[544] MapReduce and Spark

Meenakshi Syamkumar

Learning Objectives

- describe the role mappers and reducers have in MapReduce jobs
- describe Spark concepts related to data lineage (RDDs, operations, transformations, actions)
- deploy Spark with multiple workers
- write Spark code using RDD and DataFrame APIs

Outline: MapReduce and Spark

Data Lakes

Hadoop MapReduce

Spark

Review: Data Warehouse

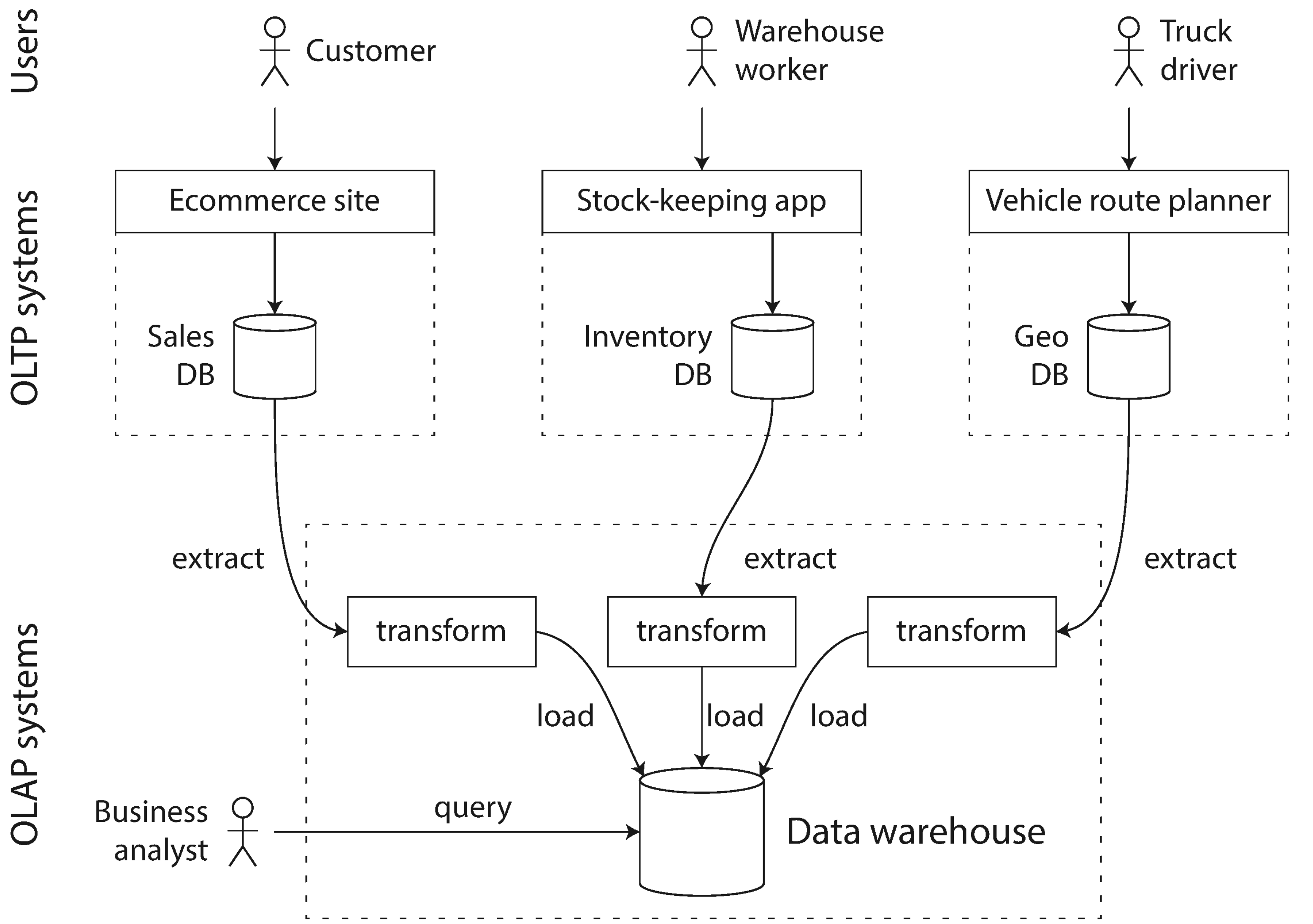
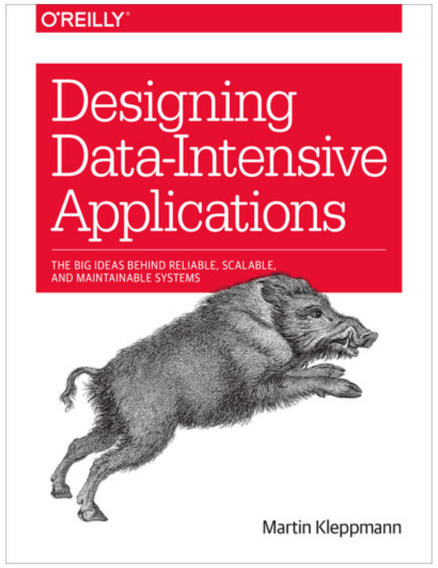


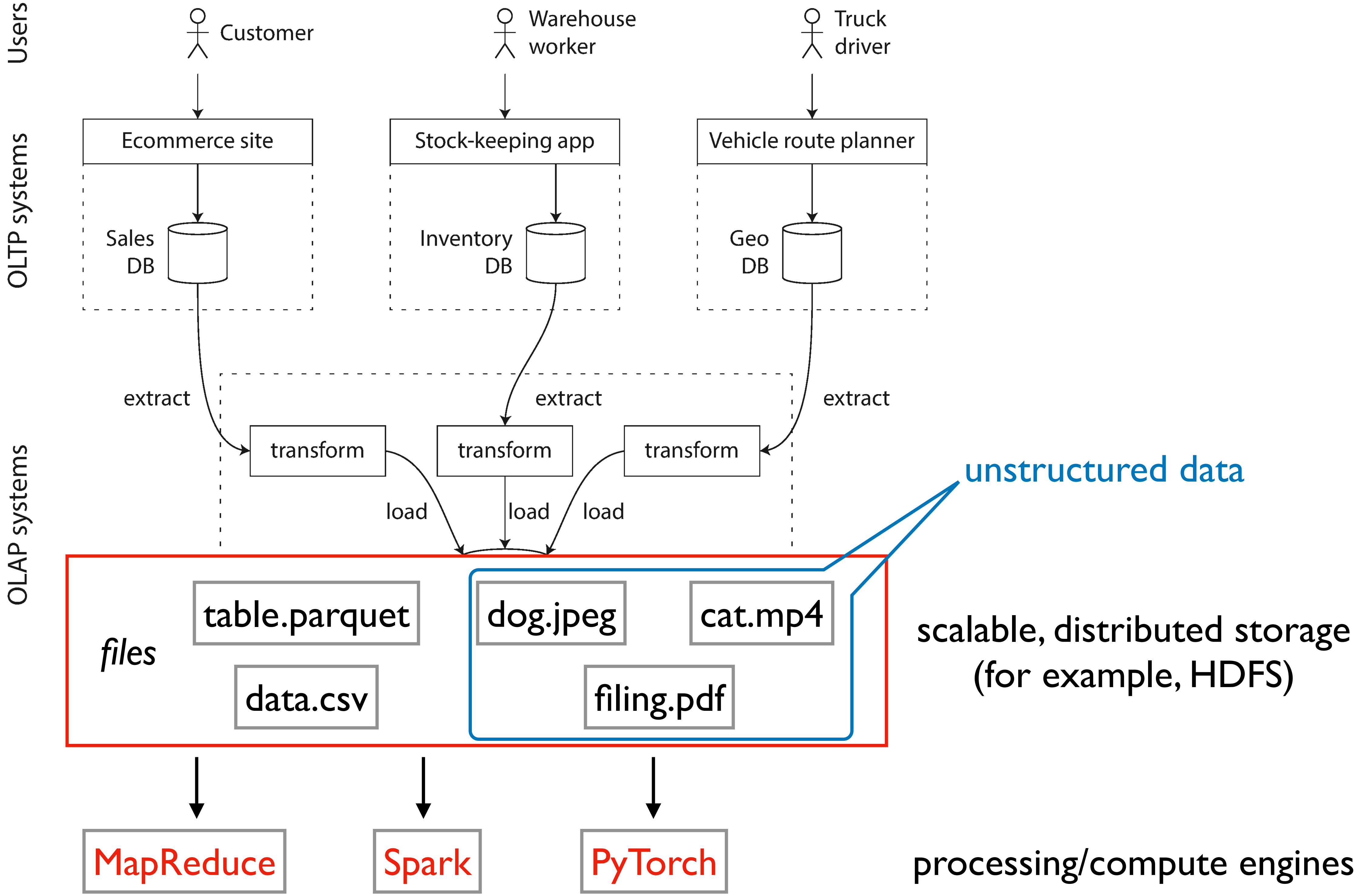
Figure 3-8. Simplified outline of ETL into a data warehouse.
(Chapter 3 of Data-Intensive Applications, by Kleppmann)


O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS
Martin Kleppmann

Data warehouse: storage + compute are tightly coupled (e.g., indexes)

- Efficient: coupling makes more optimization possible
- Limited: what if you want to do ML instead of running SQL queries?

"Data Lake" (new term for decoupled storage/compute for analytics)



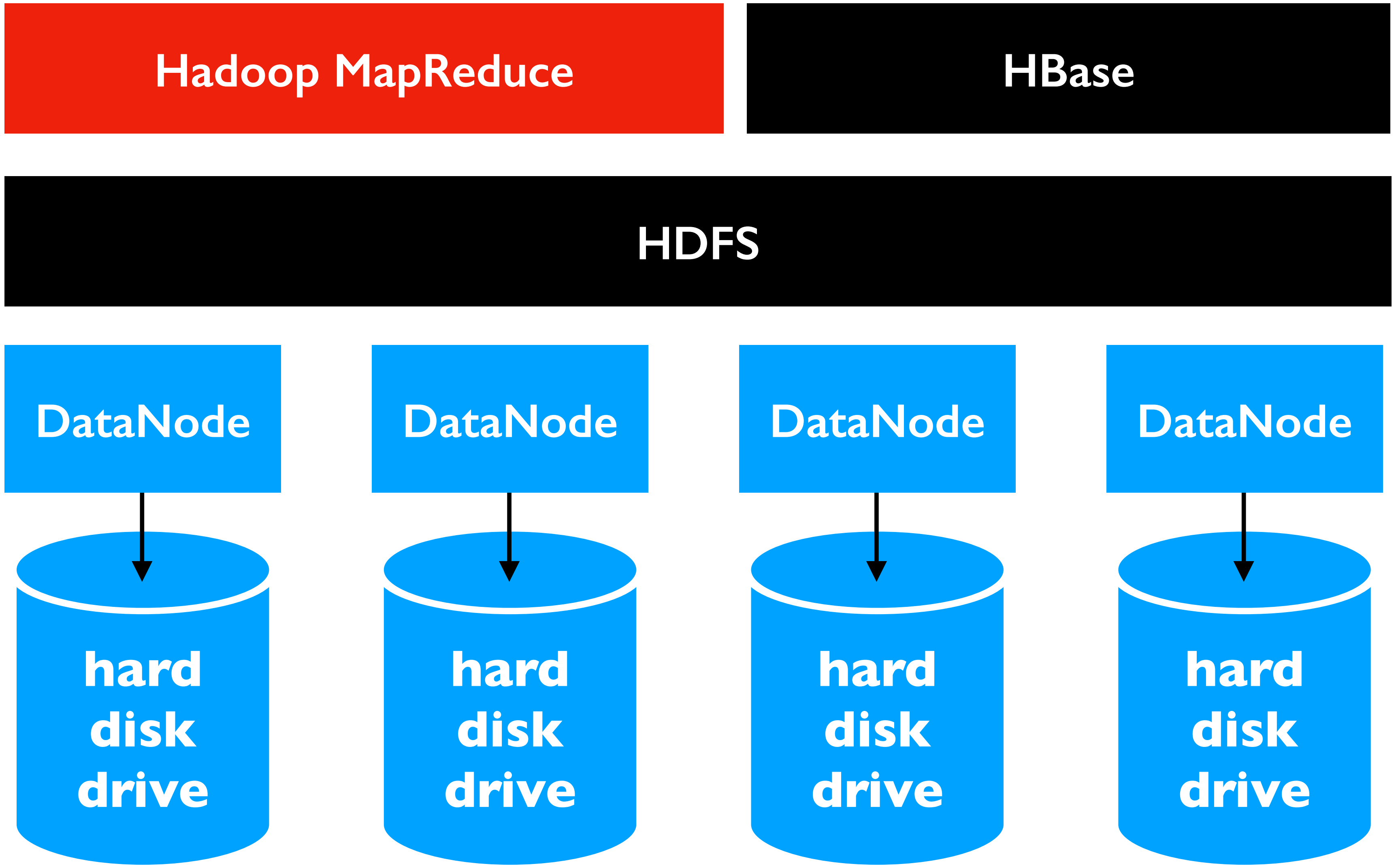
Outline: MapReduce and Spark

Data Lakes

Hadoop MapReduce

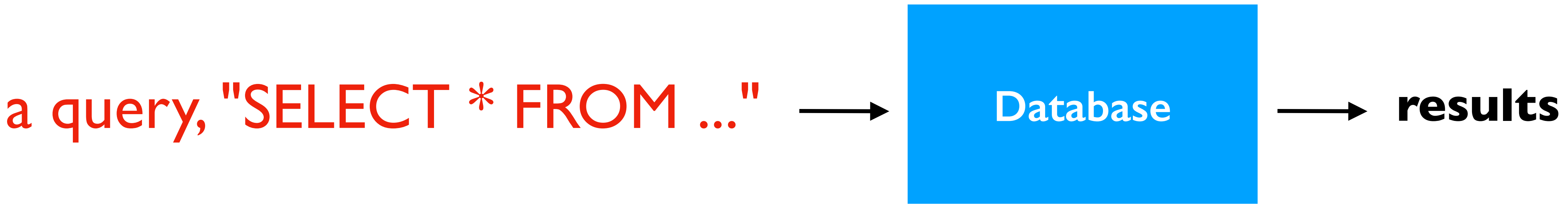
Spark

MapReduce


Hadoop MapReduce
HBase
HDFS
DataNode
DataNode
DataNode
DataNode
hard disk drive
hard disk drive
hard disk drive
hard disk drive

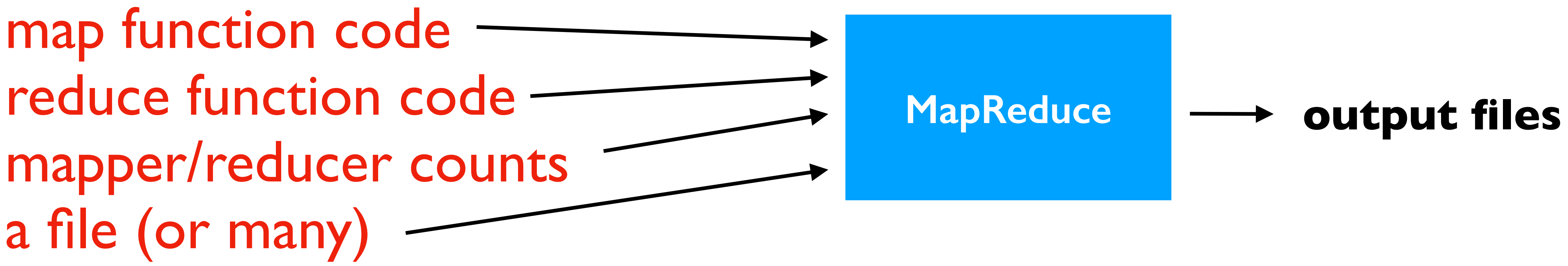
How do we answer questions?

SQL:



a query, "SELECT * FROM ..." → Database → **results**

MapReduce



map function code

reduce function code

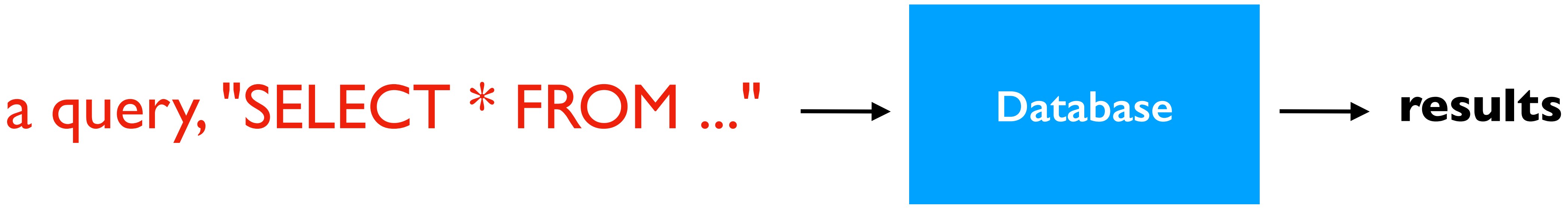
mapper/reducer counts

a file (or many)

→ MapReduce → **output files**

How do we answer questions?

SQL:

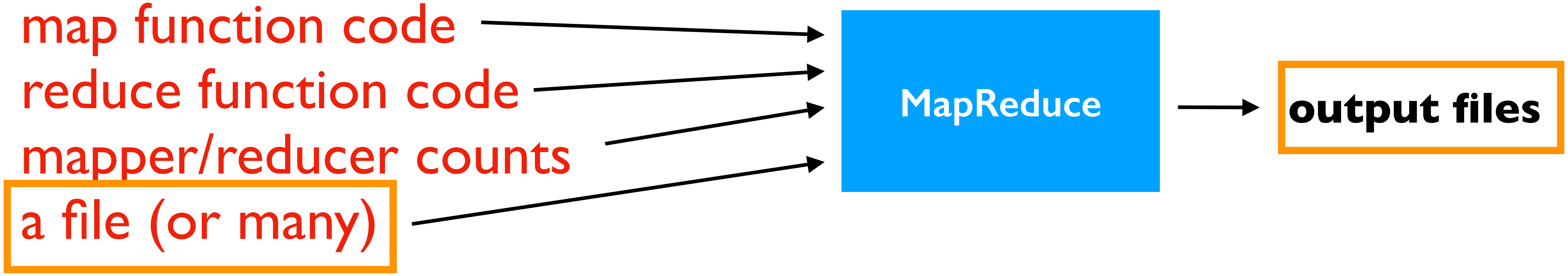


a query, "SELECT * FROM ..."

Database

results

MapReduce



map function code

reduce function code

mapper/reducer counts

a file (or many)

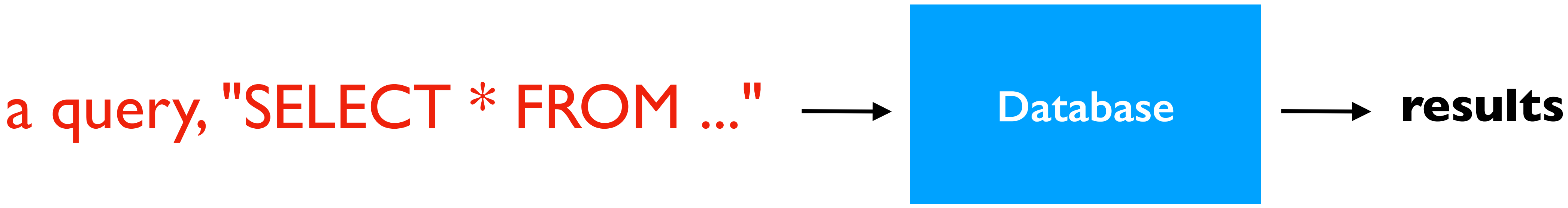
MapReduce

output files

input/output files are generally in HDFS

How do we answer questions?

SQL:

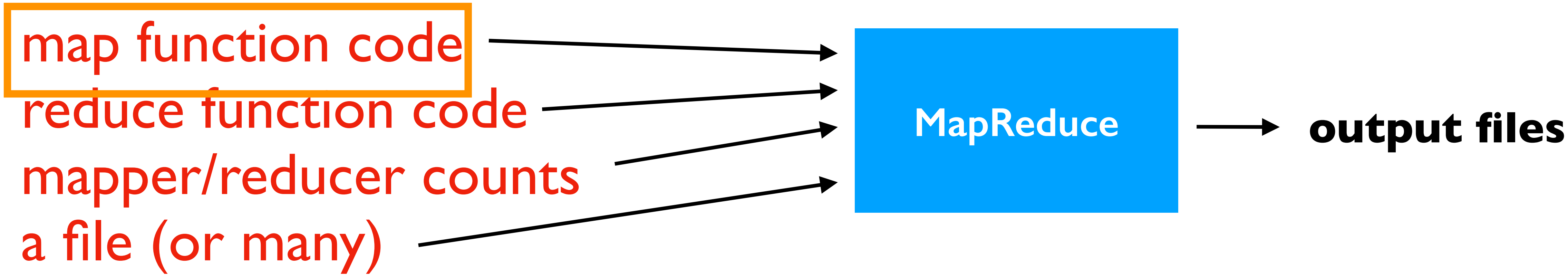


a query, "SELECT * FROM ..."

Database

results

MapReduce



map function code

reduce function code

mapper/reducer counts

a file (or many)

MapReduce

output files

Mappers by example: what are the colors of the squares?

input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

```
def map(key, value):
     ...
```

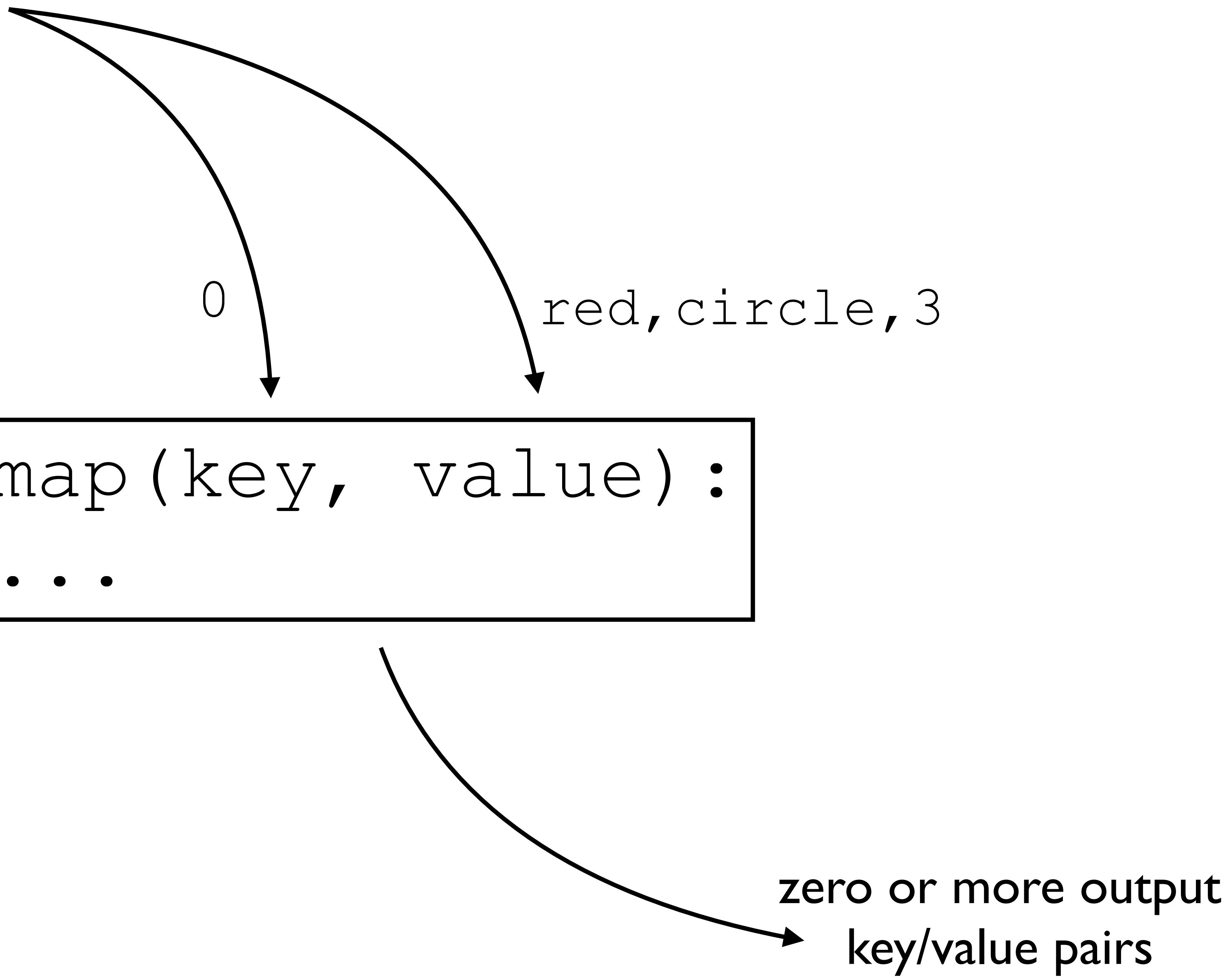
In SQL:

SELECT color FROM table WHERE shape = "square";

Mappers by example: what are the colors of the squares?

input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```



0

red,circle,3

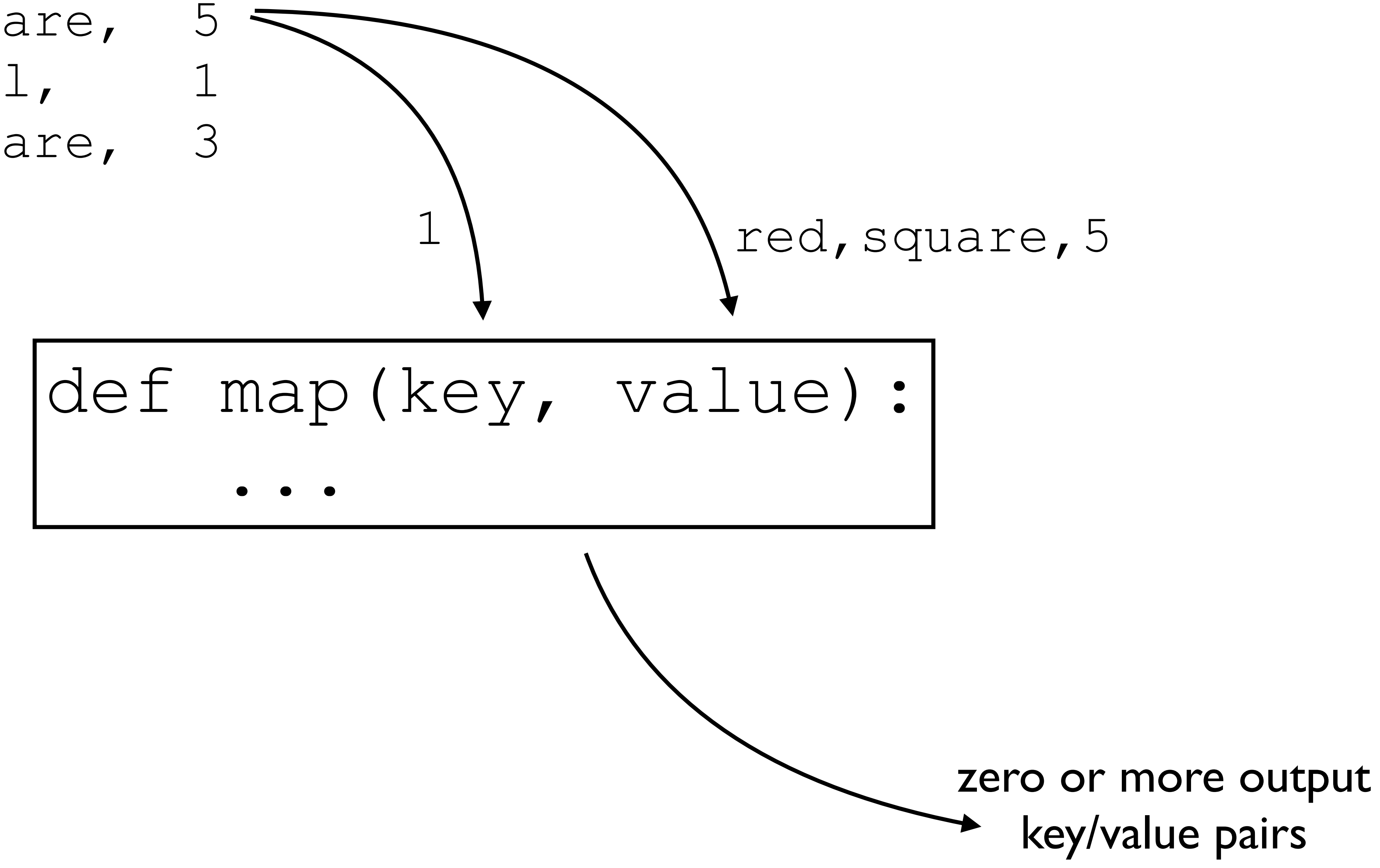
```
def map(key, value):
     ...
```

zero or more output
key/value pairs

Mappers by example: what are the colors of the squares?

input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```



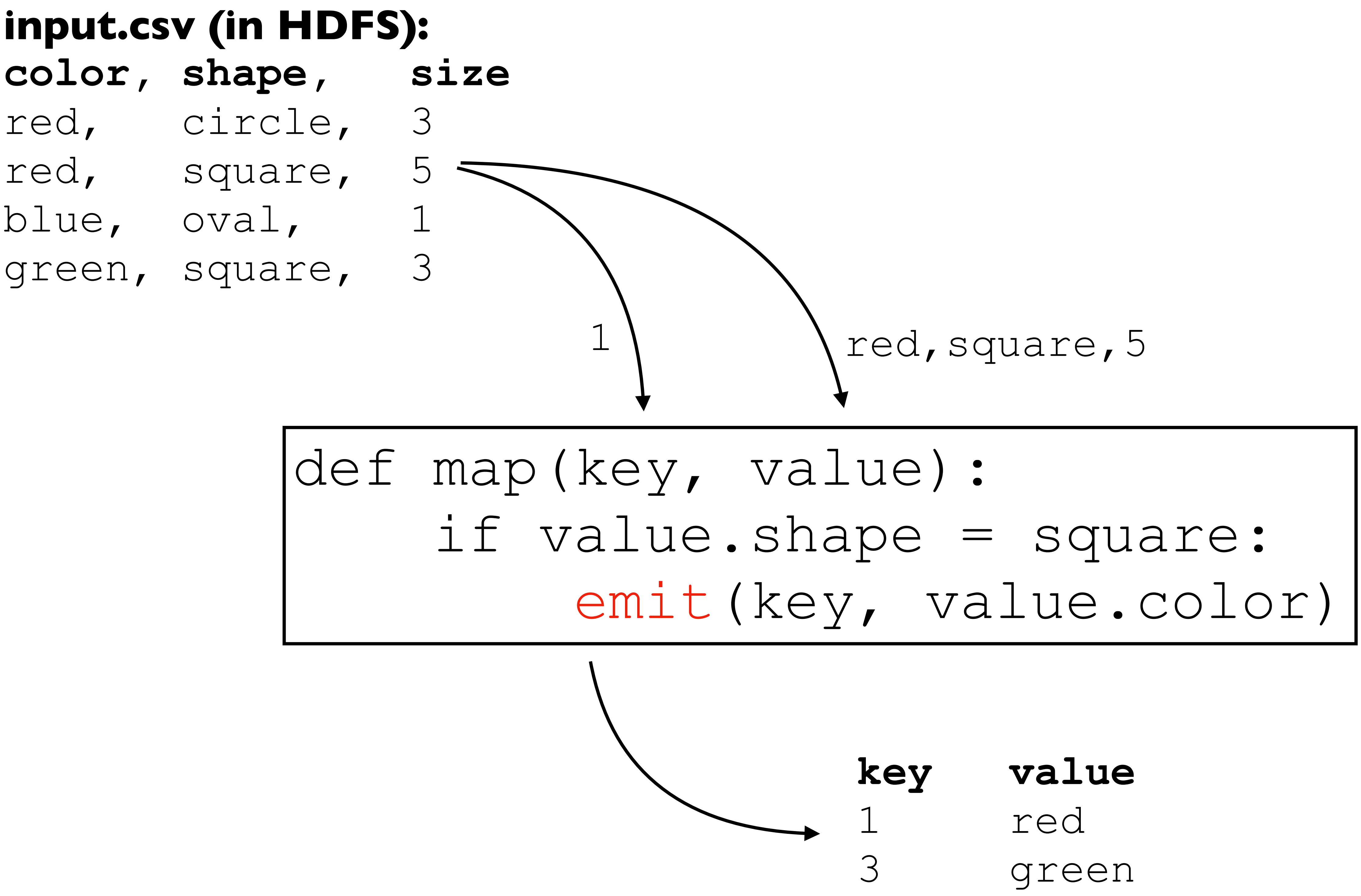
1

red,square,5

```
def map(key, value):
     ...
```

zero or more output key/value pairs

Mappers by example: what are the colors of the squares?



input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

1

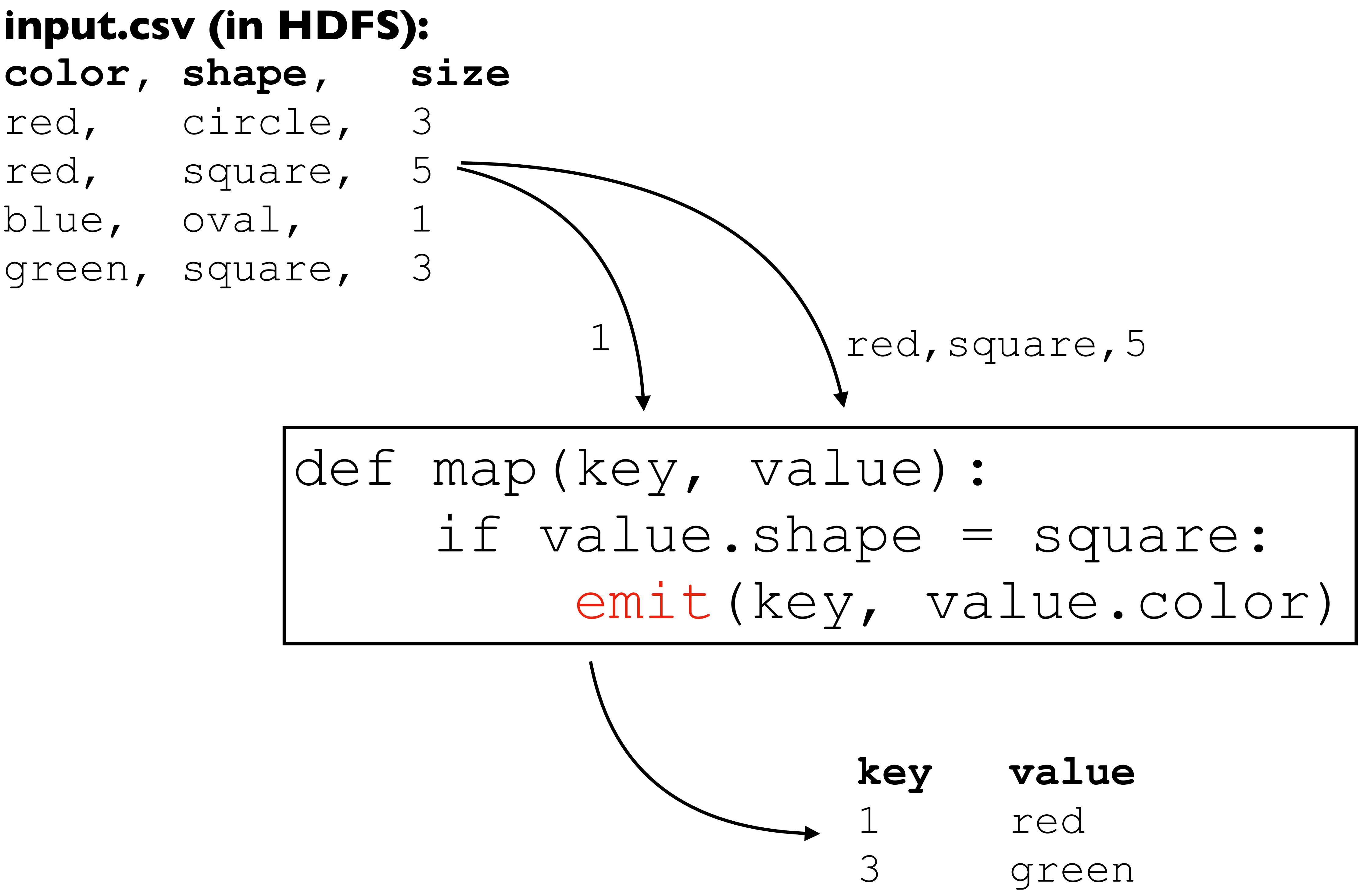
red,square,5

```
def map(key, value):
    if value.shape = square:
        emit(key, value.color)
```

key	value
1	red
3	green

Mappers by example: what are the colors of the squares?

what if the data is huge?



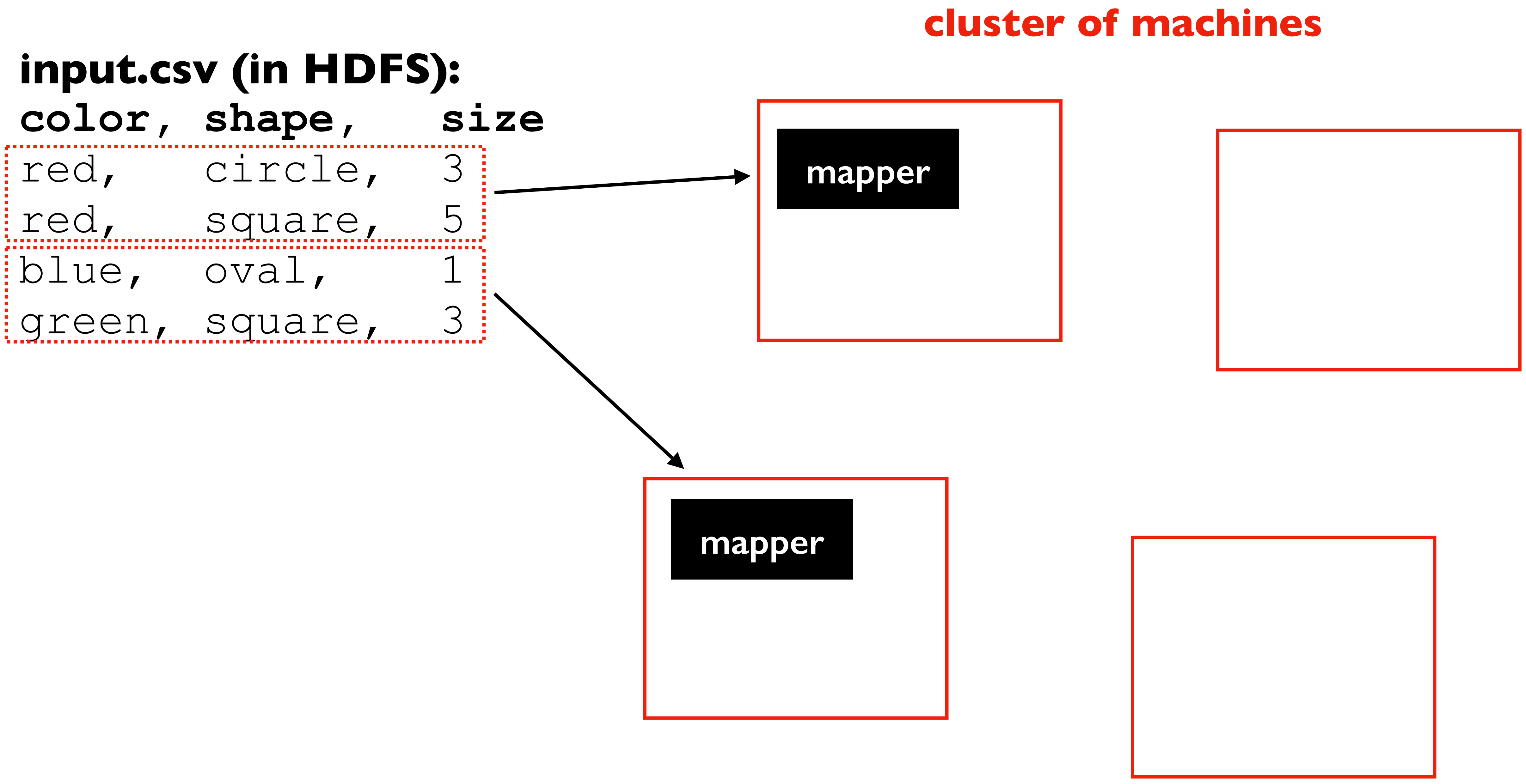
input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

```
def map(key, value):
    if value.shape = square:
        emit(key, value.color)
```

key	value
1	red
3	green

Mappers Run on Multiple Machines at Once



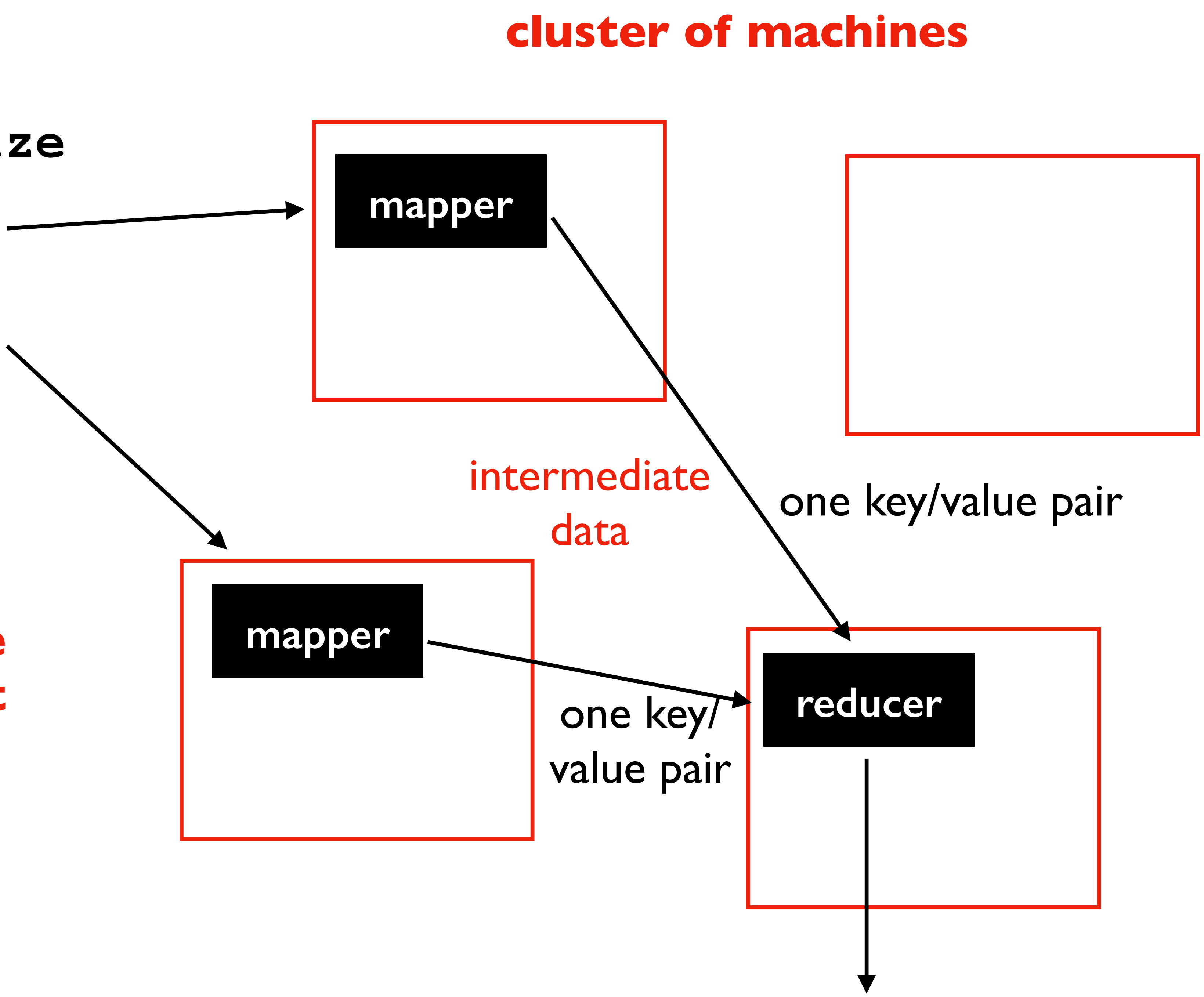
input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

Reducers

input.csv (in HDFS):

```
color, shape,    size
red,    circle,   3
red,    square,   5
blue,   oval,     1
green,  square,   3
```



a simple (default) reduce task can combine output of multiple mappers to a single file

key	**value**
1	red
3	green

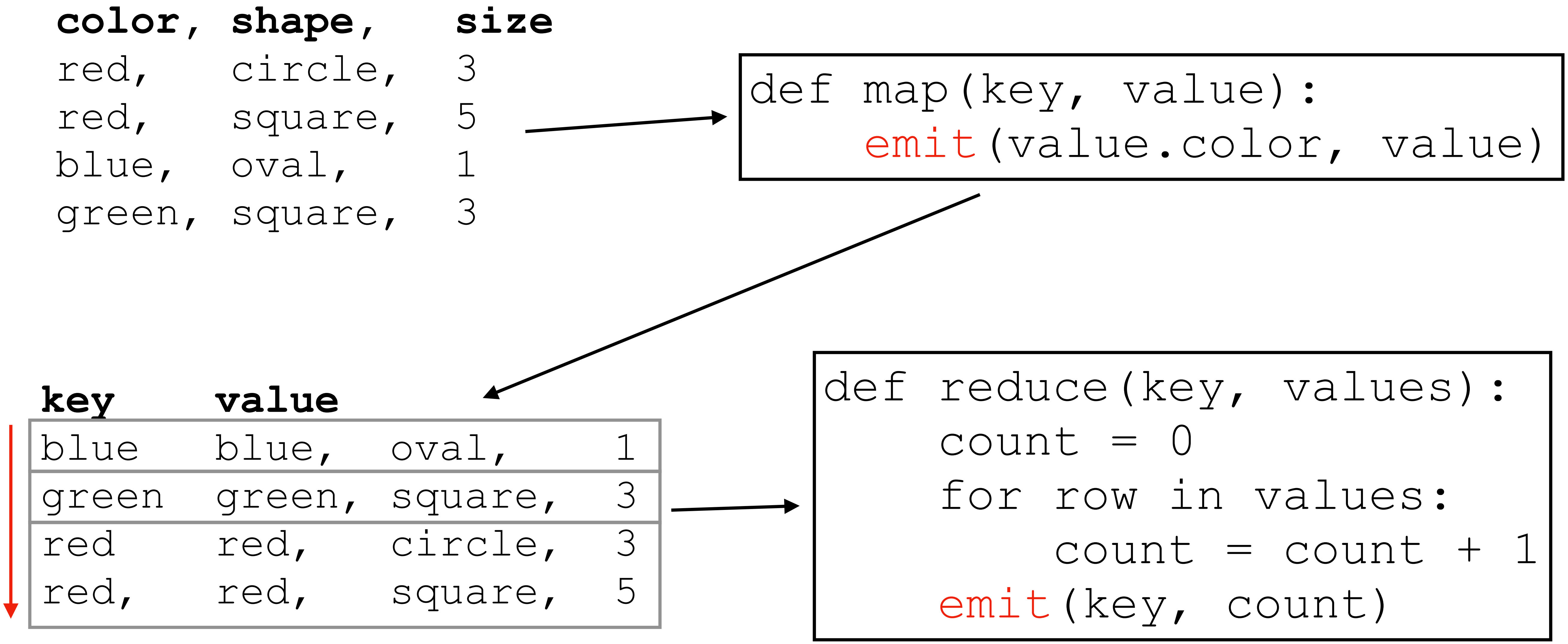
Reducers

MapReduce shuffles the data, bringing together intermediate outputs from mappers with the same key. All data with the same key is passed to a single "reduce" call. A reduce task will generally have reduce called many times with different keys.

```
def reduce(key, values):
    for row in values:
        emit(key, row)
```

reducers can output exactly their input,
OR have further computation

Reducers



```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

```
def map(key, value):
    emit(value.color, value)
```

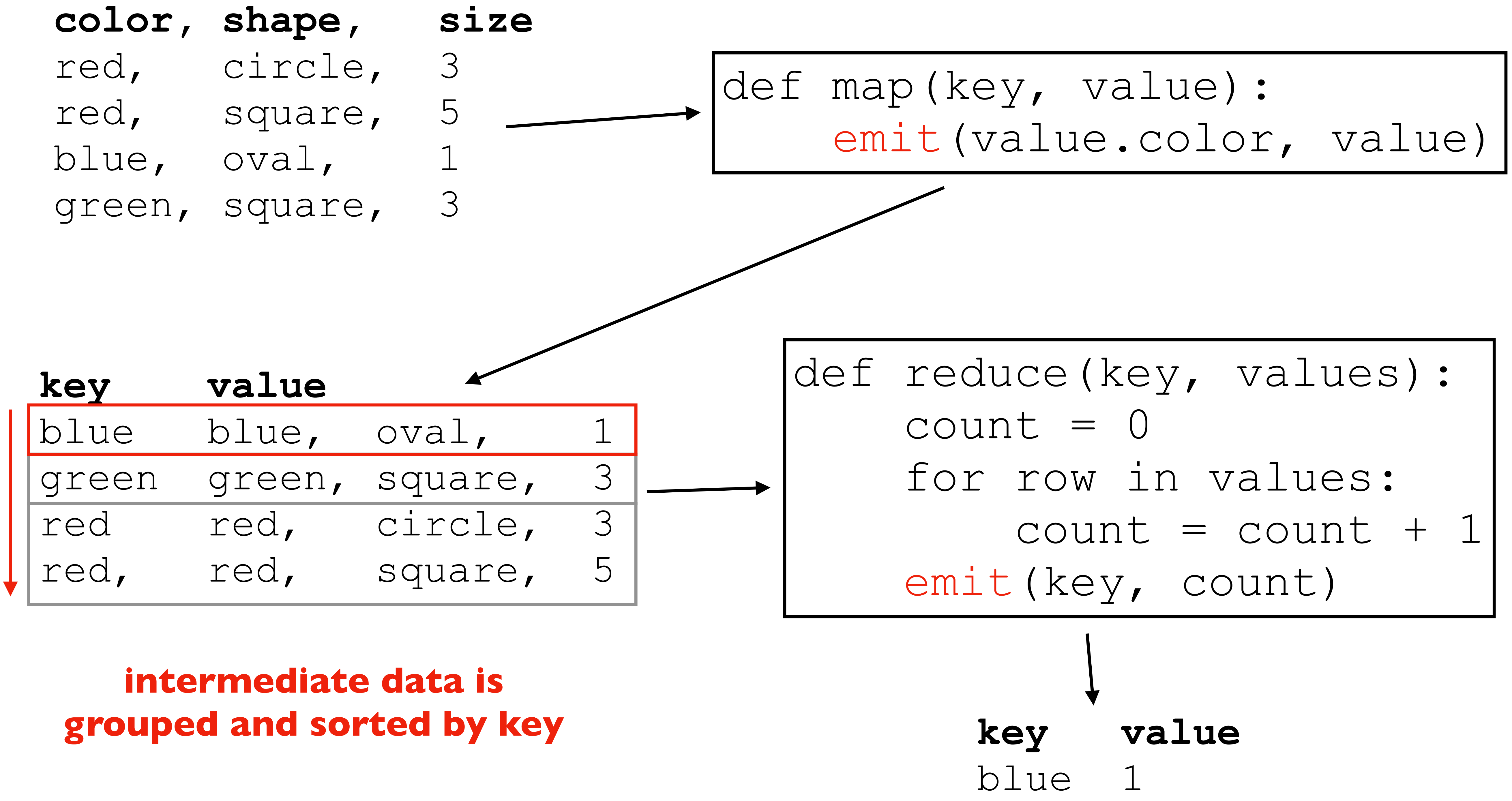
key	value		
blue	blue,	oval,	1
green	green,	square,	3
red	red,	circle,	3
red,	red,	square,	5

```
def reduce(key, values):
    count = 0
    for row in values:
        count = count + 1
    emit(key, count)
```

intermediate data is grouped and sorted by key

reduce will be called 3 times (once for each group). The calls could happen in one reduce task (or be split over many)

Reducers



```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

```
def map(key, value):
    emit(value.color, value)
```

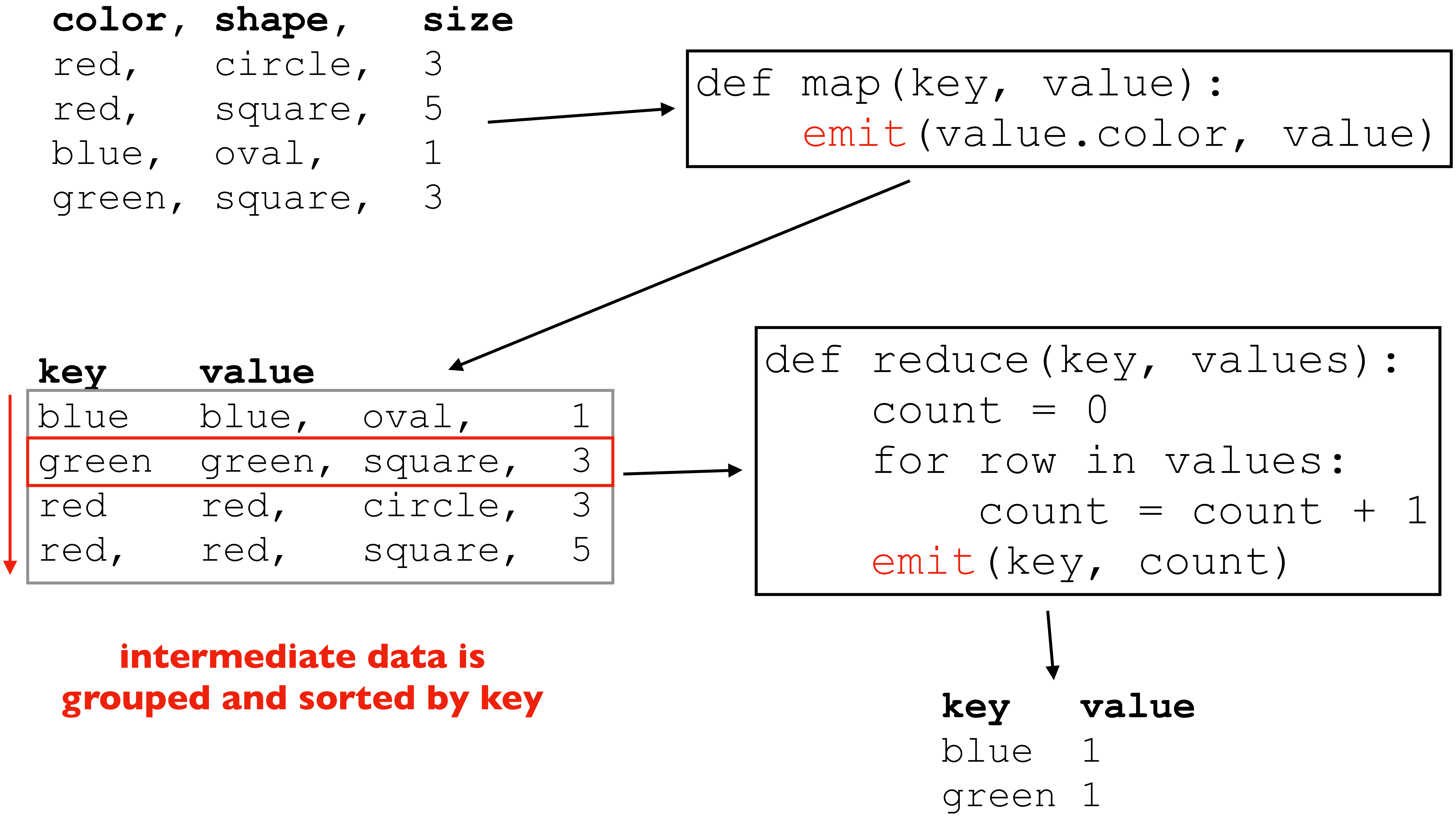
key	value		
blue	blue,	oval,	1
green	green,	square,	3
red	red,	circle,	3
red,	red,	square,	5

intermediate data is grouped and sorted by key

```
def reduce(key, values):
    count = 0
    for row in values:
        count = count + 1
    emit(key, count)
```

key	value
blue	1

Reducers



```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

```
def map(key, value):
    emit(value.color, value)
```

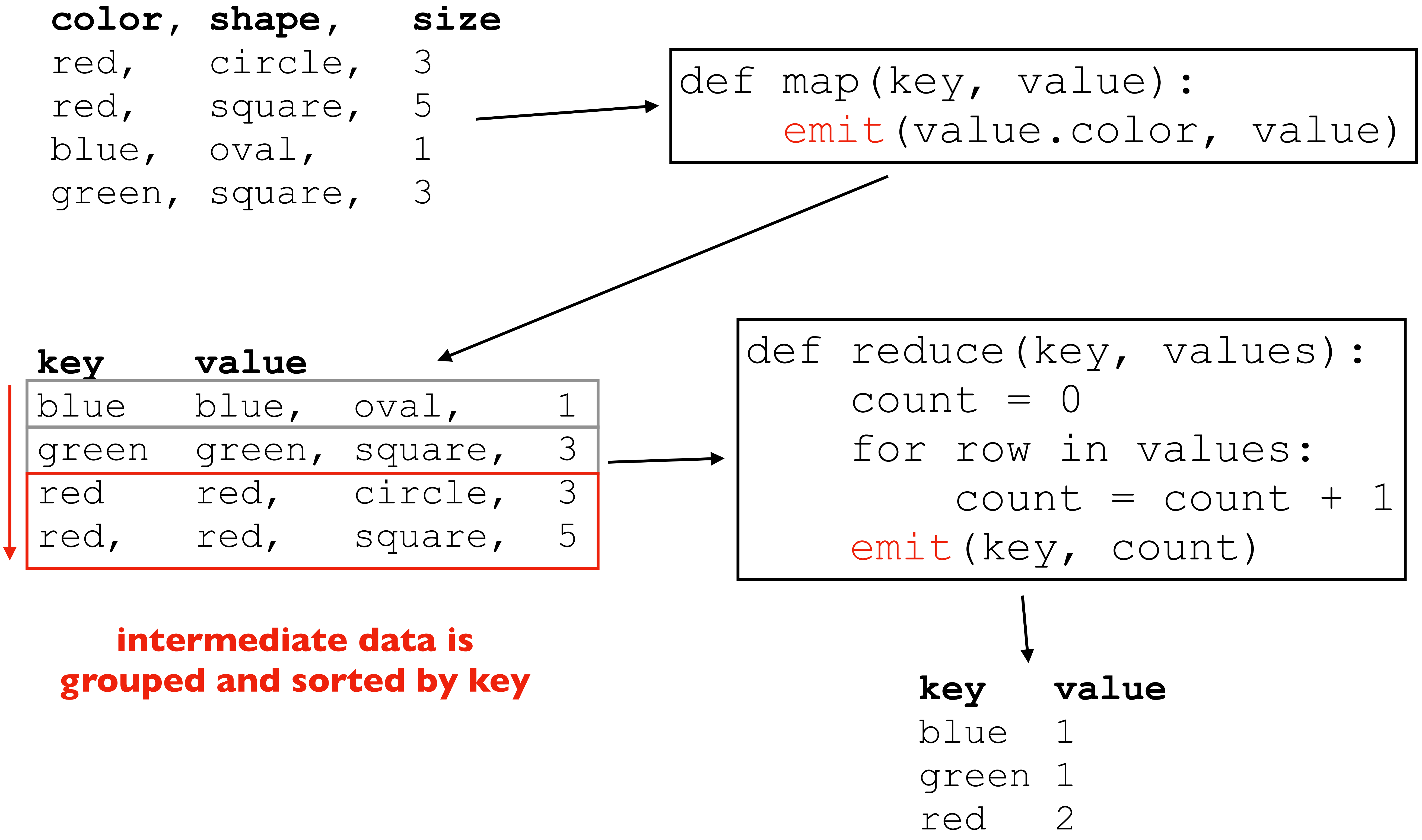
key	value
blue	blue, oval, 1
green	green, square, 3
red	red, circle, 3
red,	red, square, 5

intermediate data is grouped and sorted by key

```
def reduce(key, values):
    count = 0
    for row in values:
        count = count + 1
    emit(key, count)
```

key	value
blue	1
green	1

Reducers



```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```

```
def map(key, value):
    emit(value.color, value)
```

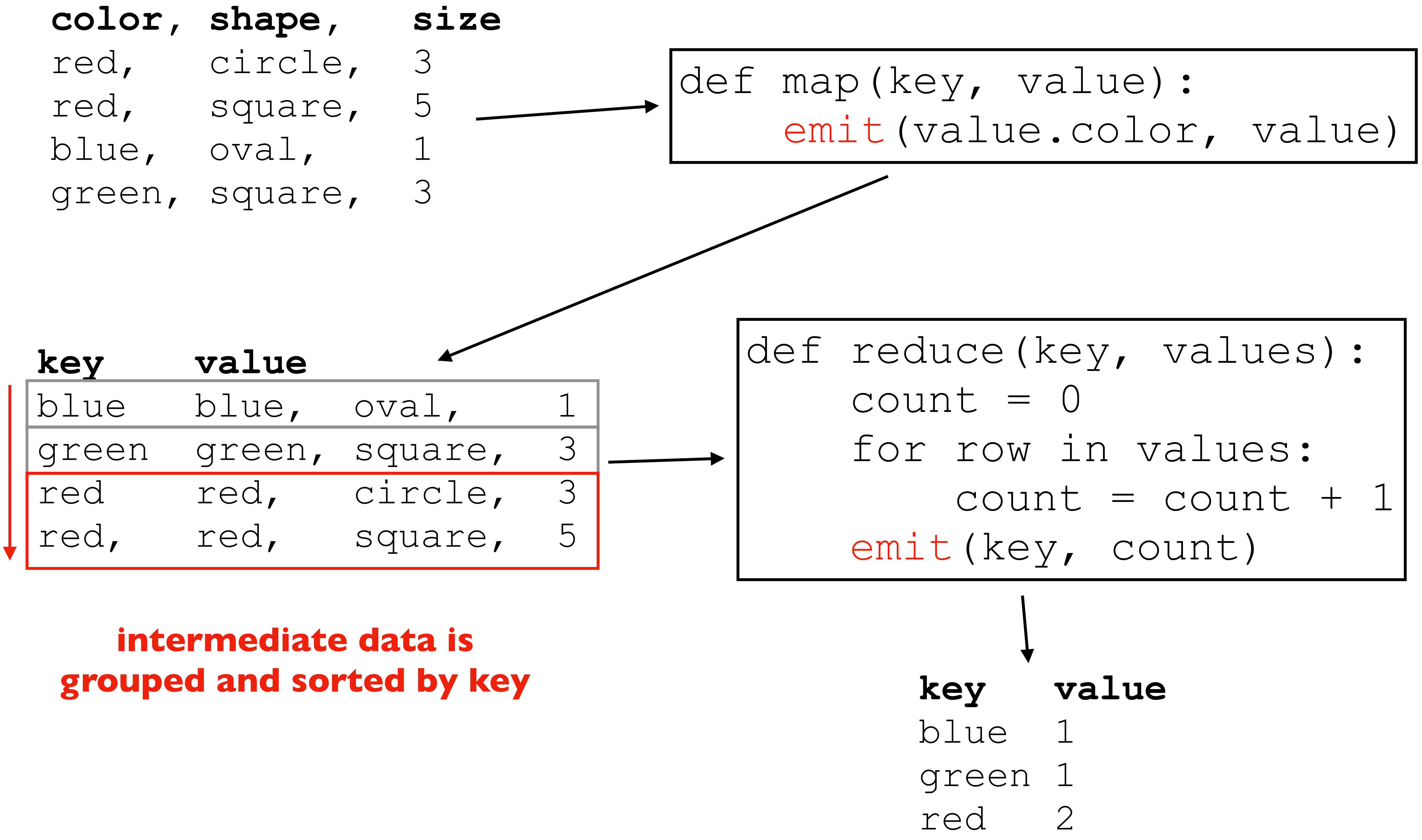
key	value		
blue	blue,	oval,	1
green	green,	square,	3
red	red,	circle,	3
red,	red,	square,	5

```
def reduce(key, values):
    count = 0
    for row in values:
        count = count + 1
    emit(key, count)
```

intermediate data is grouped and sorted by key

key	value
blue	1
green	1
red	2

What is the SQL equivalent of this MapReduce program?



color,	shape,	size
red,	circle,	3
red,	square,	5
blue,	oval,	1
green,	square,	3

```
def map(key, value):
    emit(value.color, value)
```

key	value		
blue	blue,	oval,	1
green	green,	square,	3
red	red,	circle,	3
red,	red,	square,	5

```
def reduce(key, values):
    count = 0
    for row in values:
        count = count + 1
    emit(key, count)
```

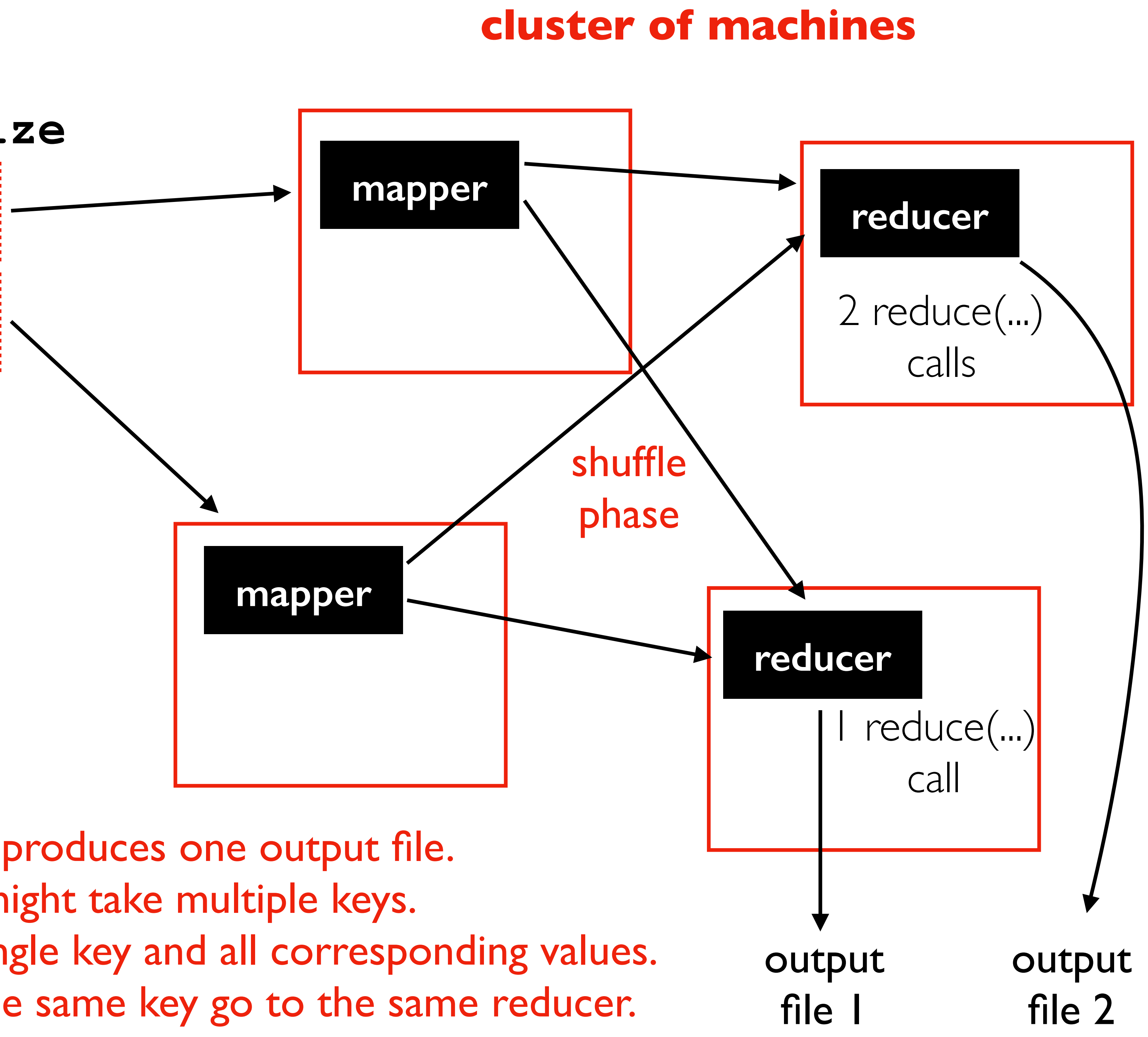
key	value
blue	1
green	1
red	2

intermediate data is grouped and sorted by key

Multiple Reducers (for big intermediate data)

input.csv (in HDFS):

```
color, shape,   size
red,   circle,  3
red,   square,  5
blue,  oval,    1
green, square,  3
```



each reduce **task** produces one output file.
a reduce **task** might take multiple keys.
a single reduce **call** takes a single key and all corresponding values.
all intermediate rows with the same key go to the same reducer.

SQL => MapReduce

Map Phase

- SELECT, WHERE, GROUP BY, JOIN

Shuffle Phase (bringing related data to same place)

- ORDER BY, GROUP BY, JOIN

Reduce Phase

- SELECT, AGGREGATE, HAVING, JOIN

MapReduce is more flexible. (for example, how to do a GROUP BY where one row goes to mutliple groups in SQL?)

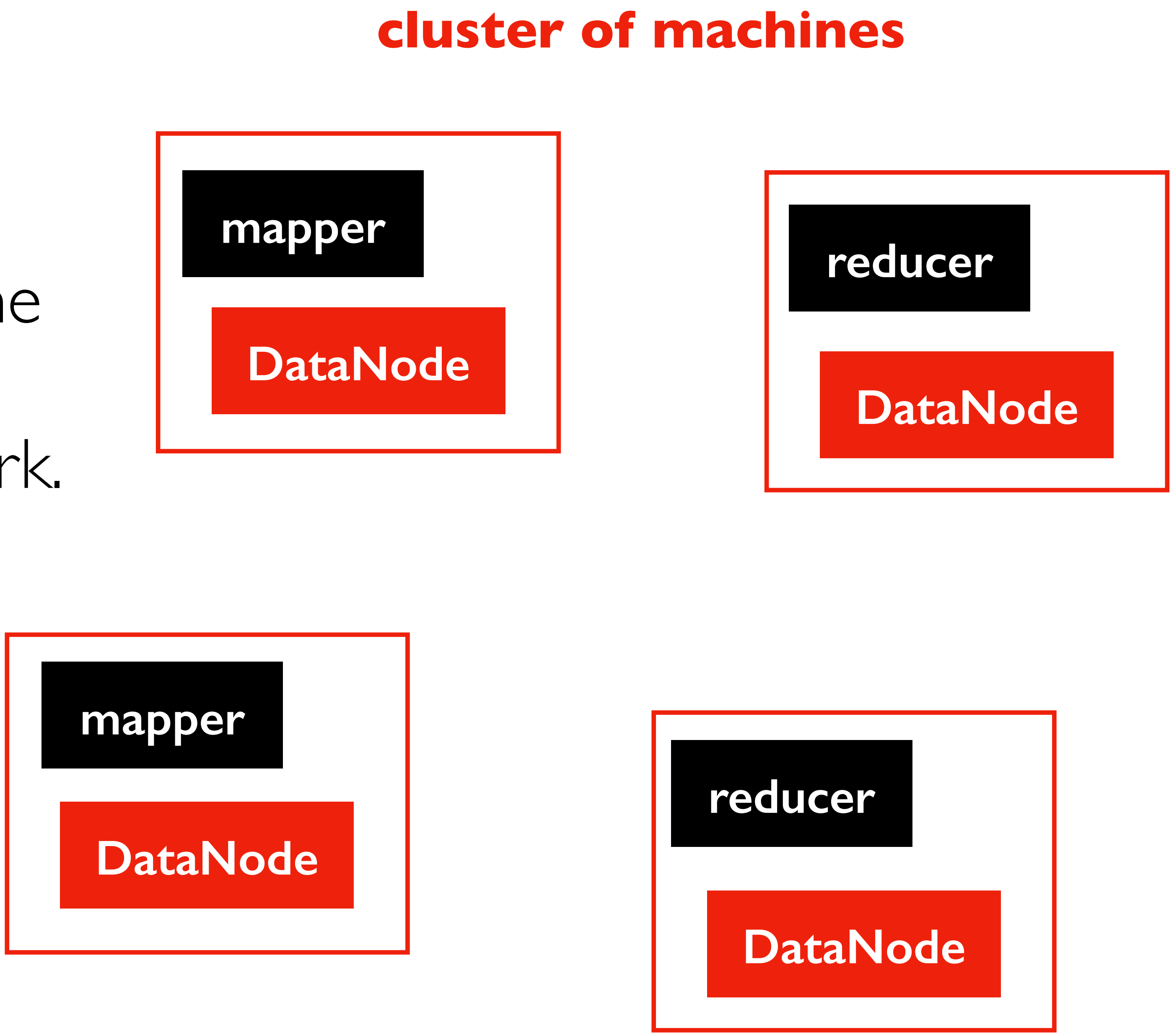
Projects like **HiveQL** try to make MapReduce more accessible.

Data Locality: Avoid Network Transfers

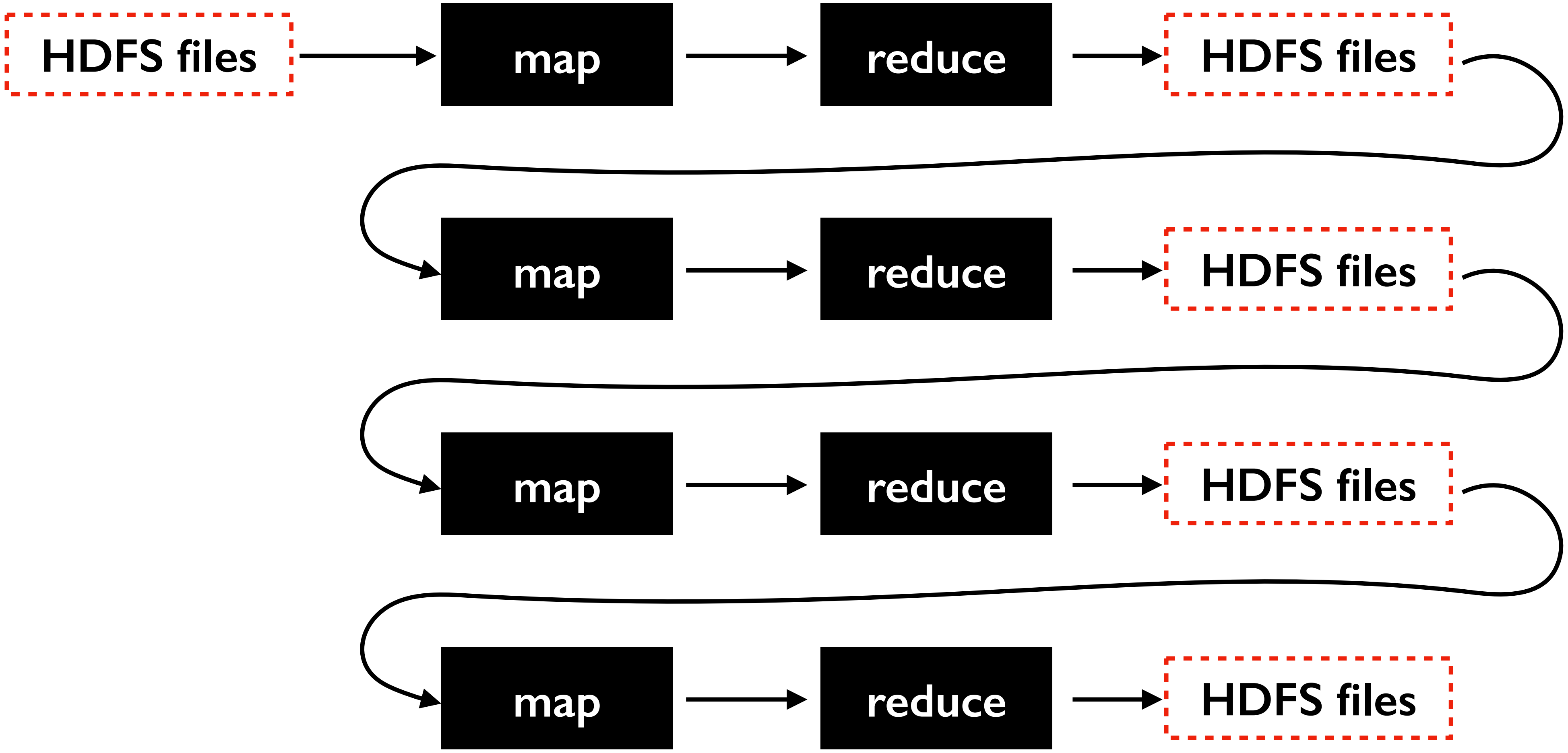
Run on same machines

- HDFS DataNodes
- MapReduce executor

Try to run mappers on machine where DataNode has needed data. Uses disk but not network.



Pipelines: Sequence of MapReduce Jobs



Efficiency: is storing intermediate data in HDFS a good idea?

- replication on data we could re-compute seems wasteful (could set replication to 1x)
- could we sometimes connect output from one stage more directly to the next?
- treating each stage independently prevents optimization tools from improving the whole pipeline

Outline: MapReduce and Spark

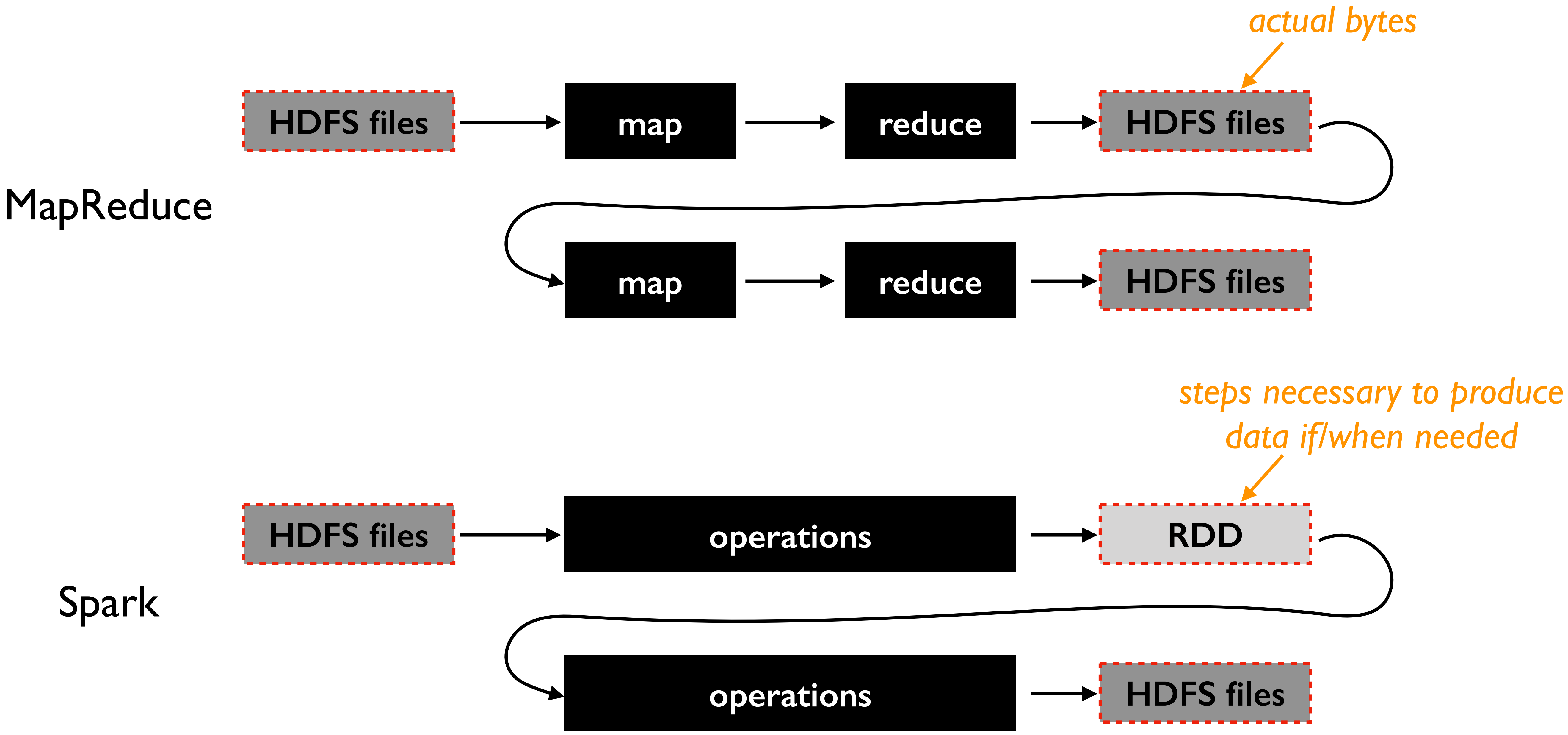
Data Lakes

Hadoop MapReduce

Spark

- Resilient Distributed Datasets (RDDs)
- SQL and DataFrames
- Deployment

Intermediate Data: MapReduce vs. Spark



Resilient Distributed Datasets (RDD)

- data lineage: record series of operations on other data necessary to obtain results
- lazy evaluation: computation only done when results needed (to write file, make plot, etc.)
- immutability: you can't change an RDD, but you can define a new one in terms of another

Data Lineage: Transformations and Actions

```
data = [
      ("A", 1),
      ("B", 2),
      ("A", 3),
      ("B", 4)
]
```

```
def mult2(row):
    return (row[0], row[1] * 2)

def onlyA(row):
    return row[0] == "A"
```

goal: get 2 times the second column wherever the first column is "A"

```
table = sc.parallelize(data)
double = table.map(mult2)
doubleA = double.filter(onlyA)
doubleA.collect()
```

```
[('A', 2),
 ('A', 6)]
```

The computation is a sequence of 4 operations. Operations come in two types:

- transformation: create a new RDD (lazy, so no execution yet). Here: parallelize, map, and filter.
- action: perform all operations in the graph to get an actual result. Here: collect.

Data Lineage: Transformations and Actions

```
data = [
    ("A", 1),
    ("B", 2),
    ("A", 3),
    ("B", 4)
]
```

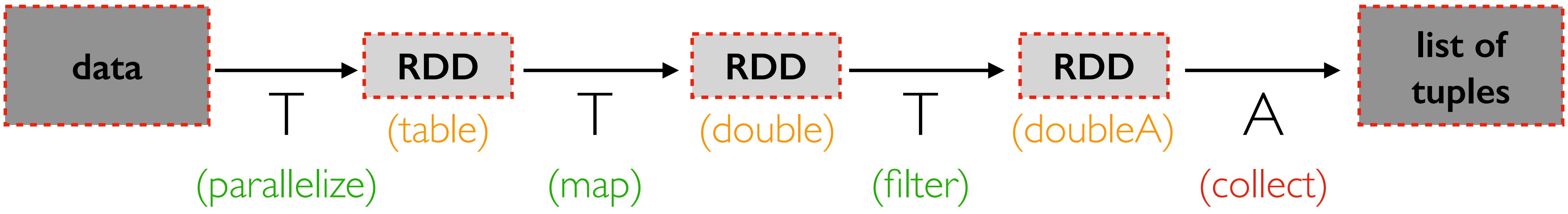
```
def mult2(row):
    return (row[0], row[1] * 2)

def onlyA(row):
    return row[0] == "A"
```

goal: get 2 times the second column wherever the first column is "A"

```
table = sc.parallelize(data)
double = table.map(mult2)
doubleA = double.filter(onlyA)
doubleA.collect()
```

```
[('A', 2),
 ('A', 6)]
```



are there alternative paths you could create from the start to end node?

Optimization

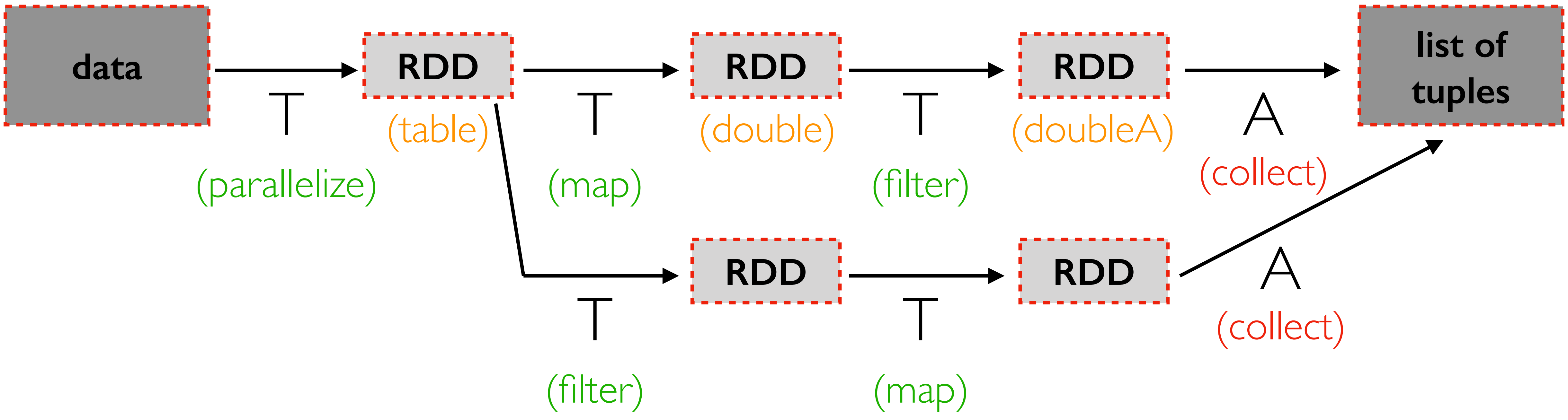
Transformation vs. action

- transformation: intermediate results (means to an end)
- action: final results we care about
- this distinction creates opportunities for **optimize**, choosing a more efficient sequence of transformations to reach the same endpoint
- tools need to know what transformations are doing (difficult with Python functions) to automatically optimize

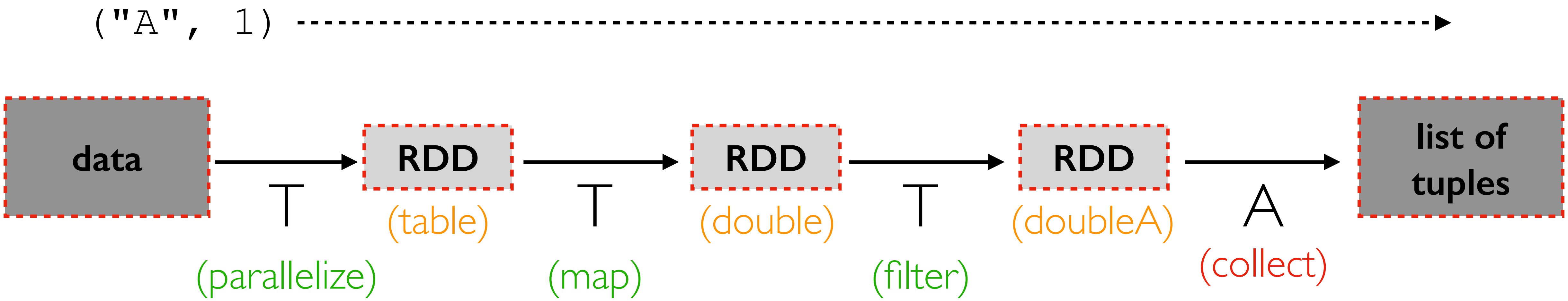
goal: get 2 times the second column wherever the first column is "A"

```
table = sc.parallelize(data)
double = table.map(mult2)
doubleA = double.filter(onlyA)
doubleA.collect()
```

```
[('A', 2),
 ('A', 6)]
```



Partitions



In what granularity should data flow through the transformations?

- whole dataset: it could all proceed through, one transformation at a time, but might not fit in memory
- row: in this pipeline, nothing prevents each row from passing through independently, but probably slower than computing in bulk
- partition: Spark users can specify the number of partitions for an RDD

```
sc.parallelize(data, 1)
    data = [
        ("A", 1),
        ("B", 2),
        ("A", 3),
        ("B", 4)
    ]
```

partition

```
sc.parallelize(data, 2)
    data = [
        ("A", 1),
        ("B", 2),
        ("A", 3),
        ("B", 4)
    ]
```

partition

partition

Tasks

Spark work

- spark code is converted to jobs, which consist of stages, which consist of tasks
- tasks:
  - run on a single CPU core
  - operate on a single partition, which is loaded entirely to memory

Choosing partition count directly affects number of tasks necessary to do a job.

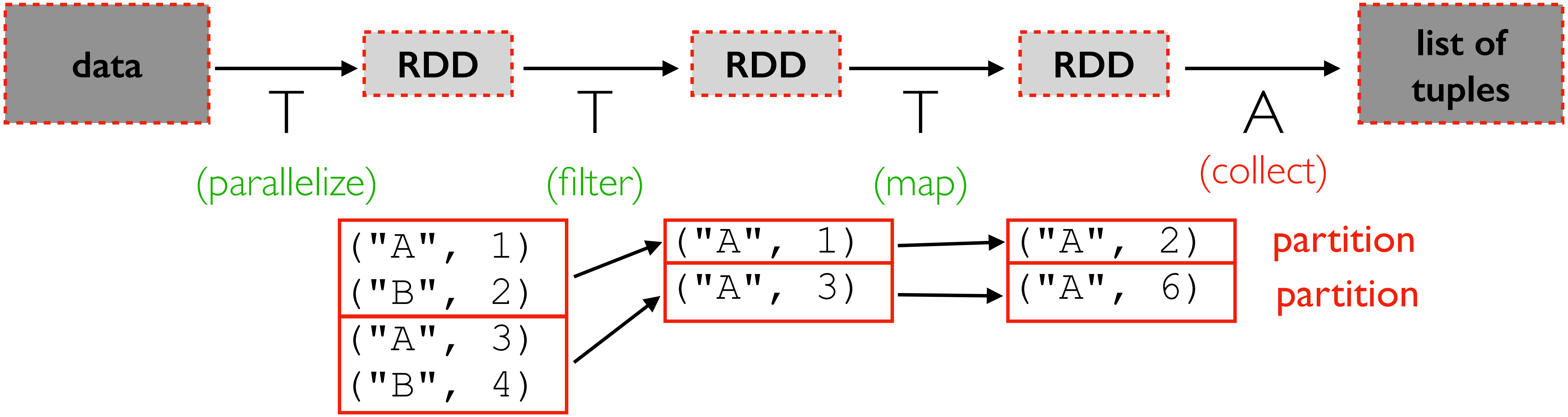
Advantages of larger partitions

- less overhead in starting tasks

Disadvantages of larger partitions

- might not have enough to use all cores that are available
- harder to balance work evenly
- uses more memory per partition at a time (is there enough)?

Repartitioning



Many operations (like filter and map) output the same number of partitions as they receive

- if the data is growing/shrinking a lot after a transformation, you might want to change the partition count
- rdd.getNumPartitions()   # check how many
- rdd2 = rdd.repartition(10)   # change how many (might be expensive!)

Examples:

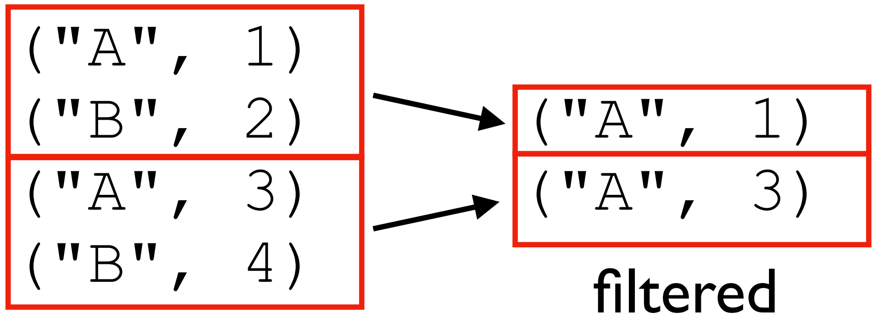
```
table.filter(onlyA).map(mult2).collect()
table.filter(onlyA).repartition(1).map(mult2).collect()
```

Transformations: Narrow vs. Wide

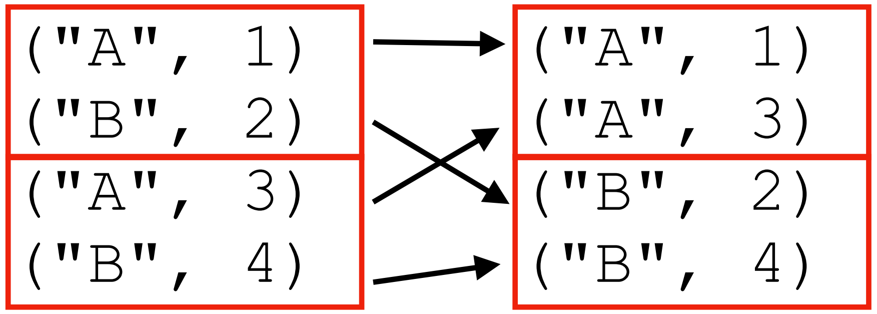
"Any transformation where a single output partition can be computed from a single input partition is a *narrow* transformation." (Learning Spark book).

Others are *wide* transformations.

```
data = [("A", 1),("B", 2),("A", 3),("B", 4)]
table = sc.parallelize(data, 2)
filtered = table.filter(lambda row: row[0] == "A")
ordered = table.sortBy(lambda row: row[0])
```



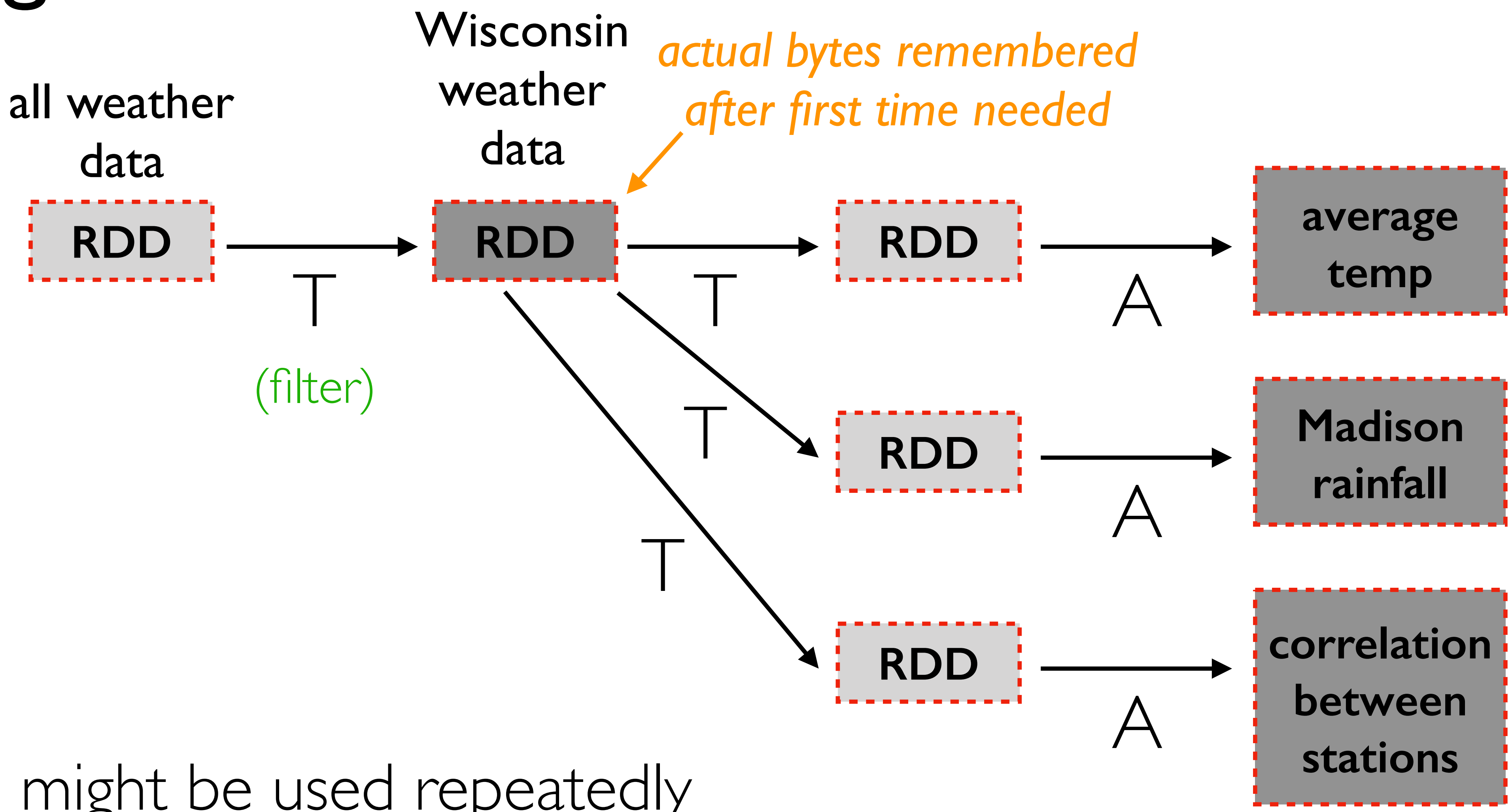
filtered
(narrow)



ordered
(wide)

Wide transformations often require network resources. Unless all input partitions are on the same machine, some will need to be transferred.

Caching



Some RDDs might be used repeatedly

- Spark might cache a copy of the computed results
- OR we can tell it to

```
all_weather = ...
wi_weather = all_weather.filter(...)
wi_weather.cache()
...
wi_weather.unpersist() # stop caching
```

Outline: MapReduce and Spark

Data Lakes

Hadoop MapReduce

Spark

- Resilient Distributed Datasets (RDDs)
- SQL and DataFrames
- Deployment

Spark SQL and DataFrames

Spark SQL

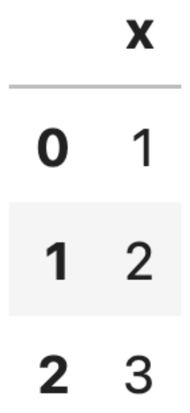
- builds on RDDs
- write standard queries (ANSI SQL:2003)
- automatic optimization possible because Spark knows what transformations are doing

DataFrame API

- builds on Spark SQL (so also optimizable)
- DataFrames are immutable because RDDs are immutable
- DataFrames aren't materialized in memory by default. Contents are computed as needed in parallel across many workers.

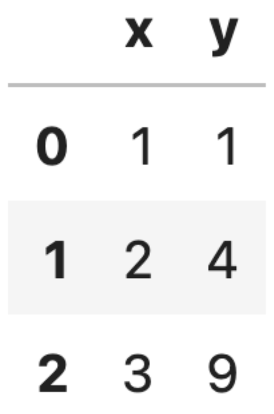
DataFrames: Pandas vs. Spark

```
pandas_df = pd.DataFrame({"x": [1,2,3]})
```



	x
0	1
1	2
2	3

```
# pandas DFs are mutable
pandas_df["y"] = pandas_df["x"] ** 2
```



	x	y
0	1	1
1	2	4
2	3	9

```
spark_df = spark.createDataFrame(pandas_df)

# could convert back:
# spark_df2.toPandas()

# cannot add column to immutable Spark DF
# can only create a new DF
spark_df2 = spark_df.withColumn("y", col("x") ** 2)
```

Outline: MapReduce and Spark

Data Lakes

Hadoop MapReduce

Spark

- Resilient Distributed Datasets (RDDs)
- SQL and DataFrames
- Deployment

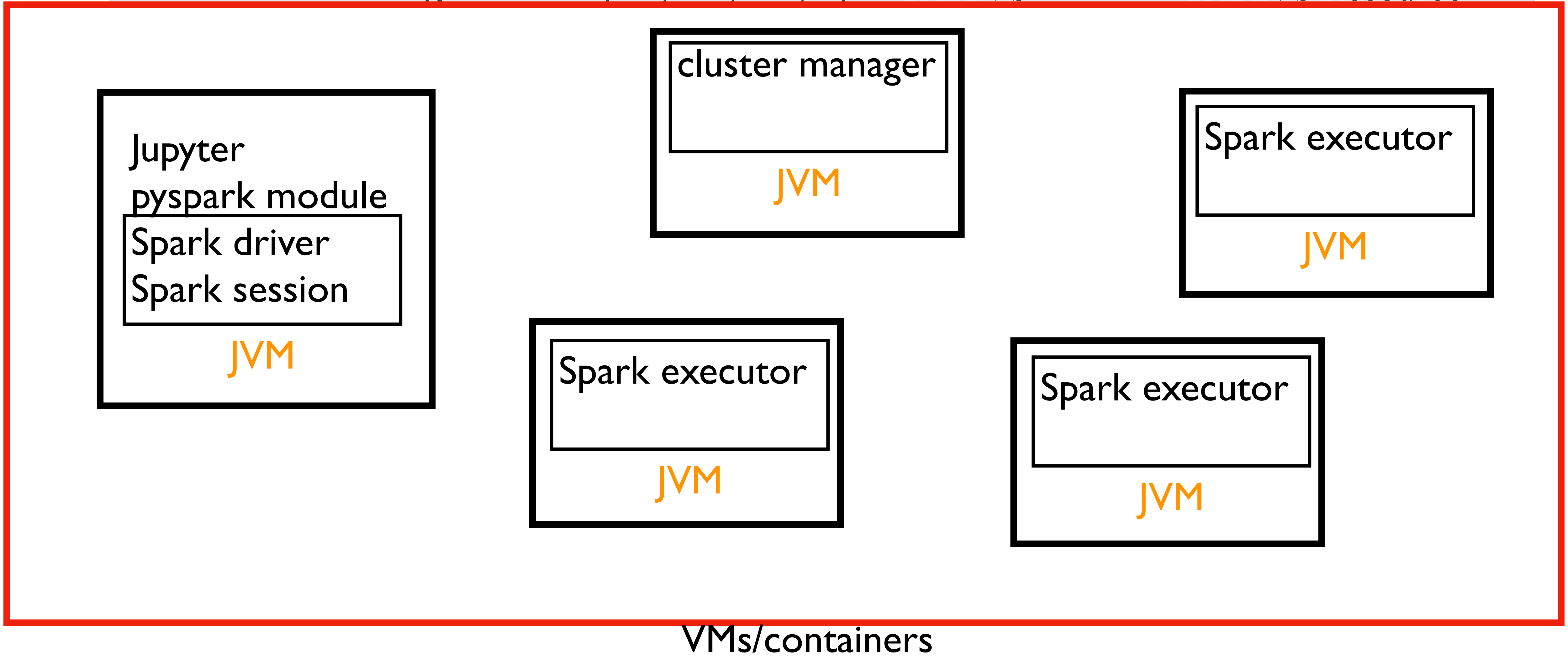
Deployment

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Table 1-1 from *Learning Spark* book

Deployment

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own	Can be allocated arbitrarily to any host in the cluster
		YARN's	YARN's Resource



Jupyter
pyspark module
Spark driver
Spark session
JVM

cluster manager
JVM

Spark executor
JVM

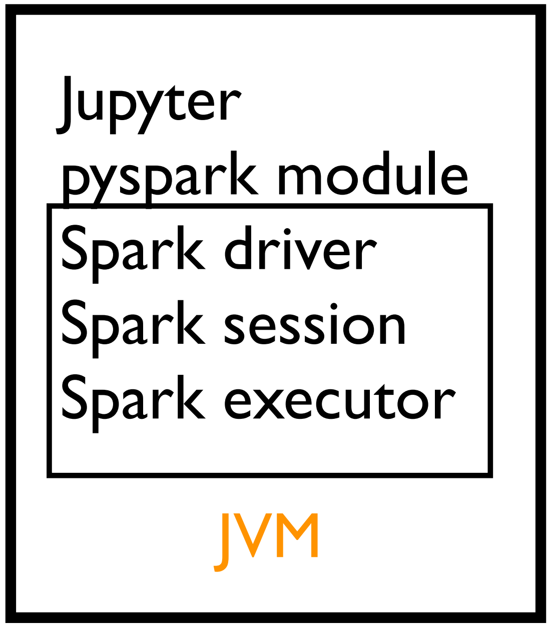
Spark executor
JVM

Spark executor
JVM

VMs/containers

Deployment

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own	Can be allocated arbitrarily to any host in the cluster
		YARN's	YARN's Resource



this mode is fine for testing/development, but misses the benefits of distributed computing

VMs/containers