LECTURE 2

MapReduce

Source

- MapReduce: Simplified Data Processing in Large Clusters
  - Jefferey Dean and Sanjay Ghemawat
  - OSDI 2004

Example Scenario

23andMe

- Genome data from roughly one million users
  - 125 MB of data per user

- Goal: Analyze data to identify genes that show susceptibility to Parkinson’s disease

Other Example Scenarios

- Ranking web pages
  - 100 billion web pages
- Selecting ads to show
  - Clickstreams of over one billion users

Lots of Data!

Although the derived tasks are simple, Petabytes or even exabytes of data

- Impossible to store data on one server
- Will take forever to process on one server

Need distributed storage and processing

How to parallelize?

Desirable Properties of Soln.

- Scalable
  - Performance grows with # of machines
- Fault-tolerant
  - Can make progress despite machine failures
- Simple
  - Minimize expertise required of programmer
- Widely applicable
  - Should not restrict kinds of processing feasible

Distributed Data Processing

- Strawman solution:
  - Partition data across servers
  - Have every server process local data
- Why won’t this work?
- Inter-data dependencies:
  - Ranking of a web page depends on ranking of pages that link to it
  - Need data from all users who have a certain gene to evaluate susceptibility to a disease

MapReduce

- Distributed data processing paradigm introduced by Google in 2004
- Popularized by open-source Hadoop framework

- MapReduce represents
  - A programming interface for data processing jobs
    - **Map** and **Reduce** functions
  - A distributed execution framework
    - **Scalable** and **fault-tolerant**

Map Operation

- The Map operation is applied to each “record” to compute a set of intermediate key value pairs.
  - Example → Temperature records between 1951 and 1955
- Map function needs to be written by the user.
- MapReduce Library groups together the values associated with a key I (e.g. year) and passes them to the Reduce function.

Reduce Operation

- Reduce function also written by user.
- Merges together the values provided to form a smaller set of values
  - (e.g., Maximum temperature seen in each year)

MapReduce: Word count

```
map(key, value): //filename, file contents
  for each word w in value:
    EmitIntermediate(w, "1");

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

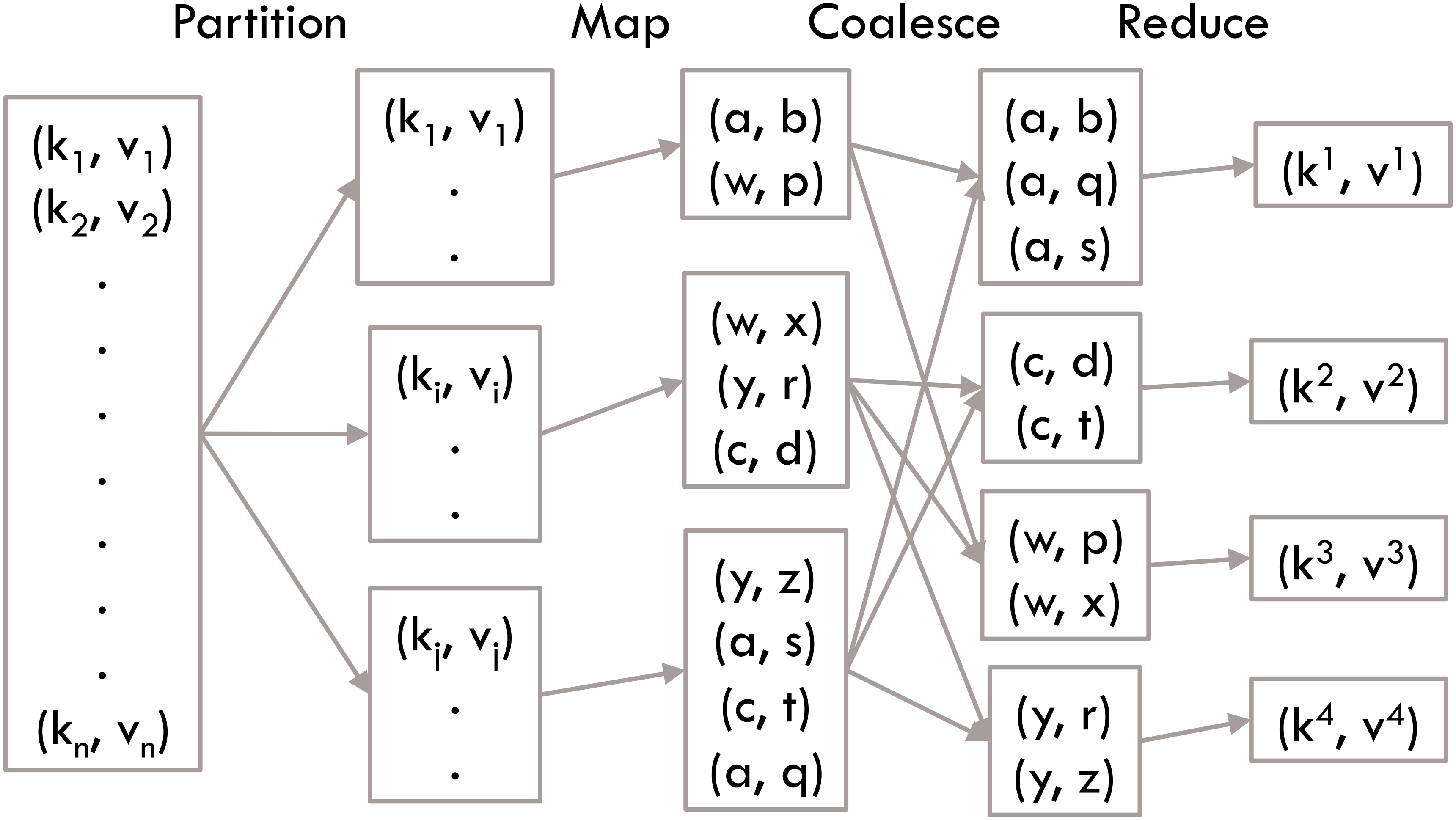
Other examples

- Distributed Grep
  - Map: Emits a line if a match is found to a pattern (key)
  - Reduce: Identity that simply shows the intermediate data
- Count of URL Access frequency
  - Map: Processes log of web page requests and outputs <URL, 1>
  - Reduce: Adds the values for the same URL and outputs <URL, count>

Execution

- Map invocations distributed across multiple machines
  - Need automatic partitioning of input data input to M splits
    - Parallelly process each split
- Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., a hash(key)mod R).

MapReduce Execution



Partition

Map

Coalesce

Reduce

(k_1, v_1)
(k_2, v_2)
.
.
.
.
.
.
.
(k_n, v_n)

(k_1, v_1)
.
.

(k_i, v_i)
.
.

(k_j, v_j)
.
.

(a, b)
(w, p)

(w, x)
(y, r)
(c, d)

(y, z)
(a, s)
(c, t)
(a, q)

(a, b)
(a, q)
(a, s)

(c, d)
(c, t)

(w, p)
(w, x)

(y, r)
(y, z)

(k^1, v^1)

(k^2, v^2)

(k^3, v^3)

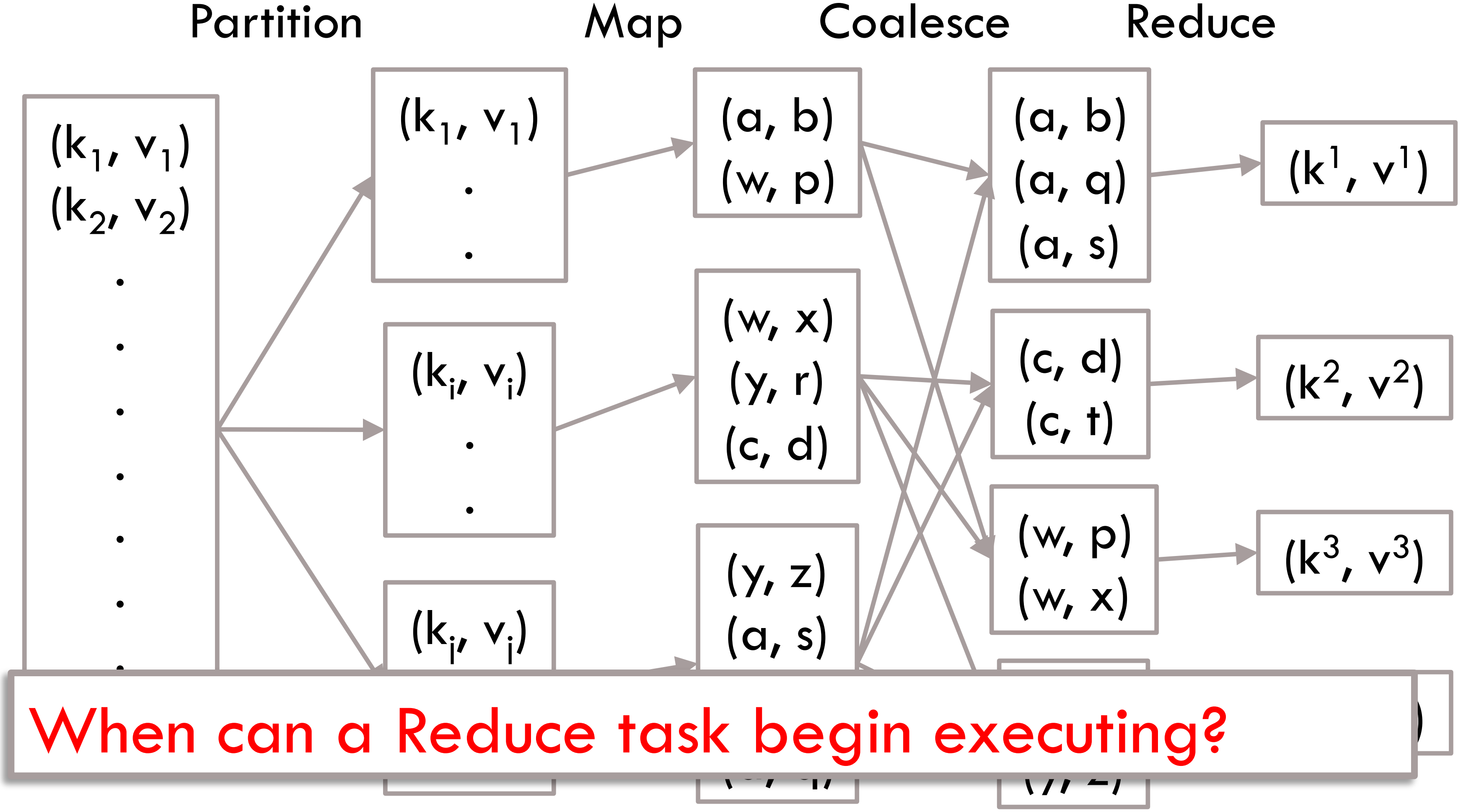
(k^4, v^4)

MapReduce: PageRank

- Compute rank for web page P as average rank of pages that link to P

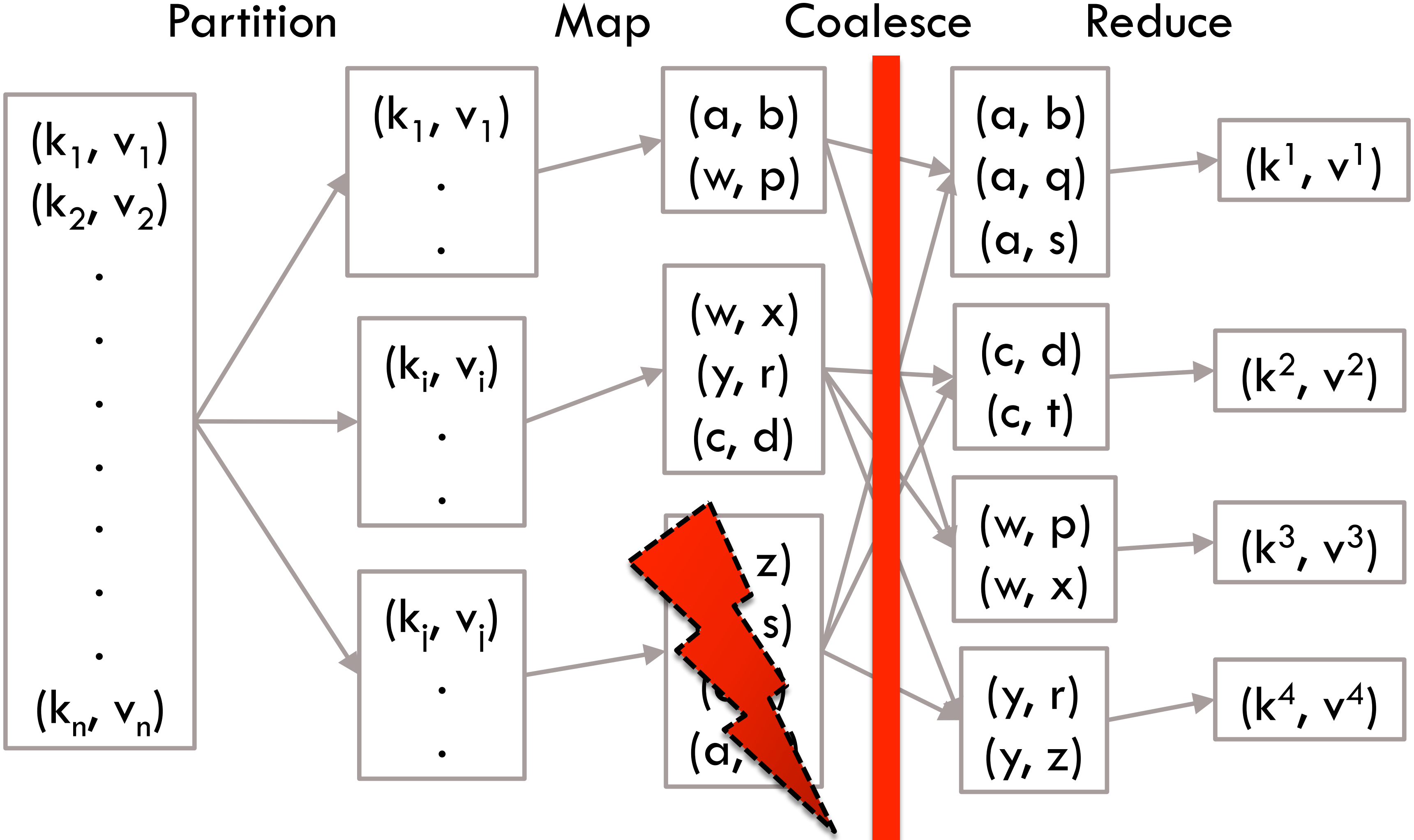
- Initialize rank for every web page to 1
- Map(a web page W, W's contents)
  - For every web page P that W links to, output (P, W)
- Reduce(web page P, {set of pages that link to P})
  - Output rank for P as average rank of pages that link to P
- Run repeatedly until ranks converge

MapReduce Execution



When can a Reduce task begin executing?

Synchronization Barrier



Partition

Map

Coalesce

Reduce

(k_1, v_1)
(k_2, v_2)
.
.
.
.
.
.
.
(k_n, v_n)

(k_1, v_1)
.
.

(k_i, v_i)
.
.

(k_j, v_j)
.
.

(a, b)
(w, p)

(w, x)
(y, r)
(c, d)

z)
s)
(a,

(a, b)
(a, q)
(a, s)

(c, d)
(c, t)

(w, p)
(w, x)

(y, r)
(y, z)

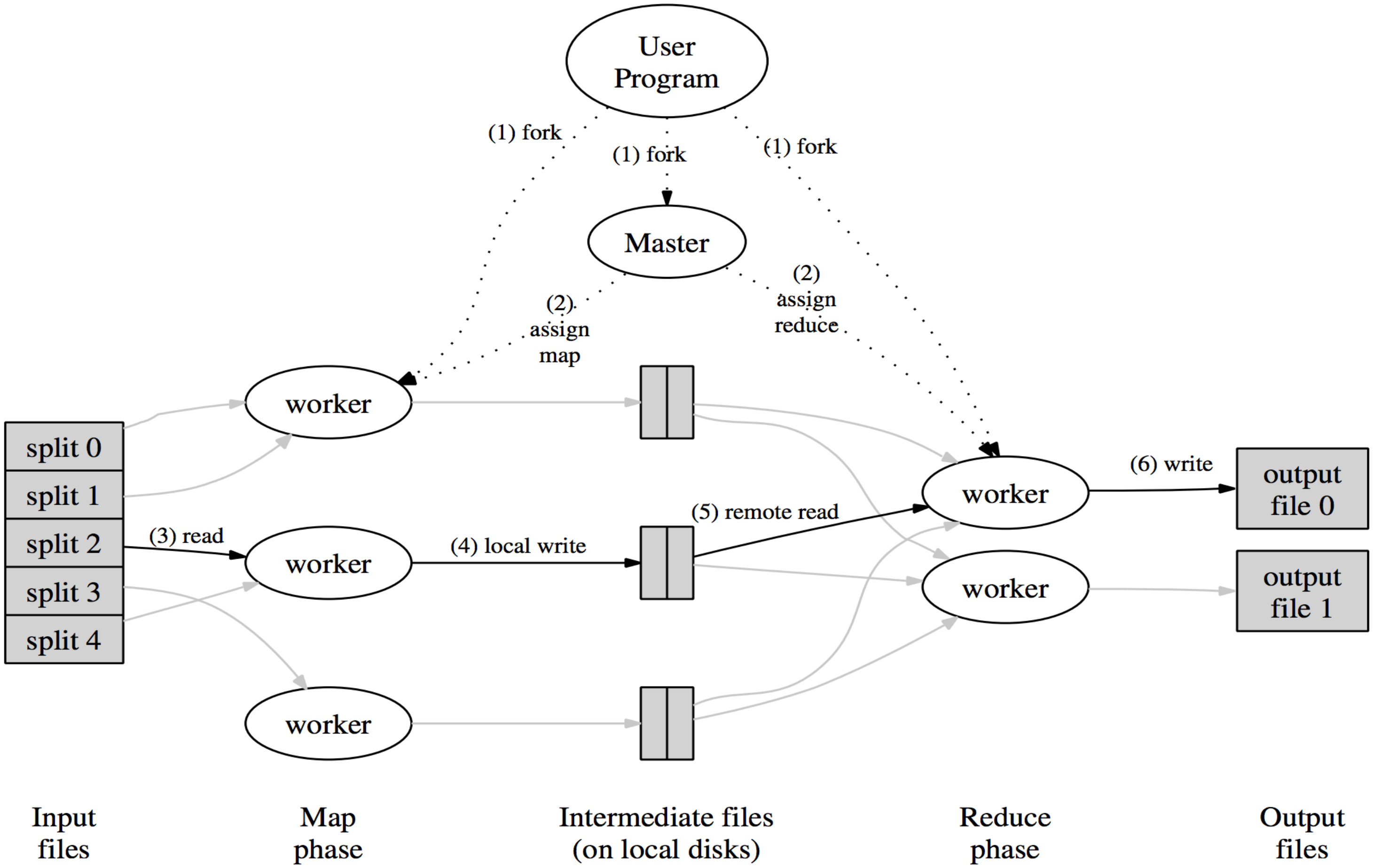
(k^1, v^1)

(k^2, v^2)

(k^3, v^3)

(k^4, v^4)

Fault Tolerance via Master



Workflow (Map)

- MapReduce library in the user program splits input files into M pieces.
- Worker assigned the map task, reads content of the corresponding input split – parses key/value pairs and passes the pair to the user-defined Map function.
  - The intermediate pair produced by Map stored in local memory

Workflow (Reduce)

- The buffered pairs are partitioned into R regions using the partitioning function (e.g., the hash)
- Locations of these pairs are sent to master who sends it to reduce workers.
- Reduce workers uses remote procedure calls to read the buffered data.
- After reading data, it groups them according to the key (sorts).
- It iterates over the intermediate data and for each key encountered.

Failures

- Worker failures
  - Master pings workers periodically.
    - No response within a certain time indicates failure.
    - Tasks reset to idle and reassigned.
      - Note that completed map tasks are re-executed since results stored on local discs and could become inaccessible.
- Master failures (unlikely)
  - Periodically, checkpoints (later) the master state (which tasks are idle, in progress, completed) and the identity of the workers.
  - Return to the last checkpoint.