SWE404/DMT413
BIG DATA ANALYTICS

Lecture 4: MapReduce and YARN

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

hadoop
Map Reduce

How to Use Data

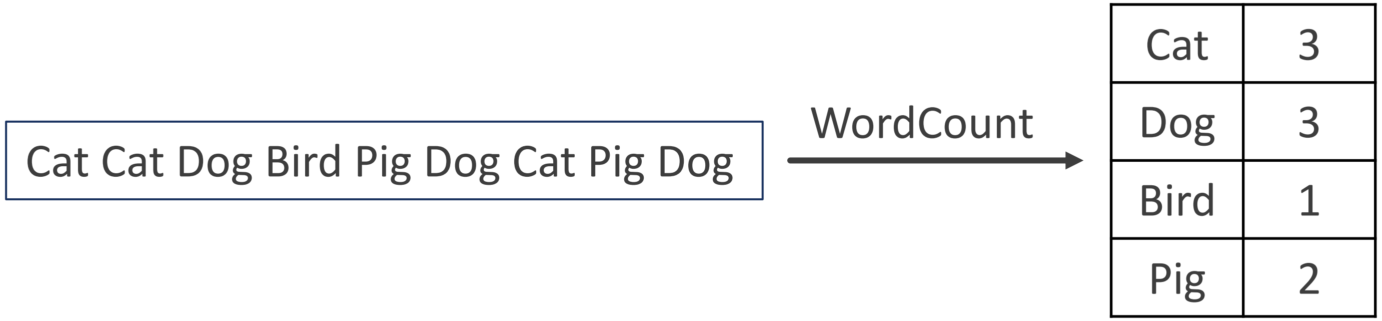
- After storing data in HDFS, the first question that comes to mind is “How can I analyze or query my data?”
  - Storing data is not the job for an analytics.
- Transferring all this data to a central node for processing is impractical.
  - If you can do this, why do you need to store data in parallel?
- HDFS has a number of the benefits : low cost, fault-tolerant, and easily scalable, to name just a few.
  - HDFS is designed for storing data in parallel. Now we need a tool to process data in parallel.
- *MapReduce* is the solution!
  - MapReduce integrates with HDFS to provide the exact same benefits for data processing.

MapReduce

- MapReduce is a *programming model*.
  - MapReduce itself is not an algorithm that you can directly use.
  - Design your application under the framework of MapReduce.
- MapReduce enables skilled programmers to write distributed applications without having to worry about the underlying distributed computing infrastructure.
  - Just like HDFS, you don’t need to care about how data is stored.
  - Similarly in MapReduce, you don’t need to care about how to write distributed algorithm once you have well defined the two functions: *map* and *reduce*.

WordCount Example

- The application WordCount requires to count the frequency of each word in a document.
  - This document may be extremly large.



Cat Cat Dog Bird Pig Dog Cat Pig Dog

Cat	3
Dog	3
Bird	1
Pig	2

Serial WordCount Algorithm

- The reality is that you would not be able to take this elegantly simple code and run it successfully on data stored in a distributed system.

```
create a two-dimensional array
    create a row for every word count
        populate the first column with the word
        populate the second column with the integer zero

read the document
    for each word
        find the row in the array that matches the word
        increment the counter in the second column by one
```

Parellel WordCount Algorithm

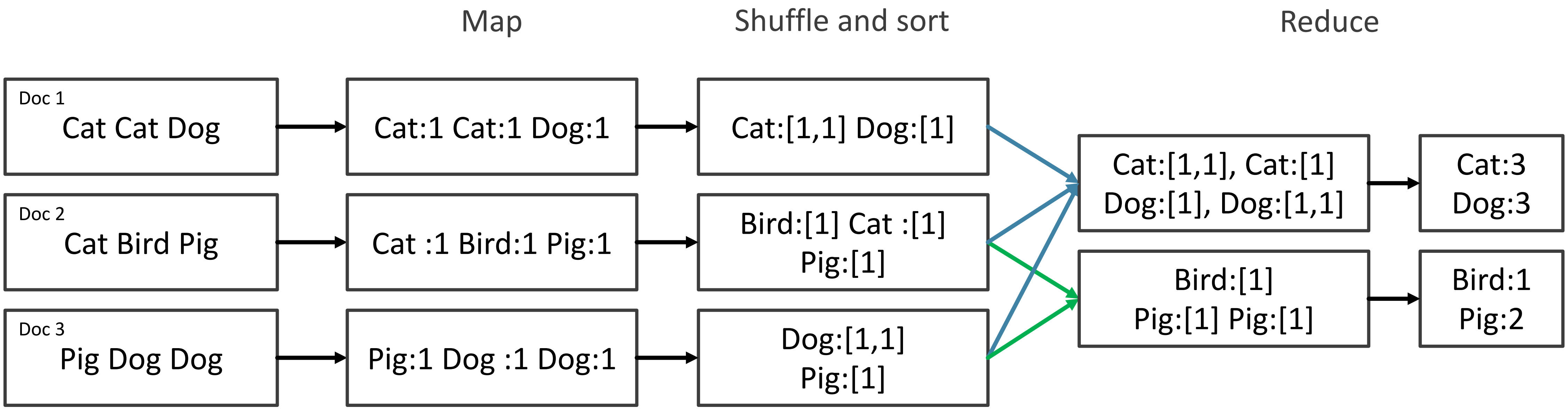
1. Map phase *(you need to implement)*
   - The map operation is run against every data block of the document individually.
   - We export a key/value pair, with the word as the key and the value being an integer one.
2. Shuffle and sort phase *(you don’t need to care)*
   - Ensure that all the values (the integer ones) are grouped together for each key.
   - Sort by key.
3. Reduce phase *(you need to implement)*
   - Add the total number of ones together for each word.

```
Map Phase:
    read the current block
        output the word and integer one as a key/value pair

Shuffle and Sort Phase:
    read the list of key/value pairs from the map phase
    group all the values with the same key together
        each key has a corresponding array of values
    sort the data by key
    output each key and its array of values

Reduce Phase:
    read the list of words and arrays of values
    for each word
        add the arrays of values together
```

MapReduce Processing Flow



Map

Shuffle and sort

Reduce

Doc 1
Cat Cat Dog

Doc 2
Cat Bird Pig

Doc 3
Pig Dog Dog

Cat:1 Cat:1 Dog:1

Cat :1 Bird:1 Pig:1

Pig:1 Dog :1 Dog:1

Cat:[1,1] Dog:[1]

Bird:[1] Cat :[1]
Pig:[1]

Dog:[1,1]
Pig:[1]

Cat:[1,1], Cat:[1]
Dog:[1], Dog:[1,1]

Bird:[1]
Pig:[1] Pig:[1]

Cat:3
Dog:3

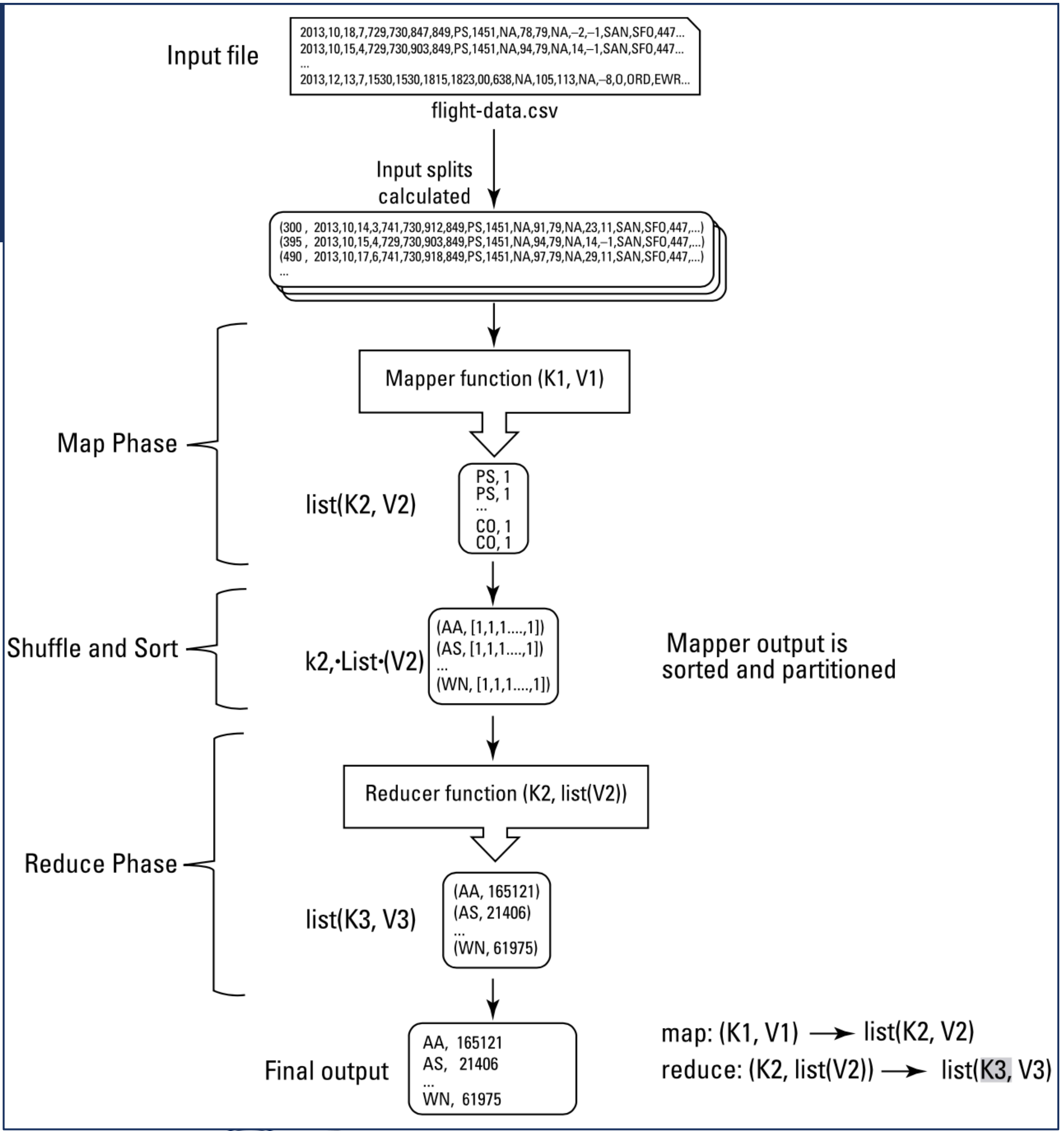
Bird:1
Pig:2

Thinking in Parallel

- Although MapReduce hides a tremendous amount of complexity, you still need to know how to program in parallel.
- To become a MapReduce programmer and thinking problems in parallel isn’t that easy.

Map Phase

- The input to the map function is organized in key/value pair as (K1, V1).
  - V1 is each record (e.g. document for WordCount) and K1 is the index.
- The map function generates a list of (K2, V2).
  - K2 and V2 are completely different from K1 and V1.
- What you need to do in this phase:
  - Delicately *design your map function* and determine what K2 is and what V2 should be for each K2 for your application.



Input file

2013,10,18,7,729,730,847,849,PS,1451,NA,78,79,NA,–2,–1,SAN,SFO,447...
2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447...
...
2013,12,13,7,1530,1530,1815,1823,00,638,NA,105,113,NA,–8,0,ORD,EWR...

flight-data.csv

Input splits calculated

(300 , 2013,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,...)
(395 , 2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447,...)
(490 , 2013,10,17,6,741,730,918,849,PS,1451,NA,97,79,NA,29,11,SAN,SFO,447,...)
...

Map Phase

Mapper function (K1, V1)

list(K2, V2)

PS, 1
PS, 1
...
CO, 1
CO, 1

Shuffle and Sort

k2,·List·(V2)

(AA, [1,1,1....,1])
(AS, [1,1,1....,1])
...
(WN, [1,1,1....,1])

Mapper output is sorted and partitioned

Reduce Phase

Reducer function (K2, list(V2))

list(K3, V3)

(AA, 165121)
(AS, 21406)
...
(WN, 61975)

Final output

AA, 165121
AS, 21406
...
WN, 61975

map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

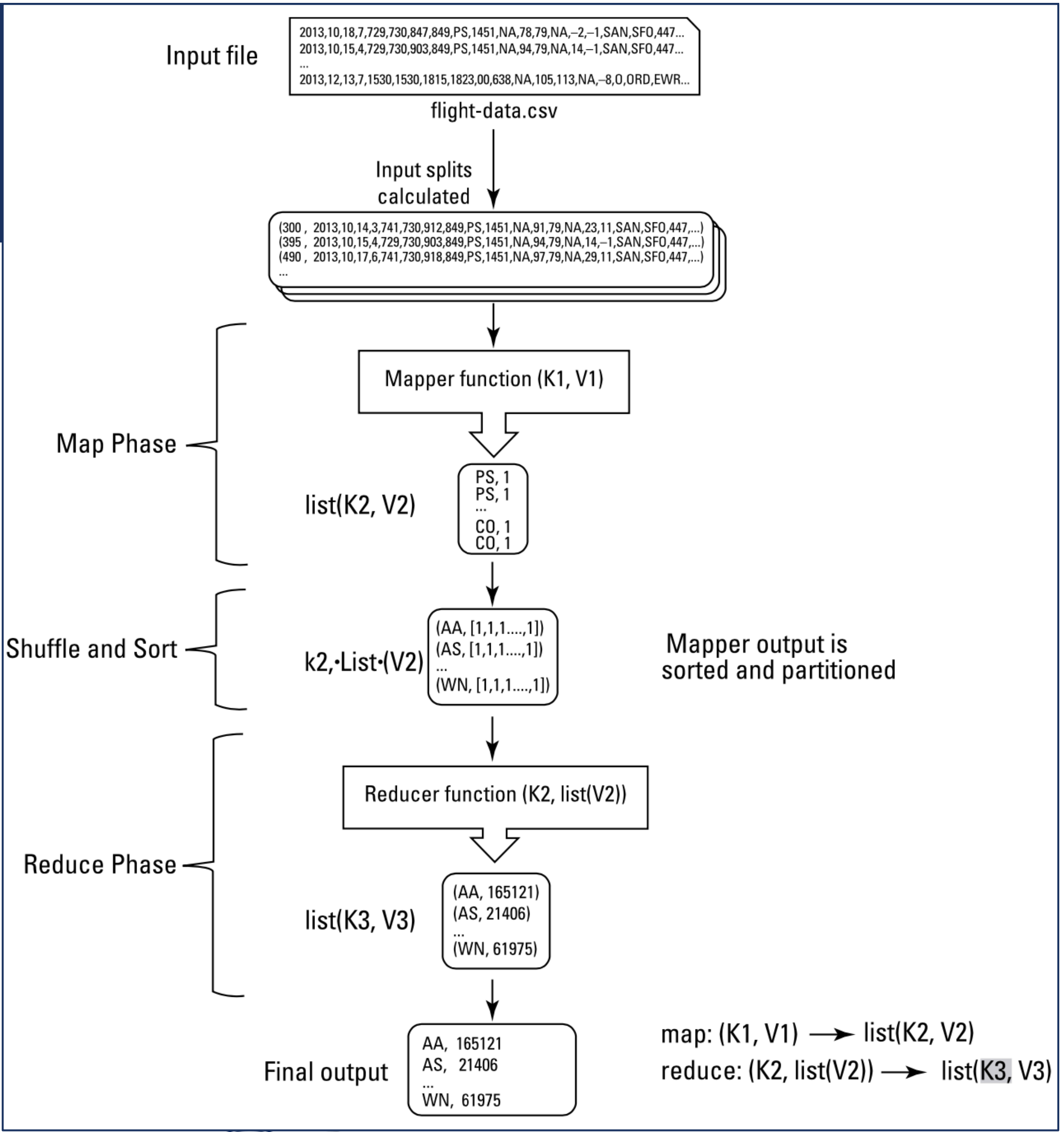
XIAMEN UNIVERSITY MALAYSIA
厦門大學馬來西亞分校

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系
Computer Science Department of Xiamen University

Shuffle and Sort Phase

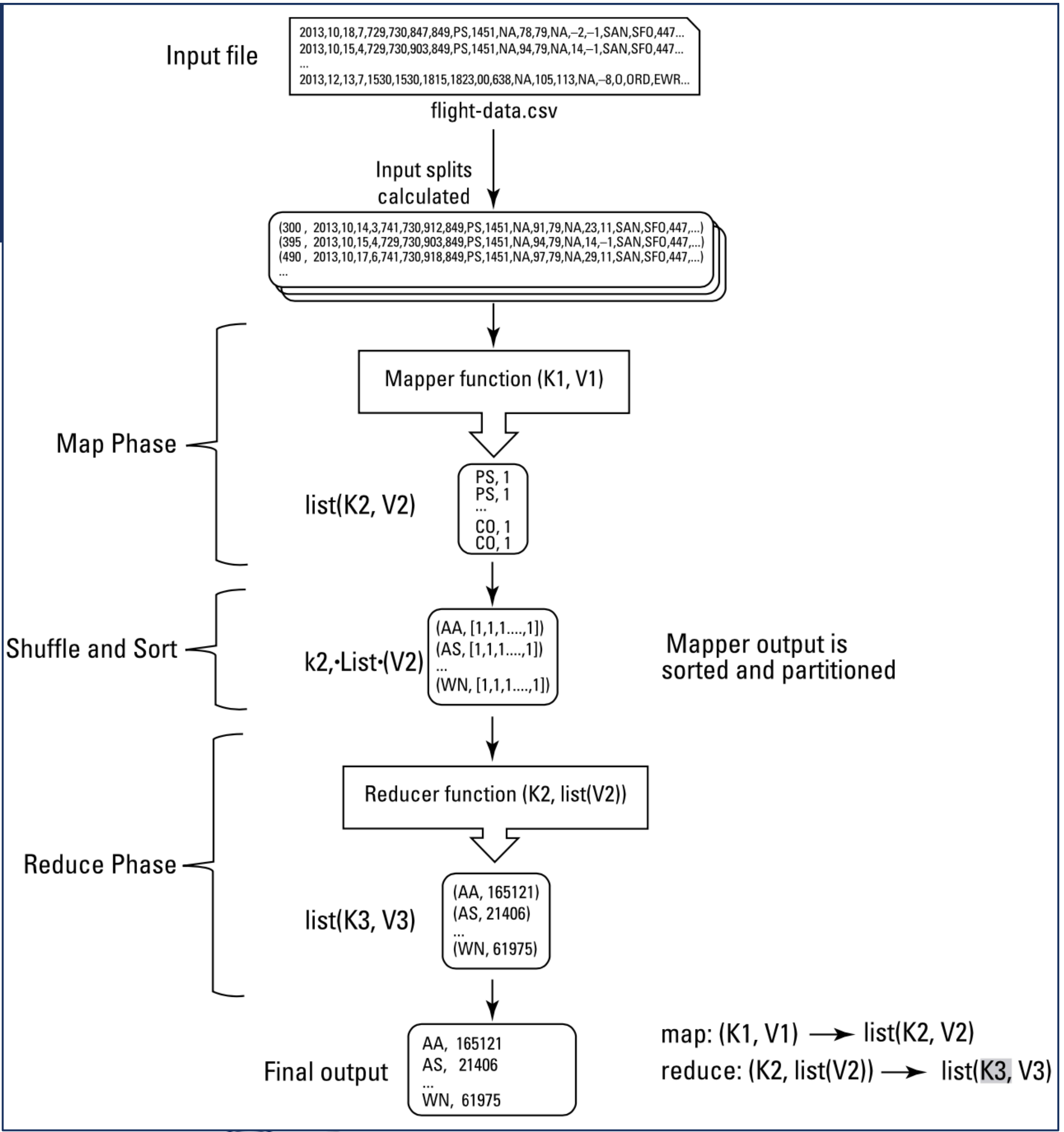
- Still on the mapper's nodes, all of the values for each K2 are grouped together, resulting in: K2, list(V2).
- K2, list(V2) is then written into the disk of the mapper's node according to K2.
  - The number of files to write depends on the number of the reducer's nodes assigned.
  - Each file contains a set of keys.
- What you need to do in this phase:
  - Nothing. It is automatically done once you defined your key in the map function.



Input file
2013,10,18,7,729,730,847,849,PS,1451,NA,78,79,NA,–2,–1,SAN,SFO,447...
2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447...
...
2013,12,13,7,1530,1530,1815,1823,00,638,NA,105,113,NA,–8,0,ORD,EWR...
flight-data.csv
Input splits calculated
(300 , 2013,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,...)
(395 , 2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447,...)
(490 , 2013,10,17,6,741,730,918,849,PS,1451,NA,97,79,NA,29,11,SAN,SFO,447,...)
...
Map Phase
Mapper function (K1, V1)
list(K2, V2)
PS, 1
PS, 1
...
CO, 1
CO, 1
Shuffle and Sort
k2,·List·(V2)
(AA, [1,1,1....,1])
(AS, [1,1,1....,1])
...
(WN, [1,1,1....,1])
Mapper output is sorted and partitioned
Reduce Phase
Reducer function (K2, list(V2))
list(K3, V3)
(AA, 165121)
(AS, 21406)
...
(WN, 61975)
Final output
AA, 165121
AS, 21406
...
WN, 61975
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

Shuffle and Sort Phase

- The output from mapper tasks isn't written to HDFS, but rather to local disk on the slave node where the mapper task was run.
- As such, it's not replicated across the Hadoop cluster.
- Why?
  - The output is processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away.
  - Storing it in HDFS with replication would be overkill.



Input file

```
2013,10,18,7,729,730,847,849,PS,1451,NA,78,79,NA,–2,–1,SAN,SFO,447...
2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447...
...
2013,12,13,7,1530,1530,1815,1823,00,638,NA,105,113,NA,–8,0,ORD,EWR...
```

flight-data.csv

Input splits calculated

```
(300 ,  2013,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,...)
(395 ,  2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447,...)
(490 ,  2013,10,17,6,741,730,918,849,PS,1451,NA,97,79,NA,29,11,SAN,SFO,447,...)
...
```

Map Phase

Mapper function (K1, V1)

list(K2, V2)

PS, 1
PS, 1
...
CO, 1
CO, 1

Shuffle and Sort

k2,·List·(V2)

(AA, [1,1,1....,1])
(AS, [1,1,1....,1])
...
(WN, [1,1,1....,1])

Mapper output is sorted and partitioned

Reduce Phase

Reducer function (K2, list(V2))

list(K3, V3)

(AA, 165121)
(AS, 21406)
...
(WN, 61975)

Final output

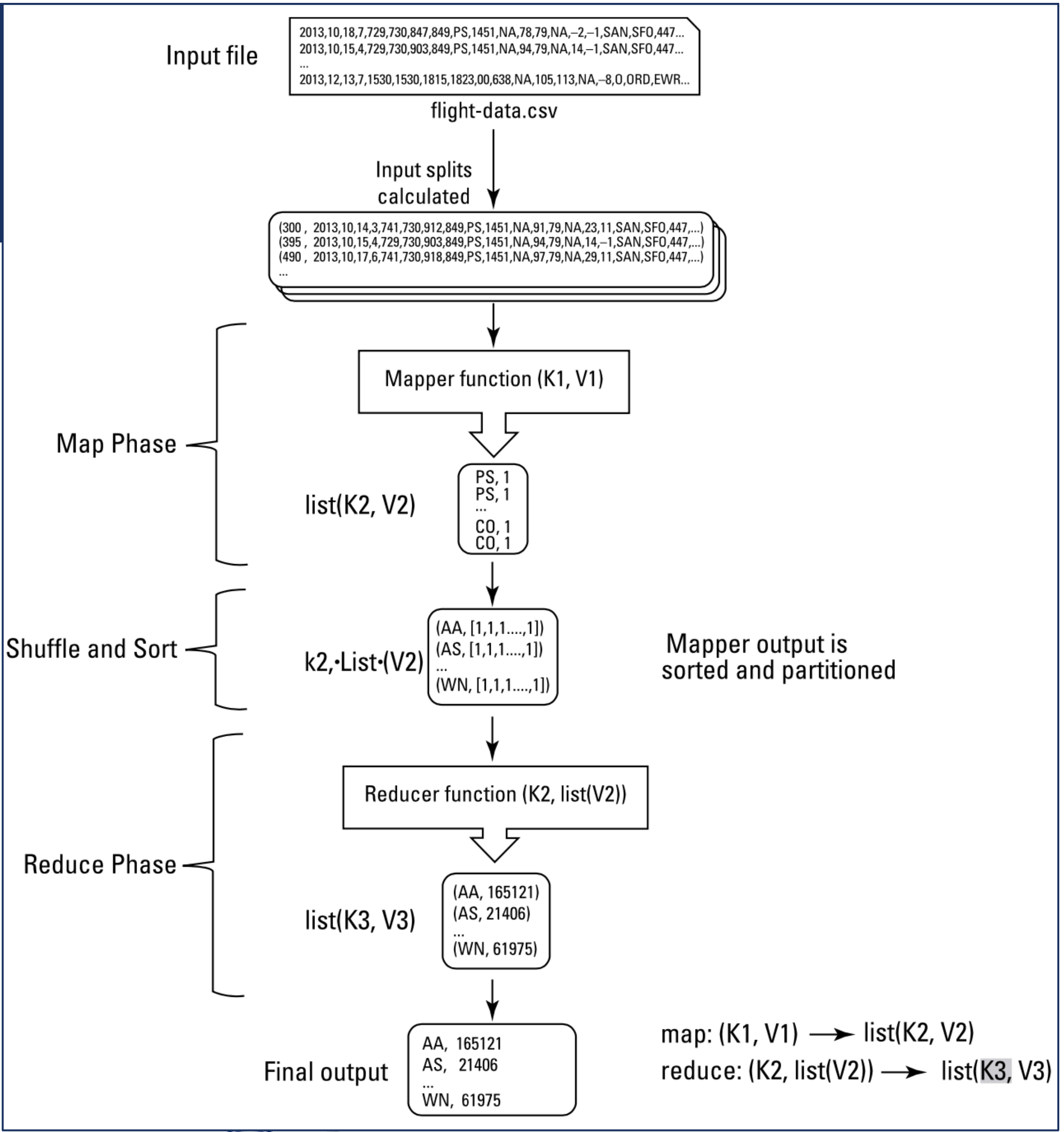
AA, 165121
AS, 21406
...
WN, 61975

map: (K1, V1) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)

Reduce Phase

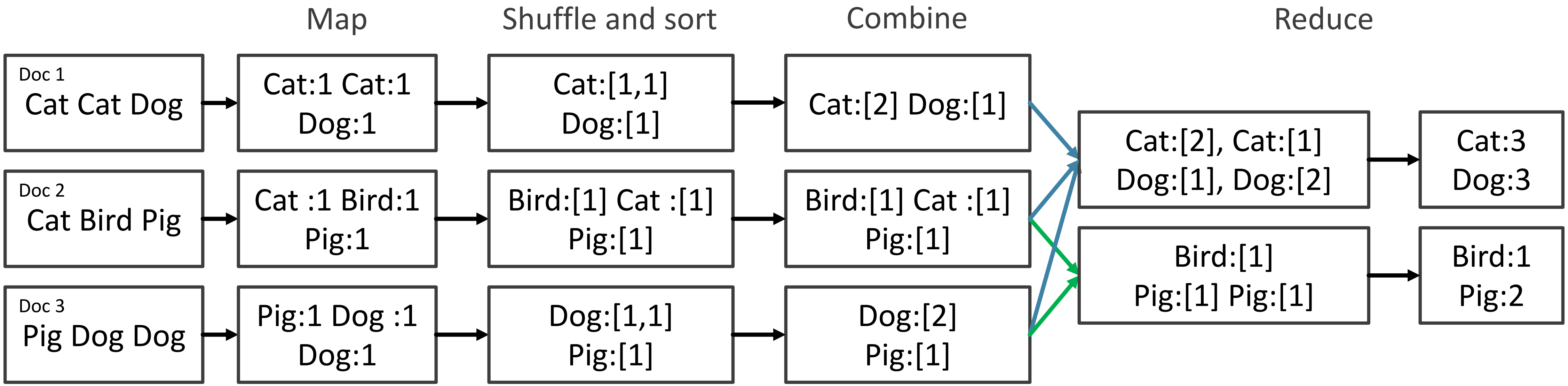
- Combine all the K2,list(V2) from the mapper and return list(K3,V3).
  - You don’t need to worry about which key is in which mapper’s node.
- The reduce task’s processing cannot begin until all mapper tasks have finished.
- What you need to do in this phase:
  - Delicately *design your reduce function* and determine how to construct V3 from V2 for your application.


Input file
2013,10,18,7,729,730,847,849,PS,1451,NA,78,79,NA,–2,–1,SAN,SFO,447...
2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447...
...
2013,12,13,7,1530,1530,1815,1823,00,638,NA,105,113,NA,–8,0,ORD,EWR...
flight-data.csv
Input splits calculated
(300 , 2013,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,...)
(395 , 2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447,...)
(490 , 2013,10,17,6,741,730,918,849,PS,1451,NA,97,79,NA,29,11,SAN,SFO,447,...)
...
Map Phase
Mapper function (K1, V1)
list(K2, V2)
PS, 1
PS, 1
...
CO, 1
CO, 1
Shuffle and Sort
k2,·List·(V2)
(AA, [1,1,1....,1])
(AS, [1,1,1....,1])
...
(WN, [1,1,1....,1])
Mapper output is sorted and partitioned
Reduce Phase
Reducer function (K2, list(V2))
list(K3, V3)
(AA, 165121)
(AS, 21406)
...
(WN, 61975)
Final output
AA, 165121
AS, 21406
...
WN, 61975
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

Combine Function

- To minimize the data transferred between map and reduce tasks, Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner function's output forms the input to the reduce function.
- It does *partial merging* of this data before it is sent over the network.
- It is optional function between map and reduce, but can't replace any of them.

Combine Function



Map

Shuffle and sort

Combine

Reduce

Doc 1
Cat Cat Dog

Cat:1 Cat:1
Dog:1

Cat:[1,1]
Dog:[1]

Cat:[2] Dog:[1]

Doc 2
Cat Bird Pig

Cat :1 Bird:1
Pig:1

Bird:[1] Cat :[1]
Pig:[1]

Bird:[1] Cat :[1]
Pig:[1]

Doc 3
Pig Dog Dog

Pig:1 Dog :1
Dog:1

Dog:[1,1]
Pig:[1]

Dog:[2]
Pig:[1]

Cat:[2], Cat:[1]
Dog:[1], Dog:[2]

Cat:3
Dog:3

Bird:[1]
Pig:[1] Pig:[1]

Bird:1
Pig:2

Master Data Structures

- The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks).
- The master assigns *M* map tasks and *R* reduce tasks.
- For each completed map task, the master stores the locations and sizes of the *R* intermediate files produced by the map task.
- Updates to this location and size information are received as map tasks are completed.

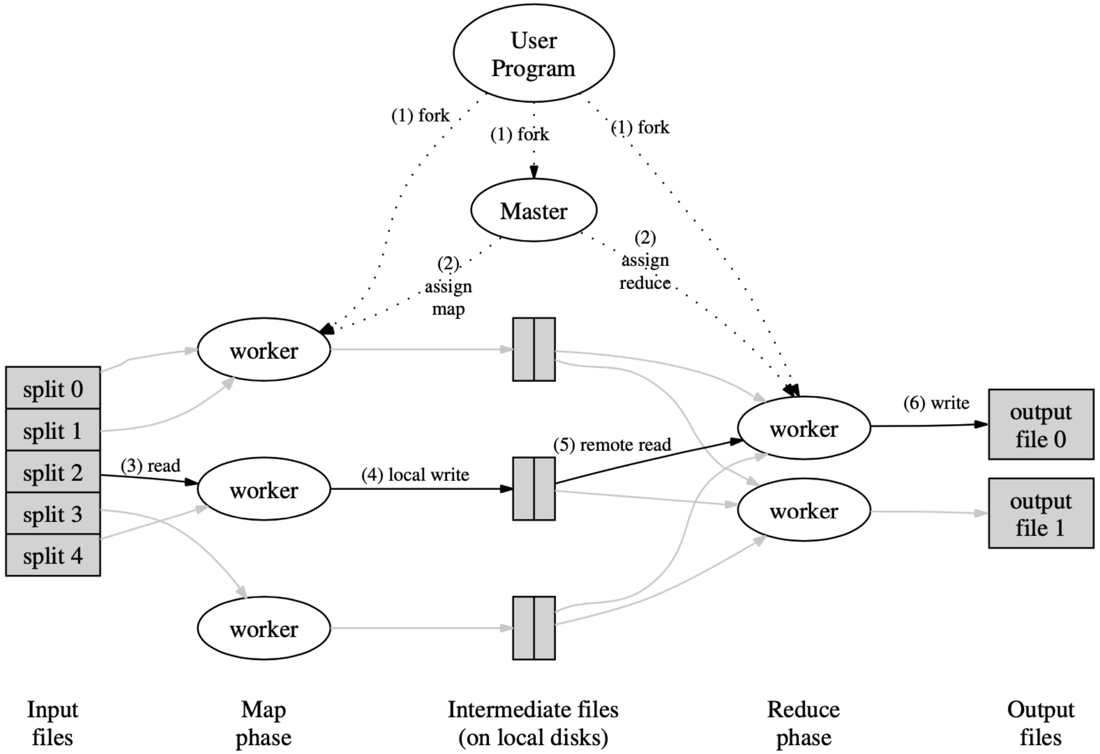


Image source: Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51, no. 1 (2008): 107-113.

Fault Tolerance

- Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.
  - Worker failure.
  - Master failure.

Worker Failure

- The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.
  - Ping is just like heartbeat, but it is sent by the master.
- Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers.
- Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

Worker Failure

- Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.
  - Not in HDFS and thus no replicas.
- Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.
  - In HDFS and can be recovered from it replicas.

Master Failure

- It is easy to make the master write periodic checkpoints of the master data structures described above.
  - If the master task dies, a new copy can be started from the last checkpointed state.
- However, if there is only a single master, the current implementation aborts the MapReduce computation if the master fails.
  - Clients can check for this condition and retry the MapReduce operation if they desire.

Task Granularity

- We divide the map phase into *M* pieces and the reduce phase into *R* pieces.
- Ideally, *M* and *R* should be much larger than the number of worker machines.
- Having each worker perform many different tasks can
  - improve dynamic load balancing;
  - speed up recovery when a worker fails;
- Thus, the many map tasks it has completed can be spread out across all the other worker machines.

Task Granularity

- In practice, we tend to choose *M* so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective)
- We make *R* a small multiple of the number of worker machines we expect to use.
- For example, we perform MapReduce computations with *M*=200,000 and *R*=5,000, using 2,000 worker machines.

Backup Tasks

- One of the common causes that increases the total time taken for a MapReduce operation is a “straggler”
  - A machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation.
  - As we know, reduce tasks should wait for the completion of all map tasks. The client should wait for the completion of all reduce tasks.
- For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s.

Backup Tasks

- Backup task is a general mechanism to alleviate the problem of stragglers.
- When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks.
  - The task is marked as completed whenever either the primary or the backup execution completes.
- It typically increases the computational resources used by the operation by no more than a few percent.
  - However, the extra resources used is worthy.
  - It significantly reduces the time to complete large MapReduce operations.

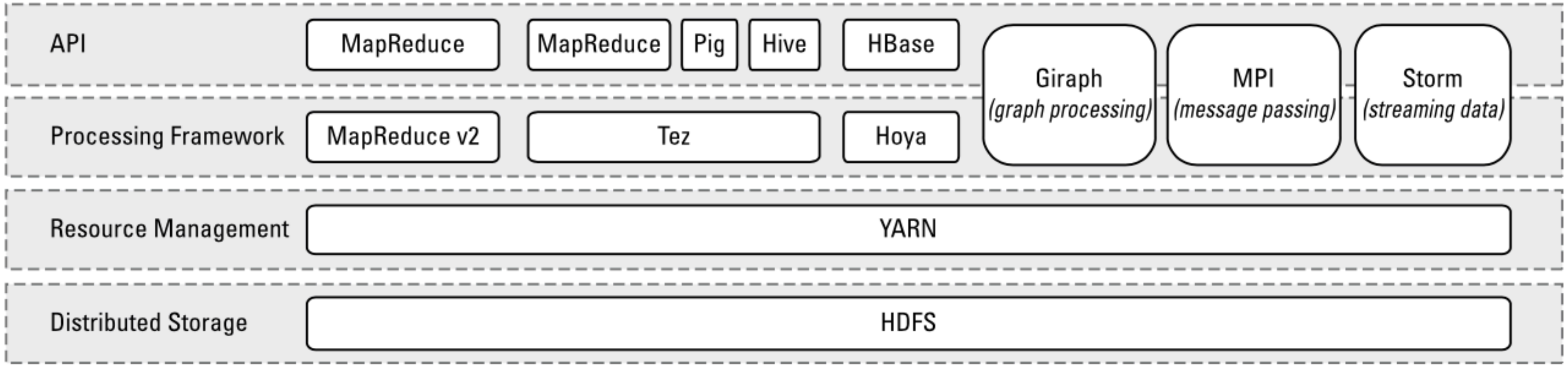
hadoop
YARN

Resource Management

- Now, we have a bunch of MapReduce tasks and HBase tasks running at the same time from different clients.
  - Each application has its own master dedicated for its own applications.
- How can we efficiently manage all the resources we have in Hadoop?

YARN

- YARN stands for Yet Another Resource Negotiator, a tool that enables other data processing frameworks to run on Hadoop.
- YARN is meant to provide a more efficient and flexible workload scheduling as well as a resource management facility.
  - Both of which will ultimately enable Hadoop to run more than just MapReduce jobs.



API: MapReduce | MapReduce | Pig | Hive | HBase

Processing Framework: MapReduce v2 | Tez | Hoya

Giraph (graph processing) | MPI (message passing) | Storm (streaming data)

Resource Management: YARN

Distributed Storage: HDFS

Image source: Figure 7-3 DeRoos, Dirk. *Hadoop for dummies*. John Wiley & Sons, 2014.

YARN

- **Distributed storage:** HDFS is still the storage layer for Hadoop.
- **Resource management:** decoupling resource management from data processing.
  - This enables YARN to provide resources to any processing framework written for Hadoop, including MapReduce and HBase.
- **Processing framework:** Because YARN is a general-purpose resource management facility, it can allocate cluster resources to any data processing framework written for Hadoop.
  - It acts as the master for all applications in Hadoop.

Resource Manager

- Resource Manager is the core component of YARN, which governs all the data processing resources in the Hadoop cluster.
- Its only tasks are to maintain a *global view* of all resources in the cluster.
  - Handling resource requests, scheduling the request, and then assigning resources to the requesting application.
- The Resource Manager, a critical component in a Hadoop cluster, should run on a dedicated master node.

Resource Manager

- The Resource Manager is completely *agnostic* with regard to both applications and frameworks.
  - “You know nothing, resource manager” – Hadoop.
- It has no concept of map or reduce tasks, it doesn’t track the progress of jobs or their individual tasks, and it doesn’t handle failovers.
- Resource Manager only does one thing: schedule workloads, and it does that job well.
  - Concentrating on one aspect while ignoring everything else.
- This high degree of separating duties is exactly what makes YARN much more scalable, able to provide a generic platform for applications.

Node Manager

- Doing scheduling only in a master node is not enough.
- Each slave node has a Node Manager daemon, which acts as a slave for the Resource Manager.
- Each Node Manager tracks the available data processing resources on its slave node and sends regular reports to the Resource Manager.
  - Colonels (Node Managers) in the battlefield (slave nodes) report to the supreme commander (Resource Manager) in the headquarter (master node).

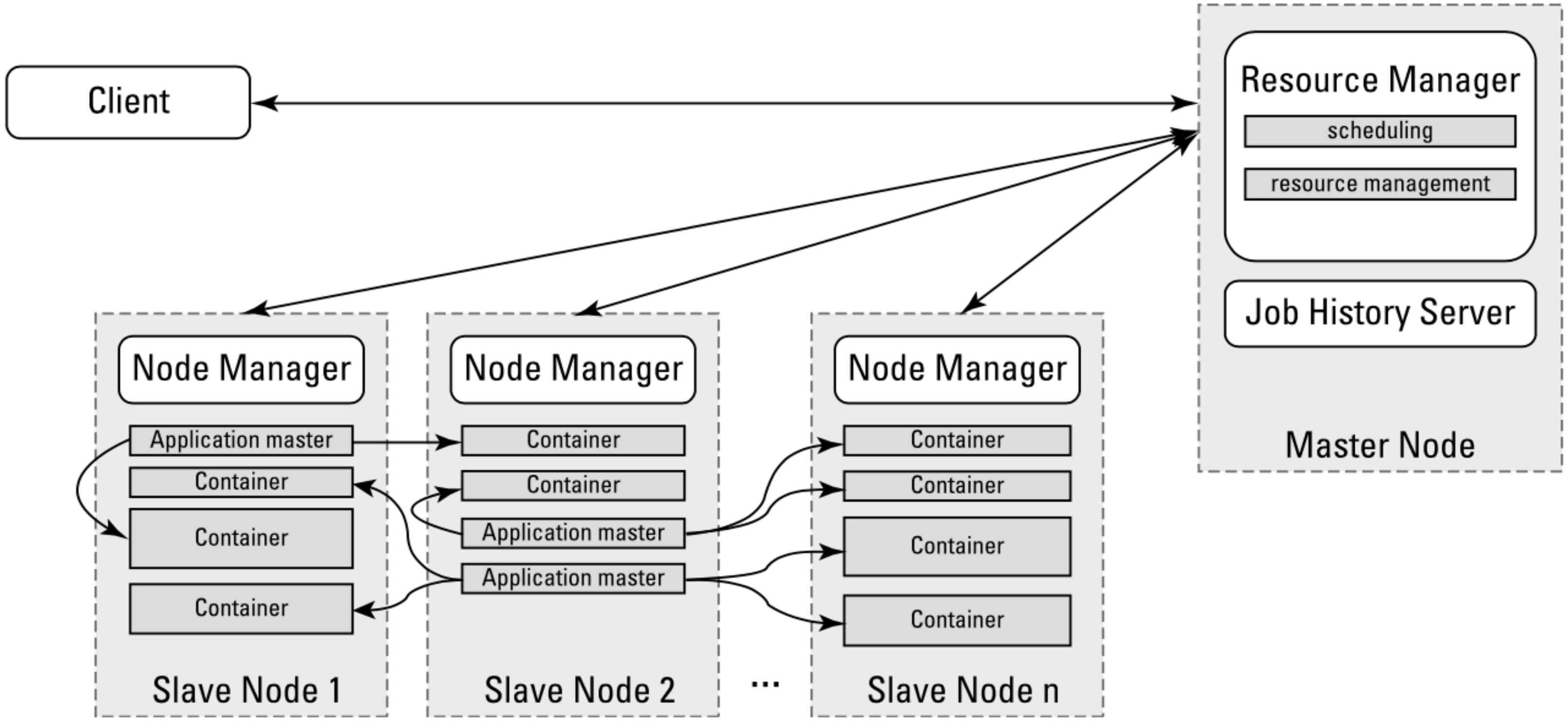
Container

- The processing resources in a Hadoop cluster are consumed in bite-size pieces called containers.
- A *container* is a collection of all the resources necessary to run an application: CPU cores, memory, network bandwidth, and disk space.
- A deployed container runs as an individual process on a slave node in a Hadoop cluster.
- All container processes running on a slave node are initially provisioned, monitored, and tracked by that slave node’s Node Manager daemon.

Application Master

- Each application running on the Hadoop cluster has its own, dedicated Application Master instance.
  - E.g. Master in MapReduce and HMaster in HBase.
- The Application Master actually runs in a container process on a slave node.
- The Application Master sends heartbeat messages to the Resource Manager with its status and the state of the application's resource needs.
- Based on the results of the Resource Manager's scheduling, it assigns *container resource leases* — basically reservations for the resources containers need — to the Application Master on specific slave nodes.
- Each application framework that's written for Hadoop must have its own Application Master implementation.

YARN



YARN daemons and application execution

Image source: Figure 7-4 DeRoos, Dirk. *Hadoop for dummies*. John Wiley & Sons, 2014.

Conclusion

After this lecture, you should know:

- Why do we need MapReduce?
- What are the key steps of MapReduce?
- What is the role of YARN in Hadoop?

Thank you!

Reference (recommend for further reading):

- **The official guide**: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Overview
- **The paper**: Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51, no. 1 (2008): 107-113.
- **The book**: Chapter 6&7, DeRoos, Dirk. *Hadoop for dummies*. John Wiley & Sons, 2014.