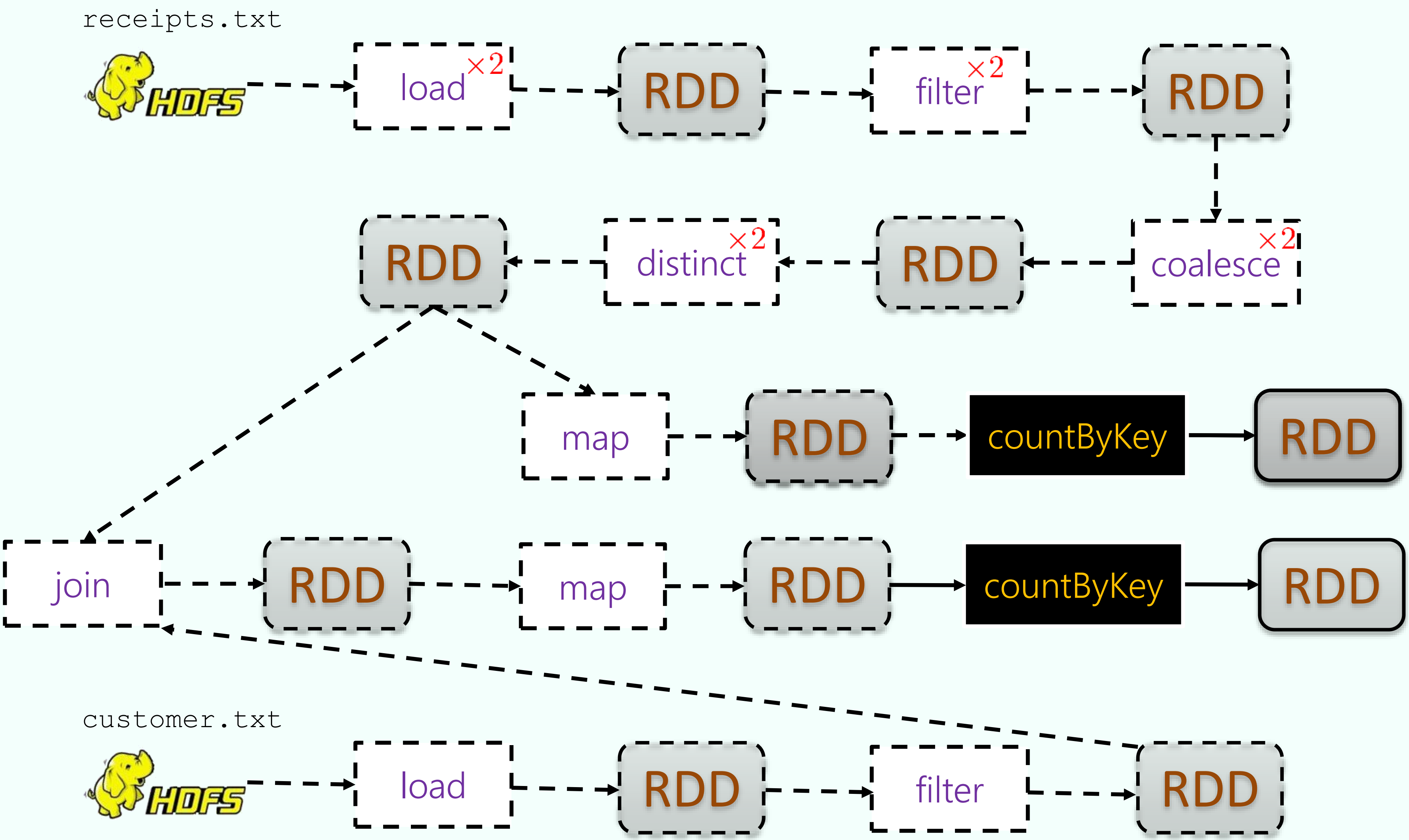
(customer,gender)

Also … № of females older than 30 buying each premium item per hour of the day

Spark: Directed Acyclic Graph (DAG)

Problem? ?

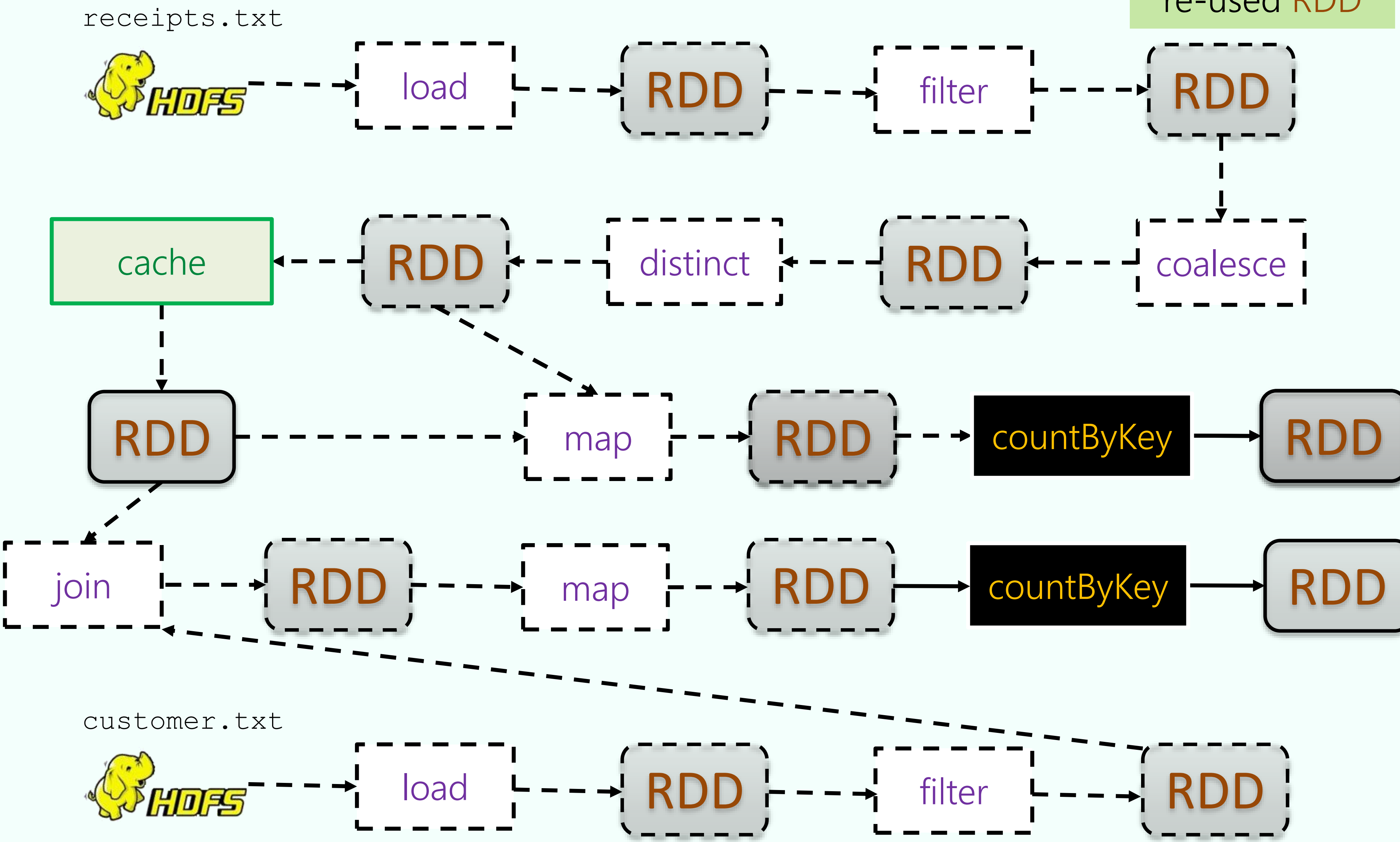
Solution? ?



Spark: Directed Acyclic Graph (DAG)

Problem?	?
Solution?	?

Materialise re-used RDD



Spark: Directed Acyclic Graph (DAG)

- Cache (aka. persist)
  - Is lazy (still needs an action to run)
  - Can use memory or disk (default memory only)

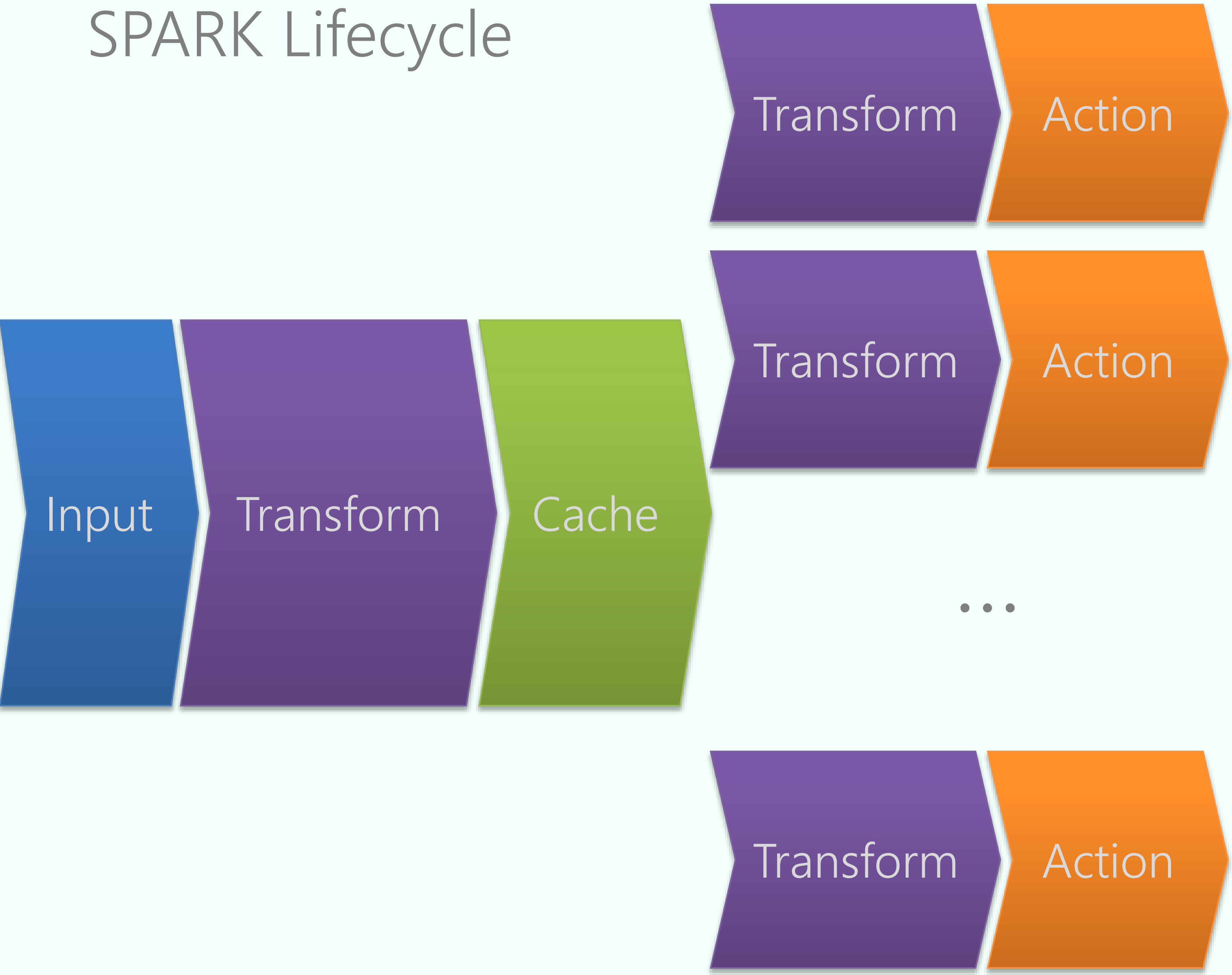
cache

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

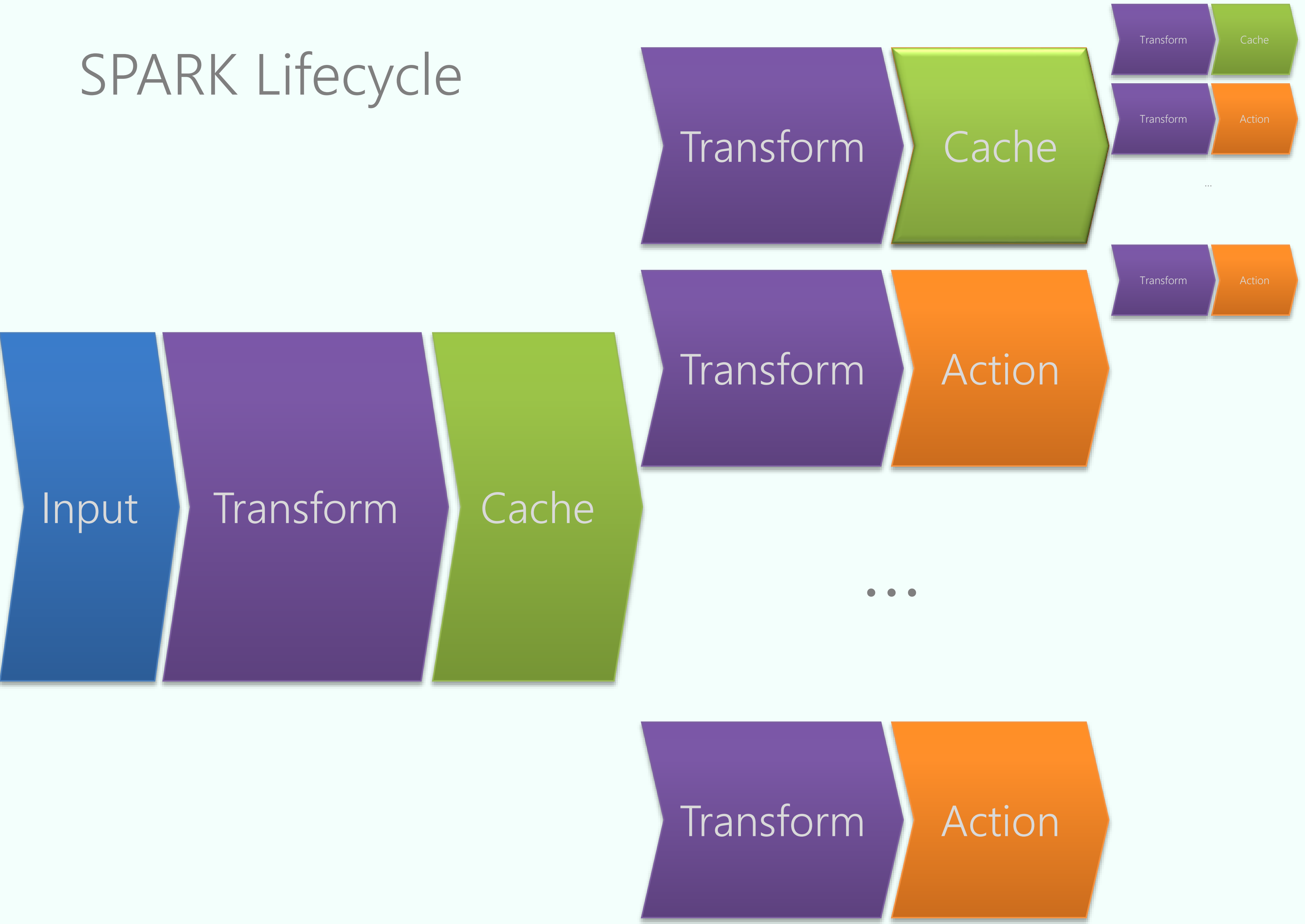
https://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence

APACHE SPARK: CORE SUMMARY

SPARK Lifecycle



SPARK Lifecycle



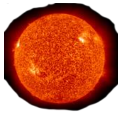
SPARK Lifecycle

- Input RDDs
- Transform RDDs
- Cache (aka. persist) reused RDDs
- Perform an Action (launching execution)
- Output to file/database/local terminal

SPARK: BEYOND THE CORE

Hadoop vs. Spark

hadoop vs Spark

Hadoop	Spark
hadoop Map Reduce	Spark 
YARN	Mesos
hadoop HDFS	Tachyon

Hadoop vs. Spark: SQL, ML, Streams, …

hadoop vs Spark

hadoop MapReduce	Spark
YARN	Mesos
hadoop HDFS	Tachyon
HIVE	Spark SQL
mahout	Spark MLlib
STORM	Spark Streaming
APACHE GIRAPH	Spark GraphX

Spark can use the disk

Apache Spark the fastest open source engine for sorting a petabyte

		Spark 100 TB *	**Spark 1 PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

A BROADER PERSPECTIVE

“Data Scientist” Job Postings (2016)

Here are the top 10 in-demand skills for data scientists:

Skills	Job skill appears in	% of jobs with skill
SQL	1987	56%
Hadoop	1713	49%
Python	1367	39%
Java	1287	36%
R	1120	32%
Hive	1099	31%
Mapreduce	768	22%
NoSQL	657	18%
Pig	561	16%
SAS	560	16%

http://visit.crowdflower.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport_2016.pdf

Questions?