

CC5212-1

PROCESAMIENTO MASIVO DE DATOS

OTOÑO 2023

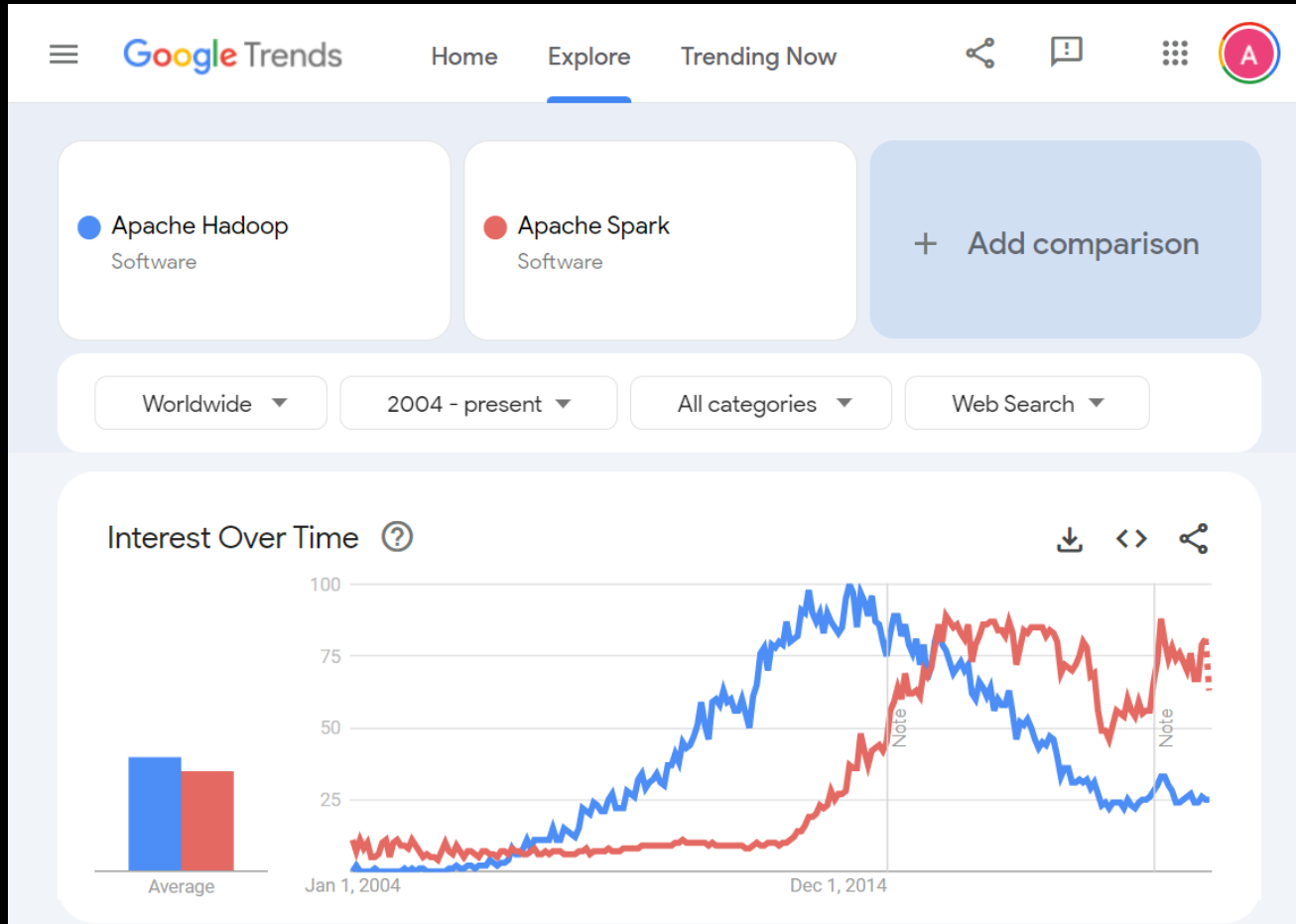
Lecture 5

Apache Spark (Core)

Aidan Hogan

aidhog@gmail.com

Spark vs. Hadoop

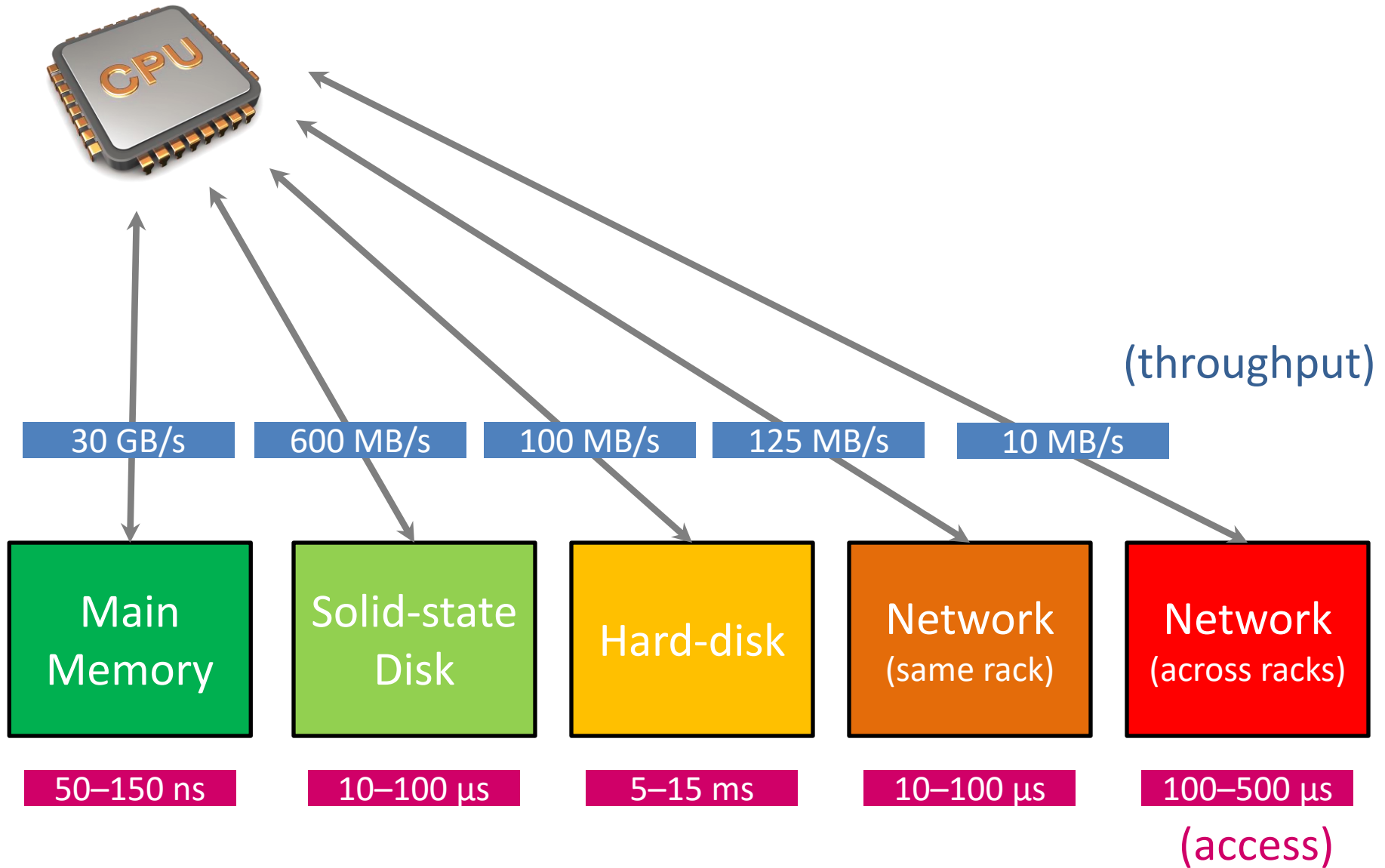


What is the main weakness of Hadoop?

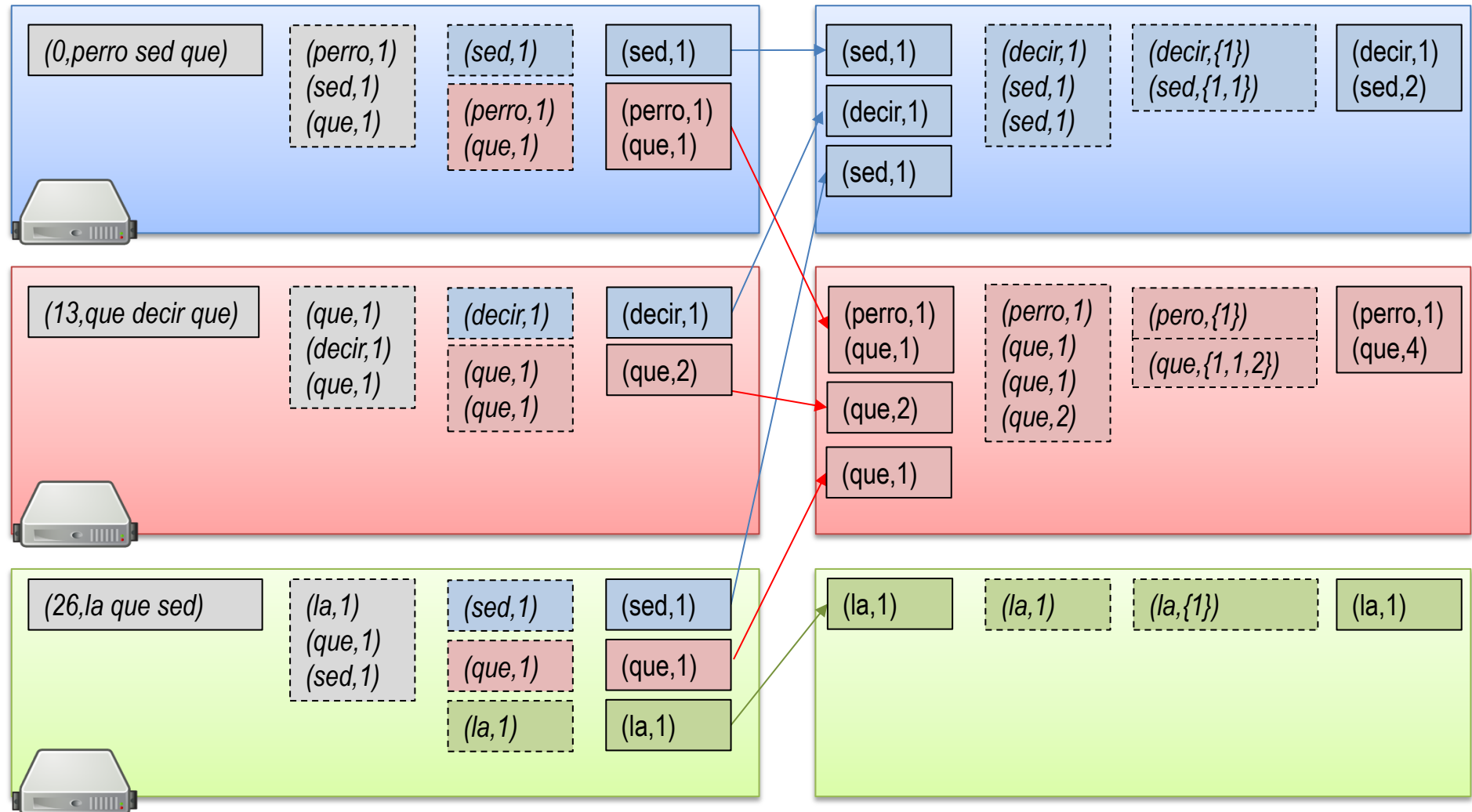
Let's see ...



Data Transport Costs



MapReduce/Hadoop

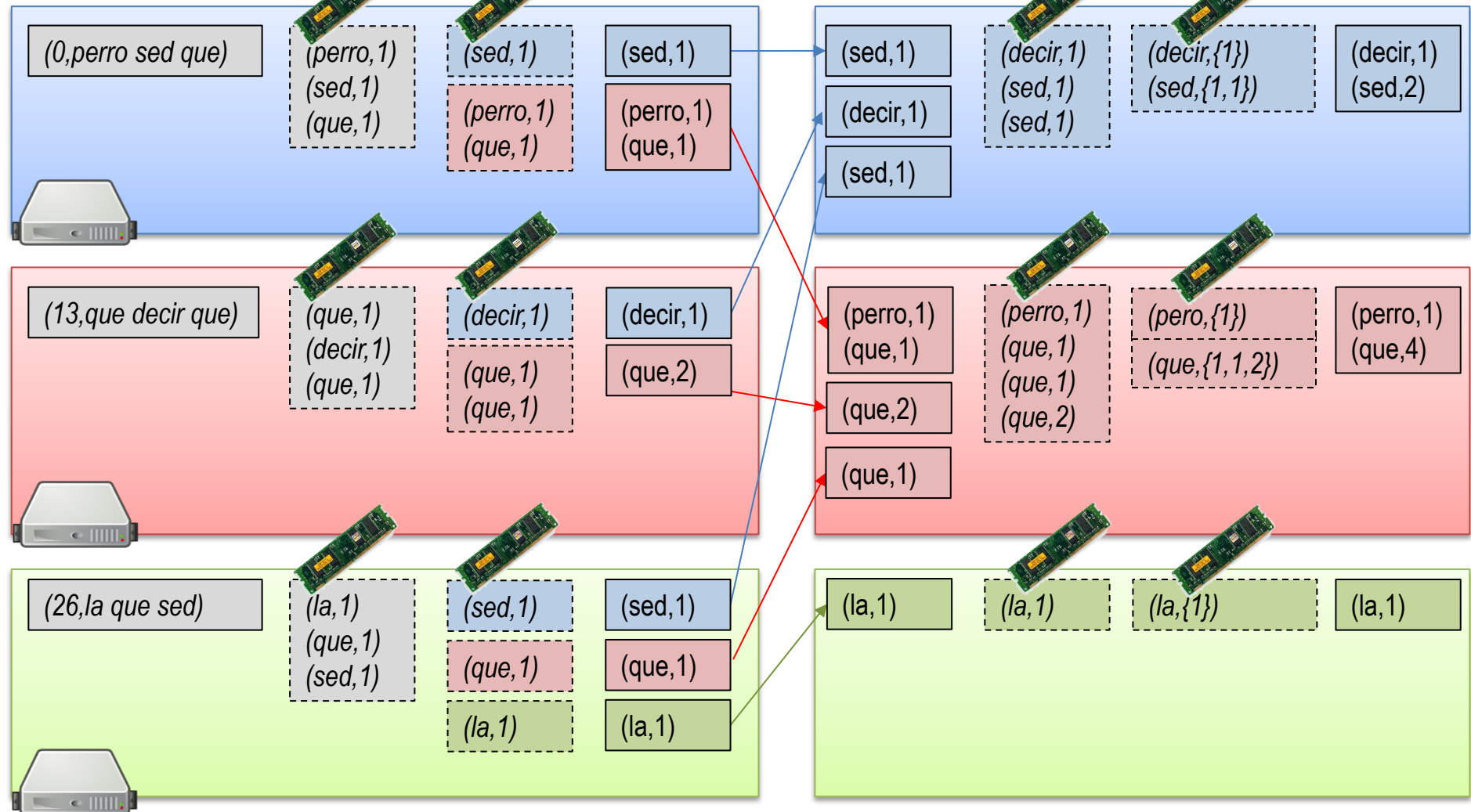




R

W R | W R

W





R

Input

Map

Partition / [Sort]

Combine

Shuffle

Merge Sort

Reduce

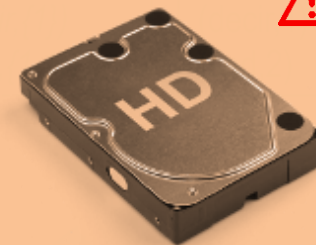
Output

W R | W R

W



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



(HDFS)



(HDFS)



(perro, 1)
(que, 4)
...

(26, la que sed)

(la, 1)
(que, 1)
(sed, 1)

(sed, 1)
(que, 1)
(la, 1)

(sed, 1)
(que, 1)
(la, 1)

(la, 1)

(la, 1)

(la, 1)

(la, 1)



R



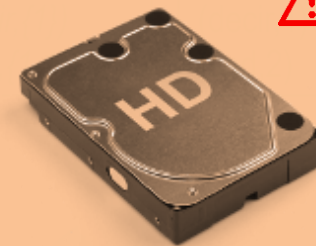
W R | W R



W



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



We saw this already counting words ...

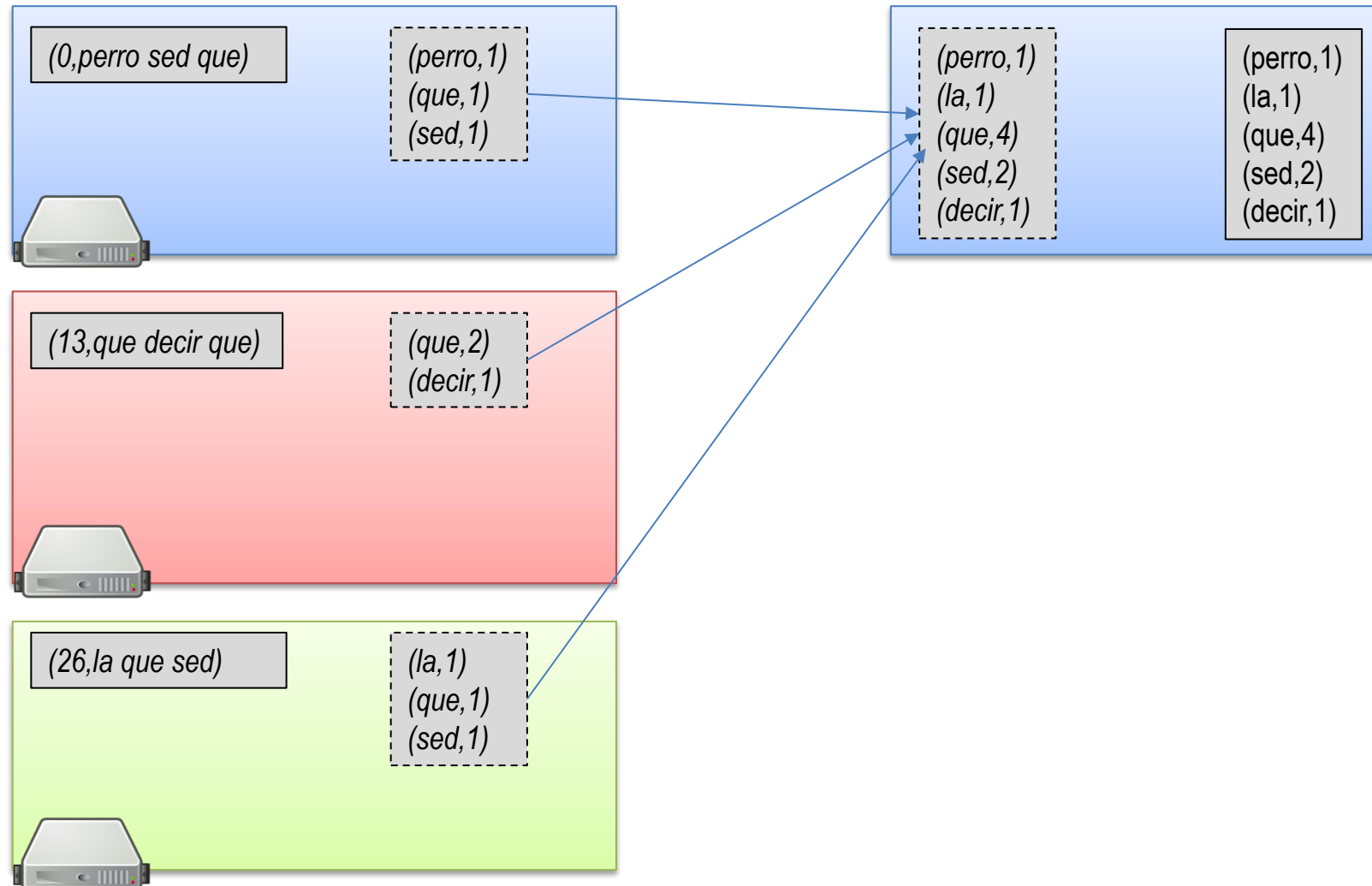
- In memory on one machine: seconds
- On disk on one machine: minutes
- Over MapReduce: minutes

Any alternative to these options?



- In memory on multiple machines: ???

Simple case: Unique words fit in memory





R



W



(0,perro sed que)

(perro, 1)
(que, 1)
(sed, 1)

(13,que decir que)

(que, 2)
(decir, 1)

(26,la que sed)

(la, 1)
(que, 1)
(sed, 1)

(perro, 1)
(la, 1)
(que, 4)
(sed, 2)
(decir, 1)

(perro, 1)
(la, 1)
(que, 4)
(sed, 2)
(decir, 1)

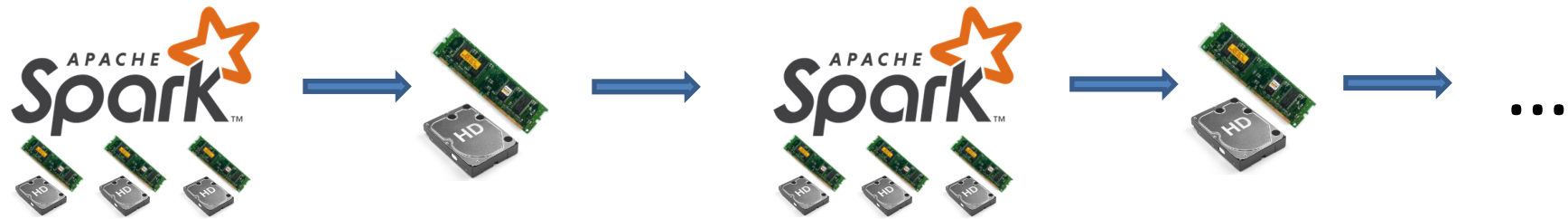


If unique words don't fit in memory?

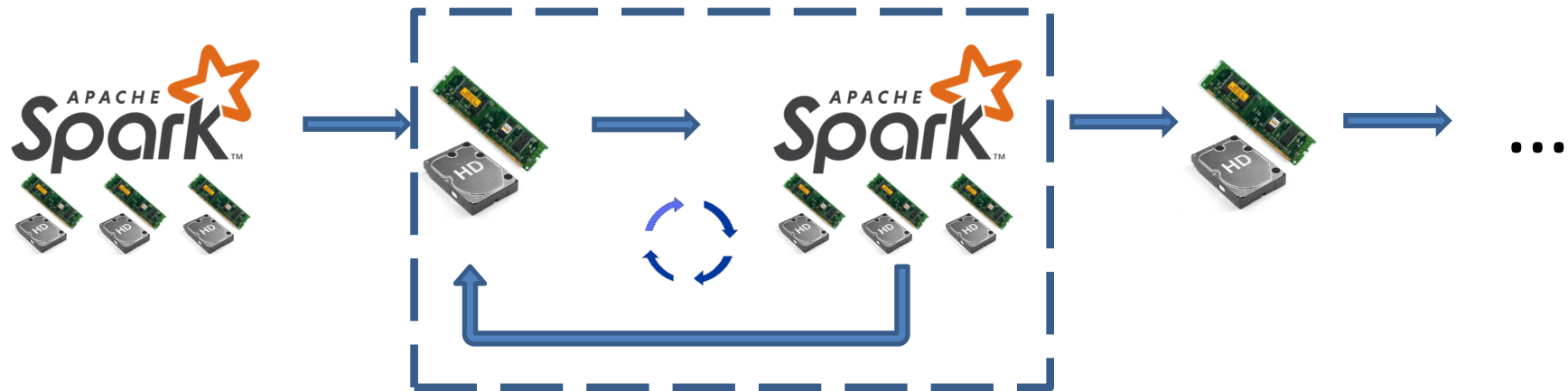
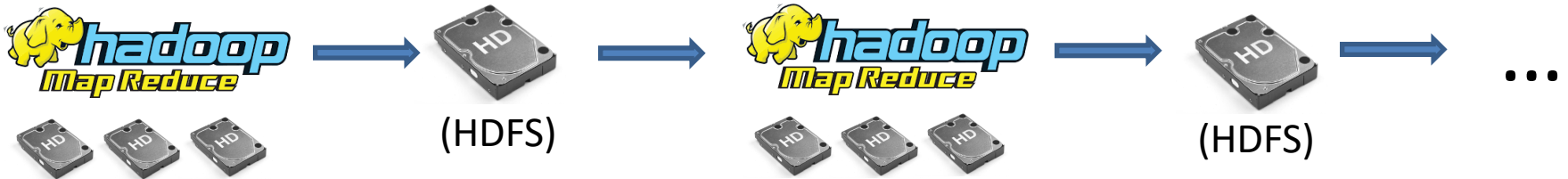
...

APACHE SPARK

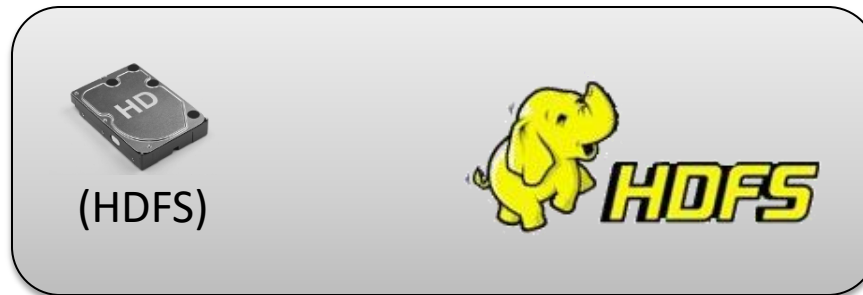
Main idea: Program with main memory



Main idea: Program (recursively) with main memory



Spark storage: Resilient Distributed Dataset



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



R



W



(0,perro sed que)

(perro, 1)
(que, 1)
(sed, 1)

(13,que decir que)

(que, 2)
(decir, 1)

(26,la que sed)

(la, 1)
(que, 1)
(sed, 1)

(perro, 1)
(la, 1)
(que, 4)
(sed, 2)
(decir, 1)

RDD

(perro, 1)
(la, 1)
(que, 4)
(sed, 2)
(decir, 1)



If unique words don't fit in memory?



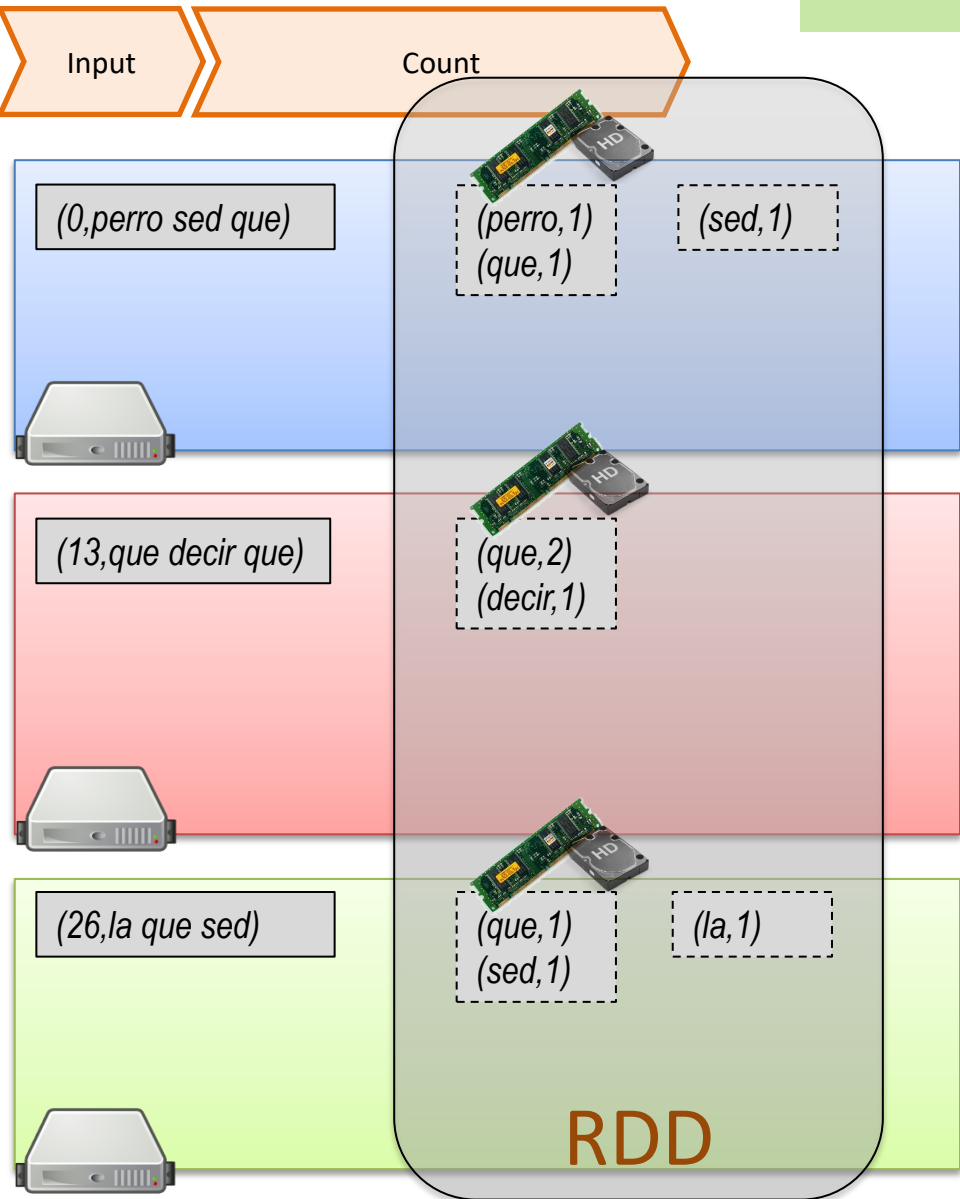
RDDs can fall back to disk

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

... (customer,item,time,price)



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)



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

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

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

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

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

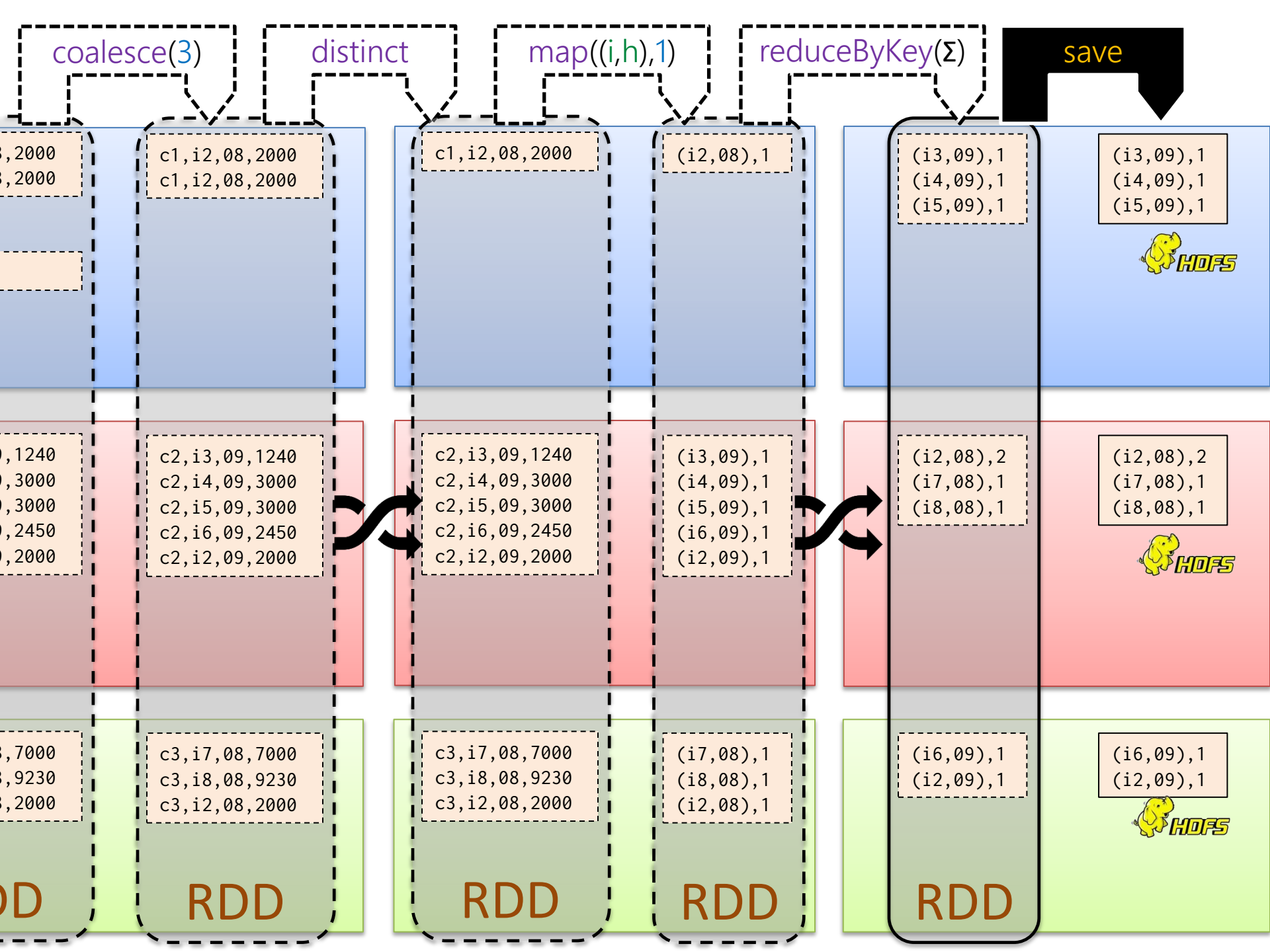
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000

RDD

RDD

RDD





APACHE SPARK:

TRANSFORMATIONS & ACTIONS

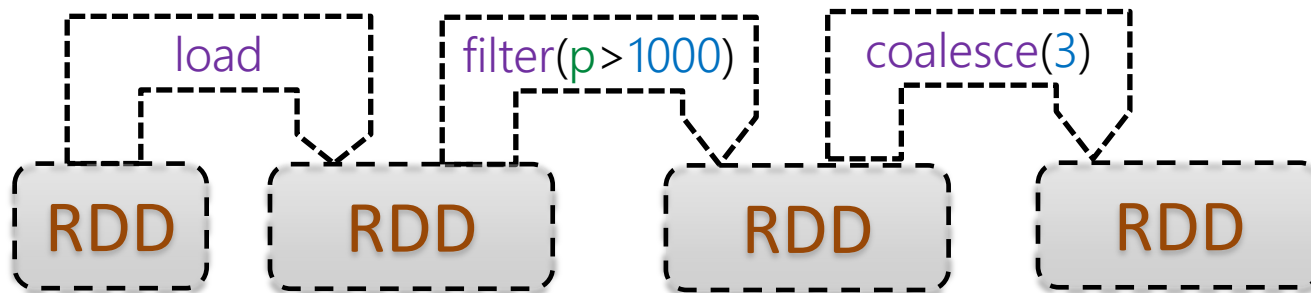
Spark: Transformations vs. Actions

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

S = RDD argument

. = simple argument

R.map(*f*)

R.intersection(*S*)

R.cogroup(*S*)

R.flatMap(*f*)

R.distinct()

R.cartesian(*S*)

R.filter(*f*)

R.groupByKey()

R.pipe(*.*)

R.mapPartitions(*f*)

R.reduceByKey(*f*)

R.coalesce(*.*)

R.mapPartitionsWithIndex(*f*)

R.aggregateByKey(*f*)

R.repartition(*.*)

R.sample(*.*, *.*, *.*)

R.sortByKey()

...


R.union(*S*)

R.join(*S*)

...

Any guesses why some are underlined?



They require a shuffle 

Transformations

R.transform()

f = function argument

S = RDD argument

x = simple argument

Transformation	Description	Example / Note
<code>R.map(f)</code>	Maps an element from R to one output value	$f : (a, b, c) \mapsto (a, c)$
<code>R.flatMap(f)</code>	Maps an element from R to zero-or more output values	$f : (a, b, c) \mapsto \{(a, b), (a, c)\}$
<code>R.filter(f)</code>	Maps every element of R that satisfies f to the output	$f : a > b$
<code>R.mapPartitions(f)</code>	Maps all elements of R to the output, calling f once per partition	$f : R \rightarrow \pi_{a,c}(R)$
<code>R.mapPartitionsWithIndex(f)</code>	Like <code>mapPartitions</code> but f has as an argument the index of the partition	$f(i) : R \rightarrow \pi_{a,c,i}(R)$
<code>R.sample(w, f, s)</code>	Takes a sample of R	w : with replacement f : fraction s : seed

Transformations

R.transform()

f = function argument

S = RDD argument

x = simple argument

R.union(*S*)

$R \cup S$

R.intersection(*S*)

$R \cap S$

R.distinct()

Remove duplicates

Transformations

Requiring a PairRDD ...

R.transform()

f = function argument

S = RDD argument

x = simple argument

R.[groupByKey\(\)](#)

Groups values by key

R.[reduceByKey\(\$f\$ \)](#)

Groups values by key and calls f to combine and reduce values with the same key

$f : (a, b) \mapsto a + b$

R.[aggregateByKey\(\$c, fc, fr\$ \)](#)

Groups values by key using c as an initial value, fc as a combiner and fr as a reducer

c : initial value
 fc : combiner
 fr : reducer

R.[sortByKey\(\[\$a\$ \]\)](#)

Order by key

a : true ascending
false descending

R.[join\(\$S\$ \)](#)

$R \bowtie S$, join by key

Also: [leftOuterJoin](#),
[rightOuterJoin](#), and
[fullOuterJoin](#)

R.[cogroup\(\$S\$ \)](#)

Group values by key in R and S together

Transformations

R.transform()

f = argumento de función

S = argumento de RDD

x = argumento simple

R.cartesian(*S*)

$R \times S$, Cartesian (cross)
product

R.pipe(*c*)

Creates a "pipe" from stdin to process data using the command *c* such as `grep`, `awk`, `perl`, etc. The result is an RDD with the output

R.coalesce(*n*)

Merges various partitions into at most *n* partitions

R.repartition(*n*)

Partitions the data again (often to rebalance) creating (at most) *n* partitions

...

...

...

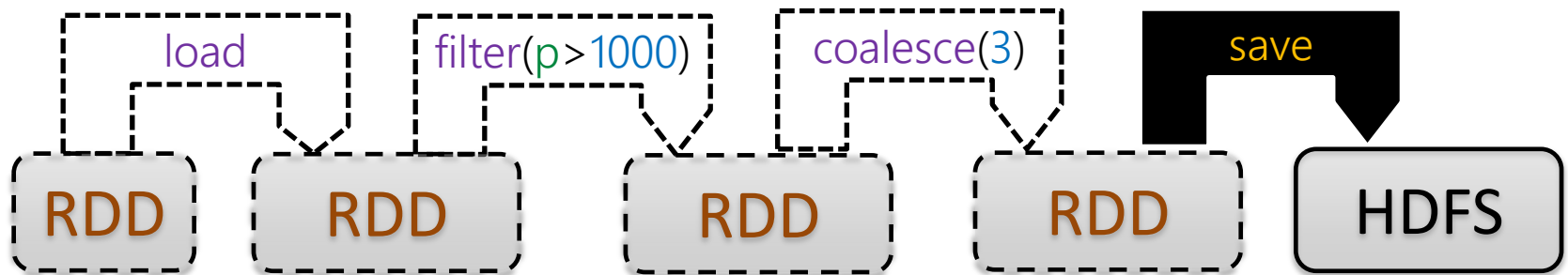
Spark: Transformations vs. Actions

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 not kept

Actions

R.action()

f = function argument

. = simple argument

R.reduce(*f*)

R.collect()

R.count()

R.first()

R.take(.)

R.takeSample(.,.)

R.takeOrdered(.)

R.saveToCassandra()

R.saveAsTextFile(.)

R.saveAsSequenceFile(.)

R.saveAsObjectFile(.)

R.countByKey()

R.foreach(*f*)

...

Actions

R.action()

f = function argument

x = simple argument

Actions	Description	Example / Notes
<code>R.reduce(f)</code>	Reduces <u>all</u> data to one value/tuple (<u>not by key!</u>)	$f : (a,b) \mapsto a + b$
<code>R.collect()</code>	Loads R as an array into the local application	
<code>R.count()</code>	Counts the elements of R	
<code>R.first()</code>	Get the first tuple of R	
<code>R.take(n)</code>	Loads an array of n values from R into the local application	
<code>R.takeSample(w,n,s)</code>	Loads a sample of n values from R into the local application	w : with replacement n : number s : seed

Actions

R.action()

f = function argument
x = simple argument

<code>R.saveAsTextFile(d)</code>	Save the data to the file system as a plain text file	d: directory
<code>R.saveAsSequenceFile(d)</code>	Save the data to the file system with the format <code>SequenceFile</code> for Hadoop	d: directory
<code>R.saveAsObjectFile(d)</code>	Save the data to the file system using native Java serialisation	d: directory
<code>R.countByKey()</code>	Count the values for each key returning the answer locally	Only for <code>PairRDD</code>
<code>R.foreach(f)</code>	Execute the function <i>f</i> for every element of R , typically to interact with something external	<i>f</i> : <code>println(r)</code>

APACHE SPARK:

TRANSFORMATIONS FOR `PairRDD`

Transformations

Requiring a PairRDD ...

R.transform()

f = function argument

S = RDD argument

x = simple argument

R.groupByKey()

Groups values by key

R.reduceByKey(f)

Groups values by key and calls f to combine and reduce values with the same key

$f : (a, b) \mapsto a + b$

R.aggregateByKey(c, fc, fr)

Groups values by key using c as an initial value, fc as a combiner and fr as a reducer

c : initial value
 fc : combiner
 fr : reducer

R.sortByKey([a])

Order by key

a : true ascending
false descending

R.join(S)

$R \bowtie S$, join by key

Also: leftOuterJoin,
rightOuterJoin, and
fullOuterJoin

How are the above different?

R.cogroup(S)

Group values by key in R y S together



Transformations

R.transform()

f = function argument

S = RDD argument

x = simple argument

<code>R.groupByKey()</code>	Groups values by key	
<code>R.reduceByKey(f)</code>	Groups values by key and calls f to combine and reduce values with the same key	$f : (a,b) \mapsto a + b$
<code>R.aggregateByKey(c, fc, fr)</code>	Groups values by key using c as an initial value, fc as a combiner and fr as a reducer	c : initial value fc : combiner fr : reducer

To sum all values for each key in **R**?



(1) `R.groupByKey().map((k, {v1, ..., vn}) \mapsto (k, sum({v1, ..., vn})))`;

(2) `R.reduceByKey((u, v) \mapsto u + v)`;

(3) `R.aggregateByKey(0, (u, v) \mapsto u + v, (u, v) \mapsto u + v)`;

(2) uses a combiner! (3) does the same, but is less concise. So (2) is best!

Transformations

R.transform()

f = function argument

S = RDD argument

x = simple argument

<code>R.groupByKey()</code>	Groups values by key	
<code>R.reduceByKey(f)</code>	Groups values by key and calls f to combine and reduce values with the same key	$f : (a,b) \mapsto a + b$
<code>R.aggregateByKey(c,fc,fr)</code>	Groups values by key using c as an initial value, fc as a combiner and fr as a reducer	c : initial value fc : combiner fr : reducer

To average all values for each key in R ?



(1) `R.groupByKey().map((k,{v1,...,vn}) \mapsto (k,avg({v1,...,vn})))`;

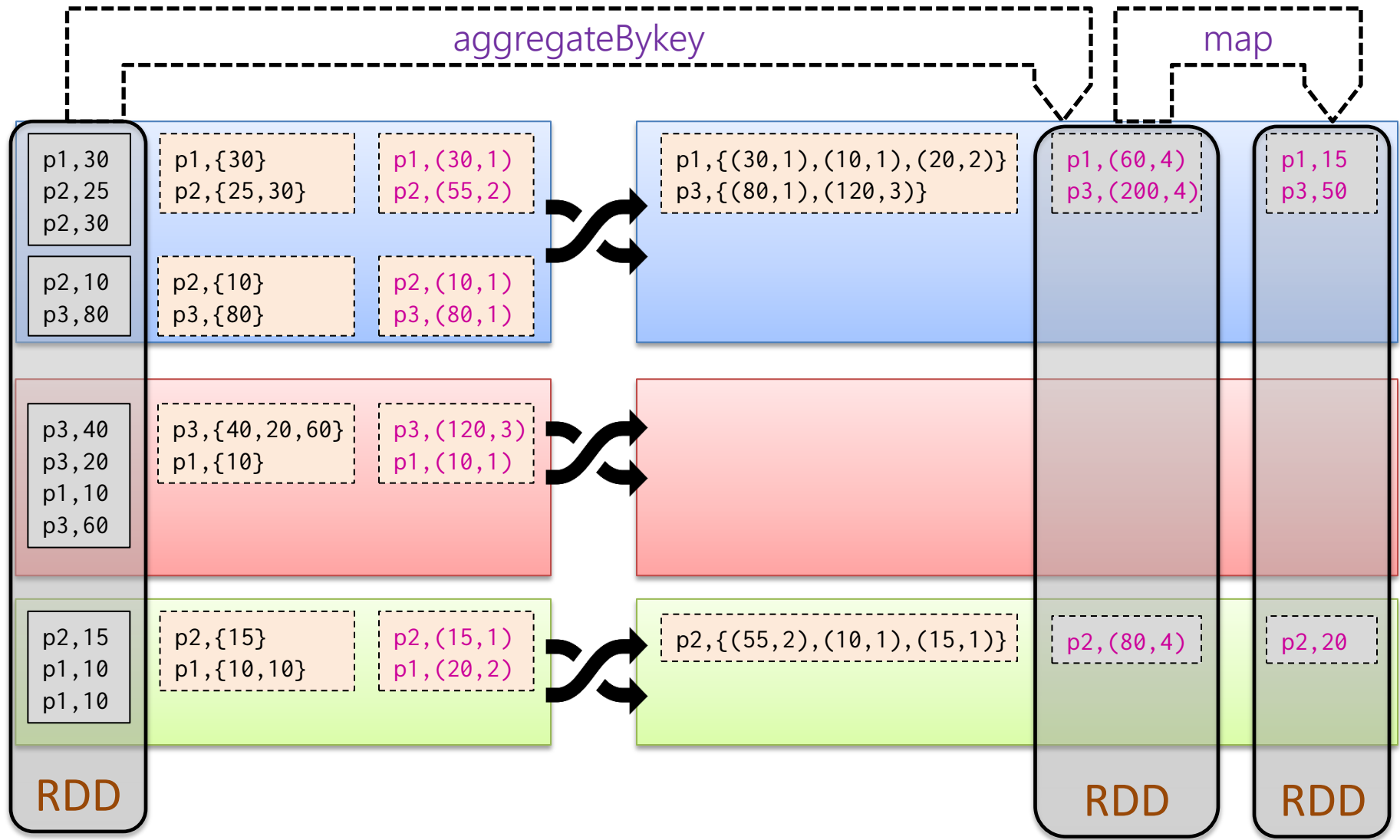
(2) `R1 = R.reduceByKey((u,v) \mapsto u + v)`;

`R2 = R.map((k,v) \mapsto (k,1)).reduceByKey((u,v) \mapsto u + v)`;

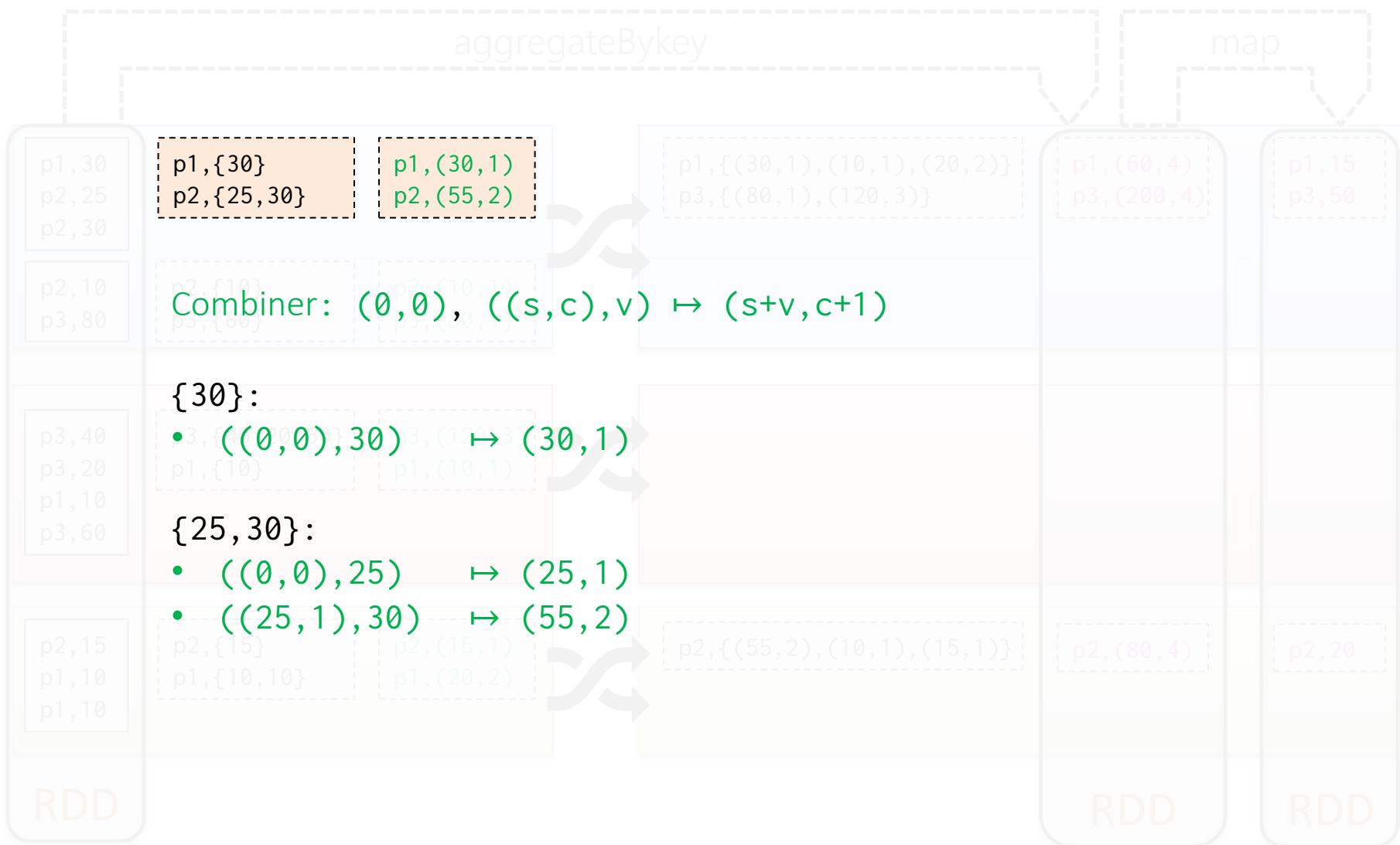
`R3 = R1.join(R2).map((k,(s,c)) \mapsto (k,s/c))`;

(3) `R.aggregateByKey((0,0), ((s,c),v) \mapsto (s+v,c+1),
((s1,c1),(s2,c2)) \mapsto (s1+s2,c1+c2))
.map((k,(s,c)) \mapsto (k,s/c))`;

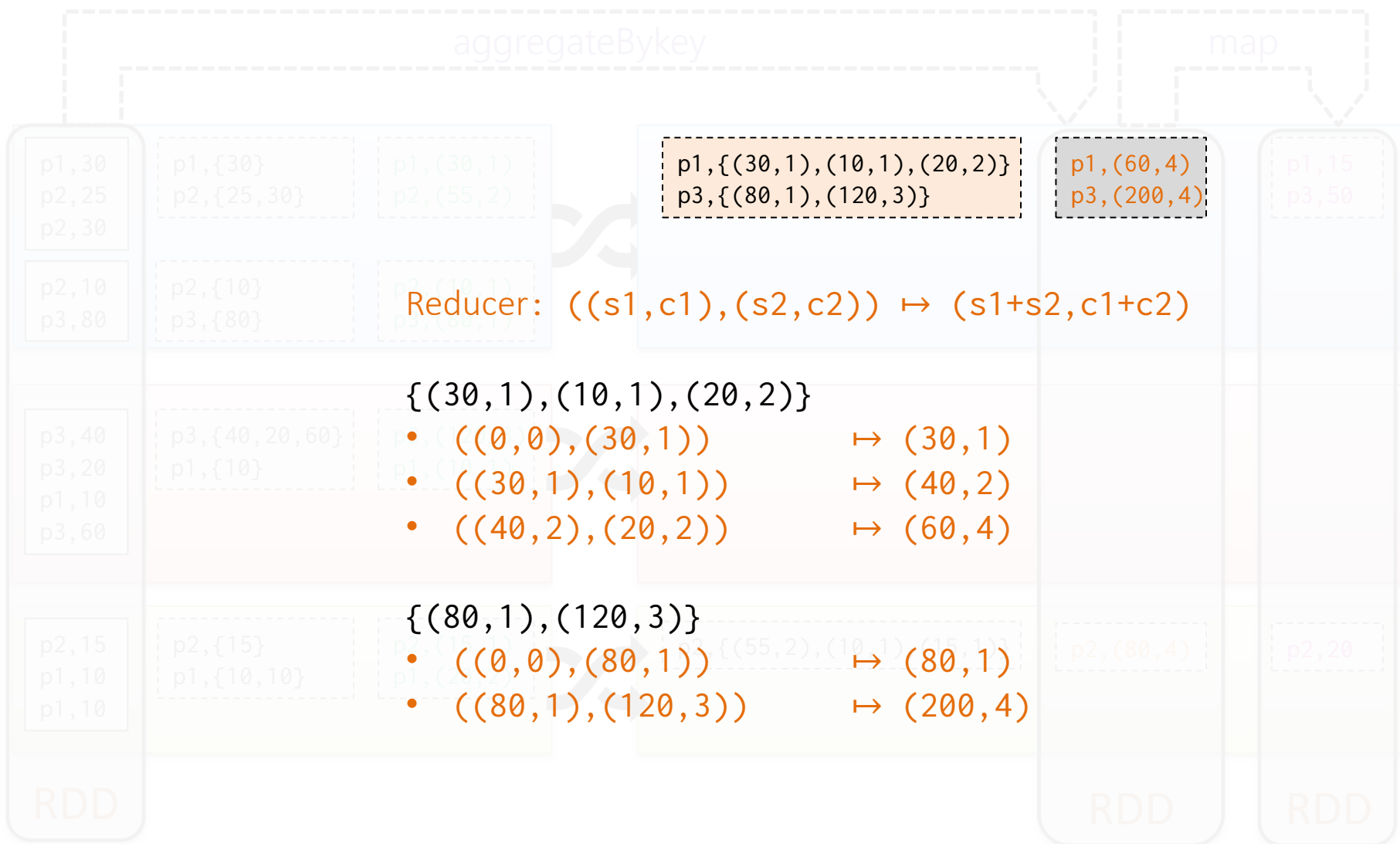
(3) has a combiner and only needs one shuffle. So (3) is best!



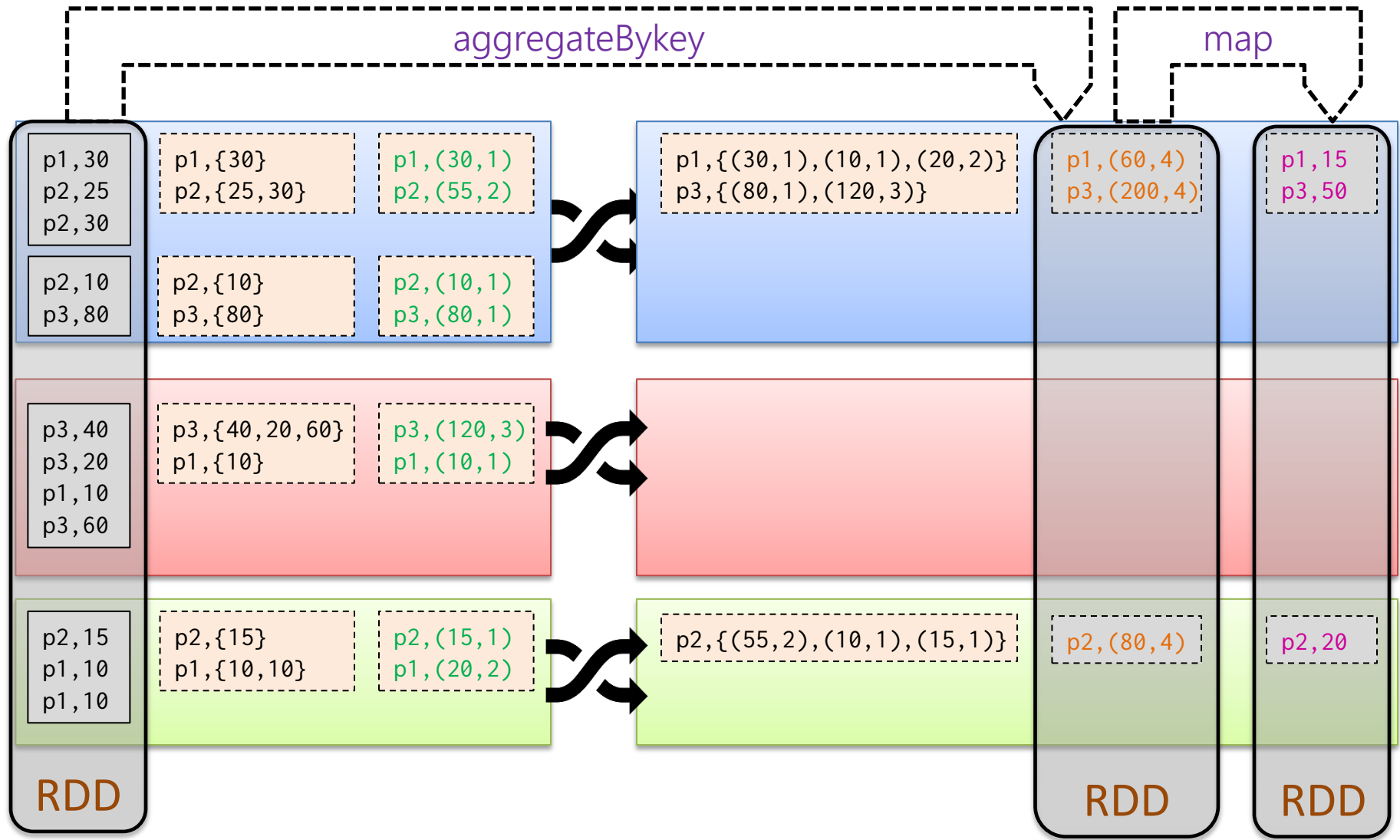
```
(3) R.aggregateByKey((0,0), ((s,c),v) ↦ (s+v,c+1),
                    ((s1,c1),(s2,c2)) ↦ (s1+s2,c1+c2))
                    .map((k,(s,c)) ↦ (k,s/c));
```



```
(3) R.aggregateByKey((0, 0), ((s, c), v) ↦ (s+v, c+1),
                    ((s1, c1), (s2, c2)) ↦ (s1+s2, c1+c2))
    .map((k, (s, c)) ↦ (k, s/c));
```



```
(3) R.aggregateByKey((0,0), ((s,c),v) -> (s+v,c+1),
                    ((s1,c1),(s2,c2)) -> (s1+s2,c1+c2))
    .map((k,(s,c)) -> (k,s/c));
```

```
(3) R.aggregateByKey((0,0), ((s,c),v) ↦ (s+v,c+1),
                    ((s1,c1),(s2,c2)) ↦ (s1+s2,c1+c2))
    .map((k,(s,c)) ↦ (k,s/c));
```

APACHE SPARK:

"DIRECTED ACYCLIC GRAPH" ("DAG")

Spark: Products by Hour

receipts.txt

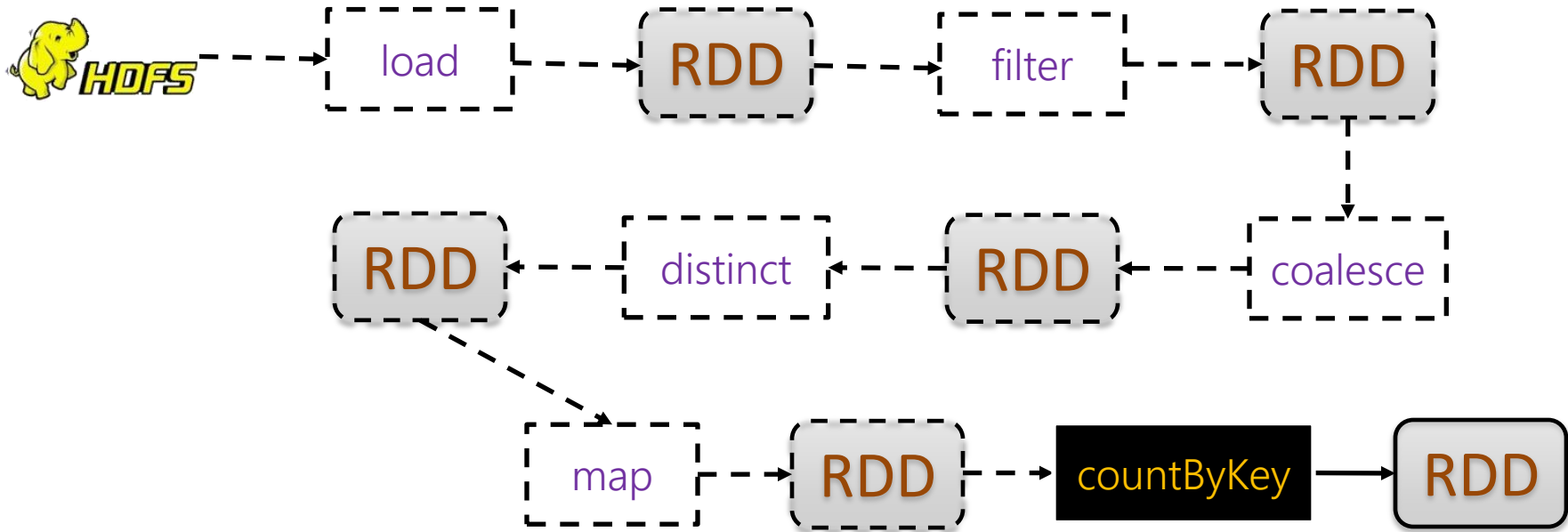


c1	i1	08	900
c1	i2	08	2000
c1	i2	08	2000
c2	i3	09	1240
c2	i4	09	3000
c2	i5	09	3000
c2	i6	09	2450
c2	i2	09	2000
c3	i7	08	7000
c3	i8	08	9230
c3	i2	08	2000
c4	i1	23	900
c4	i9	23	830
...	(customer,item,hour,price)		

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



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      non-b  73
c4      female 21
...
```



(customer,gender,age)

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

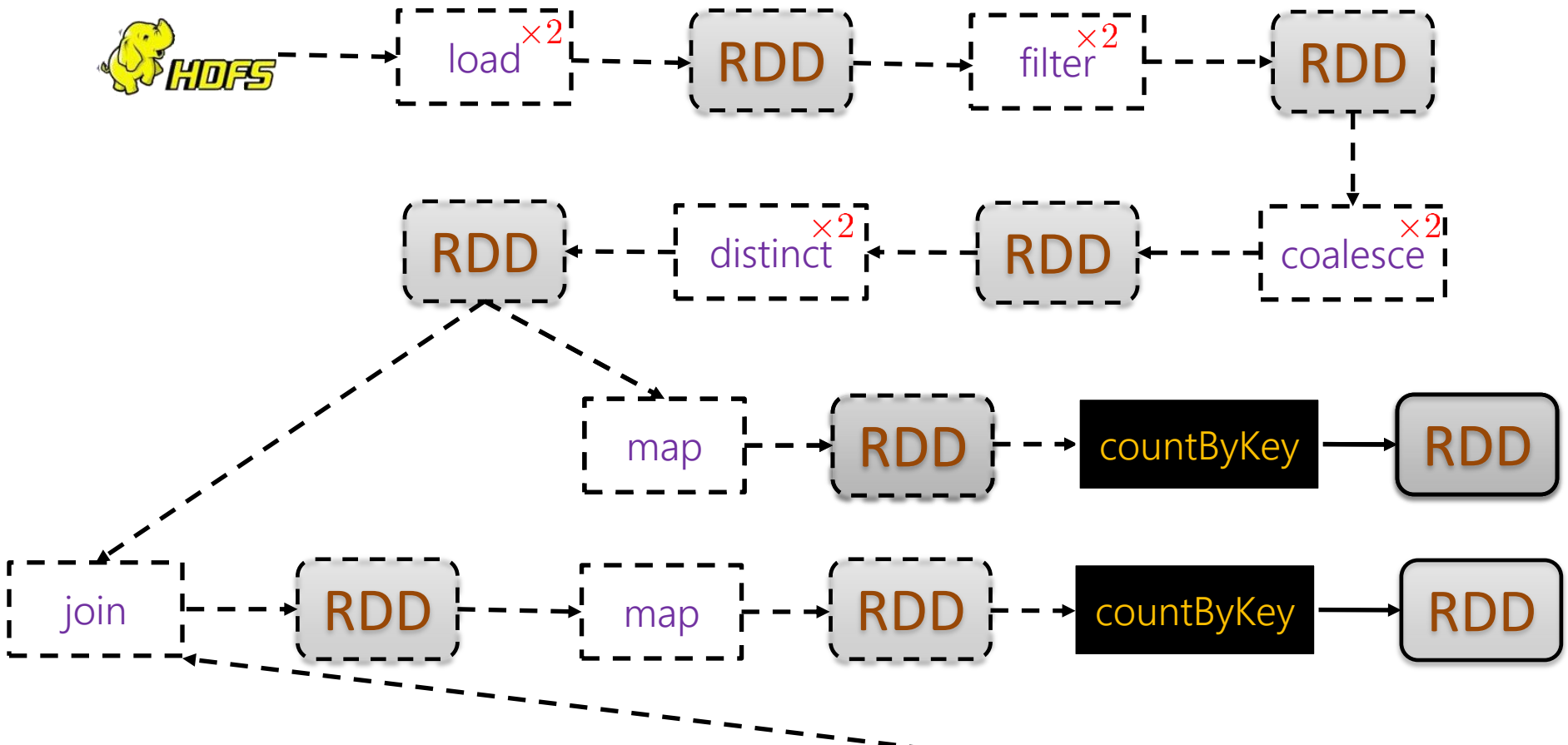
Also ... N^o of females older than 30 buying each premium item per hour of the day



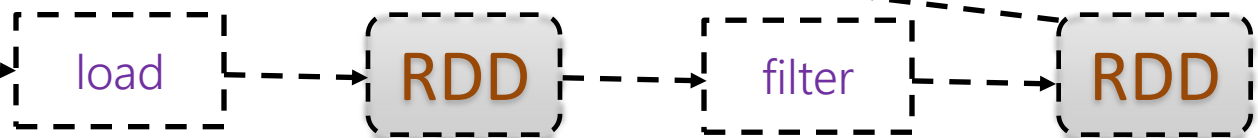
Spark: Directed Acyclic Graph (DAG)

Problem? 
Solution? 

receipts.txt



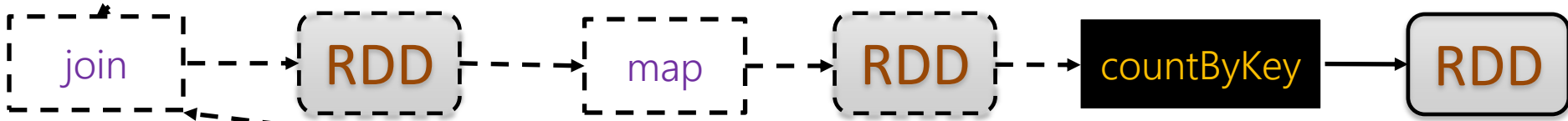
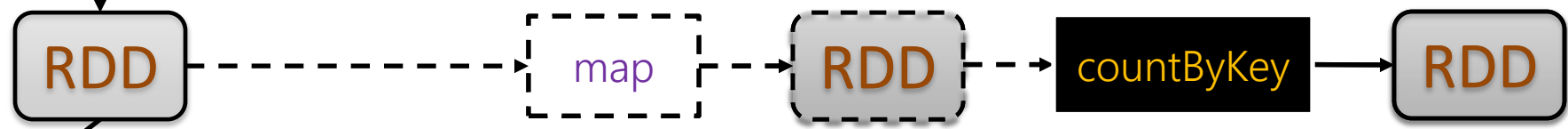
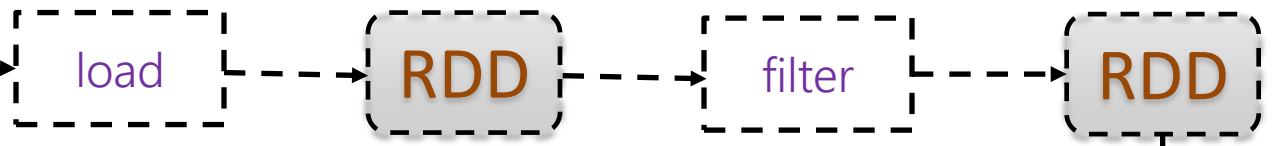
customer.txt



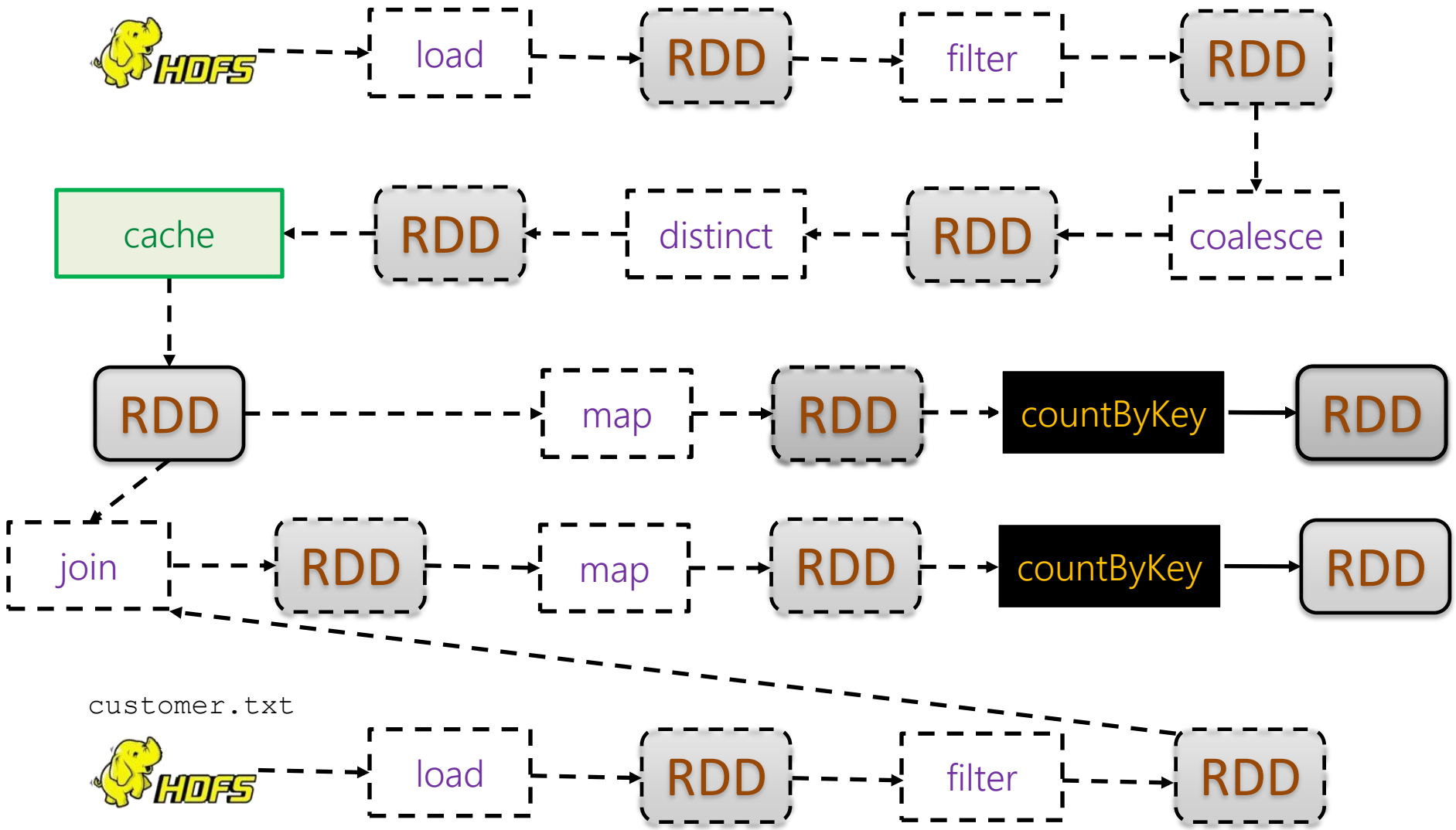
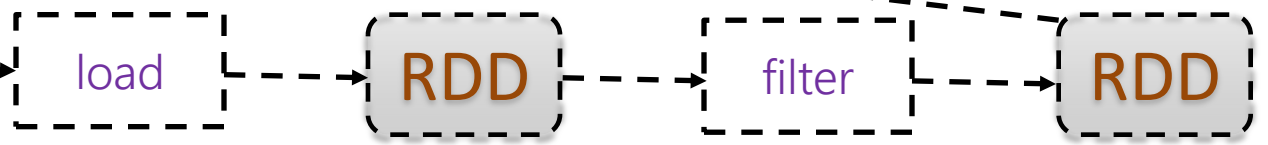
Spark: Directed Acyclic Graph (DAG)

- Problem? 
- Solution? 
- Materialise re-used RDD 

receipts.txt



customer.txt



Spark: Directed Acyclic Graph (DAG)

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

cache



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



VS



Hadoop vs. Spark: SQL, ML, Streams, ...



VS



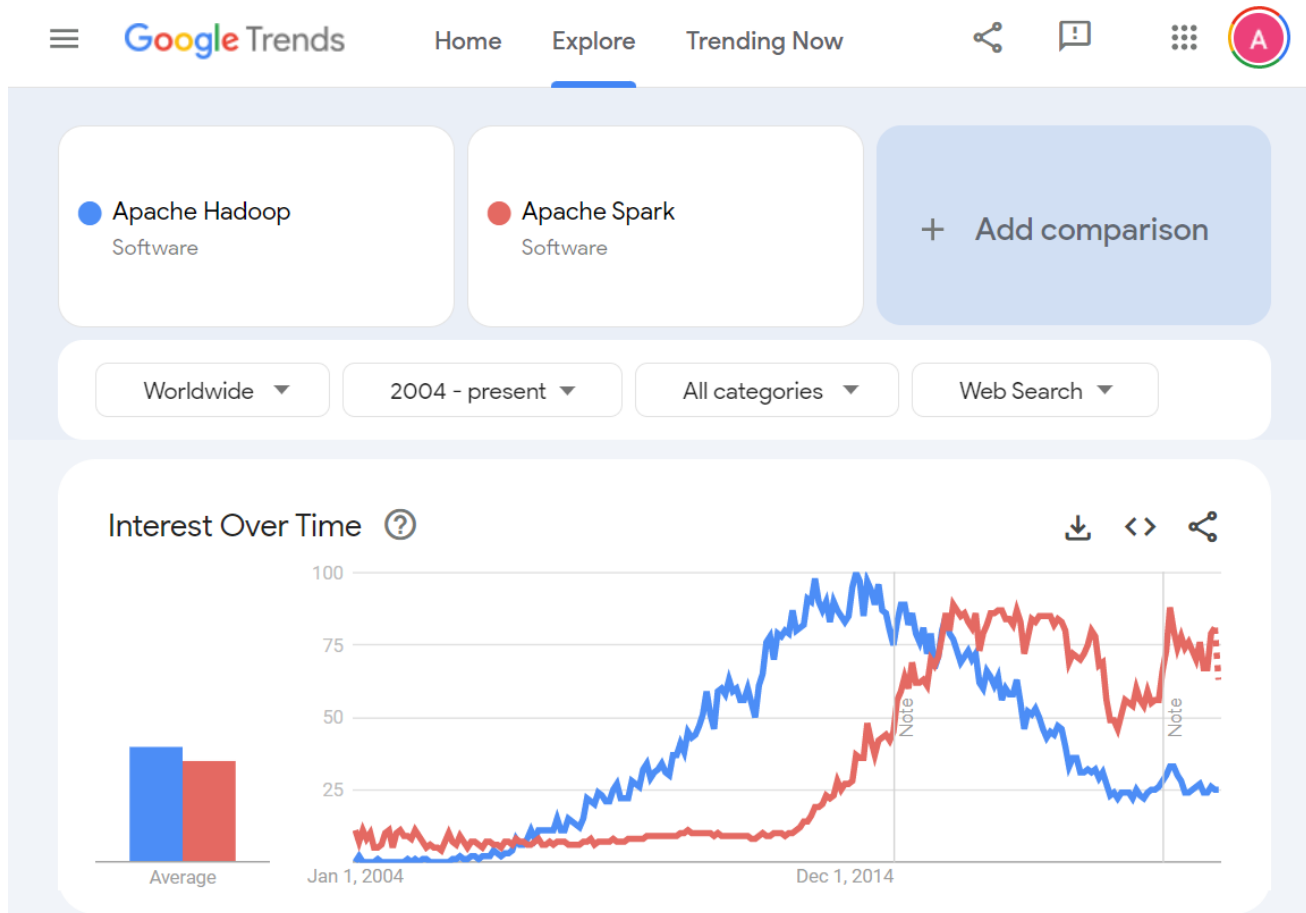
SPARK VS. HADOOP

Spark can use the disk

Apache Spark the fastest open source engine for sorting a petabyte (according to Spark)

		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)

Spark vs. Hadoop





Questions?