MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program is composed of a *map* procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a *reduce* method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

What is MapReduce?

- Terms are borrowed from functional language (e.g., Lisp)

Sum of squares:

- (map square '(1 2 3 4))
  - Output: (1 4 9 16)

  [processes each record sequentially and independently]

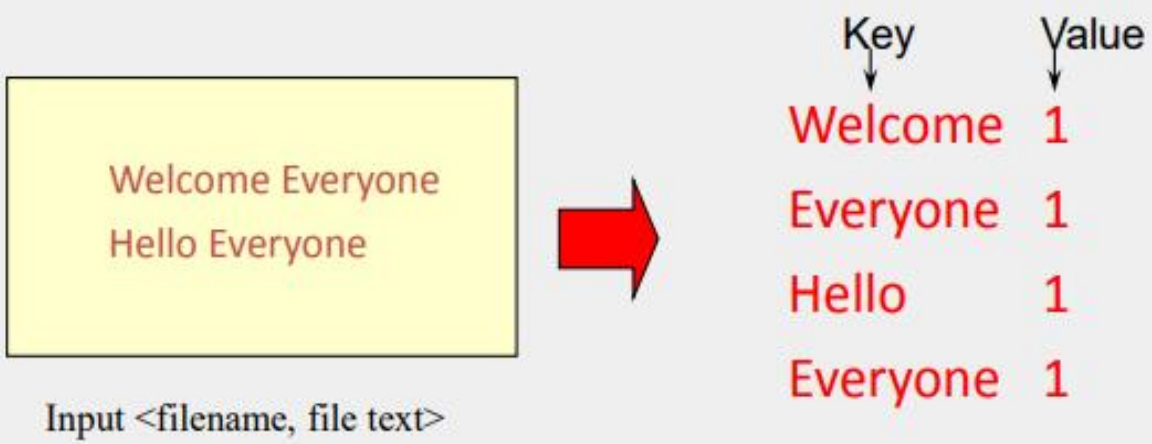
- (reduce + '(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
  - Output: 30

  [processes set of all records in batches]

- Let's consider a sample application: WordCount
  - You are given a huge dataset (e.g., Wikipedia dump or all of Shakespeare's works) and asked to list the count for each of the words in each of the documents therein.

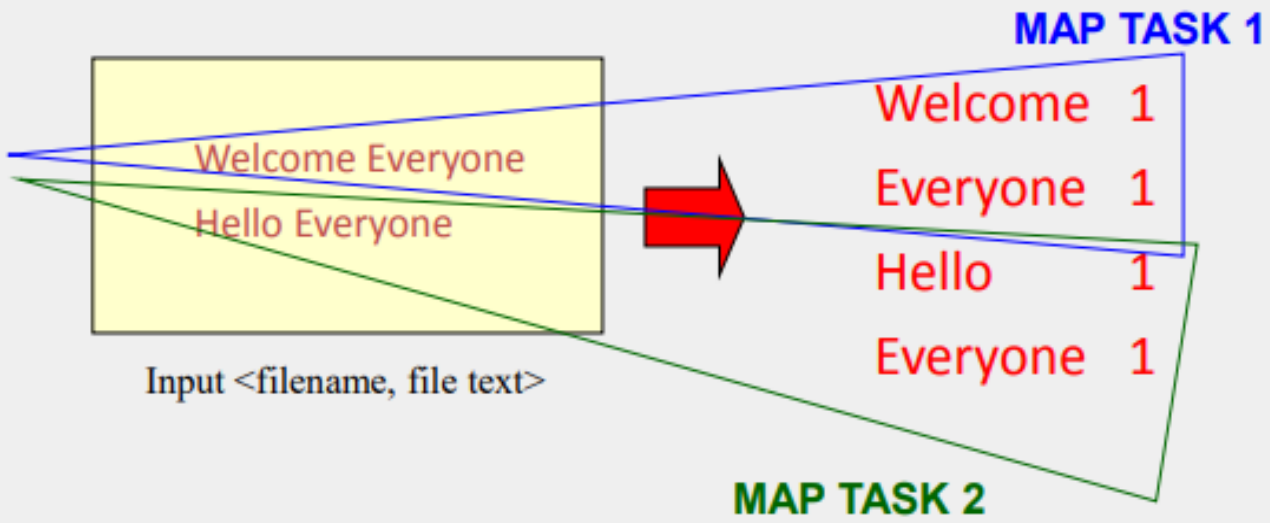
Map

- Process individual records to generate intermediate key/value pairs.



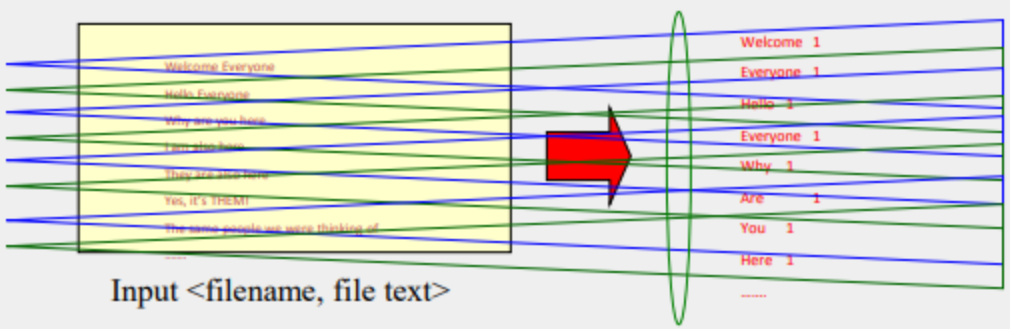
Map

- Parallelly process individual records to generate intermediate key/value pairs.



Map

- Parallelly process a large number of individual records to generate intermediate key/value pairs.



Input <filename, file text>

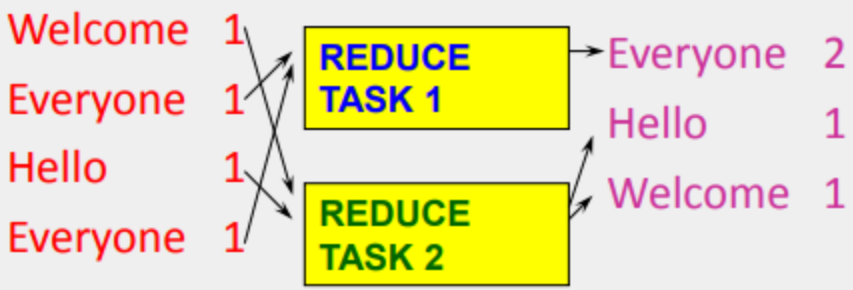
MAP TASKS

Reduce

- Reduce processes and merges all intermediate values associated per key

Reduce

- Each key assigned to one Reduce
- Parallelly processes and merges all intermediate values by partitioning keys



- Popular: *hash partitioning, i.e.,* key is assigned to reduce # = hash(key)%number of reduce servers

Programming MapReduce

Externally: For user

1. Write a Map program (short), write a Reduce program (short)
2. Submit job; wait for result
3. Need to know nothing about parallel/distributed programming!

Internally: For the Paradigm and Scheduler

1. Parallelize Map
2. Transfer data from Map to Reduce
3. Parallelize Reduce
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output

Inside MapReduce

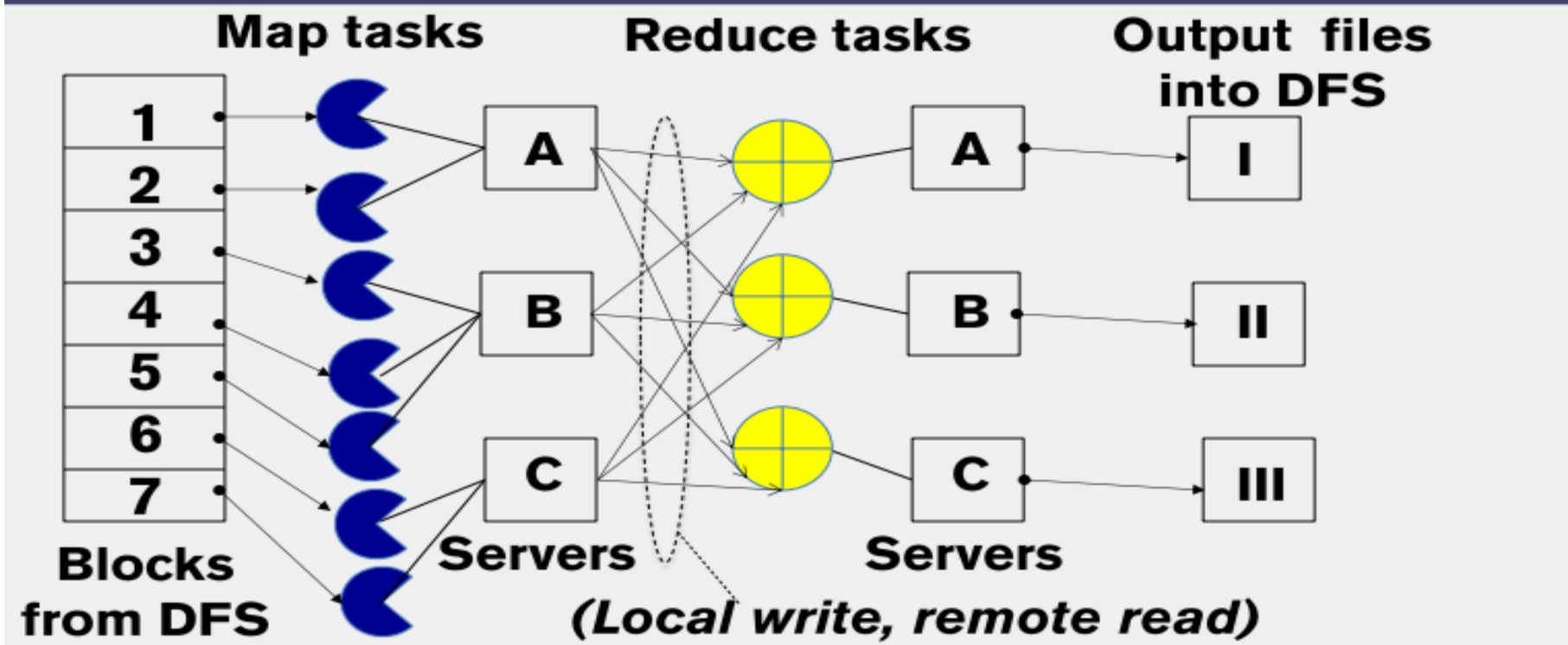
For the cloud:

1. Parallelize Map: easy! each map task is independent of the other!
   - All Map output records with same key assigned to same Reduce
2. Transfer data from Map to Reduce:
   - All Map output records with same key assigned to same Reduce task
   - Use partitioning function, e.g., hash(key)%number of reducers
3. Parallelize Reduce: easy! Each reduce task is independent of the other!
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
   - Map input: from distributed file system
   - Map output: to local disk (at Map node); uses local file system
   - Reduce input: from (multiple) remote disks; uses local file systems
   - Reduce output: to distributed file system

local file system = Linux FS, etc.

distributed file system = GFS (Google File System), HDFS (Hadoop Distributed File System)

Internal Workings of MapReduce

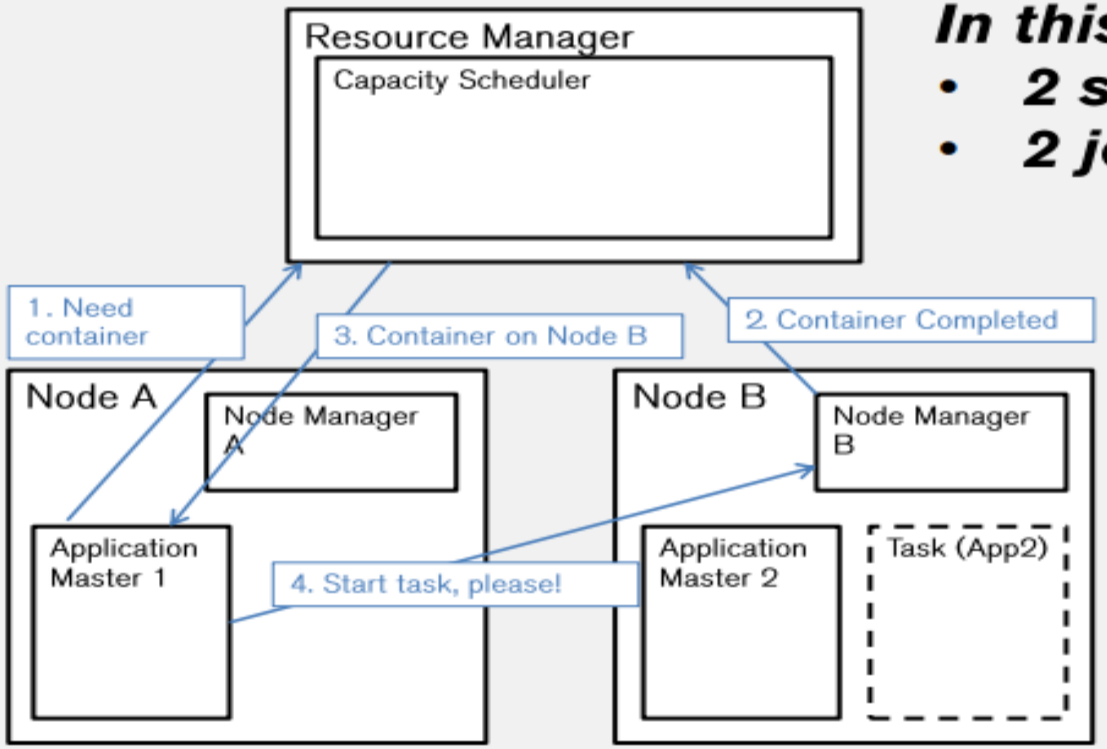


Resource Manager (assigns maps and reduces to servers)

The YARN Scheduler

- Used in Hadoop 2.x +
- YARN = Yet Another Resource Negotiator
- Treats each server as a collection of *containers*
  - Container = some CPU + some memory
- Has 3 main components
  - Global *Resource Manager (RM)*
    - Scheduling
  - Per-server *Node Manager (NM)*
    - Daemon and server-specific functions
  - Per-application (job) *Application Master (AM)*
    - Container negotiation with RM and NMs
    - Detecting task failures of that job

YARN: How a job gets a container



In this figure

- ***2 servers (A, B)***
- ***2 jobs (1, 2)***

Fault Tolerance

- Server failure
  - NM heartbeats to RM
    - If server fails, RM lets all affected AMs know, and AMs take action
  - NM keeps track of each task running at its server
    - If task fails while in-progress, mark the task as idle and restart it
  - AM heartbeats to RM
    - On failure, RM restarts AM, which then syncs up with its running tasks
- RM failure
  - Use old checkpoints and bring up secondary RM
- Heartbeats also used to piggyback container requests
  - Avoids extra messages

Slow Servers

Stragglers (slow nodes)

- The slowest machine slows the entire job down (why?)
- Due to bad disk, network bandwidth, CPU, or memory
- Keep track of "progress" of each task (% done)
- Perform backup (replicated) execution of straggler task: task considered done when first replica complete. Called speculative execution.

Locality

- Locality
  - Since cloud has hierarchical topology (e.g., racks)
  - GFS/HDFS stores 3 replicas of each of chunks (e.g., 64 MB in size)
    - Maybe on different racks, e.g., 2 on a rack, 1 on a different rack
  - MapReduce attempts to schedule a map task on
    - A machine that contains a replica of corresponding input data, or failing that,
    - On the same rack as a machine containing the input, or failing that,
    - Anywhere

MapReduce: Summary

- MapReduce uses parallelization + aggregation to schedule applications across clusters
- Need to deal with failure
- Plenty of ongoing research work in scheduling and fault-tolerance for MapReduce and Hadoop