MapReduce

CS 4740: Cloud Computing

Fall 2024

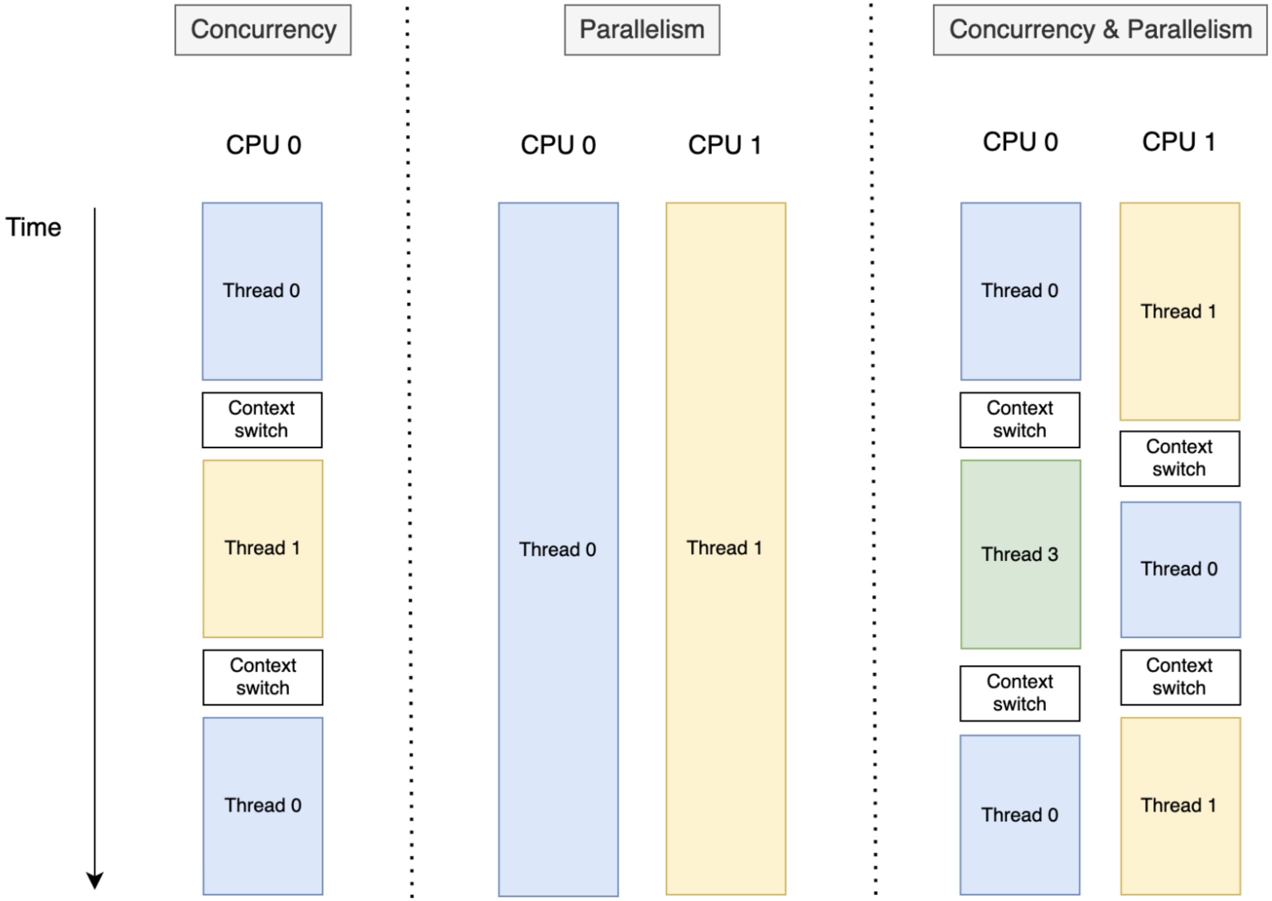
Lecture 4

Yue Cheng

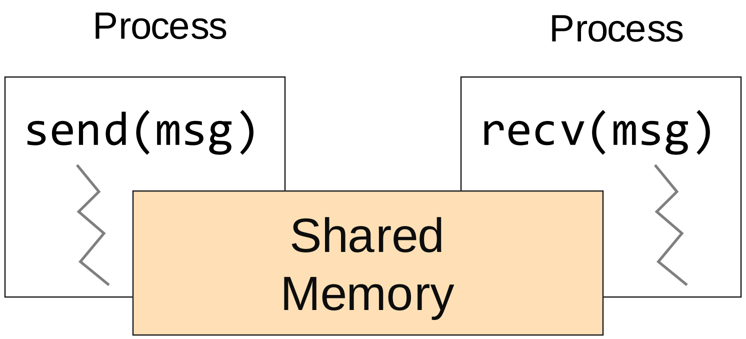
Some material taken/derived from:
- Princeton COS-418 materials created by Michael Freedman and Wyatt Lloyd.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

Recap: Parallelism vs. concurrency

"Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once."

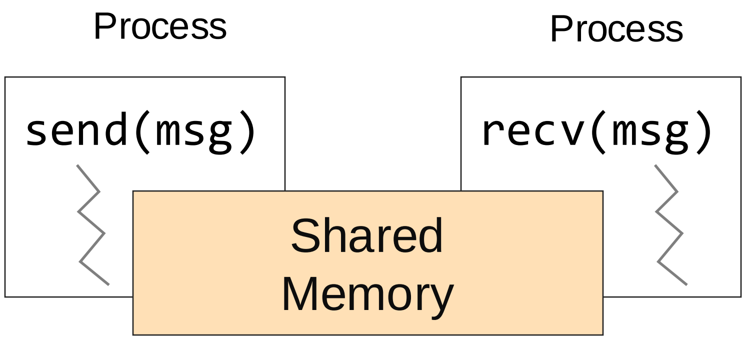


Recap: Shared memory



- Shared memory: multiple processes to share data via memory
- Applications must locate and and map shared memory regions to exchange data

Recap: Shared memory vs. Message passing



Process

Process

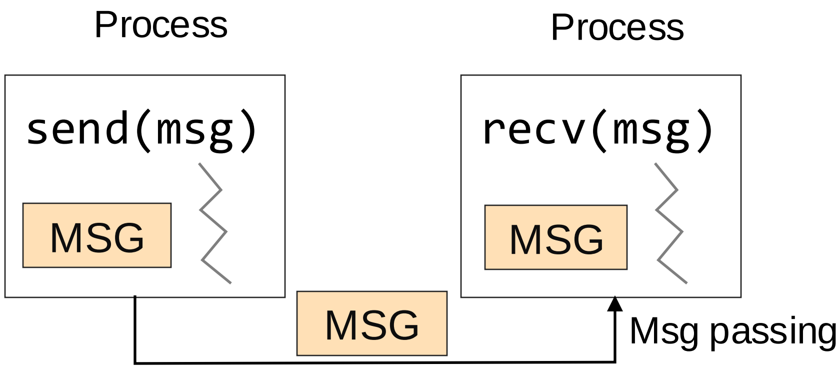
`send(msg)`

`recv(msg)`

Shared Memory

- Shared memory: multiple processes to share data via memory

- Applications must locate and and map shared memory regions to exchange data



Process

Process

`send(msg)`

`recv(msg)`

MSG

MSG

MSG

Msg passing

- Message passing: exchange data explicitly via message passing

- Application developers define protocol and exchanging format, number of participants, and each exchange

Recap: Shared memory vs. Message passing

- Easy to program; just like a single multi-threaded machines
- Hard to write high-performance apps:
  - Cannot control which data is local or remote (remote mem. access much slower)
- Hard to mask failures

Recap: Shared memory vs. Message passing

- Easy to program; just like a single multi-threaded machines
- Hard to write high-performance apps:
  - Cannot control which data is local or remote (remote mem. access much slower)
- Hard to mask failures

- Message passing: can write very high-performance apps
- Hard to write apps:
  - Need to manually decompose the app, and move data
- Need to manually handle failures

Shared memory: Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX (e.g., Linux) OSes

Shared memory: Pthread

```
void *myThreadFun(void *vargp) {
      sleep(1);
      printf(“Hello world\n”);
      return NULL;
}

int main() {
      pthread_t thread_id_1, thread_id_2;
      pthread_create(&thread_id_1, NULL, myThreadFun, NULL);
      pthread_create(&thread_id_2, NULL, myThreadFun, NULL);
      pthread_join(thread_id_1, NULL);
      pthread_join(thread_id_2, NULL);
      exit(0);
}
```

Message passing: MPI

- MPI – Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing

- Portable: one standard, many implementations
  - Available on almost all parallel machines in C and Fortran
  - De facto standard platform for the HPC community

Message passing: MPI

```
int main(int argc, char **argv) {
      MPI_Init(NULL, NULL);

      // Get the number of processes
      int world_size;
      MPI_Comm_size(MPI_COMM_WORLD, &world_size);

      // Get the rank of the process
      int world_rank;
      MPI_Comm_rank(MPI_COMM_WORLD, *world_rank);

      // Print off a hello world message
      printf(“Hello world from rank %d out of %d processors\n”,
            world_rank, world_size);

      // Finalize the MPI environment
      MPI_Finalize();
}
```

Message passing: MPI

```
mpirun -n 4 -f host_file ./mpi_hello_world
```

```
int main(int argc, char **argv) {
      MPI_Init(NULL, NULL);

      // Get the number of processes
      int world_size;
      MPI_Comm_size(MPI_COMM_WORLD, &world_size);

      // Get the rank of the process
      int world_rank;
      MPI_Comm_rank(MPI_COMM_WORLD, *world_rank);

      // Print off a hello world message
      printf(“Hello world from rank %d out of %d processors\n”,
            world_rank, world_size);

      // Finalize the MPI environment
      MPI_Finalize();
}
```

MapReduce

The big picture (motivation)

- Datasets are too big to process using a single computer

The big picture (motivation)

- Datasets are too big to process using a single computer
- Good parallel processing engines are rare (back then in the late 90s)

The big picture (motivation)

- Datasets are too big to process using a single computer

- Good parallel processing engines are rare (back then in the late 90s)

- Want a parallel processing framework that:
  - is **general** (works for many problems)
  - is **easy to use** (no locks, no need to explicitly handle communication, no race conditions)
  - can **automatically parallelize** tasks
  - can **automatically handle machine failures**

Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, commodity hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete despite failures

Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, commodity hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete despite failures
- **Key question:** how can every Google engineer be imbued with the ability to write parallel, scalable, distributed, fault-tolerant code?
- **Solution:** abstract out the redundant parts
- Restriction: relies on job semantics, so restricts which problems it works for

Application: Word Count

```
cat data.txt
     | tr –s ‘[[:punct:][:space:]]’ ‘\n’
     | sort | uniq -c
```

```
SELECT count(word), word FROM data
     GROUP BY word
```

Deal with multiple files?

1. Compute word counts from individual files

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

3. Compute word count on merged outputs

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
    - Compute word counts from individual files
    - Collect results, wait until all finished

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

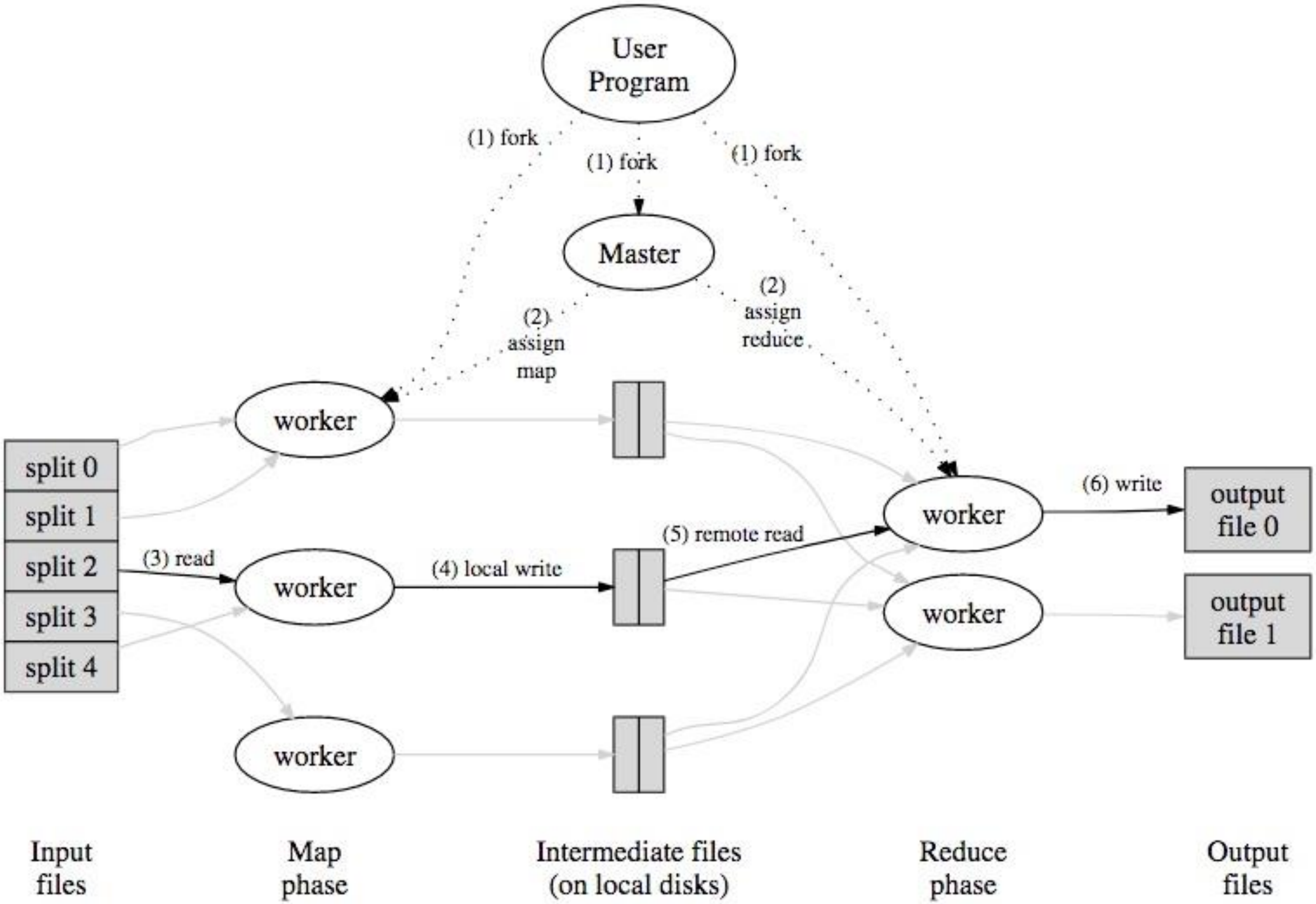
3. Compute word count on merged intermediates

MapReduce: Programming interface

- `map(k1, v1) → list(k2, v2)`
  - Apply function to (`k1, v1`) pair and produce set of intermediate pairs (`k2, v2`)

- `reduce(k2, list(v2)) → list(k3, v3)`
  - Apply aggregation (reduce) function to values
  - Output results

MapReduce data flows



MapReduce visualization

How to count the number of occurrences for each unique color?

input.csv (in GFS)

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

Map

How to count the number of occurrences for each unique color?

input.csv (in GFS)

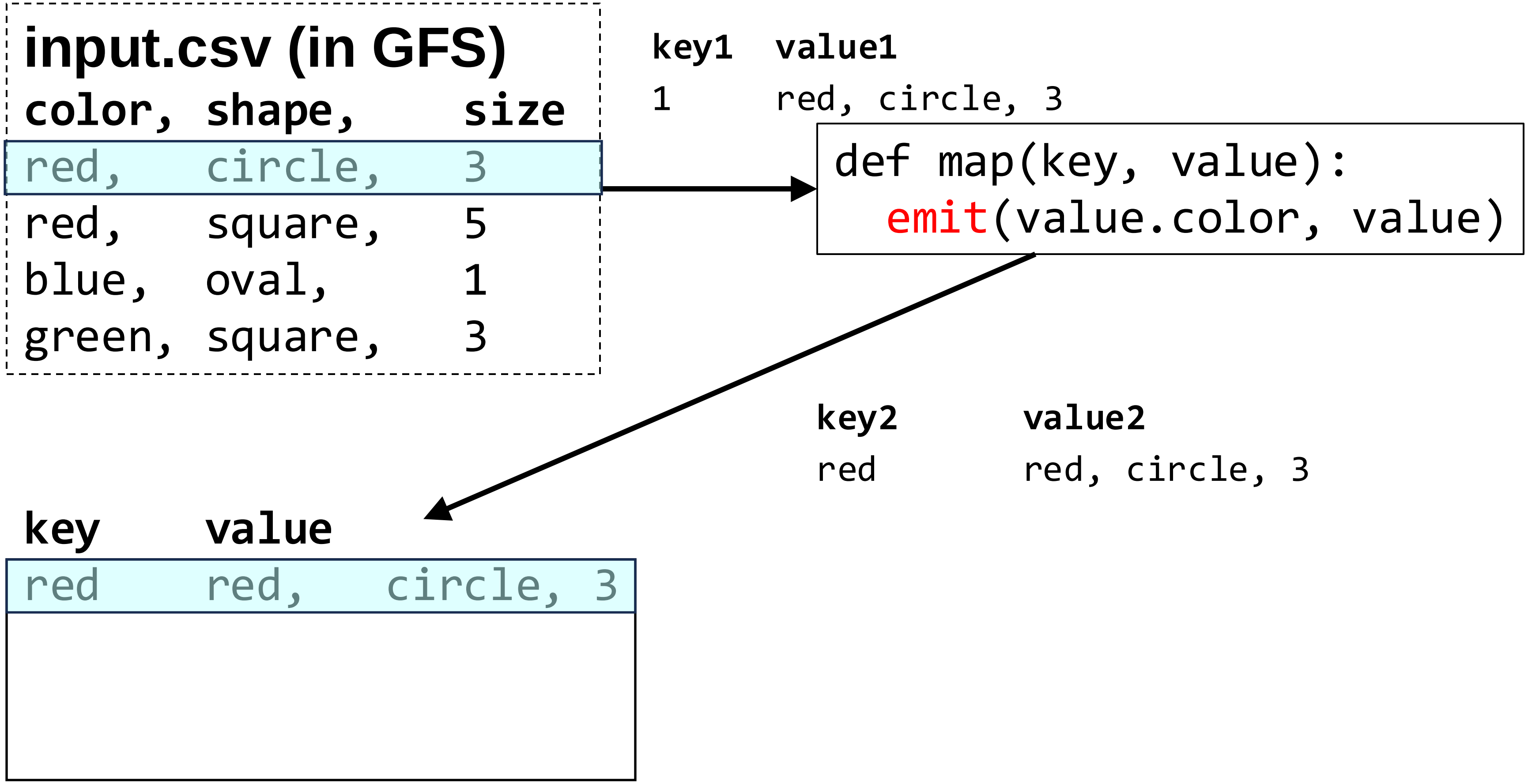
```
color, shape,    size
red,   circle,   3
red,   square,   5
blue,  oval,     1
green, square,   3
```

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

Map will be called 4 times (once for each line of the input file).

Map

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

key1	value1
1	red, circle, 3

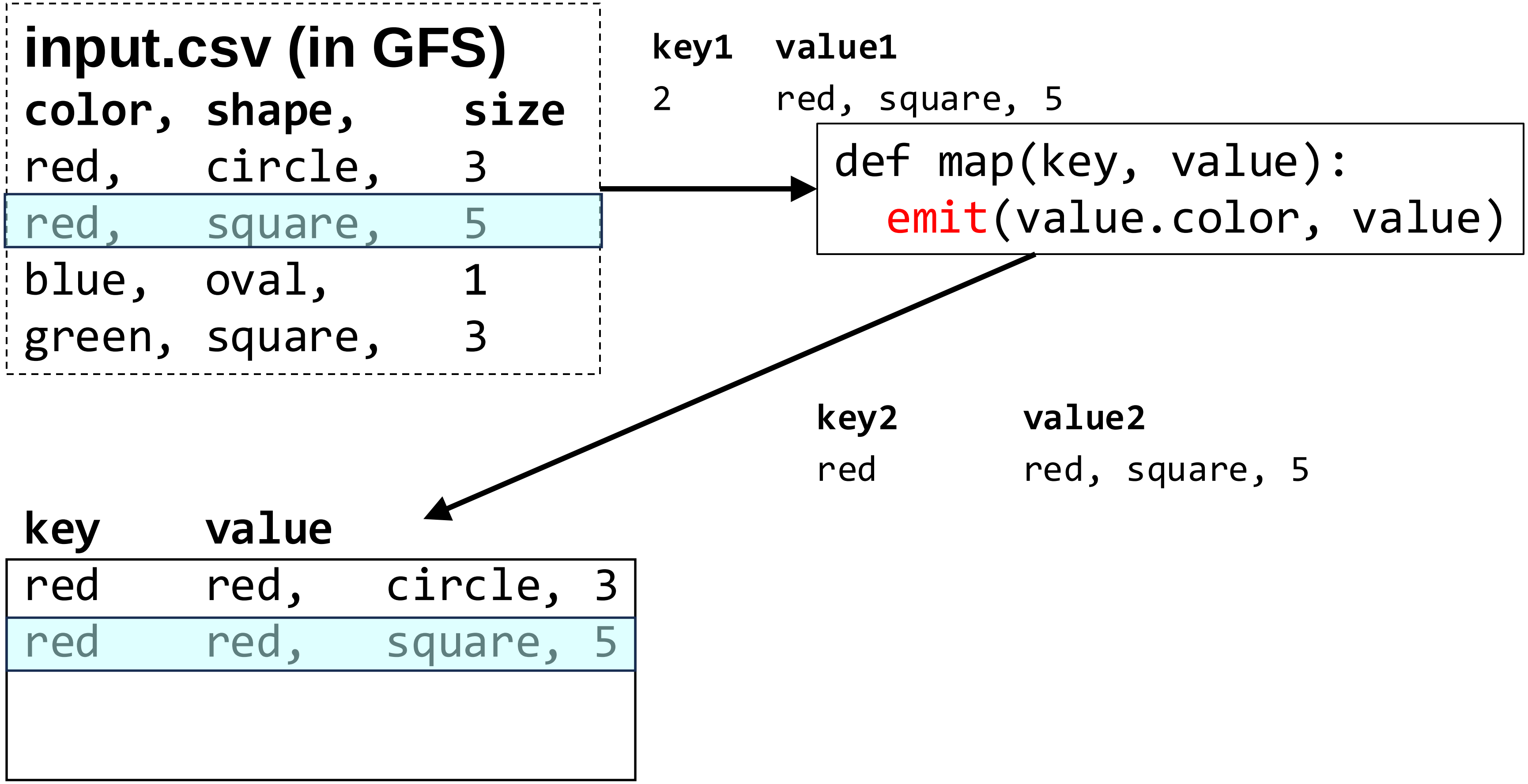
```
def map(key, value):
  emit(value.color, value)
```

key2	value2
red	red, circle, 3

key	value
red	red, circle, 3

Map

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

key1	value1
2	red, square, 5

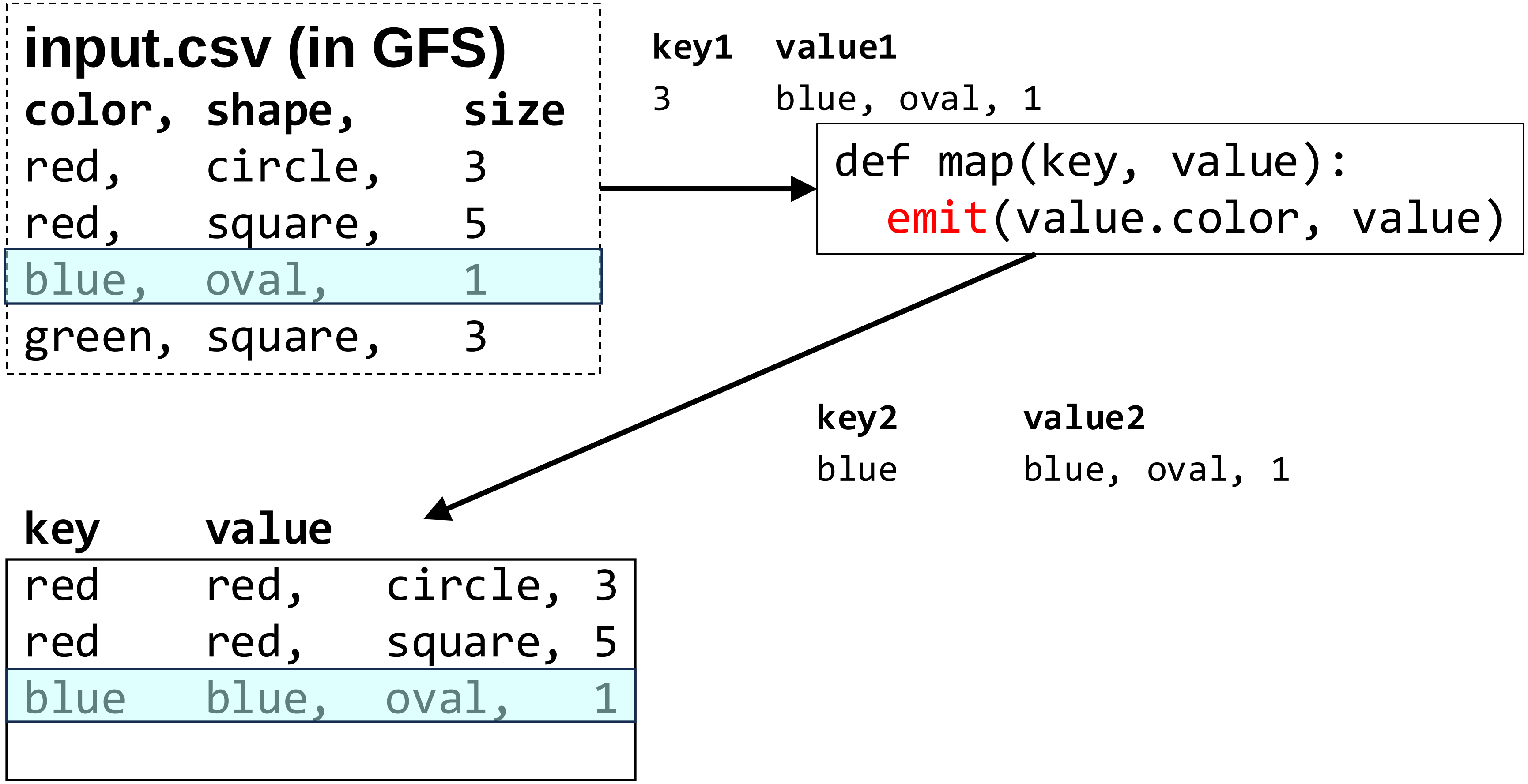
```
def map(key, value):
  emit(value.color, value)
```

key2	value2
red	red, square, 5

key	value
red	red, circle, 3
red	red, square, 5

Map

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

key1	value1
3	blue, oval, 1

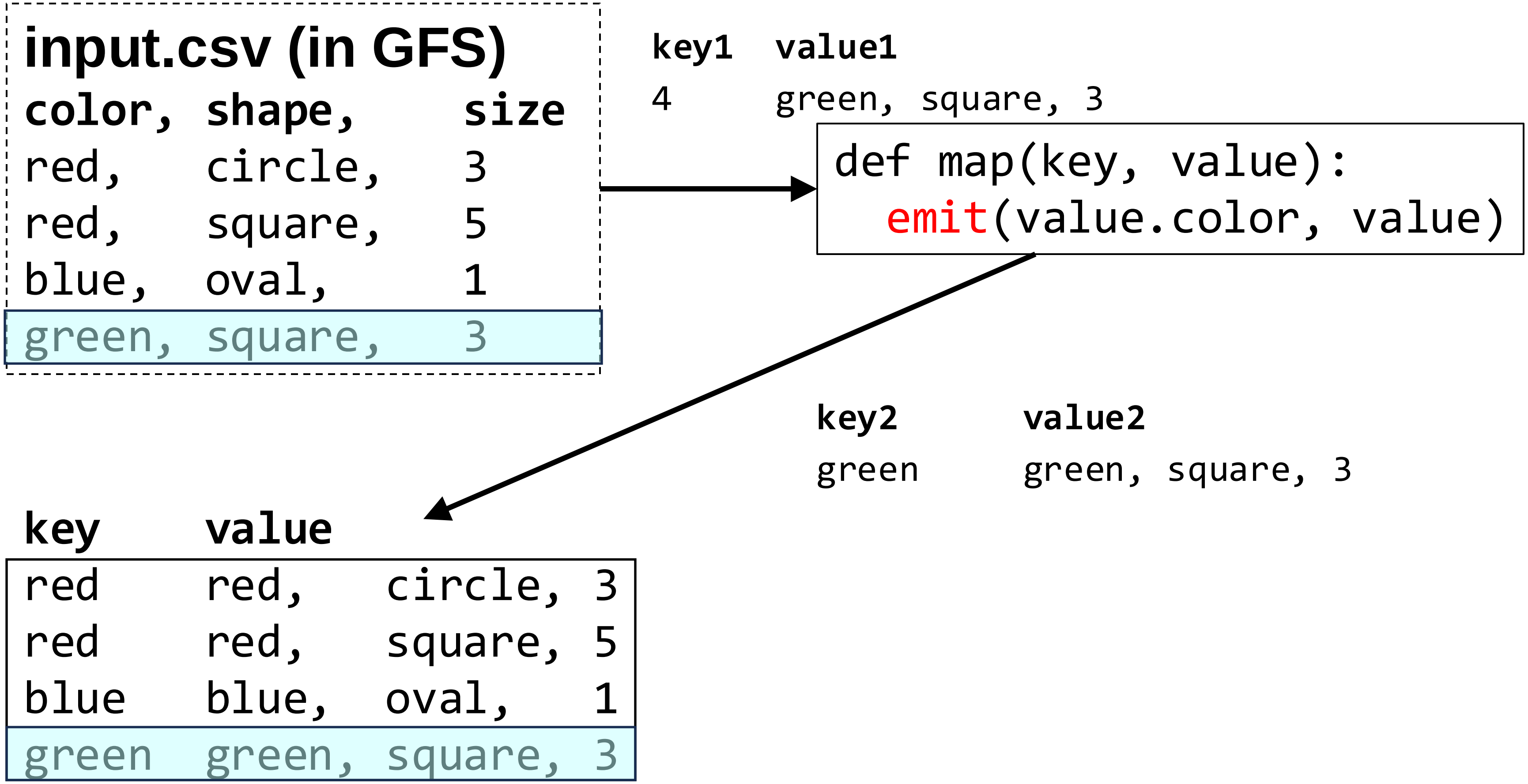
```
def map(key, value):
  emit(value.color, value)
```

key2	value2
blue	blue, oval, 1

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

Map

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

key1	value1
4	green, square, 3

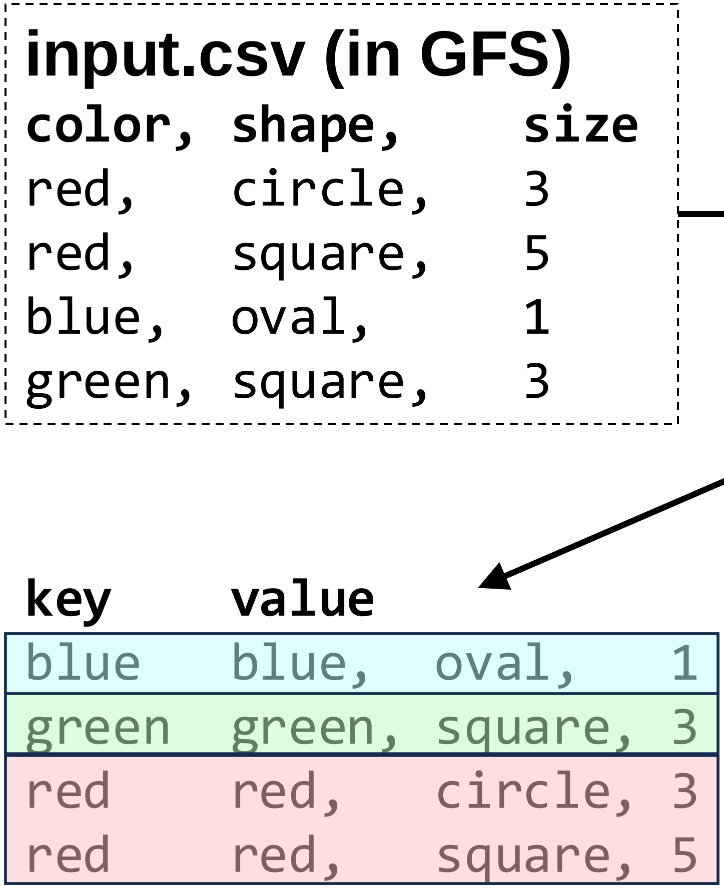
```
def map(key, value):
  emit(value.color, value)
```

key2	value2
green	green, square, 3

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

Reduce

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

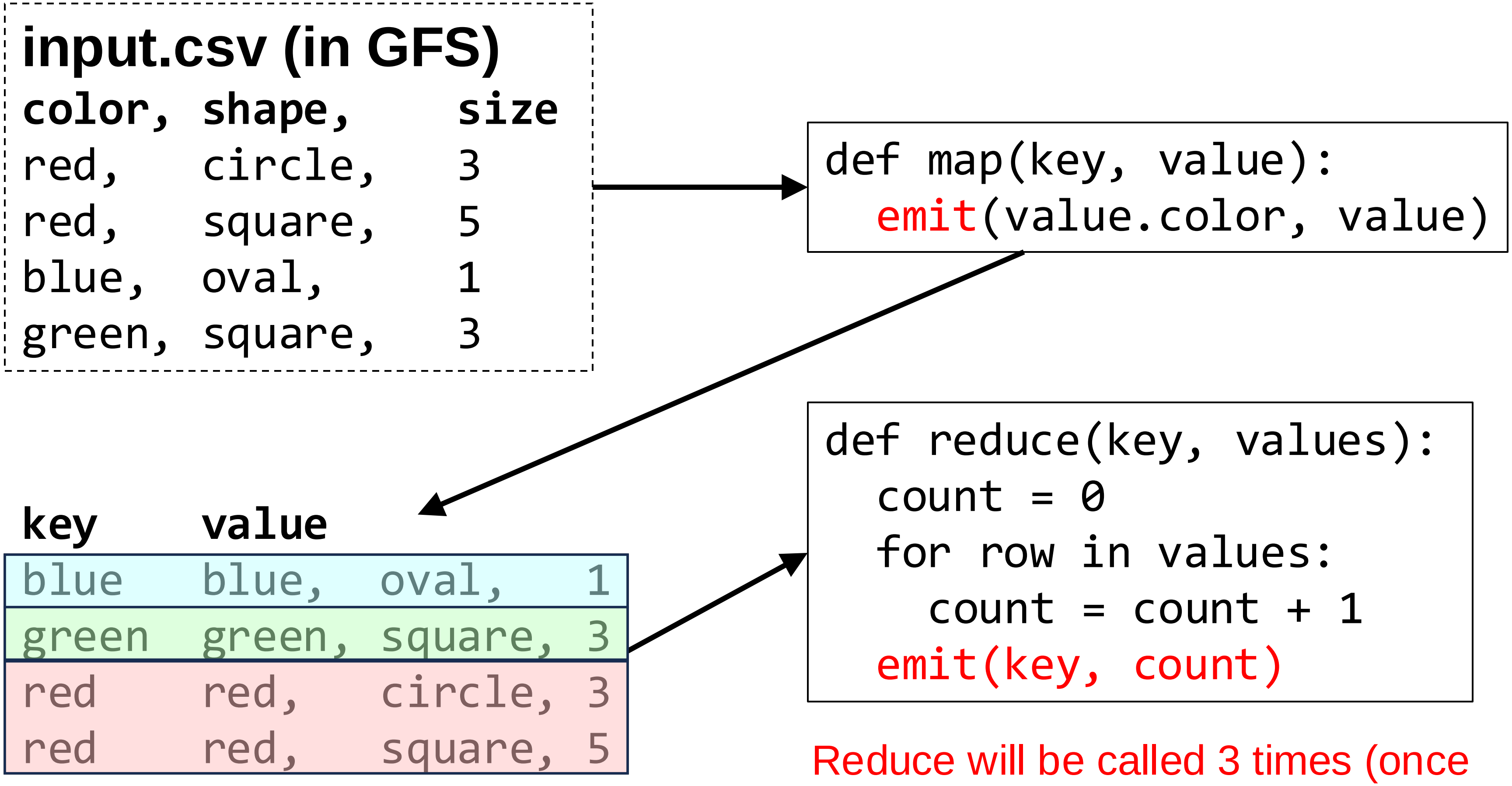
```
def map(key, value):
  emit(value.color, value)
```

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.

Reduce

How to count the number of occurrences for each unique color?



input.csv (in GFS)

```
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

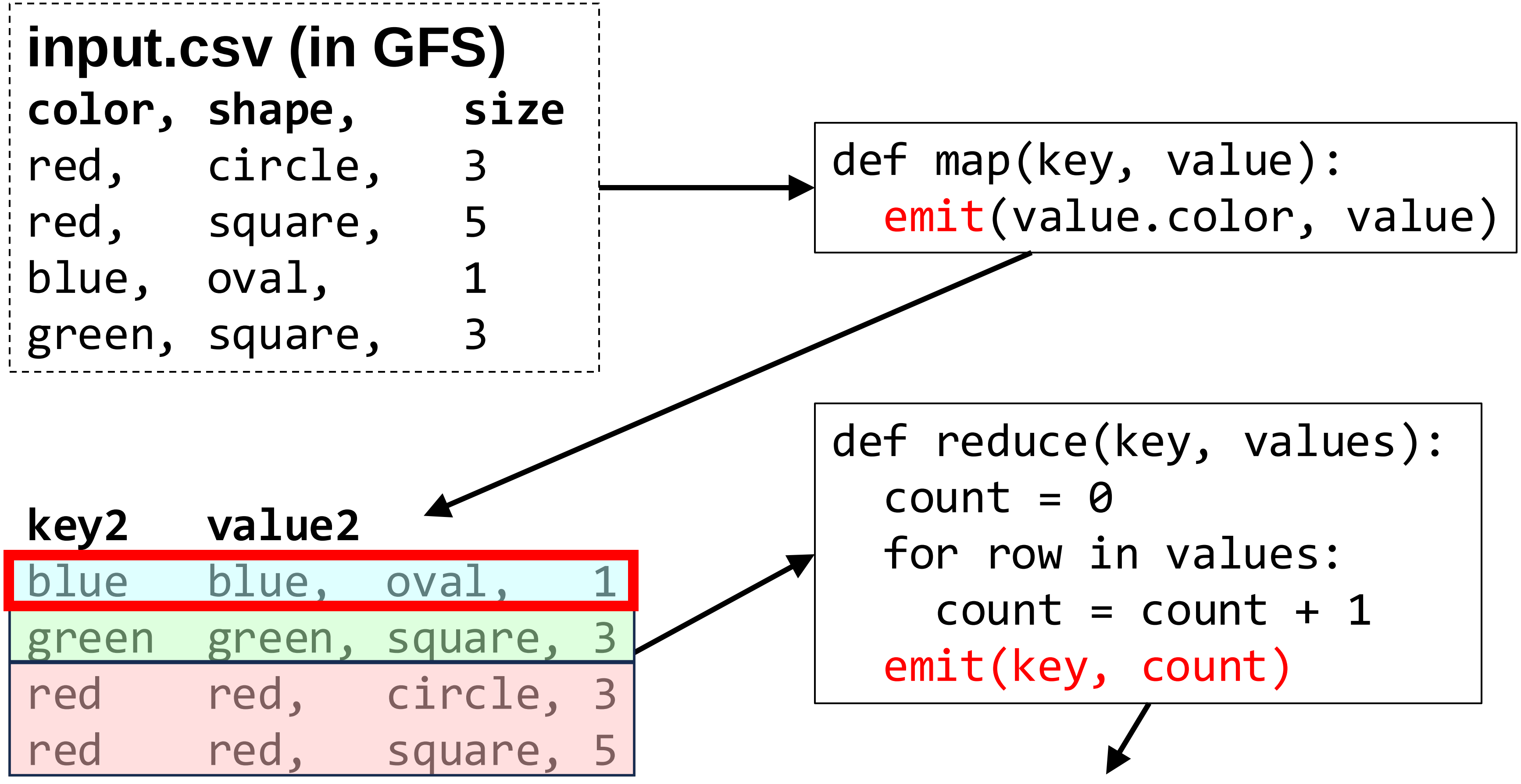
```
def reduce(key, values):
  count = 0
  for row in values:
    count = count + 1
  emit(key, count)
```

Intermediate data is **grouped** and **sorted** by key.

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

Reduce

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

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

```
key2   value2
blue   blue,  oval,   1
green  green, square, 3
red    red,   circle, 3
red    red,   square, 5
```

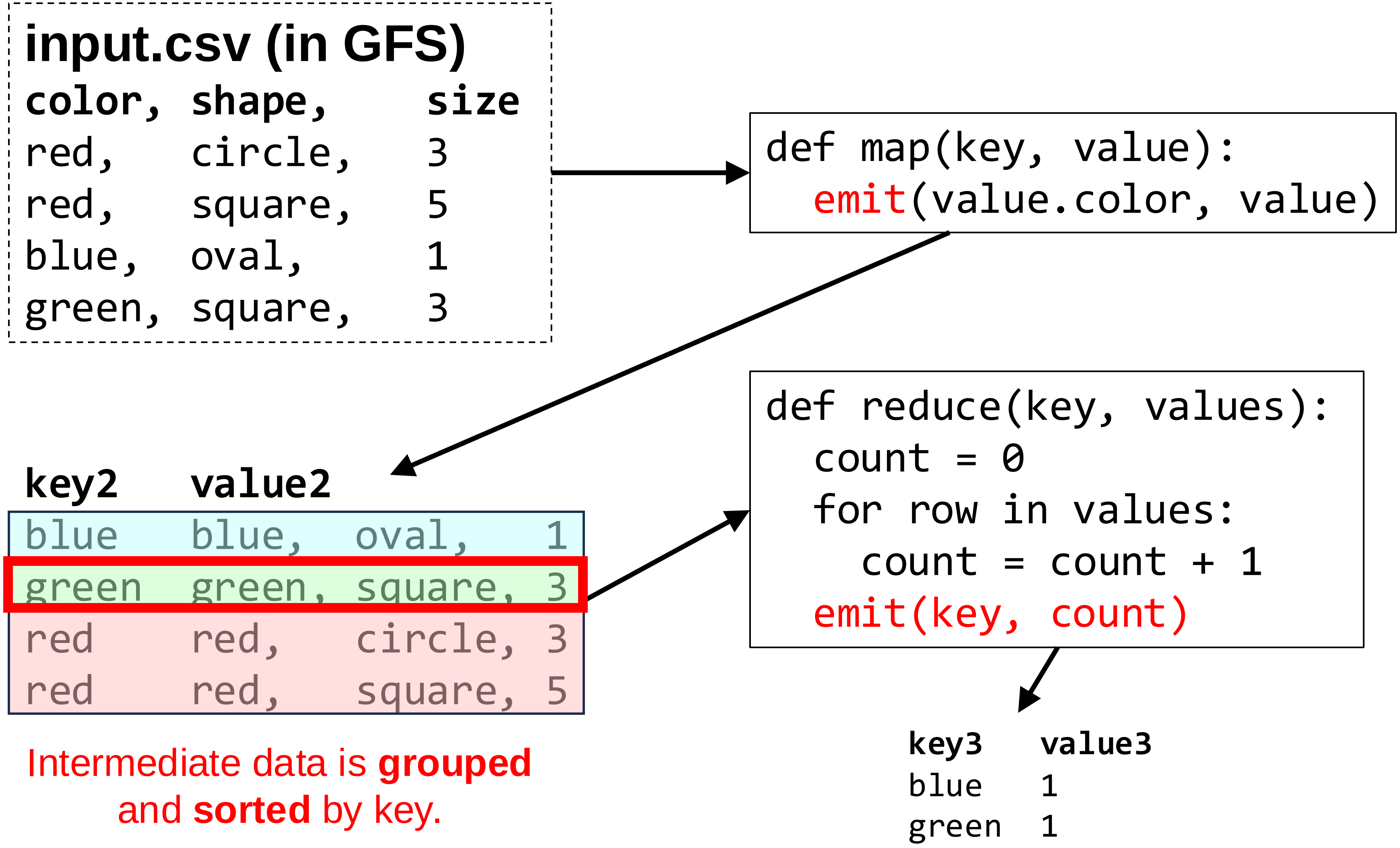
Intermediate data is **grouped** and **sorted** by key.

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

```
key3   value3
blue   1
```

Reduce

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

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

key2	value2		
blue	blue,	oval,	1
green	green,	square,	3
red	red,	circle,	3
red	red,	square,	5

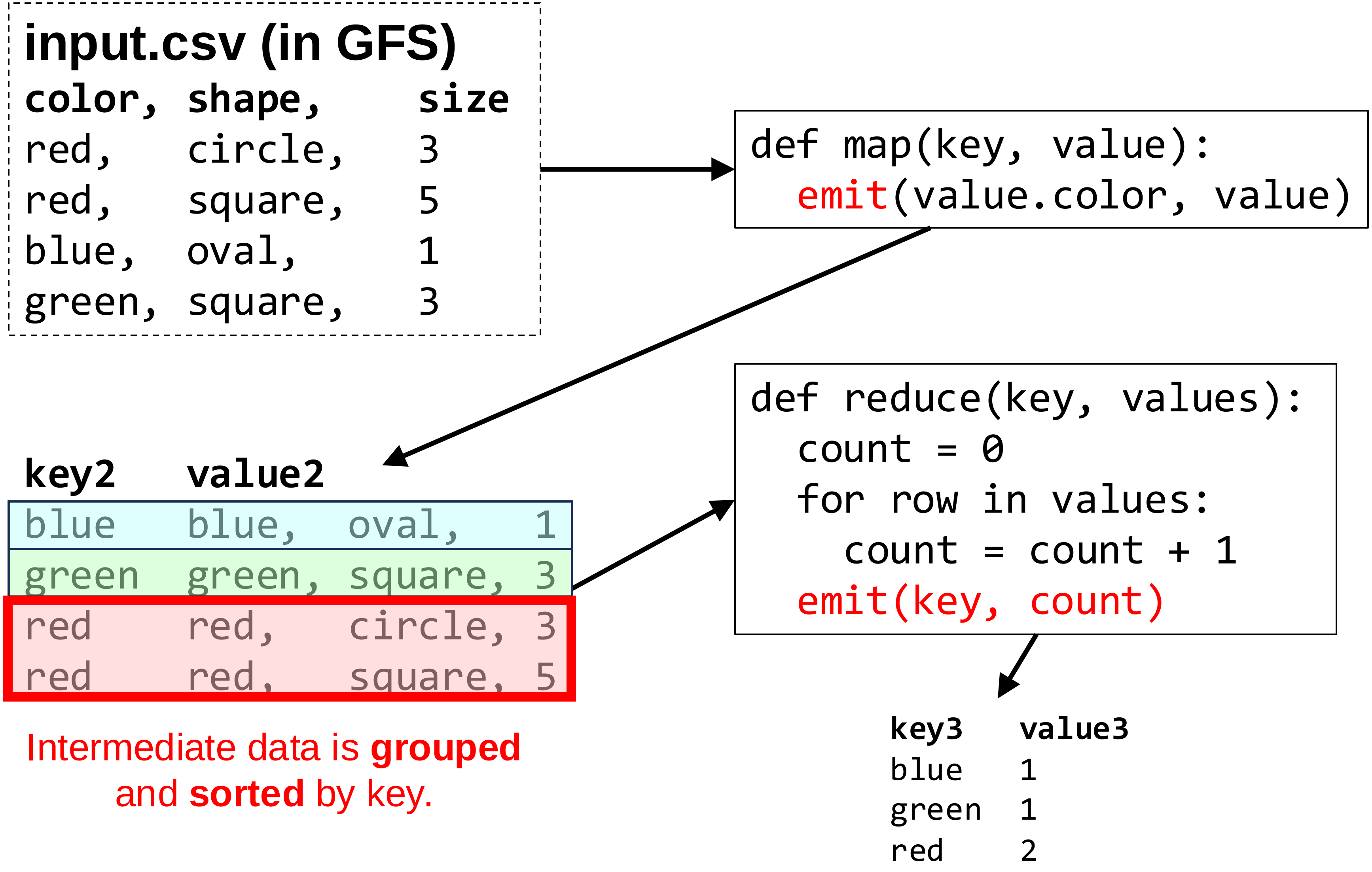
Intermediate data is **grouped** and **sorted** by key.

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

key3	value3
blue	1
green	1

Reduce

How to count the number of occurrences for each unique color?



input.csv (in GFS)

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

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

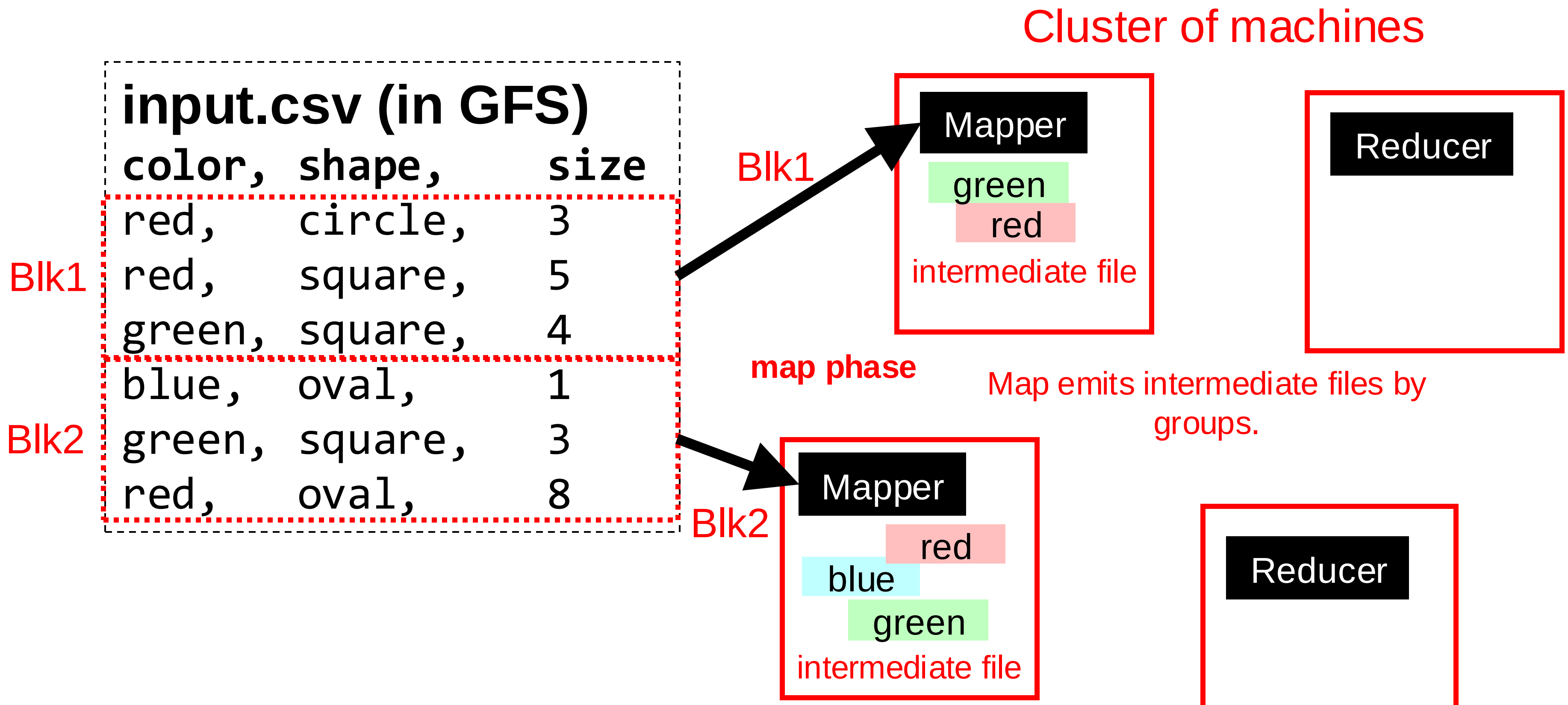
key2	value2		
blue	blue,	oval,	1
green	green,	square,	3
red	red,	circle,	3
red	red,	square,	5

Intermediate data is **grouped** and **sorted** by key.

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

key3	value3
blue	1
green	1
red	2

Multiple reducers (for big intermediate data)



Cluster of machines

input.csv (in GFS)

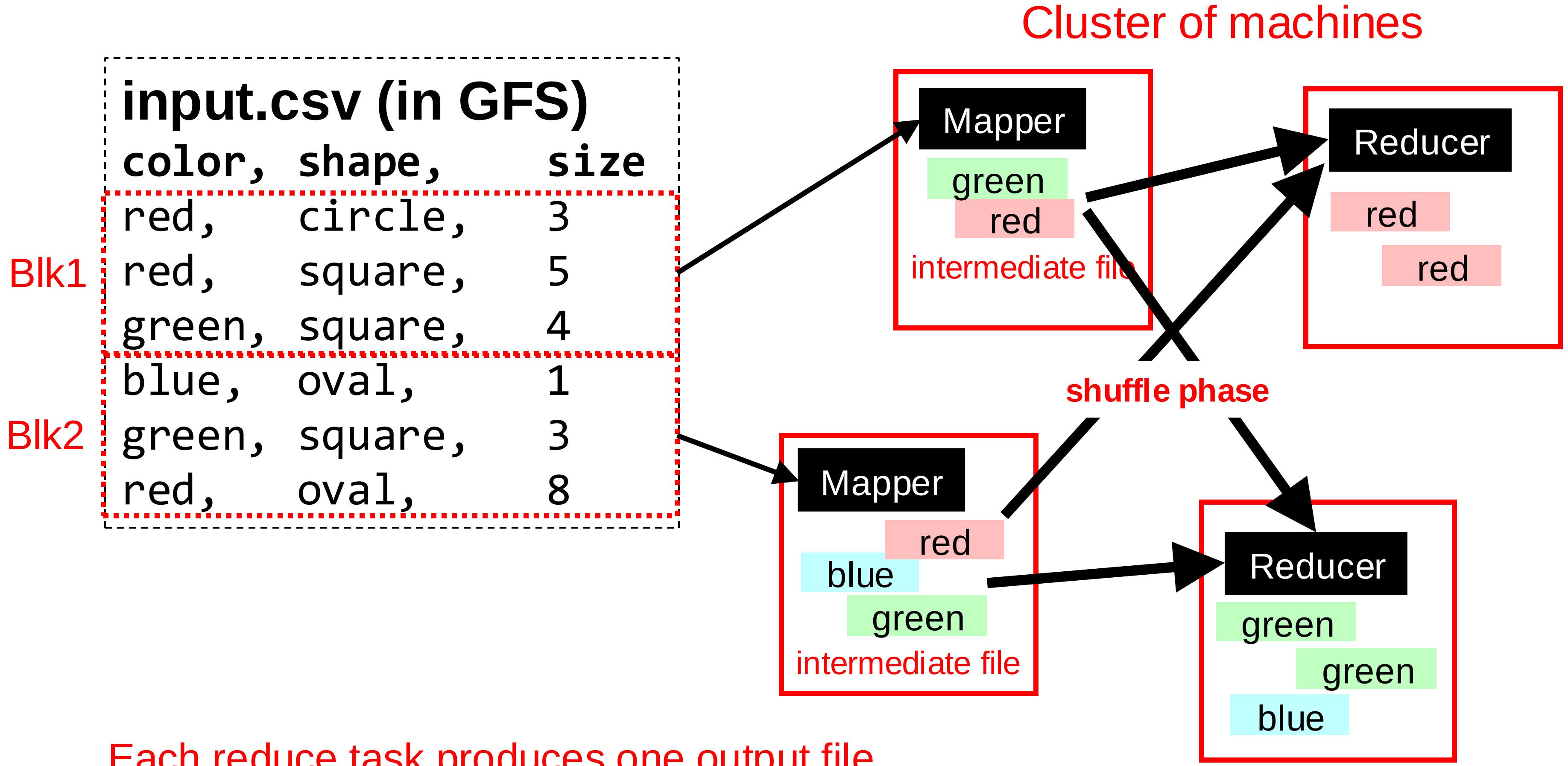
```
color, shape,   size
red,   circle,  3
red,   square,  5
green, square,  4
blue,  oval,    1
green, square,  3
red,   oval,    8
```

map phase

Map emits intermediate files by groups.

Each reduce task produces one output file.
A reduce task might take multiple keys.
All intermediate rows with the same key go to the same reducer.

Multiple reducers (for big intermediate data)



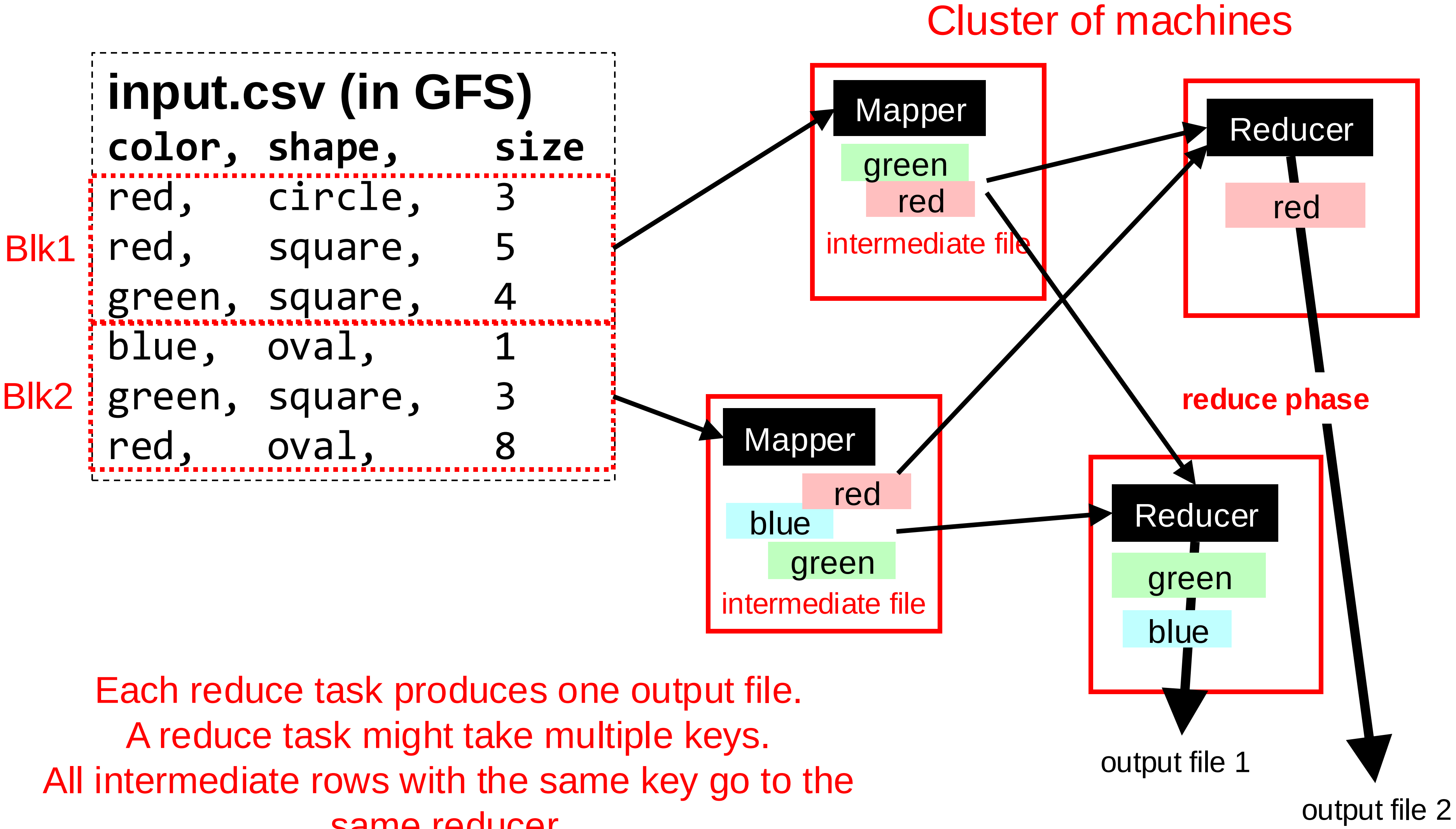
input.csv (in GFS)

```
color, shape,   size
red,   circle,   3
red,   square,   5
green, square,   4
blue,  oval,     1
green, square,   3
red,   oval,     8
```

Each reduce task produces one output file.
A reduce task might take multiple keys.
All intermediate rows with the same key go to the same reducer.

Reducer collects all intermediate files of its assigned keys (groups).

Multiple reducers (for big intermediate data)



input.csv (in GFS)

```
color, shape,   size
red,   circle,  3
red,   square,  5
green, square,  4
blue,  oval,    1
green, square,  3
red,   oval,    8
```

Reducer dumps final results to GFS.

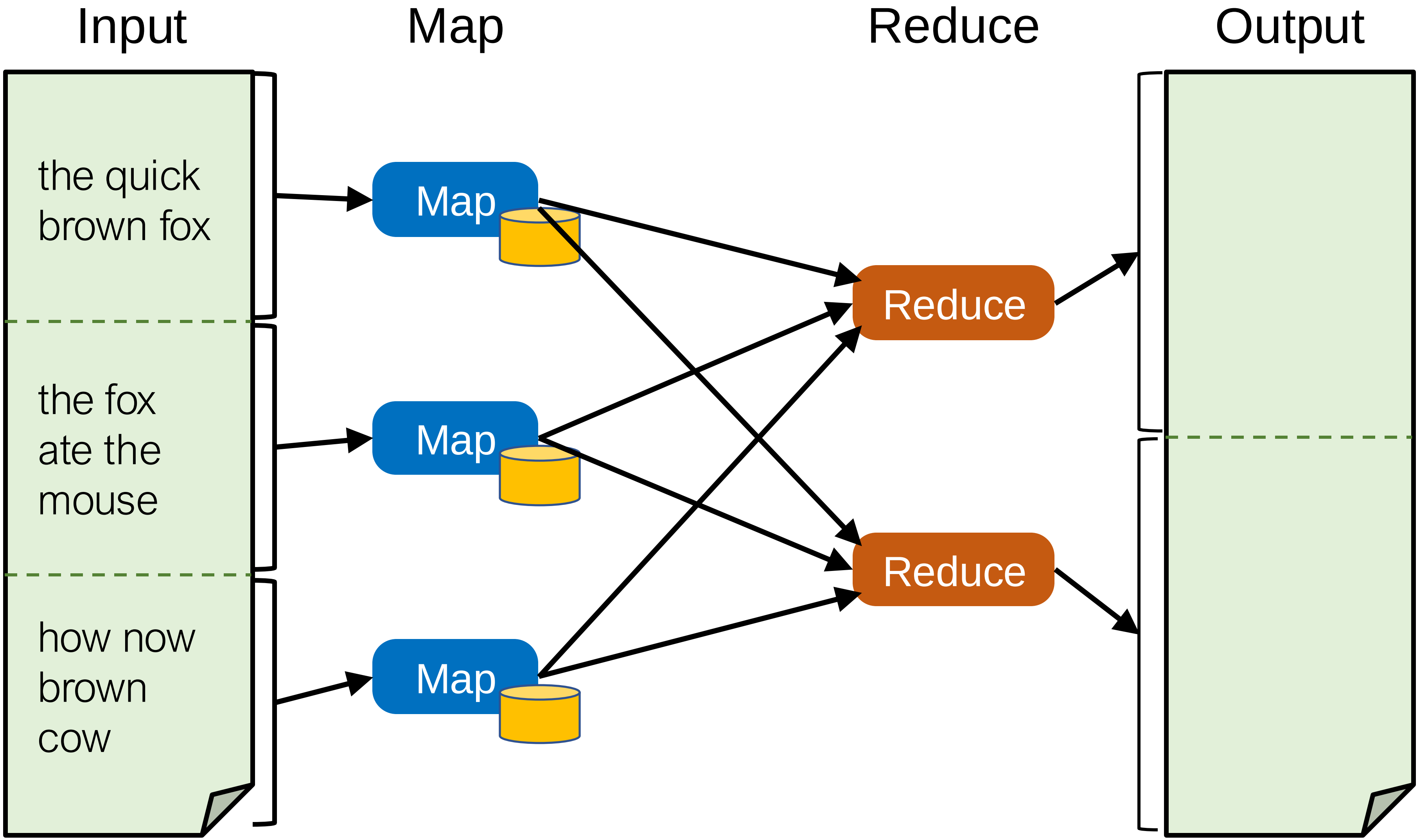
Each reduce task produces one output file.
A reduce task might take multiple keys.
All intermediate rows with the same key go to the same reducer.

MapReduce: Word Count

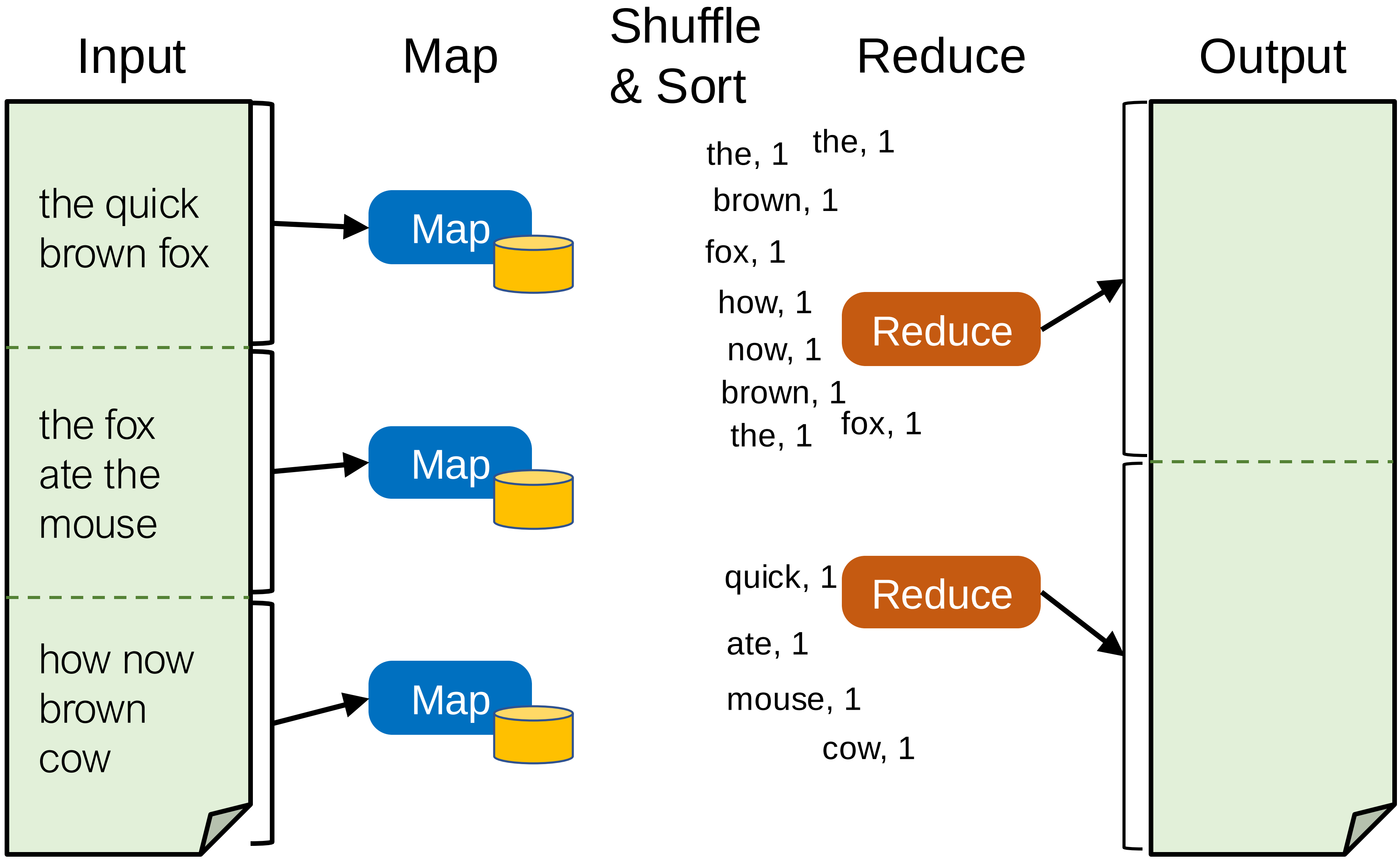
```
map(key, value):
    for each word w in value:
        EmitIntermediate(w, “1”);

reduce(key, values):
    int result = 0;
    for each v in values:
        results += ParseInt(v);
    Emit(AsString(result));
```

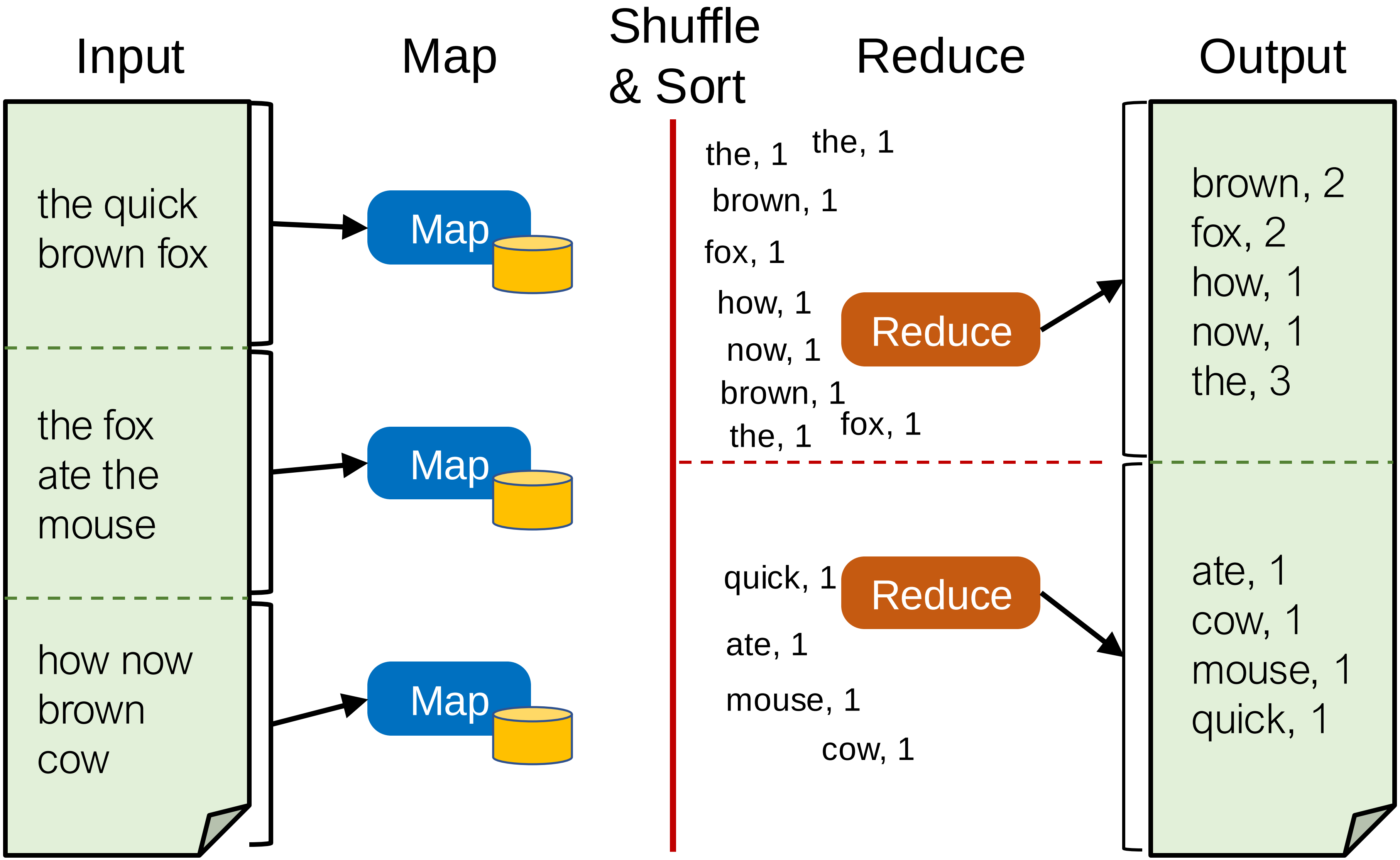
Word Count execution



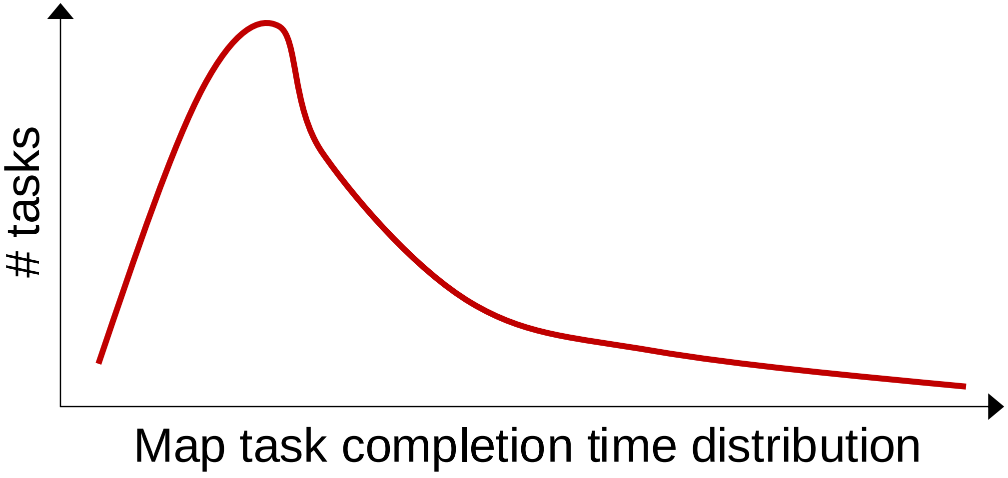
Word Count execution



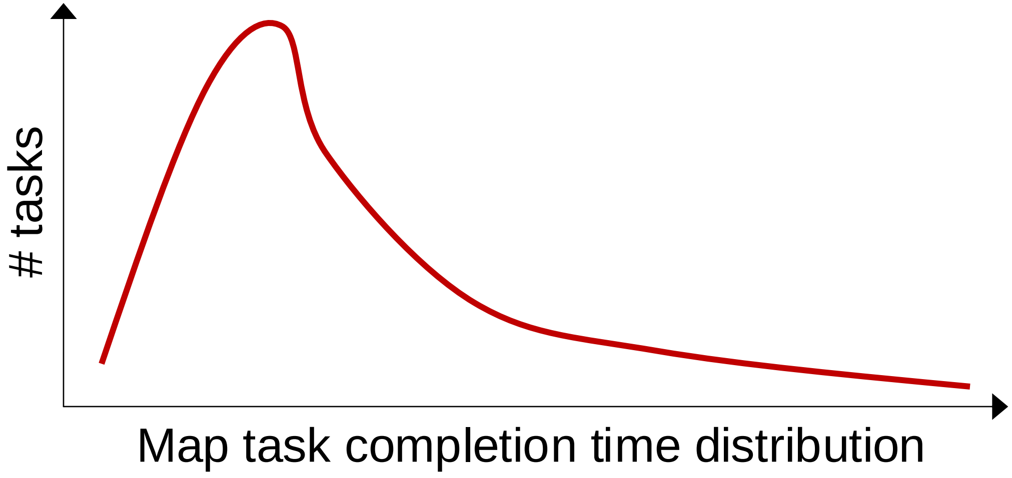
Word Count execution



Stragglers



Stragglers



- **Tail execution time** means some executors (always) finish late (**tail latency**)

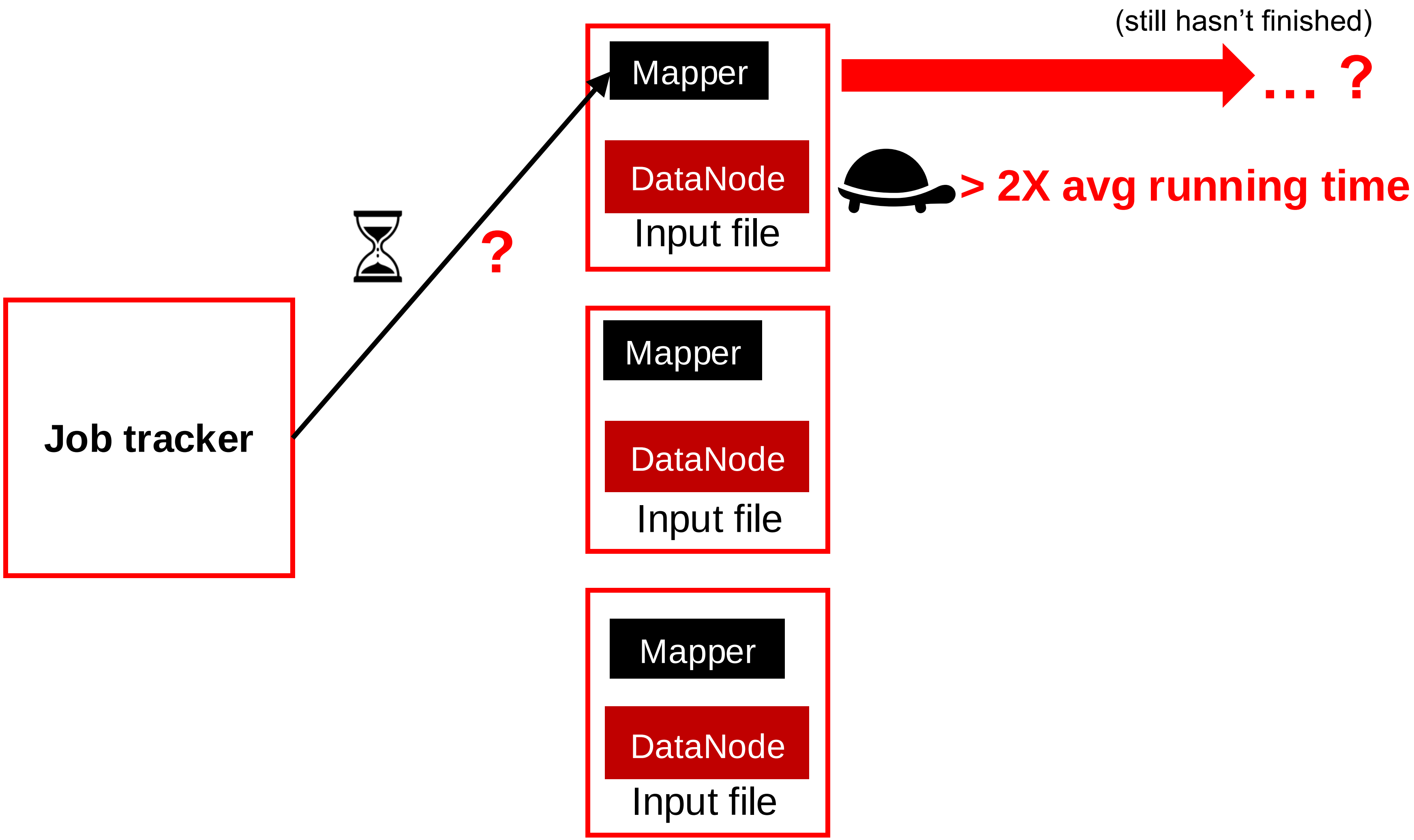
Q: How can MapReduce work around this?

  - Hint: its approach to **fault-tolerance** provides the right tool

Resilience against stragglers?

- If a task is going slowly (i.e., straggler):
  - Launch second copy of task (**backup task**) on another node
  - Take the output of whichever finishes first

Resilience against stragglers



Cluster of machines

(still hasn't finished)

Mapper

DataNode

Input file

… ?

> 2X avg running time

?

Job tracker

Mapper

DataNode

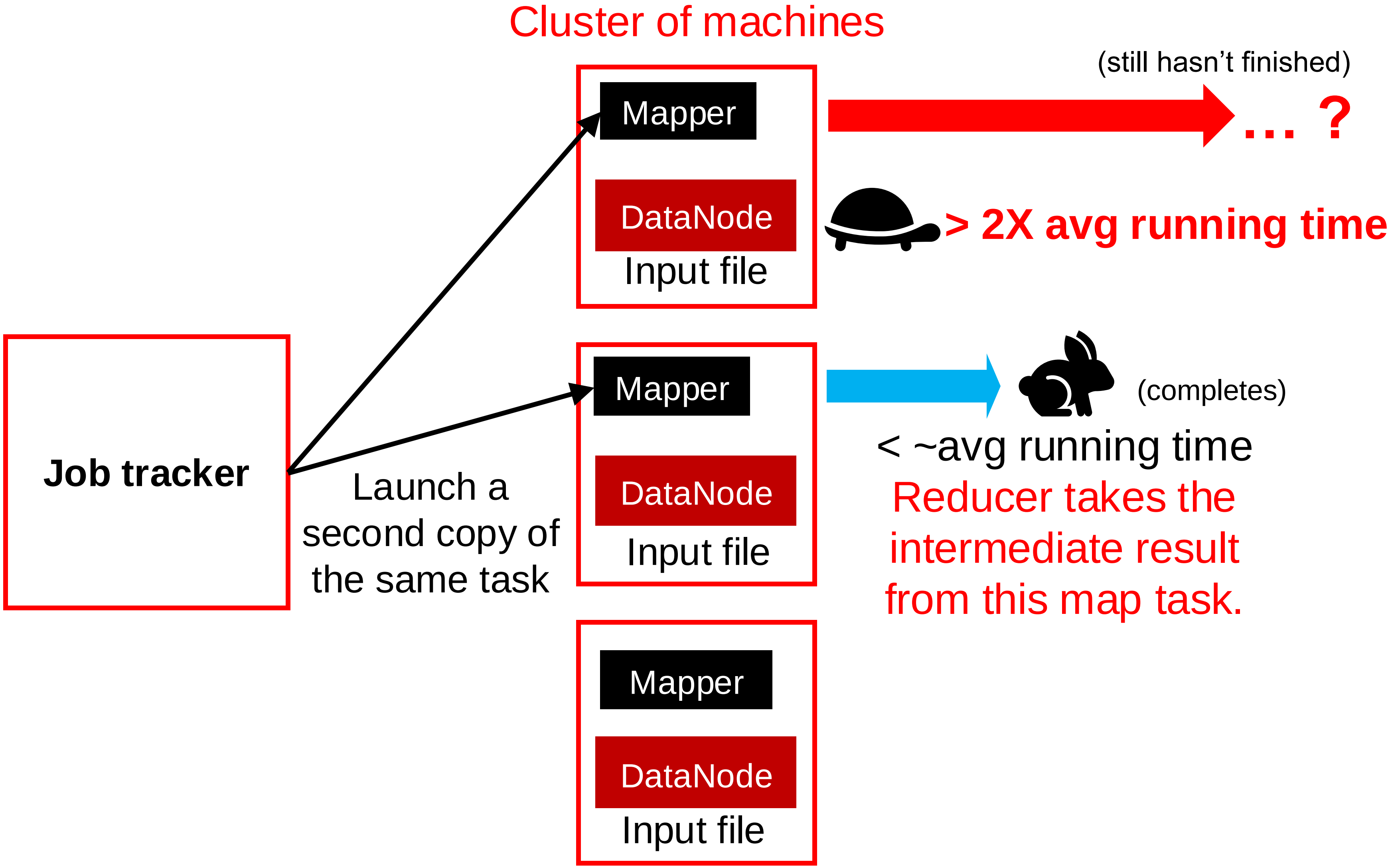
Input file

Mapper

DataNode

Input file

Resilience against stragglers



Cluster of machines

Job tracker

Launch a second copy of the same task

Mapper

DataNode

Input file

Mapper

DataNode

Input file

Mapper

DataNode

Input file

(still hasn't finished)

… ?

> 2X avg running time

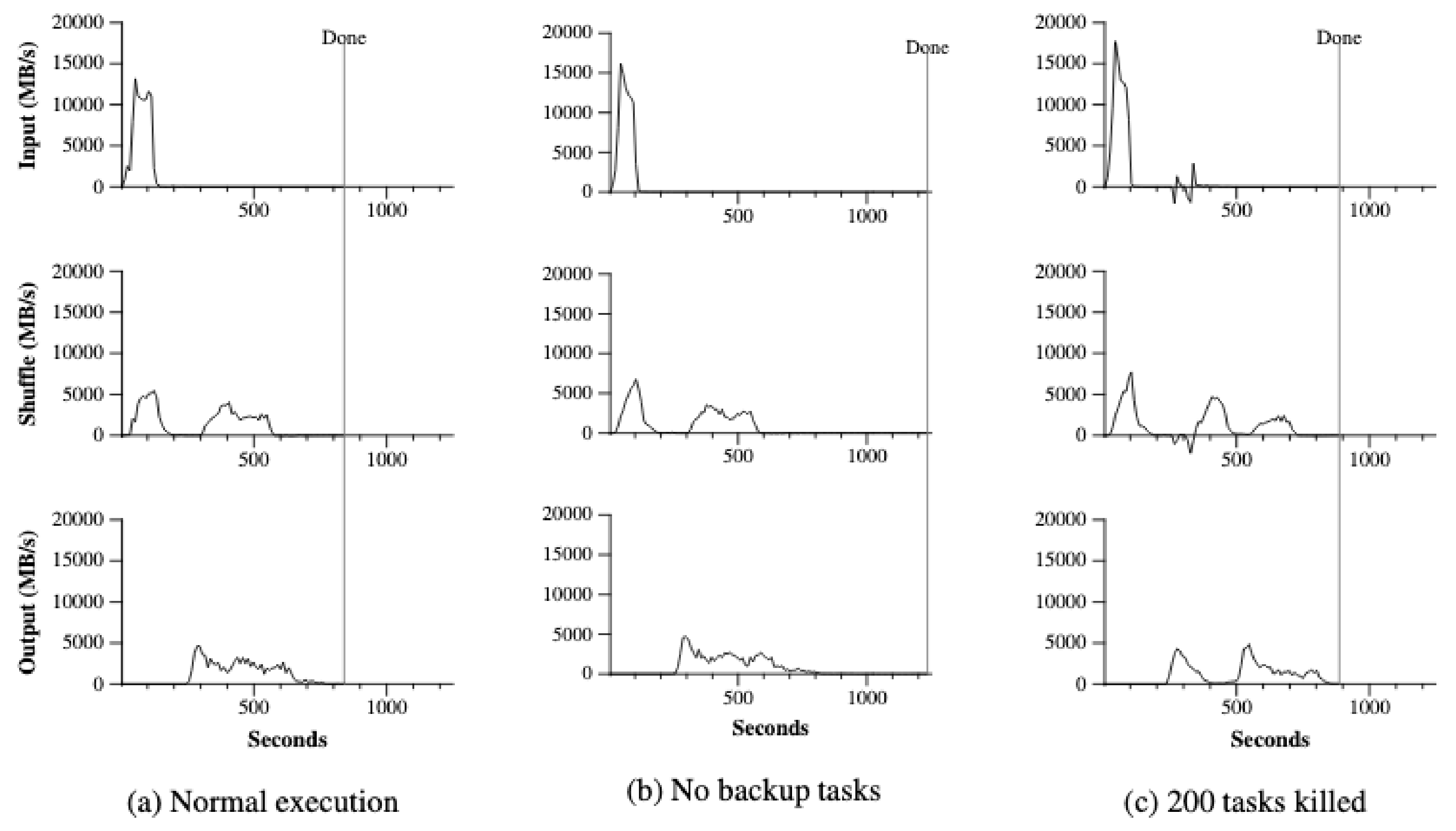
(completes)

< ~avg running time

Reducer takes the intermediate result from this map task.

Would backup tasks cause correctness issue in MapReduce jobs?

Discussion: MapReduce eval (paper)



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed