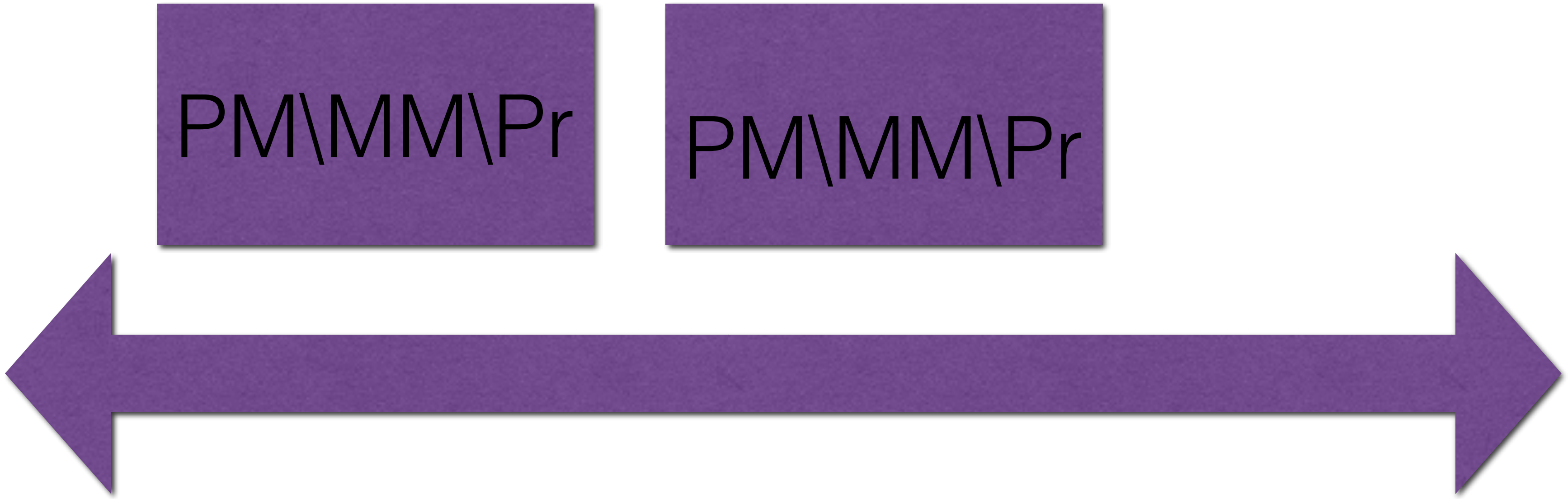
Module 3 & 4

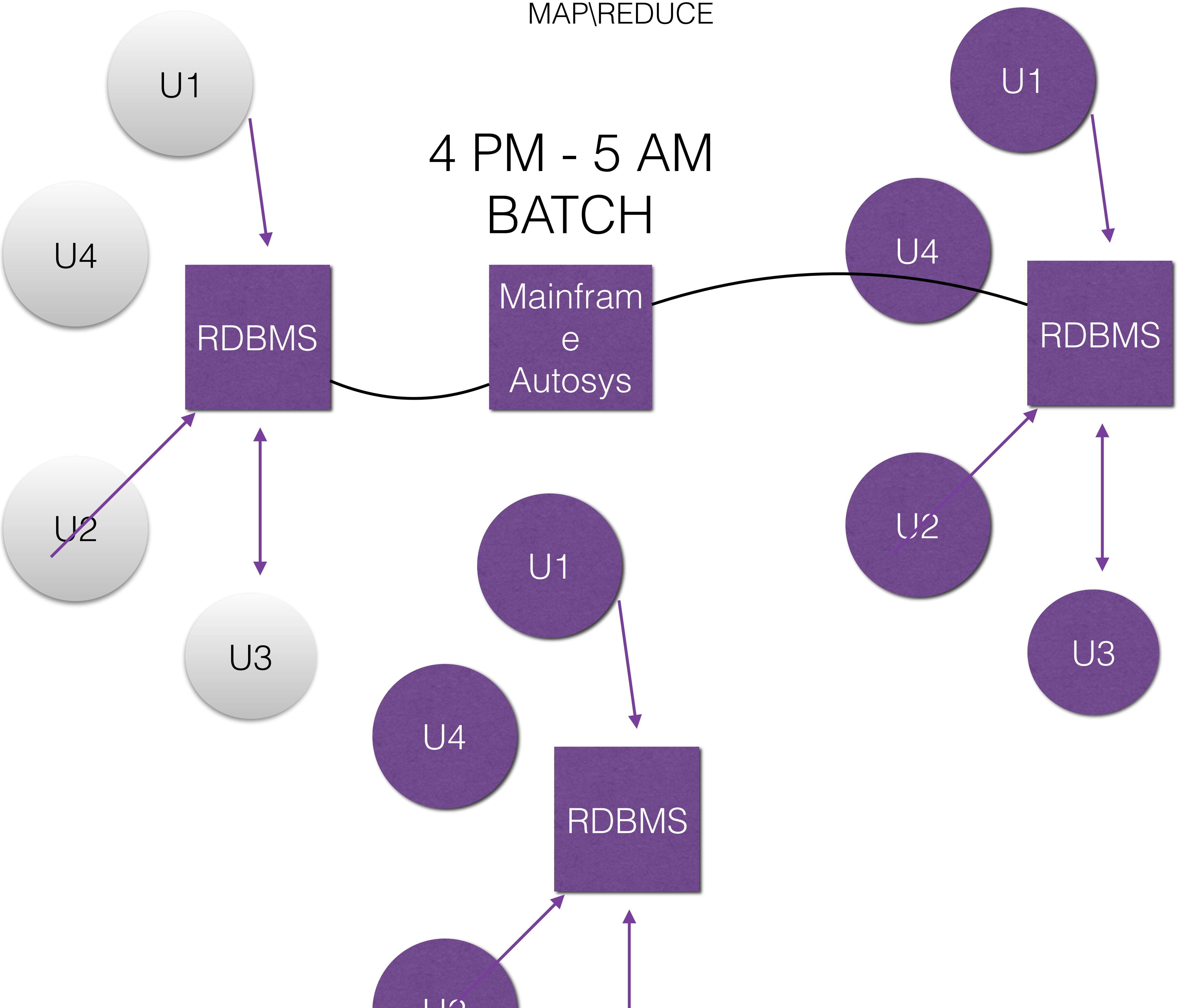


PM\MM\Pr

PM\MM\Pr

JOB1 - Get Data
JOB2 - Process & generate o/p
MAP\REDUCE

4 PM - 5 AM
BATCH



What is Spark?

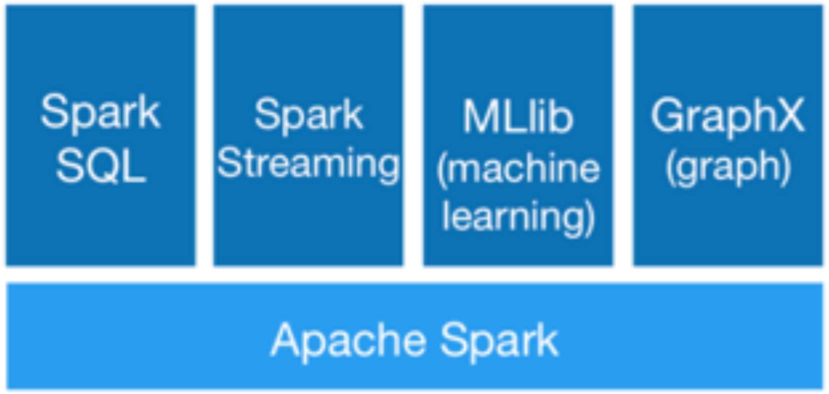
- Apache Spark™ is a fast and general engine for large-scale data processing.
- It guarantees:

A. ***SPEED:*** Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

- Apache Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.

B. ***EASE OF USE****: Write applications quickly in Java, Scala, Python, R. Spark offers over* ***80 high-level operators*** *that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.*

C. ***GENERALITY:*** *Combine SQL, streaming, and complex analytics. Spark powers a stack of libraries including* ***SQL and DataFrames****,* ***MLlib for machine learning****,* ***GraphX****, and* ***Spark Streaming.*** *You can combine these libraries seamlessly in the same application.*



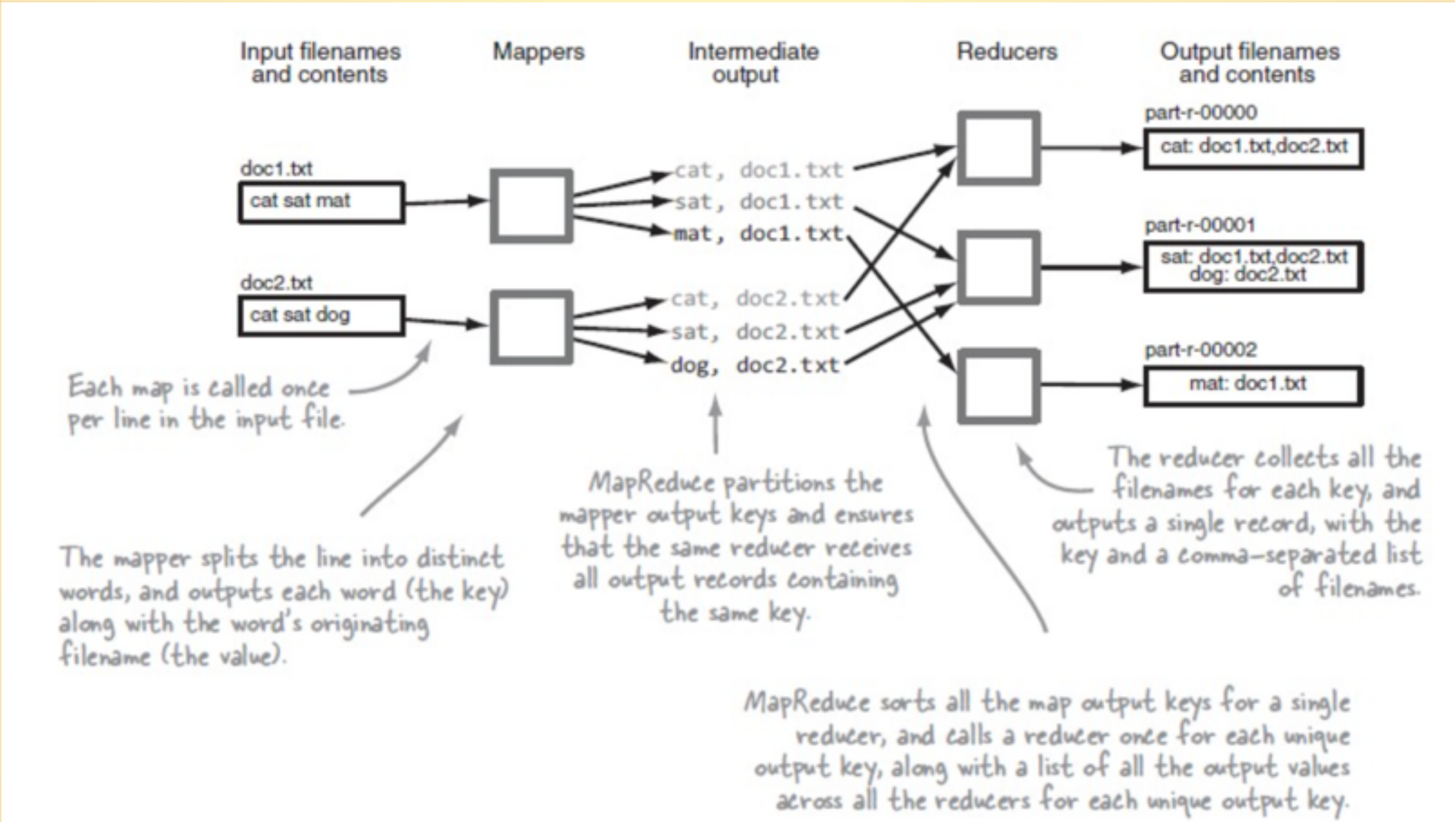
D. ***RUNS EVERYWHERE:*** *Spark runs on* ***Hadoop****,* ***Mesos, standalone****, or in the* ***cloud****. It can access* ***diverse data sources including HDFS, Cassandra, HBase, and S3****.*

SPARK with Scala

SCALA doc for available SPARK Libraries

- https://spark.apache.org/docs/preview/api/scala/index.html#package

Map Reduce Architecture



Each map is called once per line in the input file.

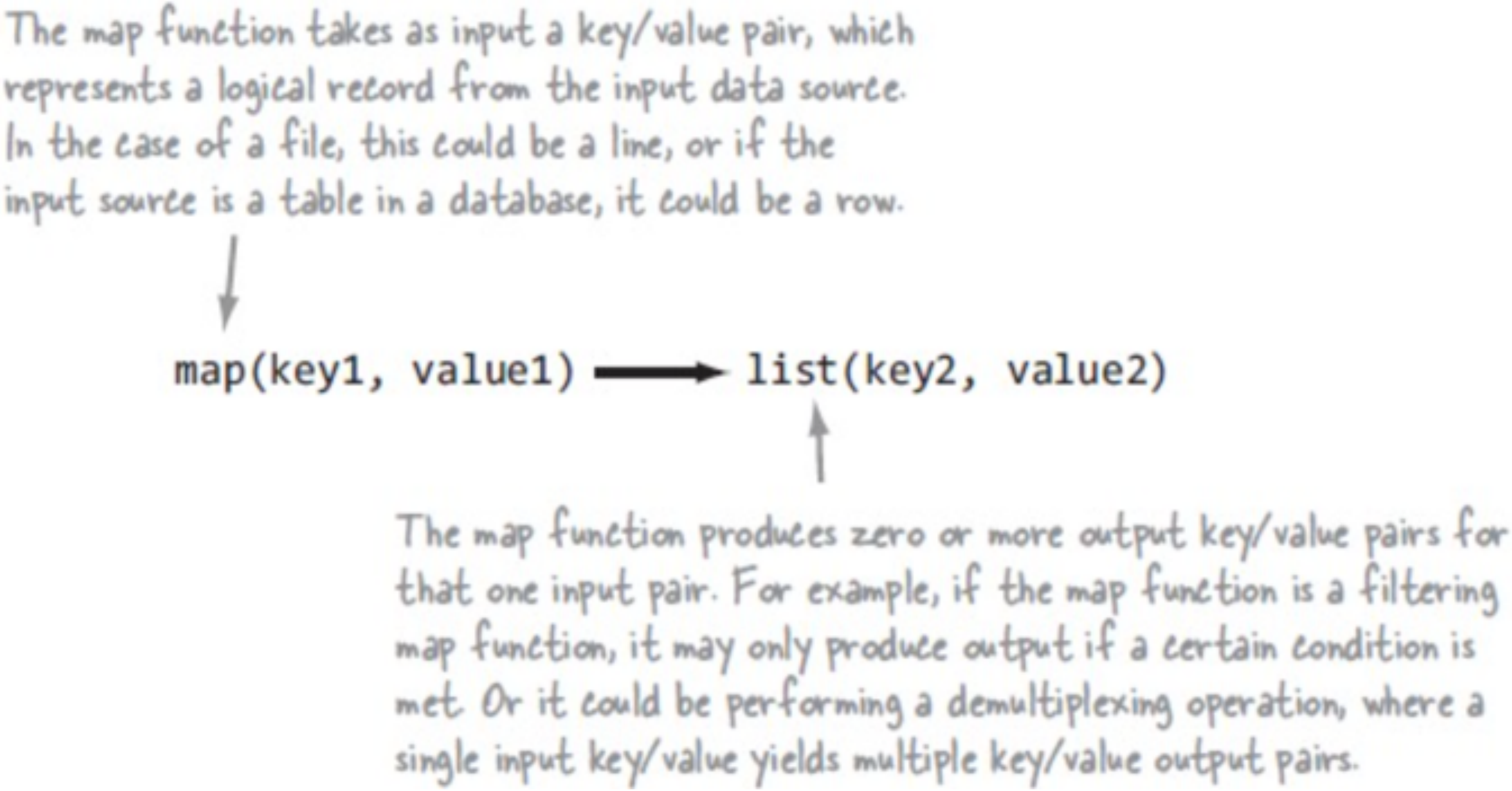
The mapper splits the line into distinct words, and outputs each word (the key) along with the word's originating filename (the value).

MapReduce partitions the mapper output keys and ensures that the same reducer receives all output records containing the same key.

The reducer collects all the filenames for each key, and outputs a single record, with the key and a comma-separated list of filenames.

MapReduce sorts all the map output keys for a single reducer, and calls a reducer once for each unique output key, along with a list of all the output values across all the reducers for each unique output key.

Map Output

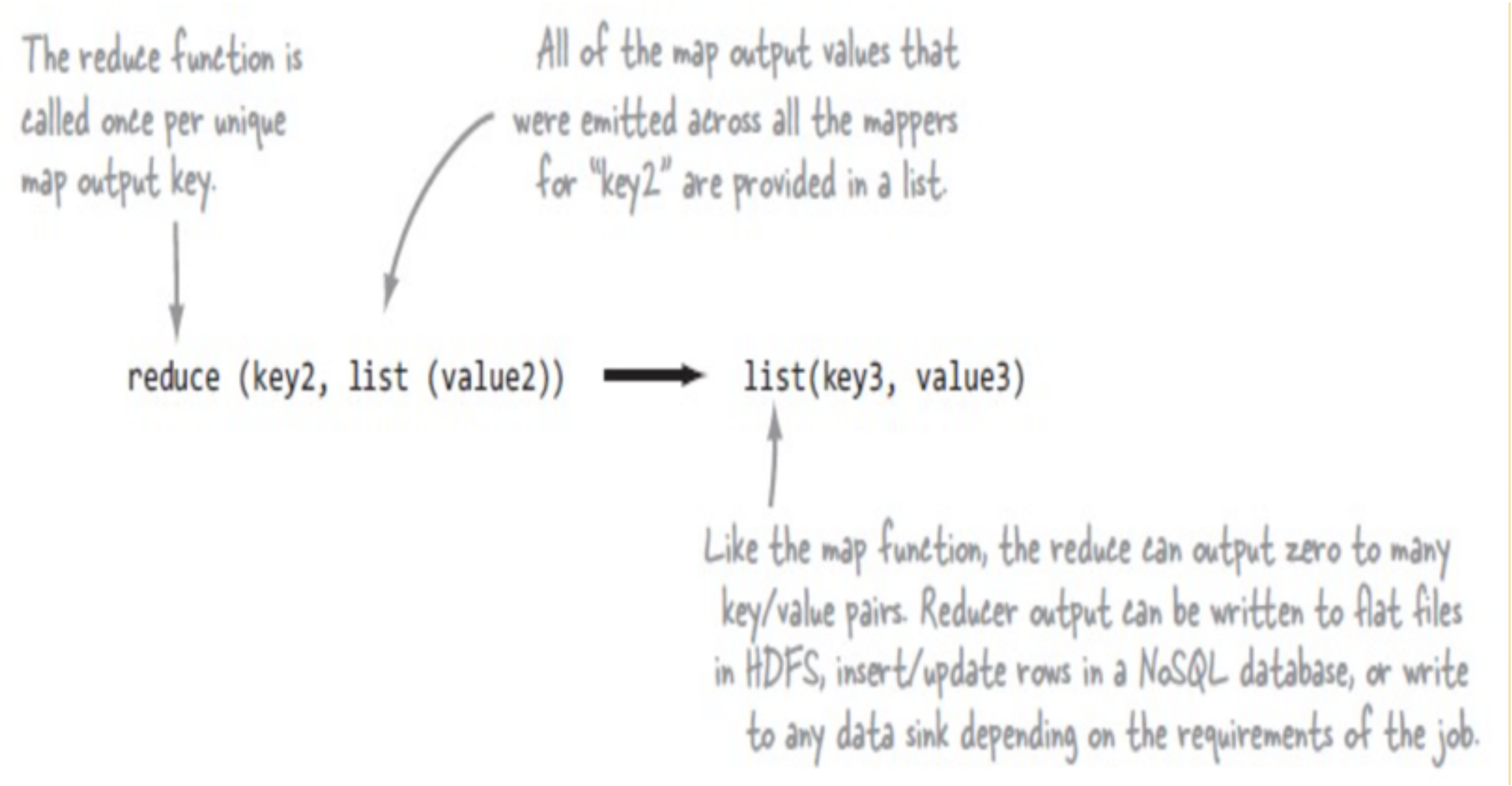


The map function takes as input a key/value pair, which represents a logical record from the input data source. In the case of a file, this could be a line, or if the input source is a table in a database, it could be a row.

```
map(key1, value1) ——→ list(key2, value2)
```

The map function produces zero or more output key/value pairs for that one input pair. For example, if the map function is a filtering map function, it may only produce output if a certain condition is met. Or it could be performing a demultiplexing operation, where a single input key/value yields multiple key/value output pairs.

Reducer Output



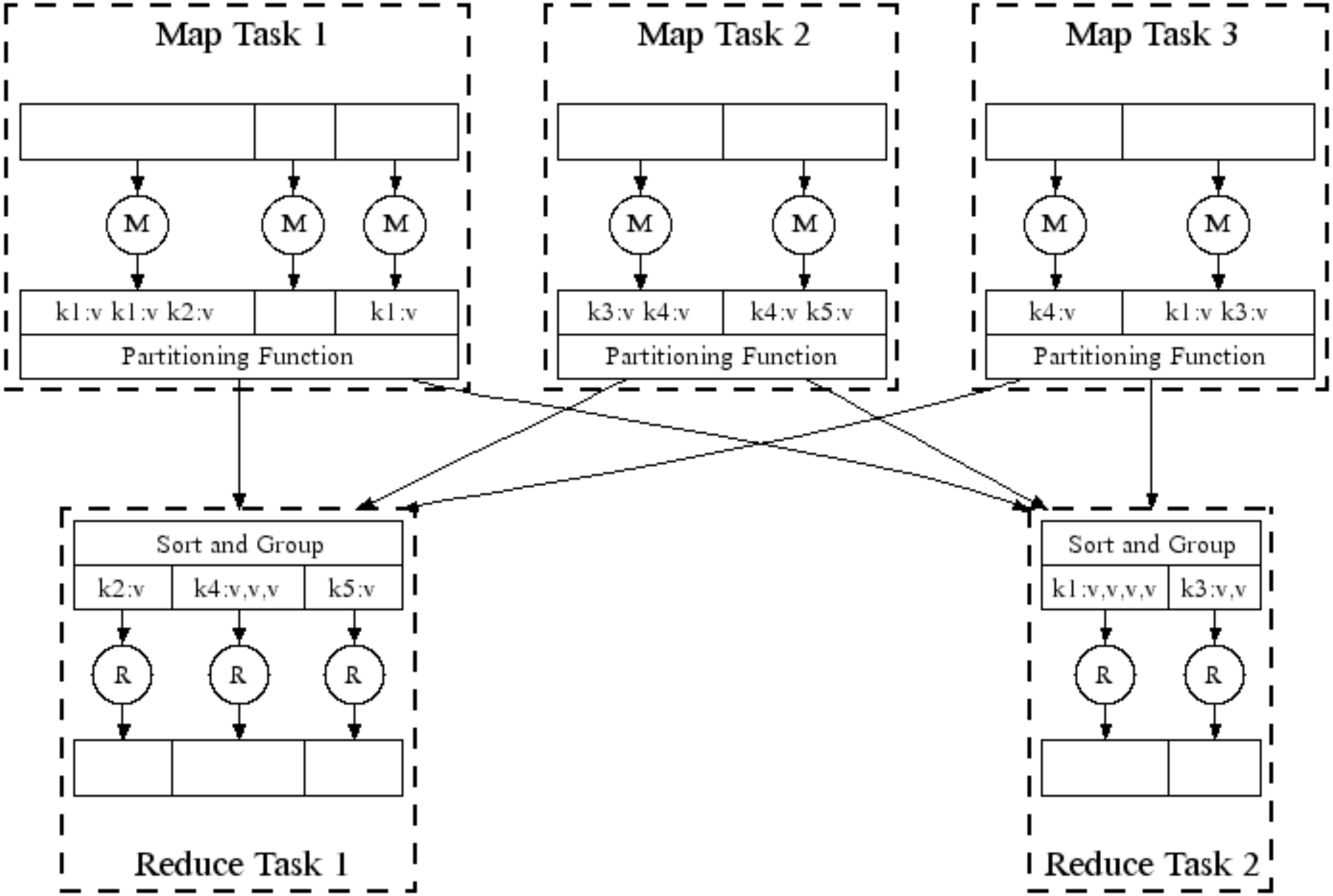
The reduce function is called once per unique map output key.

All of the map output values that were emitted across all the mappers for "key2" are provided in a list.

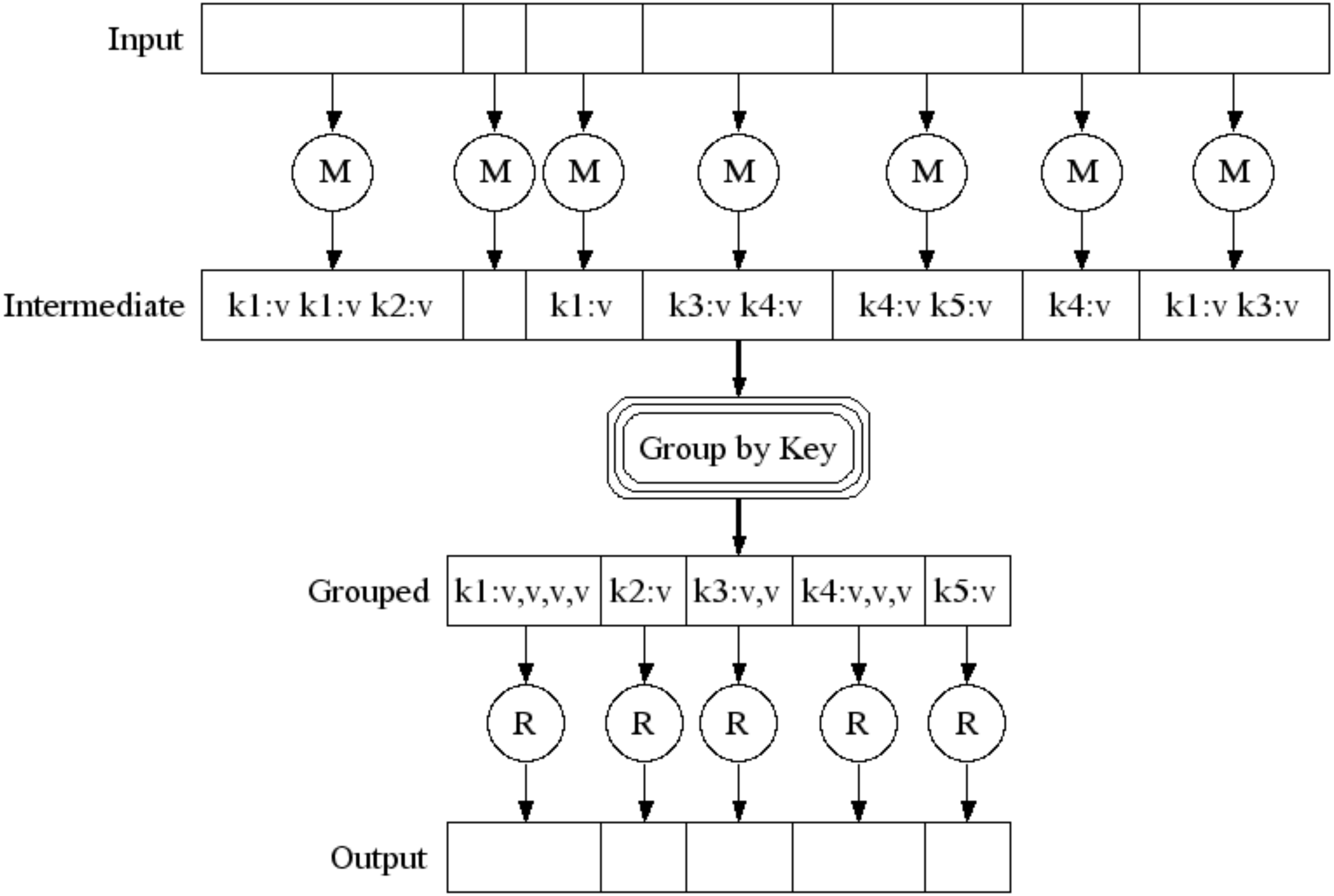
```
reduce (key2, list (value2)) ⟶ list(key3, value3)
```

Like the map function, the reduce can output zero to many key/value pairs. Reducer output can be written to flat files in HDFS, insert/update rows in a NoSQL database, or write to any data sink depending on the requirements of the job.

Parallel Execution



Map Reduce Execution



Input

M M M M M M M

Intermediate: k1:v k1:v k2:v | | k1:v | k3:v k4:v | k4:v k5:v | k4:v | k1:v k3:v

Group by Key

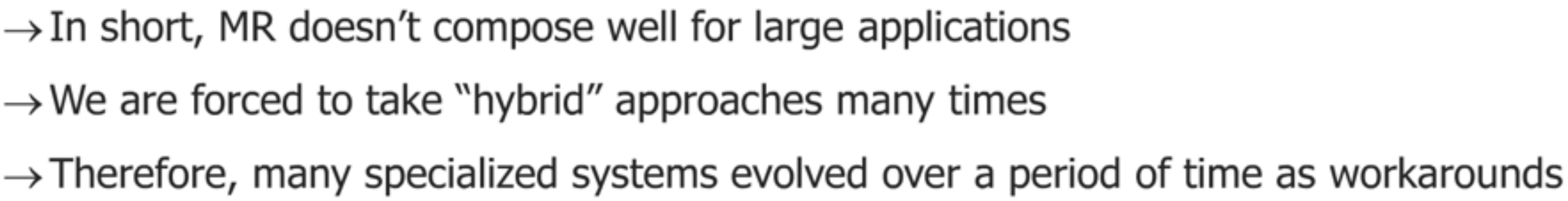
Grouped: k1:v,v,v,v | k2:v | k3:v,v | k4:v,v,v | k5:v

R R R R R

Output

Map Reduce Limitations

- Difficult to Program an algo in Native Map Reduce
- Not good for low latency processing as:
    - Lots of Disk I\O
    - Processing in Stages: Reducer cant start until Mapper is complete
    - Not fit for small batch.
    - Many categories of algos not supported or are very difficult to implement: For e.g., the ones which need iterations
- The problem of “***stragglers***” (slow workers)
    - Other jobs consuming resources on machine
    - Bad disks with soft errors transfer data very slowly
    - Weird things: processor caches disabled (!!)

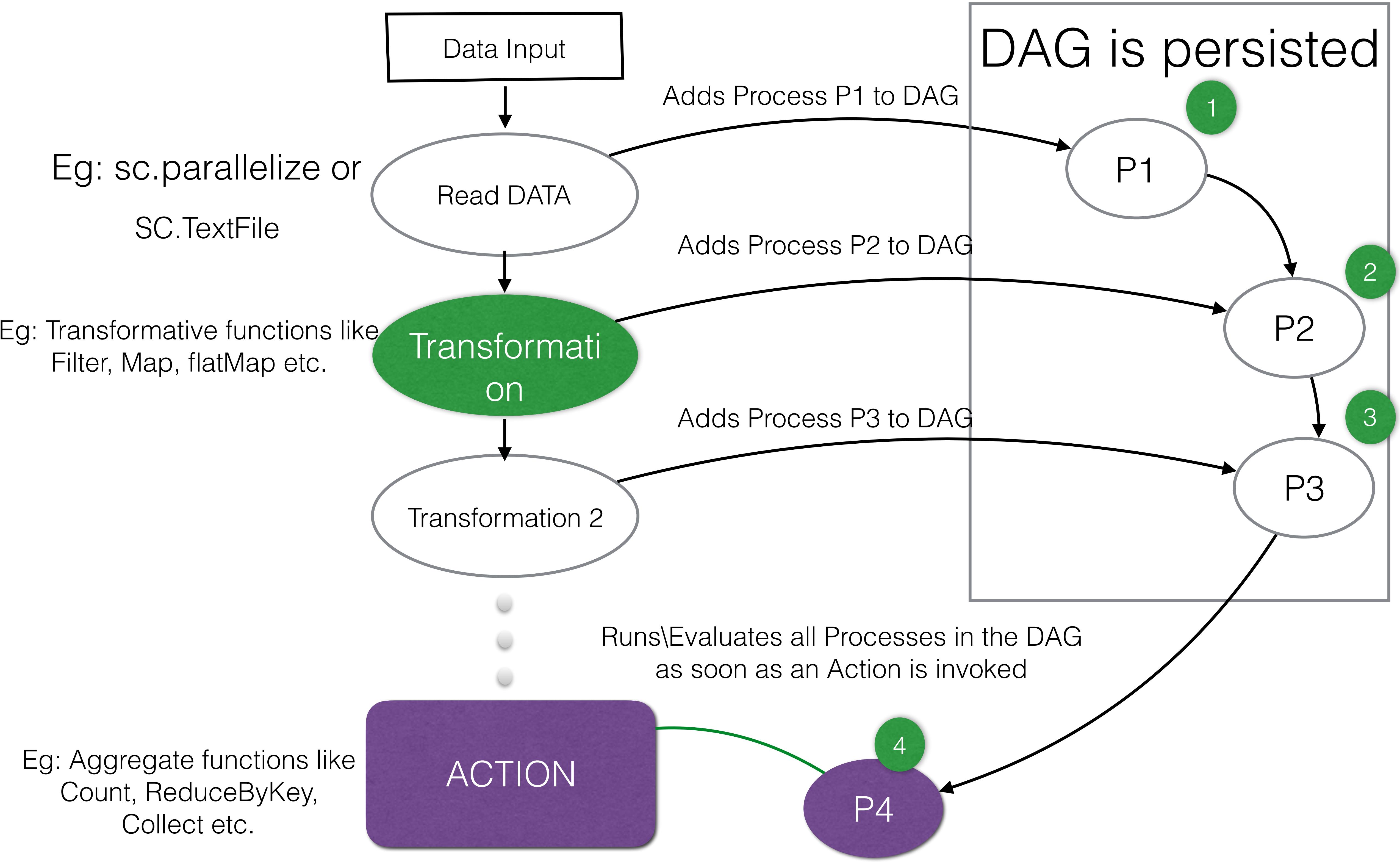


→ In short, MR doesn’t compose well for large applications

→ We are forced to take “hybrid” approaches many times

→ Therefore, many specialized systems evolved over a period of time as workarounds

What is a DAG - Direct Acyclic Graph Depicting LAZY VAL



Data Input

Eg: sc.parallelize or

SC.TextFile

Read DATA

Eg: Transformative functions like Filter, Map, flatMap etc.

Transformati on

Transformation 2

Eg: Aggregate functions like Count, ReduceByKey, Collect etc.

ACTION

DAG is persisted

Adds Process P1 to DAG

Adds Process P2 to DAG

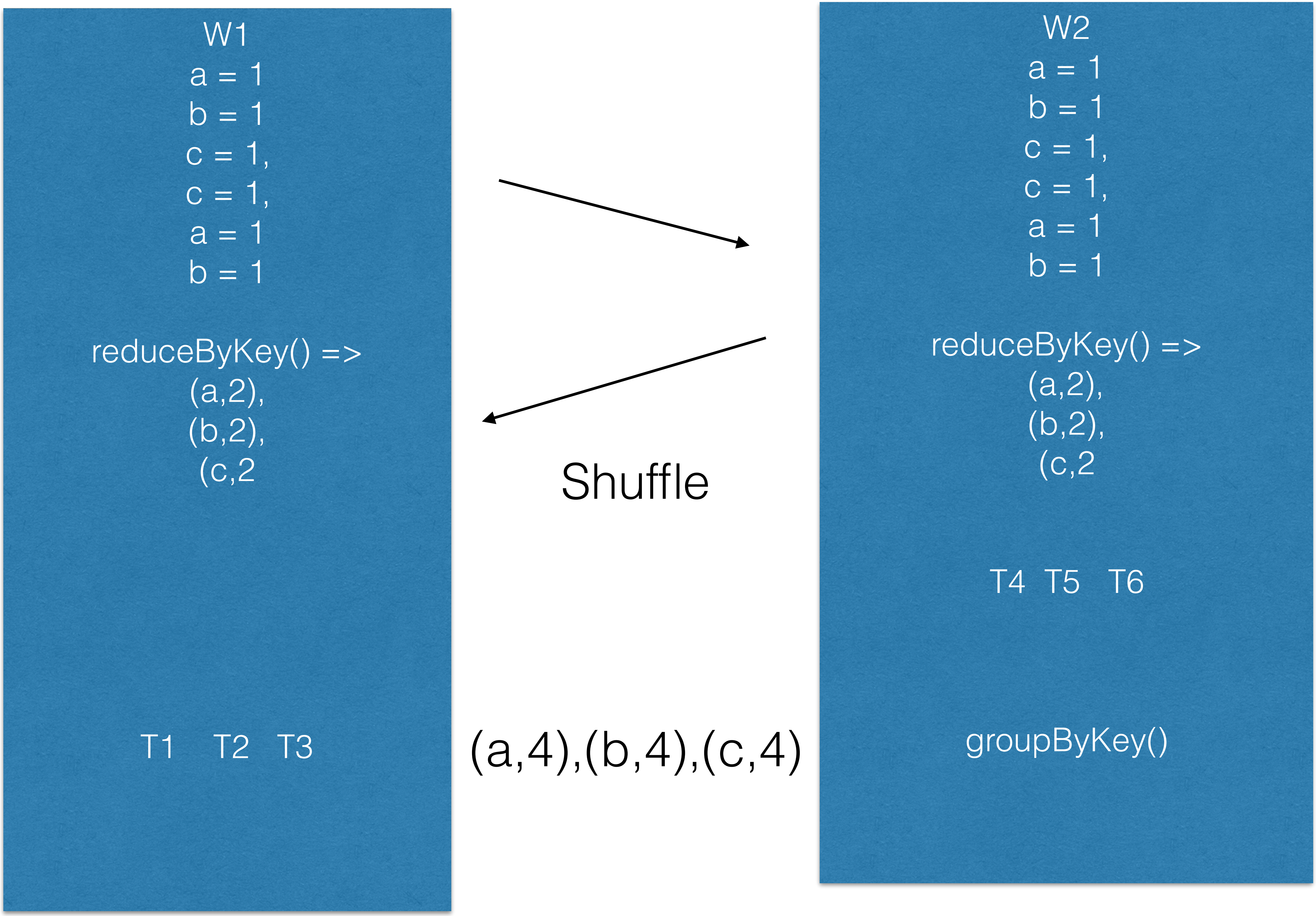
Adds Process P3 to DAG

Runs\Evaluates all Processes in the DAG as soon as an Action is invoked

RDD is:

— Smallest unit of Abstracted data set in spark

— it is partitioned so that it can be worked upon in parallel



W1

a = 1

b = 1

c = 1,

c = 1,

a = 1

b = 1

reduceByKey() =>

(a,2),

(b,2),

(c,2

T1 T2 T3

Shuffle

(a,4),(b,4),(c,4)

W2

a = 1

b = 1

c = 1,

c = 1,

a = 1

b = 1

reduceByKey() =>

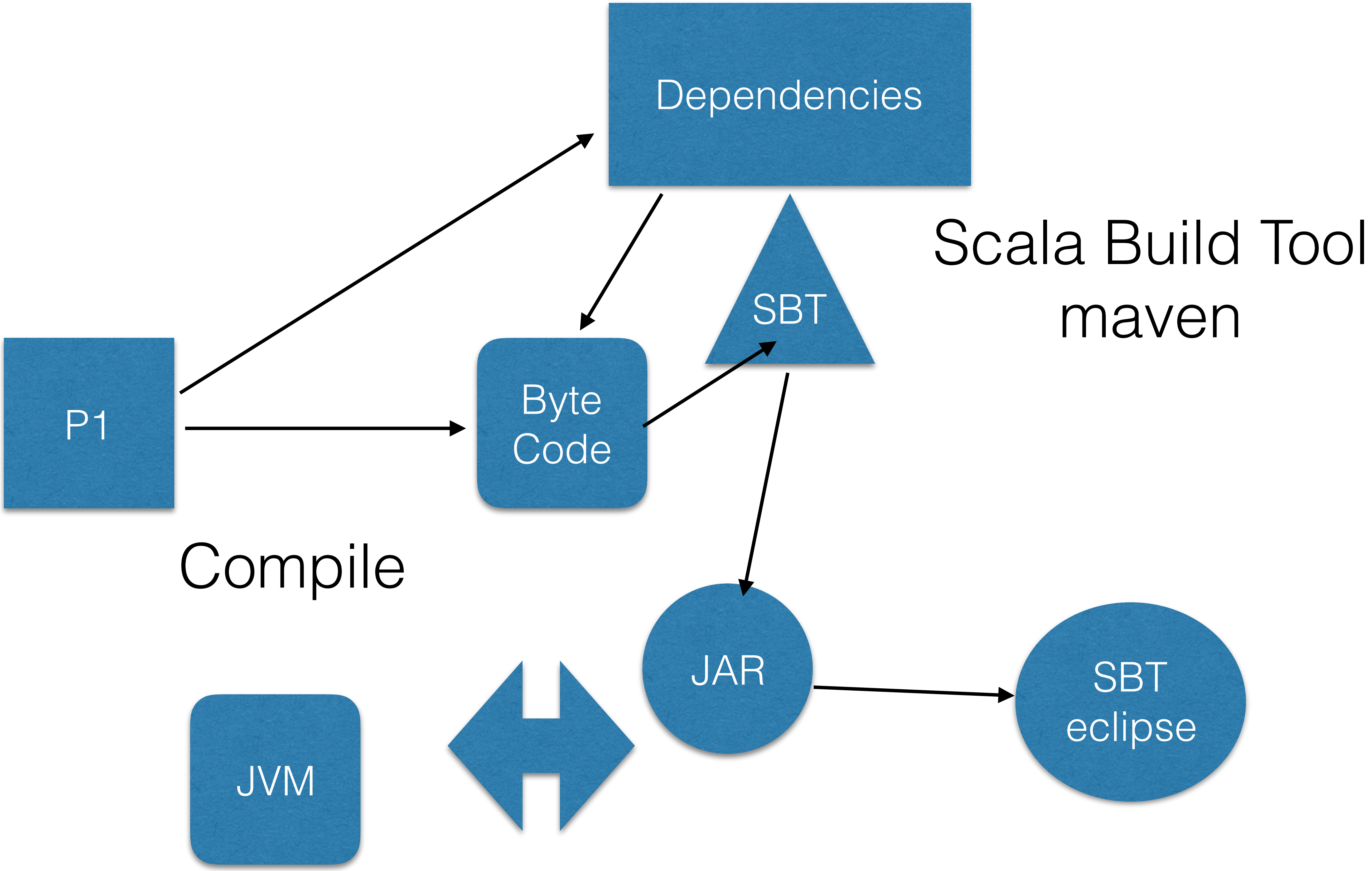
(a,2),

(b,2),

(c,2

T4 T5 T6

groupByKey()



Dependencies

Scala Build Tool
maven

SBT

P1

Byte Code

Compile

JAR

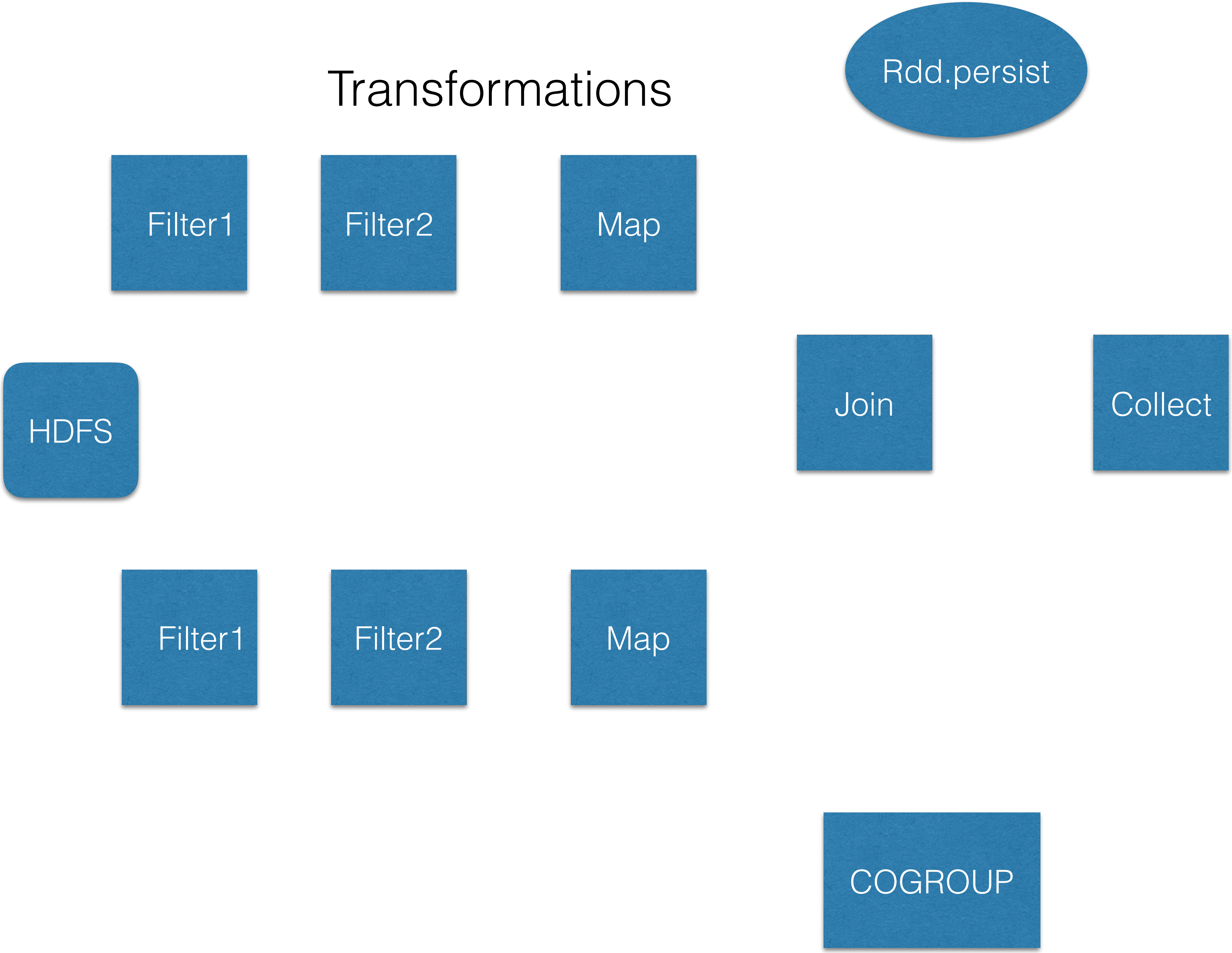
SBT eclipse

JVM

build.sbt

1) sbt compile
2) sbt package
3) sbt eclipse

Transformations



Rdd.persist

Filter1

Filter2

Map

HDFS

Join

Collect

Filter1

Filter2

Map

COGROUP