Simple MapReduce with Ruby and Rinda

Josh Carter
2007@joshcarter.com

MapReduce: Simplified Data Processing on Large Clusters

-Google paper by Jeffrey Dean

and Sanjay Ghemawat

2 key functions:

Map: processes a key/value pair to generate intermediate key/value pairs

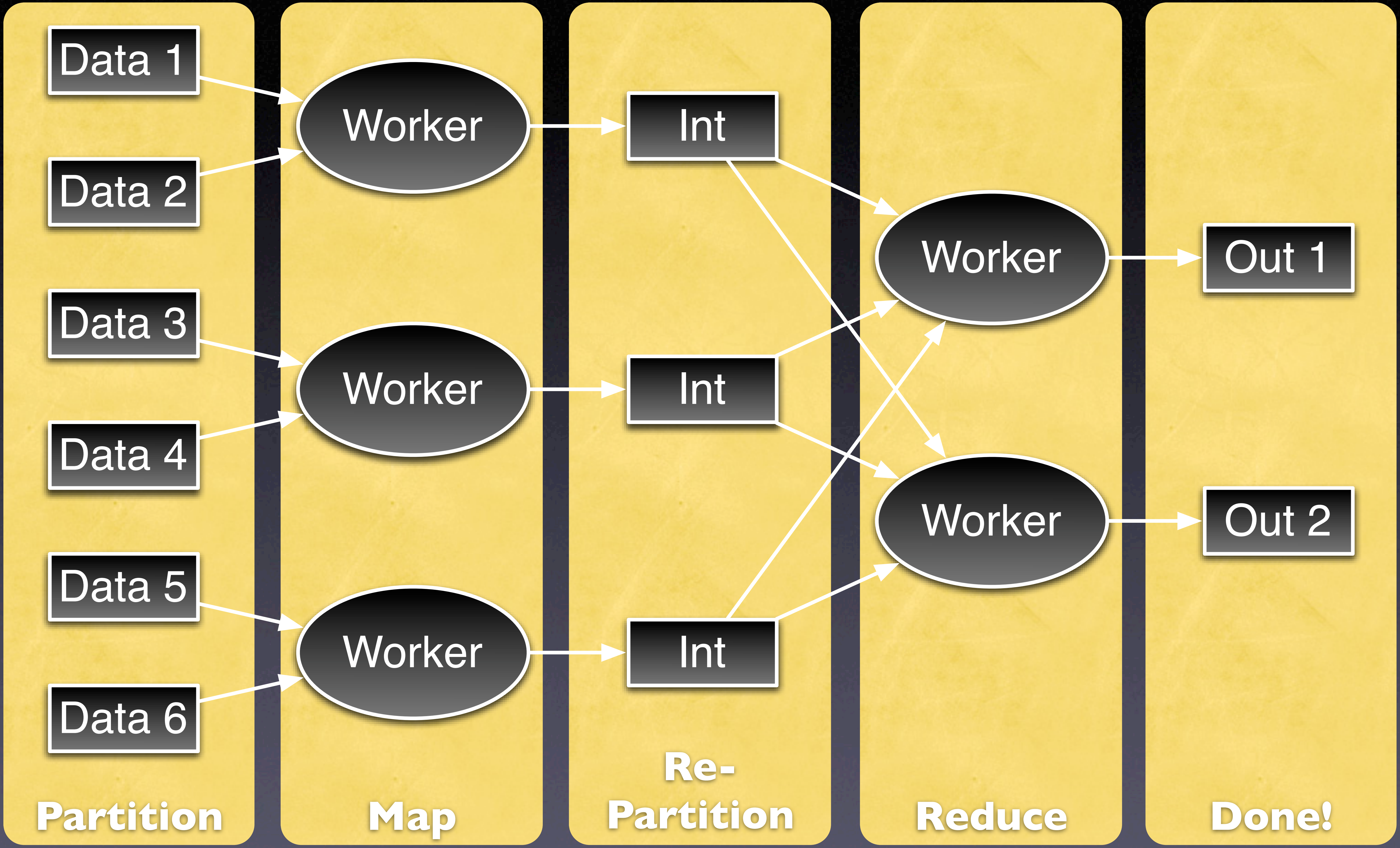
Reduce: merges all intermediate values associated with the same intermediate key

Example: counting words in a big file

Map: Process file, emit [word,1] pairs

Reduce: Add all values for same word, emit [word, total]

The big diagram that explains everything



Partitioning: we'll get to that later... *(time permitting)*

Now, about **Ruby...**

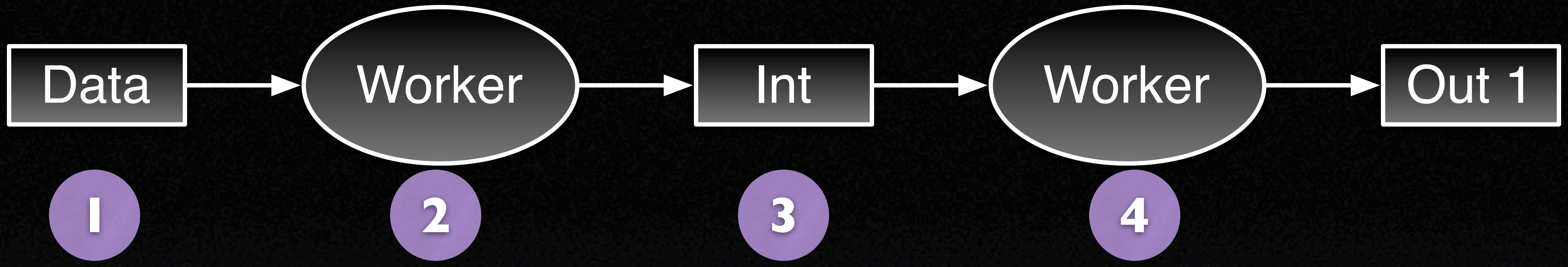
Easy way to distribute code and data: **Rinda/DRb**

Rinda: provides TupleSpace

Anyone can write to TS

Anyone can take from TS

(See “Blackboard” chapter in *The Pragmatic Programmer*)



```
map_data = Partitioner::simple_partition_data(@data, @map_tasks)   1
map_tasks = Array.new

(0..@map_tasks - 1).each do |i|
  map_tasks << WorkerTask.new(i + 1, map_data[i], @map)
end

map_data = run_tasks("map", map_tasks)   2

reduce_data = @partition.call(map_data, @reduce_tasks)   3
reduce_tasks = Array.new

(0..@reduce_tasks - 1).each do |i|
  reduce_tasks << WorkerTask.new(i + 1, reduce_data[i], @reduce)
end

run_tasks("reduce", reduce_tasks)   4
```

So what’s a task?

```
class WorkerTask
  attr_reader :task_id, :data, :process

  def initialize(task_id, data, process)
    @task_id = task_id
    @data    = data
    @process = process
  end

  def run
    @process.call @data
  end
end
```

the code is just a lambda, and DRb serializes it for you!

Shipping tasks around

Master:

```
ts.write(['task', DRb.uri, task])
```

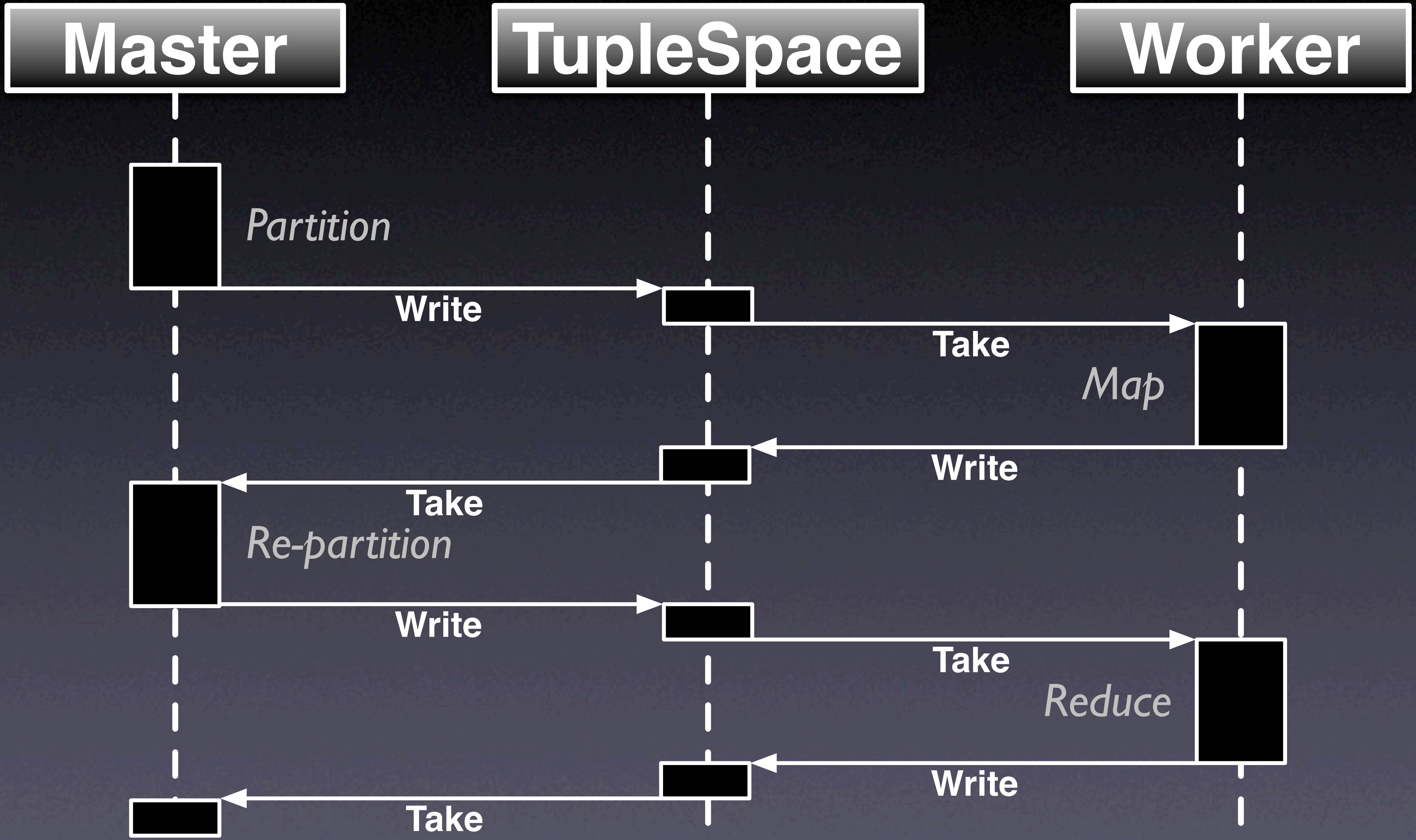
Worker:

```
tuple = ts.take(['task', nil, nil])
task = tuple[2]
result = task.run
ts.write(['result', tuple[1],
          task.task_id, result])
```

Master:

```
tuple = ts.take(['result', DRb.uri,
                task.task_id, nil])
```

Less code, more pictures



Example: word count

```
# Map: take string, return one pair
# of [word, 1] for each word.
job.map = lambda do |lines|
  result = Array.new

  lines.each do |line|
    next if line.empty?

    line.scan(/\w+/).each do |word|
      result << [ word, 1 ]
    end
  end

  result
end
```

Example: word count

```
# Reduce: Combine [word, 1] pairs into
# [word, count].
job.reduce = lambda do |pairs|
  counts = Hash.new

  pairs.each do |pair|
    word, count = pair[0], pair[1]

    counts[word] ||= 0
    counts[word] += count
  end

  counts
end
```

Demo

(see, it's not all boring stuff)

Partitioning: the previous example only works if all [word,1] pairs for each word go to the same reduce task.

Partitioning: master process is responsible for divvying up intermediate data to reduce tasks.

See `Partitioner::array_data_split_by_first_entry()`

One problem: word count distributed via MapReduce is *slower* than simple local process.

Reason: problem is I/O bound already, adding more I/O just makes it worse!

Solutions: location of the data in I/O-bound problems is key. Google keeps the data local to the servers.

Solutions:

Keep data local to workers.

Don’t send data, ship URLs.

Don’t send code, send system calls to fast (C, etc.) apps.

Topics not covered:

Worker failure

Ordering guarantees

Skipping bad records

(etc, see paper)

Play with it yourself!

```
http://multipart-mixed.com/software/simple_mapreduce_in_ruby.html
```

```
http://labs.google.com/papers/mapreduce.html
```