# MapReduce:

## Simplified Data Processing on Large Clusters

J. Dean, S. Ghemawat, OSDI, 2004.

# Motivation: Large scale data processing

We want to:

- Extract data from large datasets
- Run on big clusters of computers
- Be easy to program

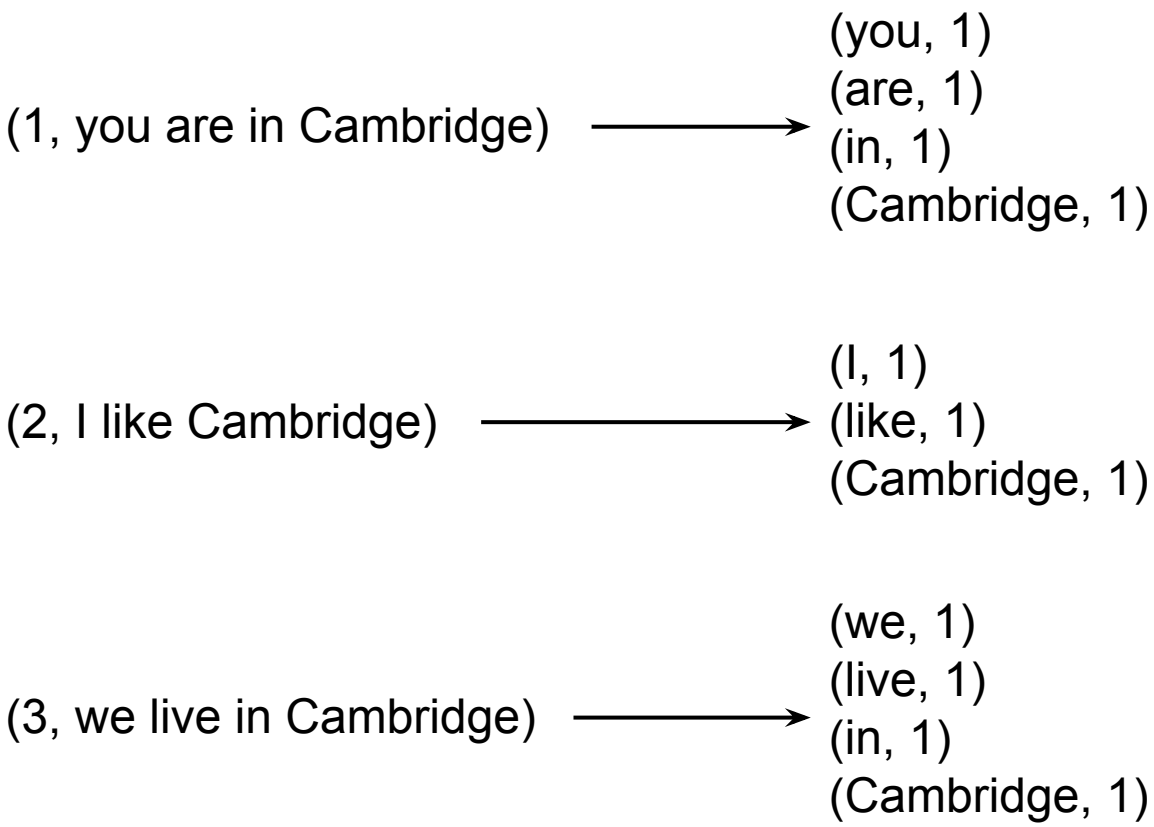
# Solution: MapReduce

A new programming model: Map & Reduce

Provides:

- Automatic parallelization and distribution
- Fault tolerance
- I/O scheduling
- Status and monitoring

# Map



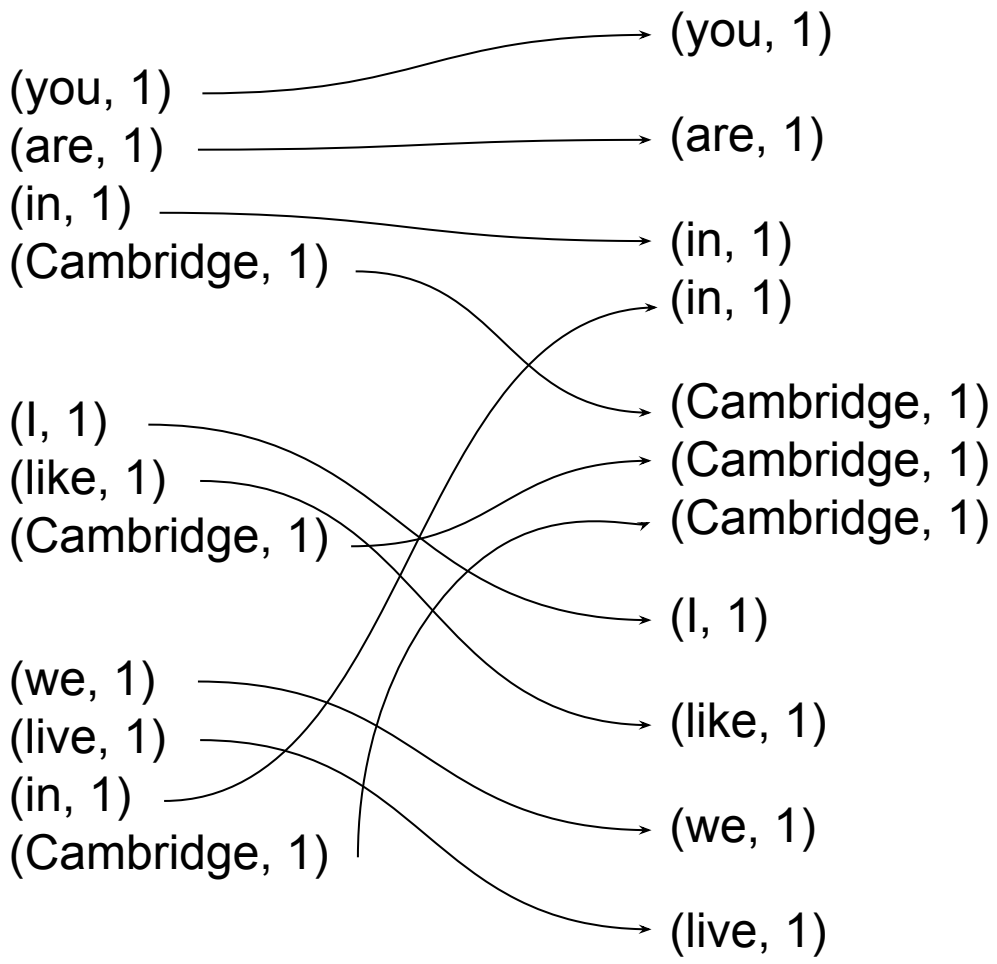

map (in_key, in_value) → list(out_key, intermediate_value)

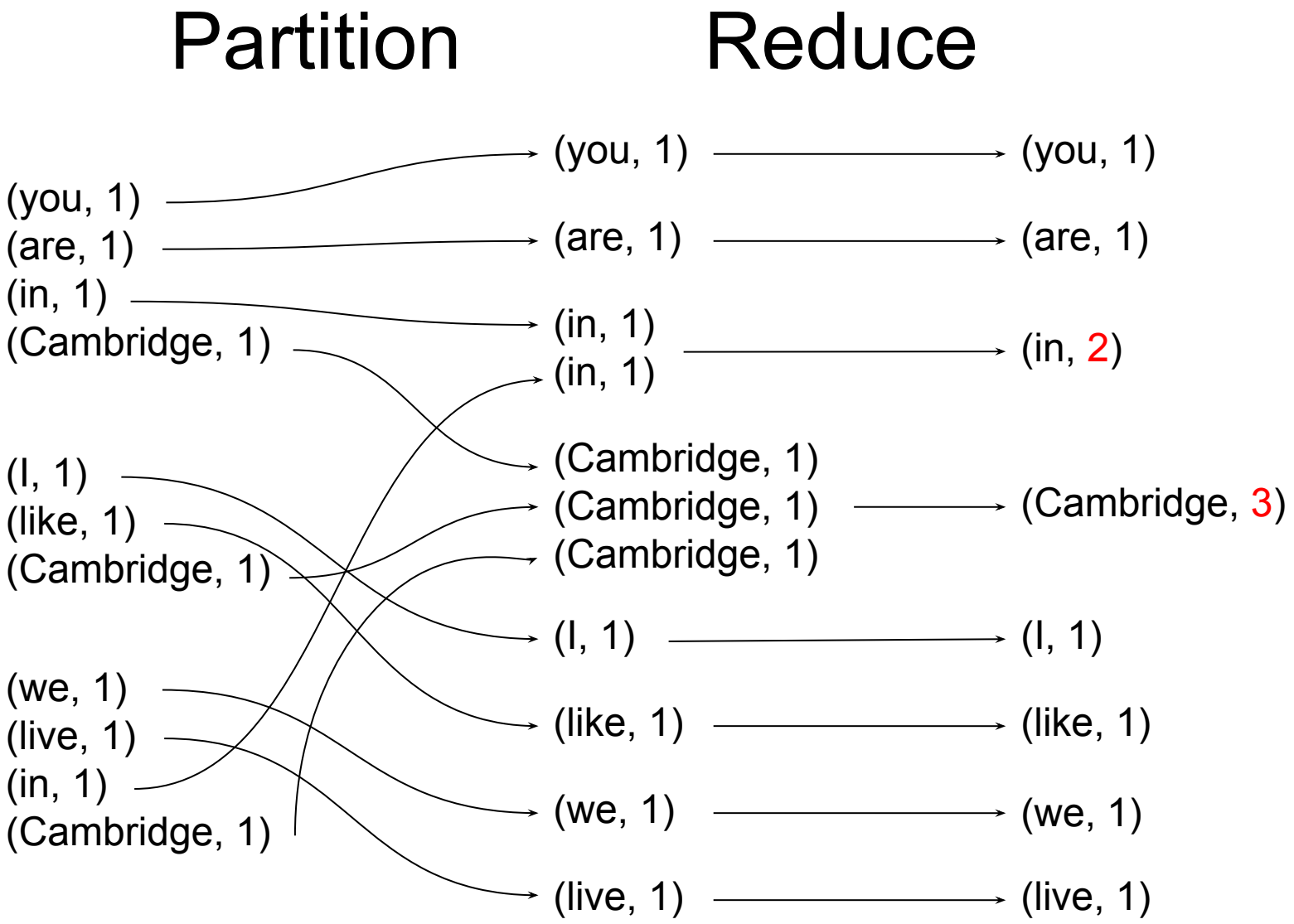
(you, 1)
(are, 1)
(in, 1)
(Cambridge, 1)

(I, 1)
(like, 1)
(Cambridge, 1)

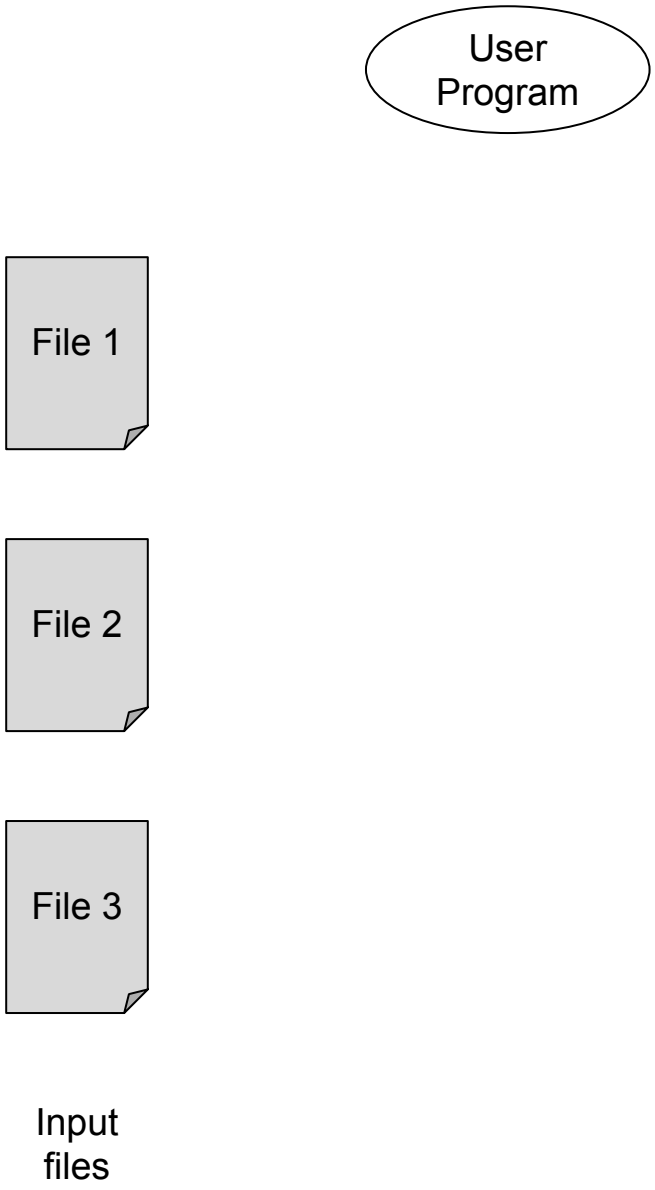
(we, 1)
(live, 1)
(in, 1)
(Cambridge, 1)

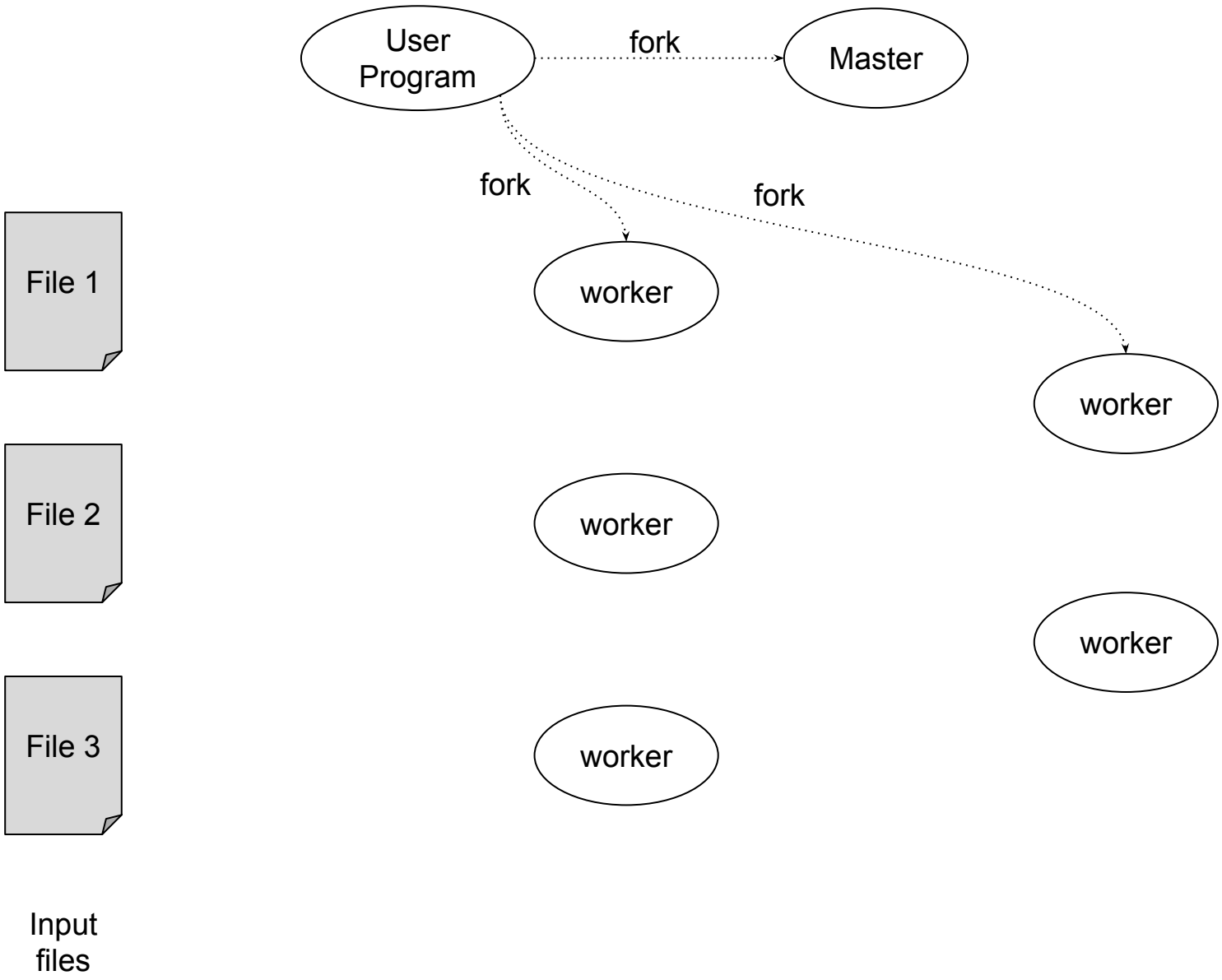
# Partition

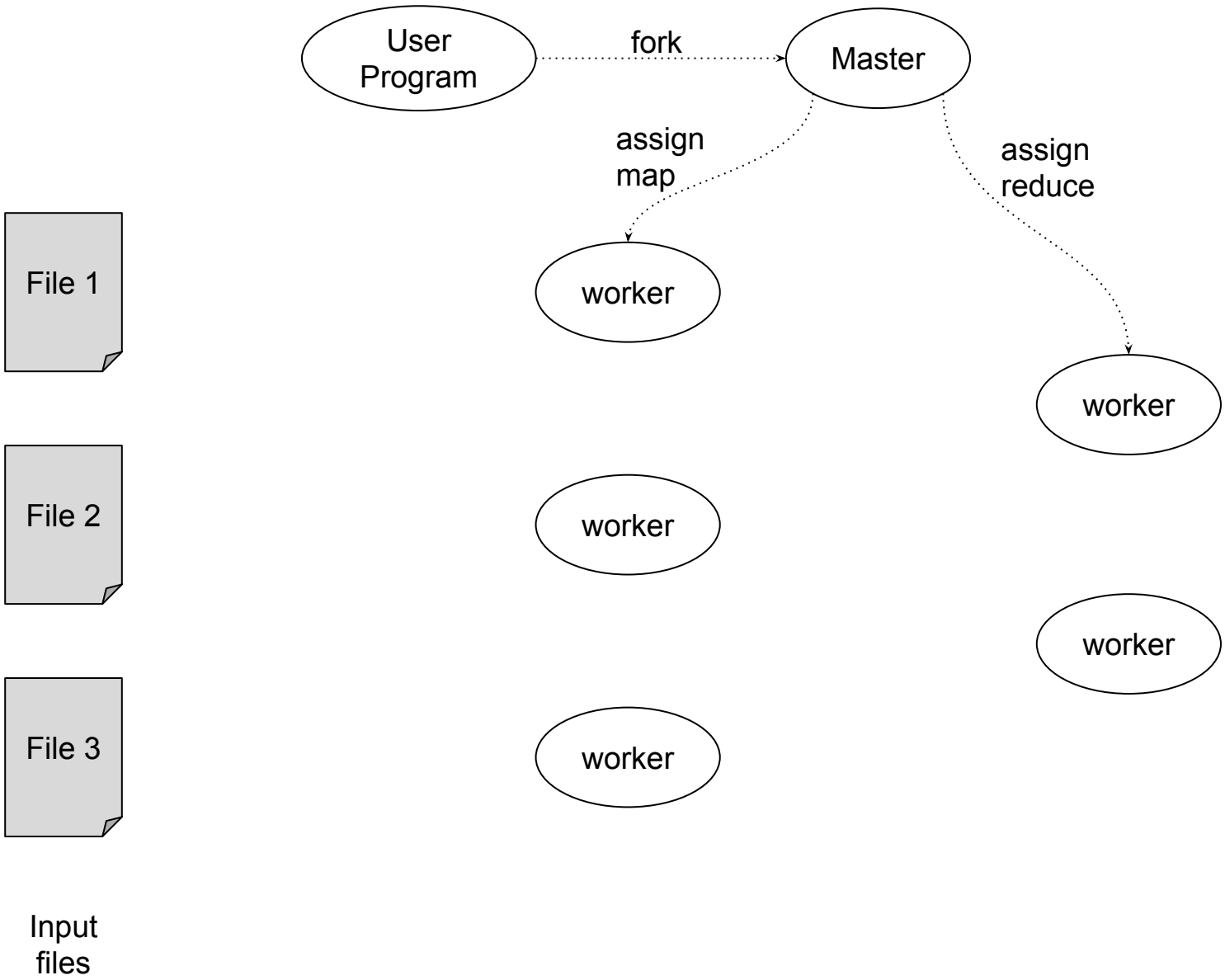

(you, 1)
(are, 1)
(in, 1)
(Cambridge, 1)
(I, 1)
(like, 1)
(Cambridge, 1)
(we, 1)
(live, 1)
(in, 1)
(Cambridge, 1)
(you, 1)
(are, 1)
(in, 1)
(in, 1)
(Cambridge, 1)
(Cambridge, 1)
(Cambridge, 1)
(I, 1)
(like, 1)
(we, 1)
(live, 1)

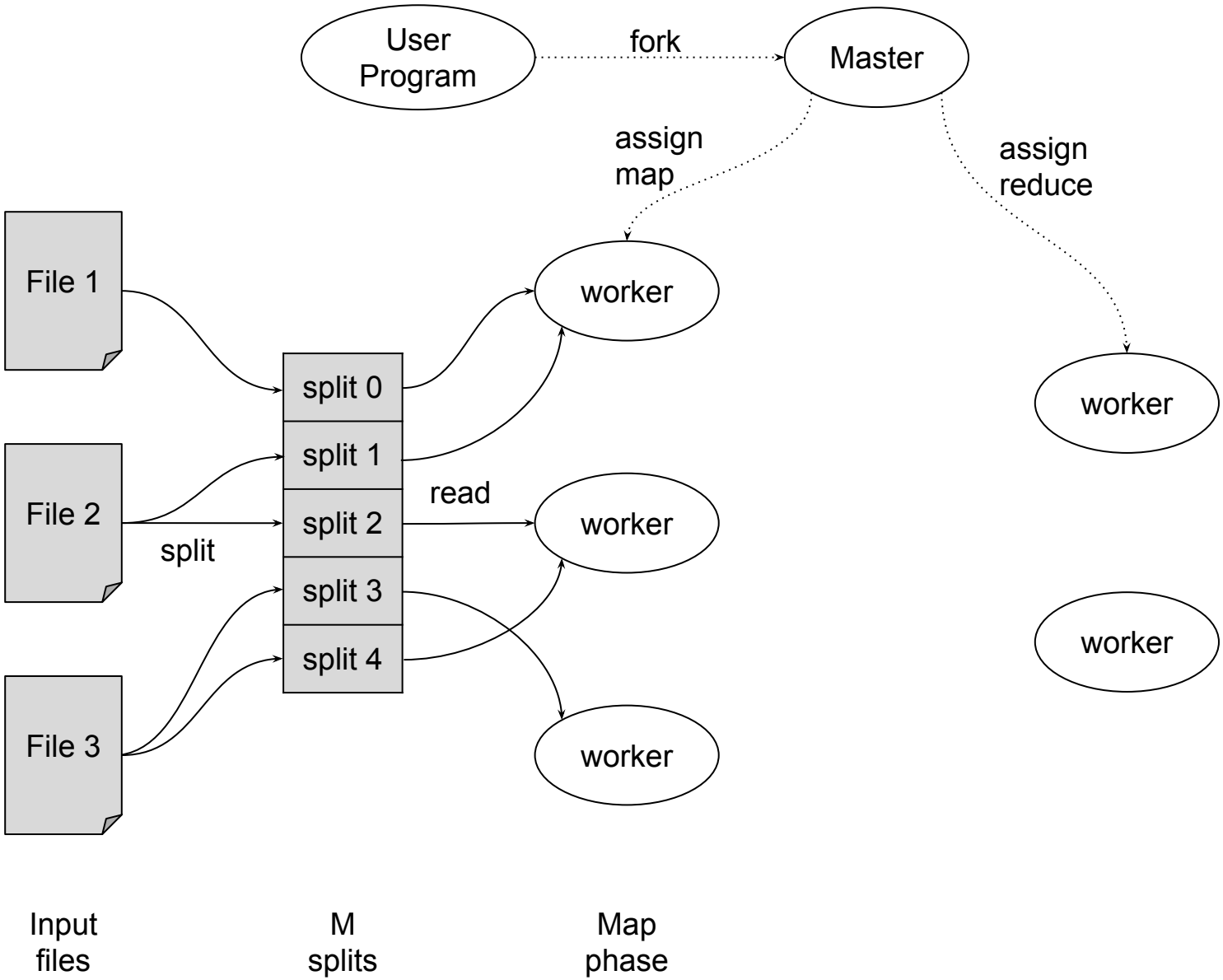


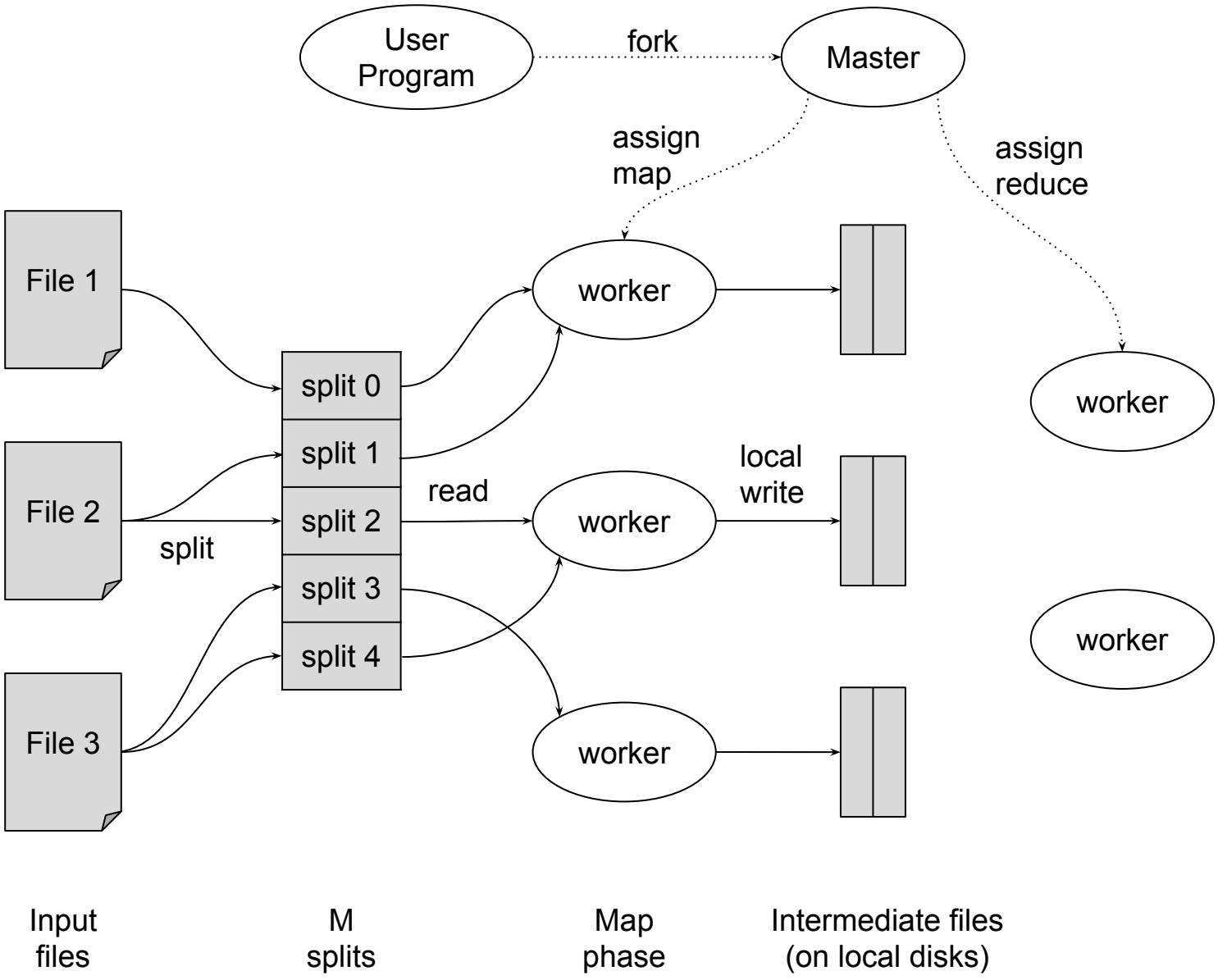

reduce (out_key, list(intermediate_value)) -> list(out_value)

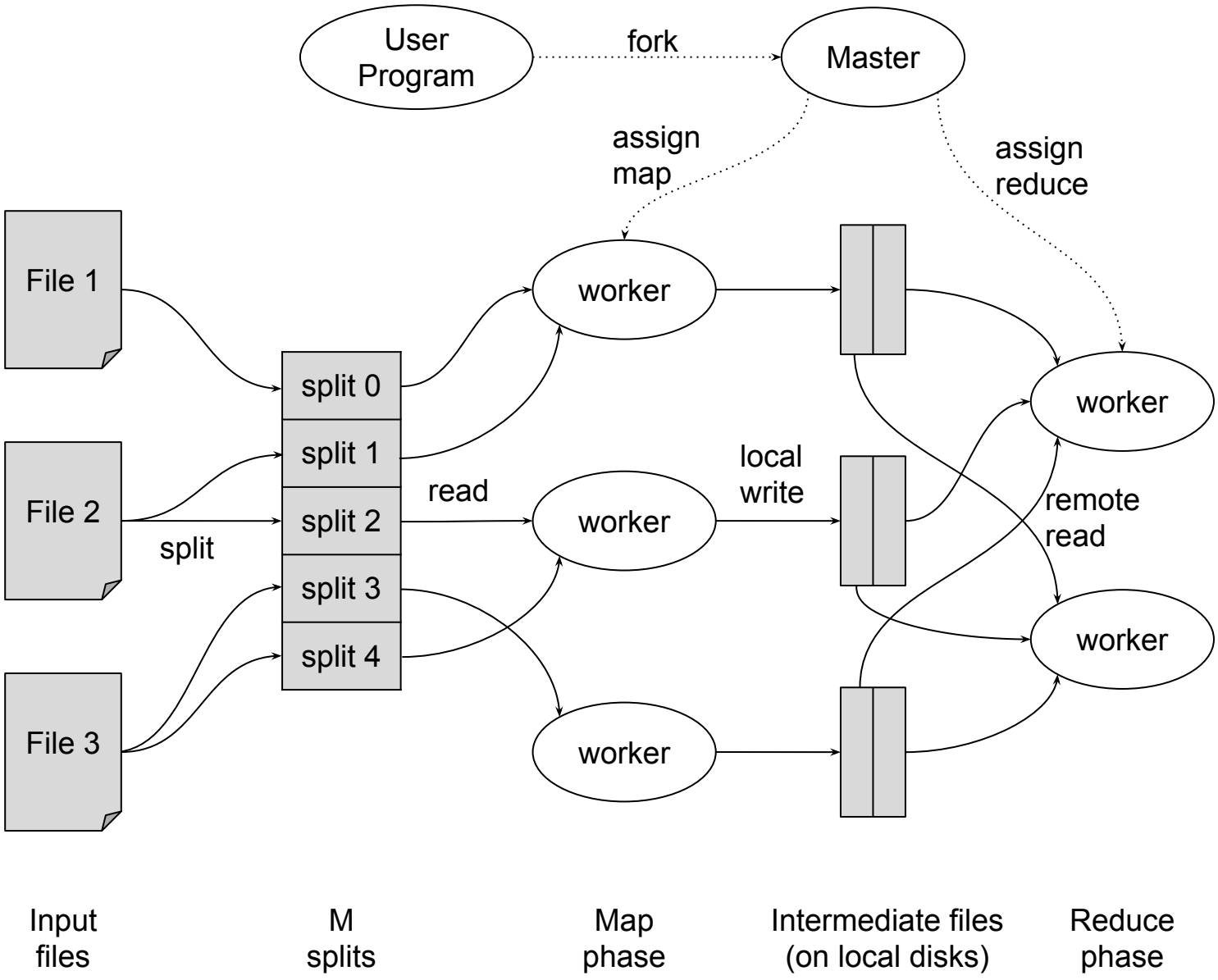

User Program
File 1
File 2
File 3
Input files

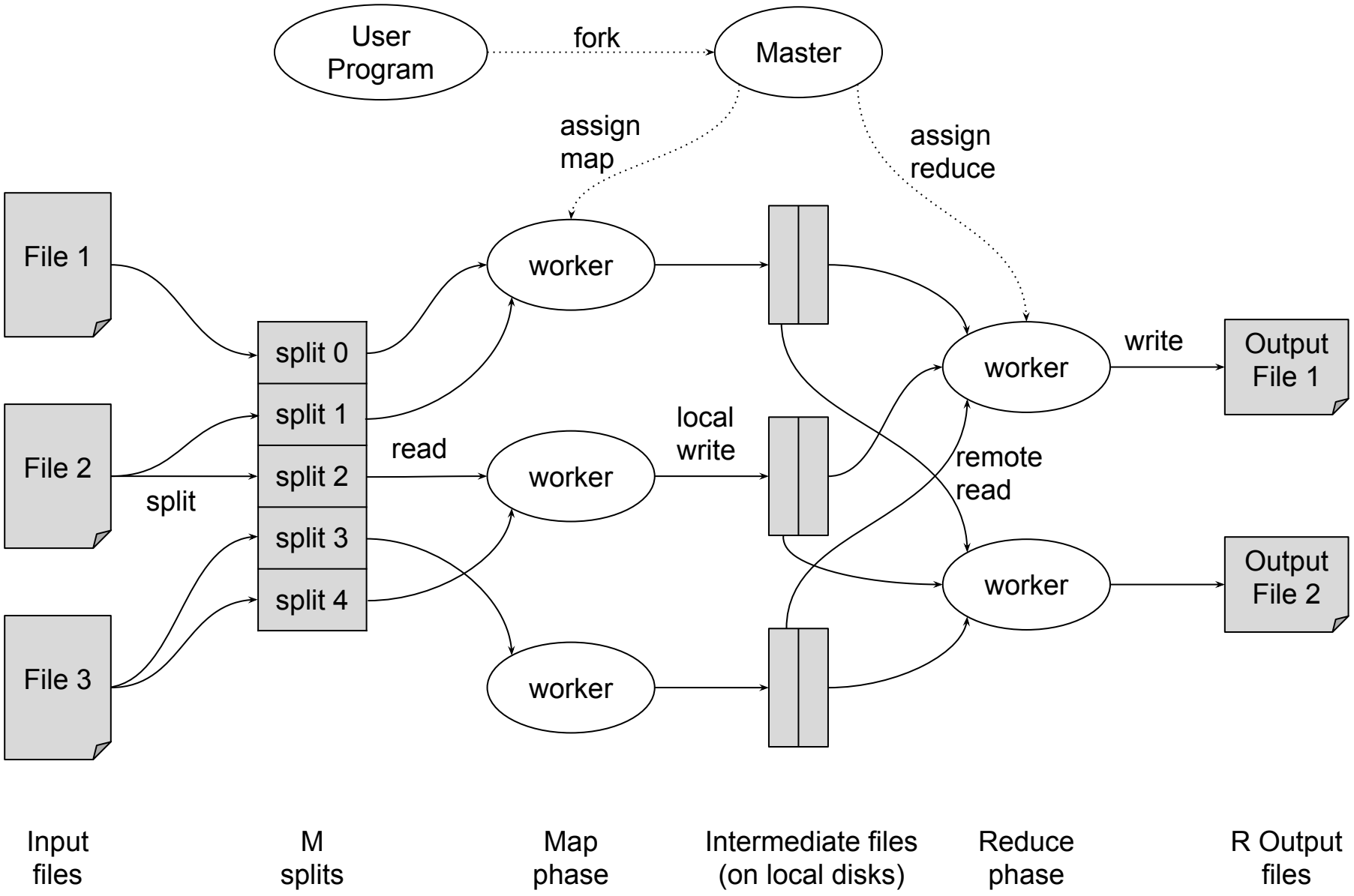

User Program
fork
Master
fork
fork
worker
worker
worker
worker
worker
File 1
File 2
File 3
Input files


User Program
fork
Master
assign map
assign reduce
File 1
File 2
File 3
worker
worker
worker
worker
worker
Input files


User Program
fork
Master
assign map
assign reduce
File 1
File 2
File 3
split
split 0
split 1
split 2
split 3
split 4
read
worker
worker
worker
worker
worker
Input files
M splits
Map phase


User Program
fork
Master
assign map
assign reduce
File 1
File 2
File 3
split
split 0
split 1
split 2
split 3
split 4
read
worker
worker
worker
worker
worker
local write
Input files
M splits
Map phase
Intermediate files (on local disks)


User Program
fork
Master
assign map
assign reduce
File 1
File 2
File 3
split
split 0
split 1
split 2
split 3
split 4
read
worker
worker
worker
local write
worker
worker
remote read
Input files
M splits
Map phase
Intermediate files (on local disks)
Reduce phase


User Program
fork
Master
assign map
assign reduce
File 1
File 2
File 3
split
split 0
split 1
split 2
split 3
split 4
read
worker
worker
worker
local write
worker
worker
write
remote read
Output File 1
Output File 2
Input files
M splits
Map phase
Intermediate files (on local disks)
Reduce phase
R Output files

# Fine task granularity

M so that data is between 16MB and 64MB

R is small multiple of workers

E.g. M = 200,000, R = 5,000 on 2,000 workers

Advantages:

- dynamic load balancing
- fault tolerance

# Fault tolerance

## Workers:

- Detect failure via periodic heartbeat
- Re-execute completed and in-progress *map* tasks
- Re-execute in progress *reduce* tasks
- Task completion committed through master

## Master:

- Not handled - failure unlikely

# Refinements

Locality optimization

Backup tasks

Ordering guarantees

Combiner function

Skipping bad records

Local execution

# Performance

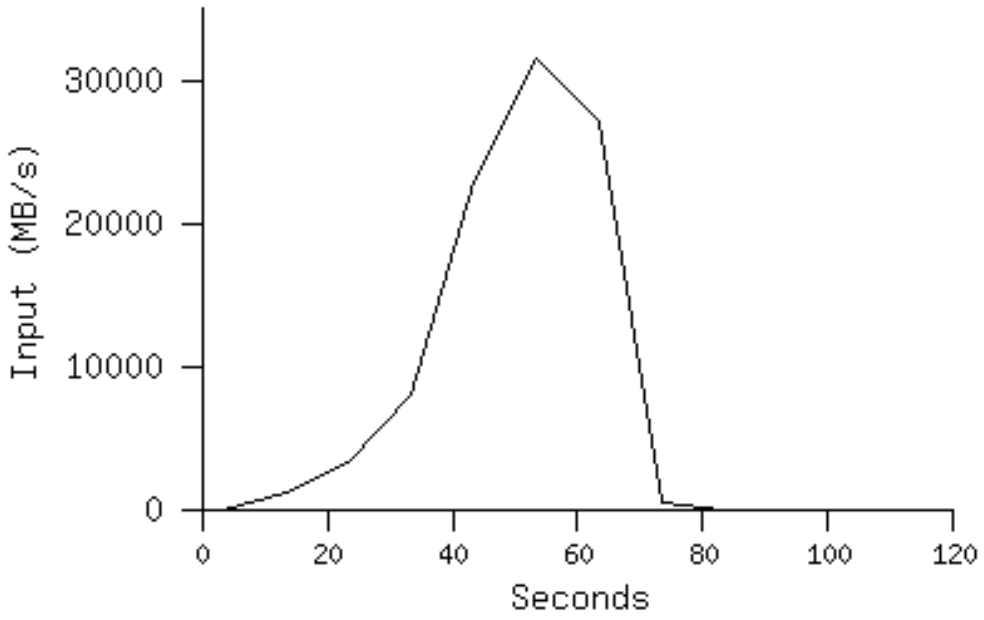
Tests run on 1800 machines:

- Dual 2GHz Intel Xeon processors
  with Hyper-Threading enabled
- 4GB of memory
- Two 160GB IDE disks
- Gigabit Ethernet link

2 Benchmarks:

| | |
|---|---|
| MR_Grep | $10^{10}$ x 100 byte entries, 92k matches |
| MR_Sort | $10^{10}$ x 100 byte entries |

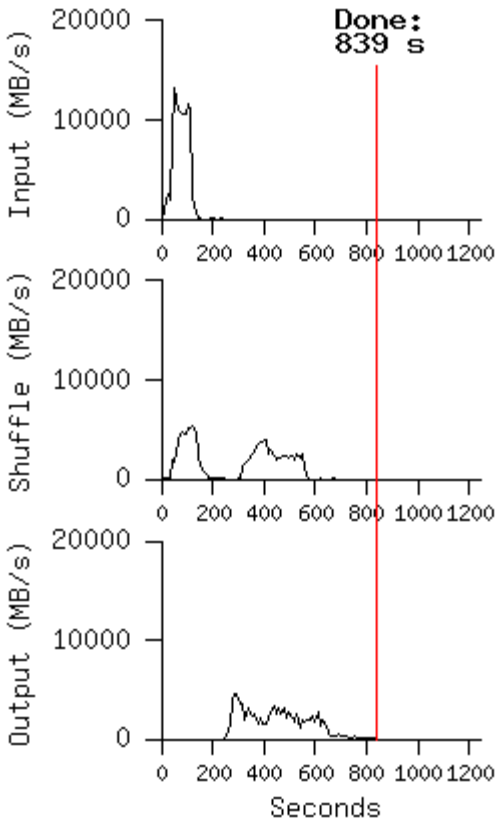
# MR_Grep



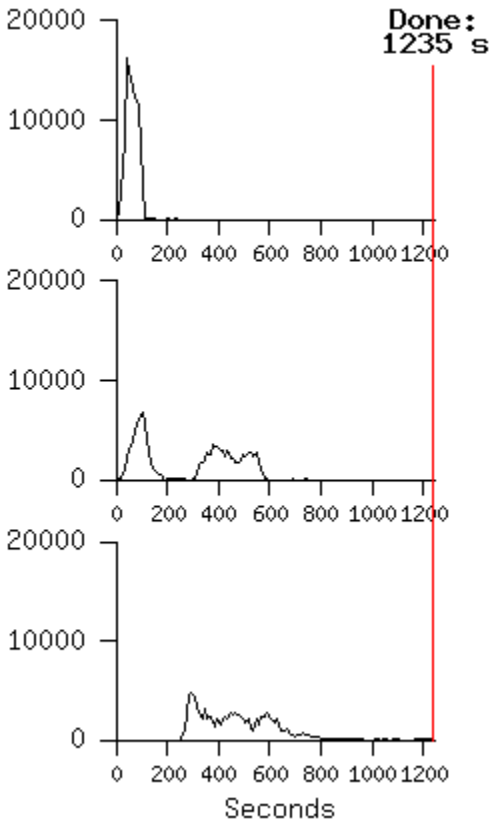

150 seconds run (startup overhead of ~60 seconds)

# MR_Sort

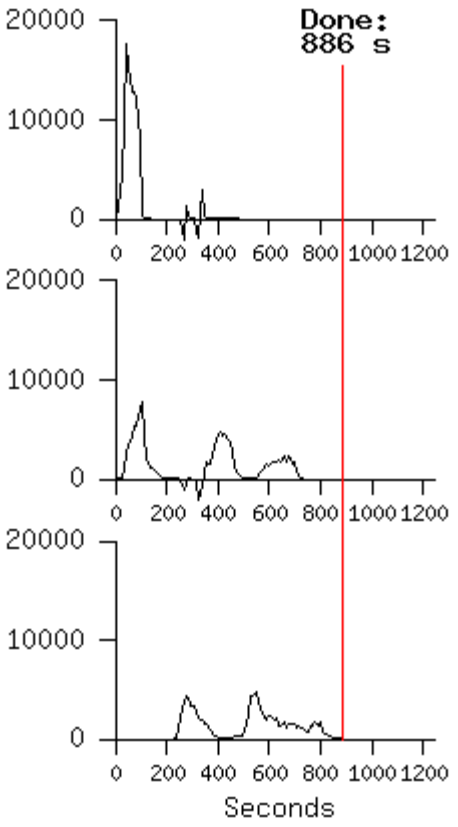
Normal execution




No backup tasks




200 tasks killed



# Experience

Rewrite of the indexing system
for Google web search

Large scale machine learning

Clustering for Google News

Data extraction for Google Zeitgeist

Large scale graph computations

# Conclusions

MapReduce:

- useful abstraction
- simplifies large-scale computations
- easy to use

However:

- expensive for small applications
- long startup time (~1 min)
- chaining of map-reduce phases?