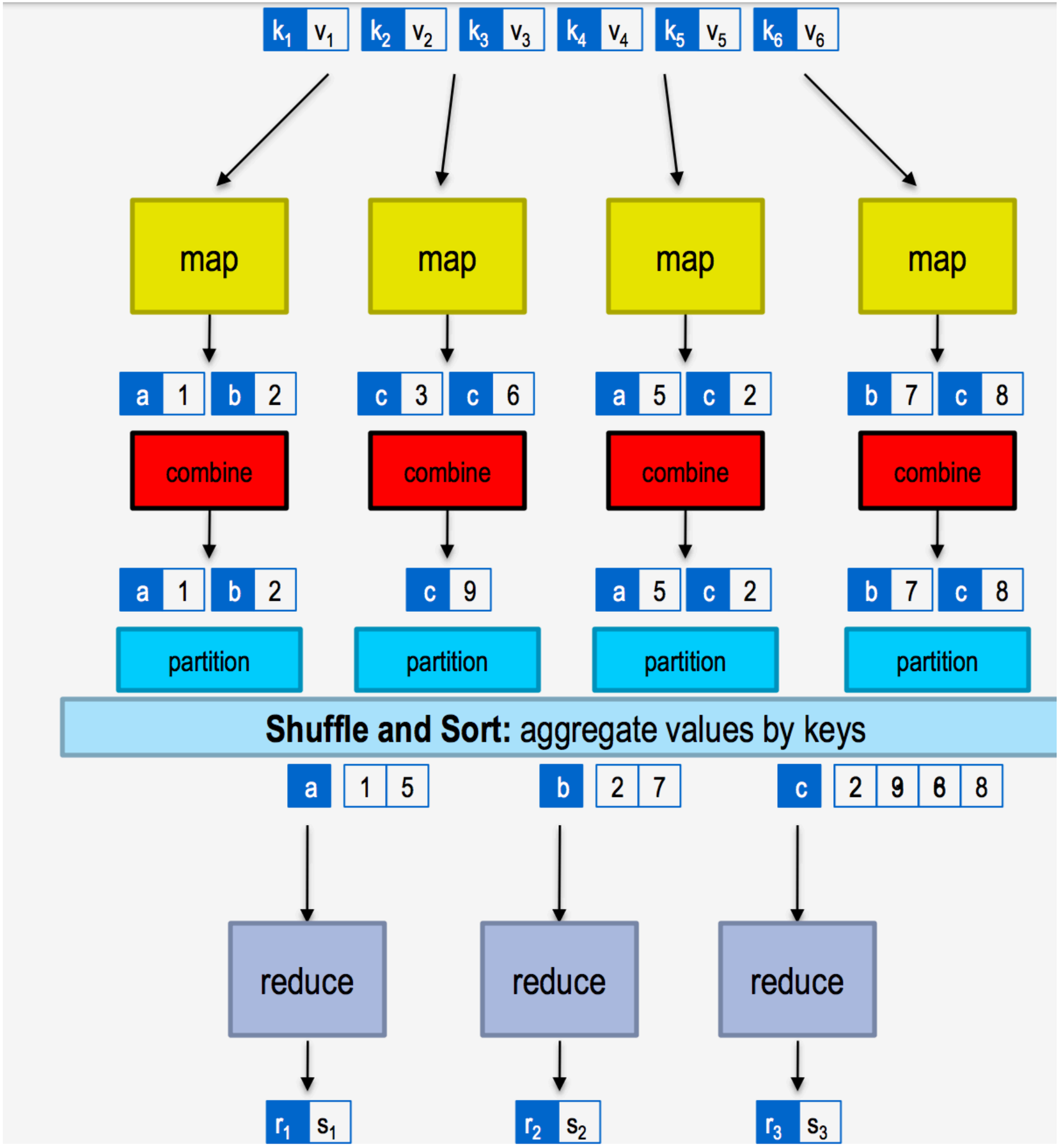
Why MapReduce

- We want to process large amount of data (Terabytes, Petabytes, and more …)
- We want to parallelize across hundred/thousands of CPUs
- MapReduce takes care of data distribution, and executing user program in parallel


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Map Output

- The output from the map is not the final output
  - However in some cases it could be
  - so the output from the map is written on the local disk of the machine running the map function
- while map is working, it keeps data in memory
- if the size in memory exceeds a threshold
- then output will be written on disk but before that
  - run the combiner to reduce the amount of data

Shuffle & Sort

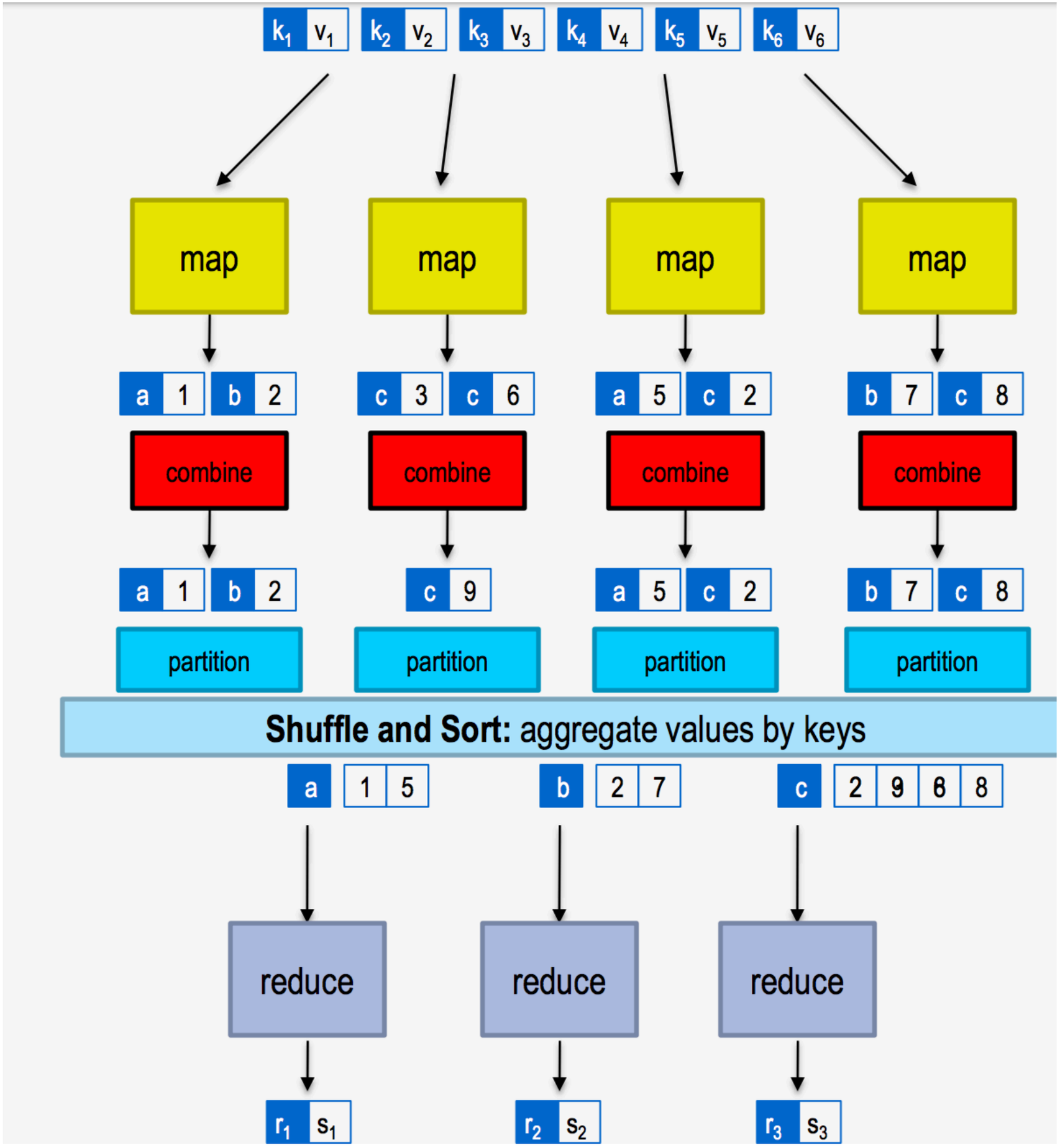
- The process by which the system performs the sort & transfer of map output to the reduce as input
- Sort covers the sort and grouping of intermediate results
- Sorting takes care of
  - grouping all values by the same key
  - sorting based on the keys, not values
- Sorting is important because if the key is different we know that there is a new key (new key-value pairs) and reduce can start

Partitioning

- Is responsible for dividing the intermediate key-value pairs (from the map) based on the keys, and assigning them to reducers

- all key-values of the same key will end up at the same partition

- default partitioner is the hash(key) % num-of-reducers


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Custom Partitioner

- is a process that allows you to store the results in different reducers, based on the user condition
- By setting a partitioner to partition by the key, we can guarantee that,
  - records for the same key will go to the same reducer.
  - only one reducer receives all the records for that particular key

Number of Map & Reduce tasks

- Number of map task depends on the number of blocks for the input data
    - To increase parallelism, people can decrease the block size
    - If the input data is split into 1000 blocks, then we can have 1000 map task
    - one node can execute multiple map tasks
- Number of reducers can be set in the configuration

Implementation

- One master node in the cluster
- Master partitions input files into M splits
  - M map tasks, equal to M splits
    - one worker might get multiple splits
- Master assigns workers to process the M map tasks
- Workers write their output on their local disk
  - combine (if configured), then partition into R partitions
- Master assigns workers to R reduce tasks
- Reduce workers write to HDFS

Interesting Implementation Details

- Worker Failure
  - master node pings workers periodically
  - if worker is down, then assign its splits to all other running workers
    - this is good for load balancing

Interesting Implementation Details

- Backup tasks
  - an optimization implemented by Hadoop to deal with slow workers
- There is a barrier between map phase & reduce phase
  - the map phase is as fast as slowest map task
- Finishing the job depends on the slowest reduce phase
- The machine that takes unusual time to finish its task is called straggler
- when a straggler machine is discovered then the framework start an identical task on different machine
  - then simply use the result from the machine that finishes first

Understanding Data Transformation

- From start to finish, there are four fundamental transformation. Data is

    1. transformed from input files and fed to the mappers (workers running the map tasks)

    2. transformed by the mappers

    3. sorted, grouped, and presented to the reducers (workers running the reduce tasks)

    4. transformed by reducers and written to output files

Write a MapReduce Program

- What do we need:
  - Input format to split data
  - a mapper class that contains the map function
  - a reducer class that contains the reduce function
  - Output format to produce final output
- Hadoop contains an already built-in classes for formatting the input & the output; for example TextInputFormat and TextOutputFormat
  - users can also create their own input/output format class
- The InputFormat & the OutputFormat are defined in the job configuration

Pay attention to key-value types

- It is important to use appropriate types for keys & values
- At each stage, make sure that inputs and outputs match up
  - types of input keys & values for the mappers must be the same as the types of the output keys & values generated by the input format
  - types of output keys & values from mappers must match the types of input keys & values for reducer
  - types of output keys & values from the reducers must match the types of keys & values of the output format

MapReduce Code

Word Count example