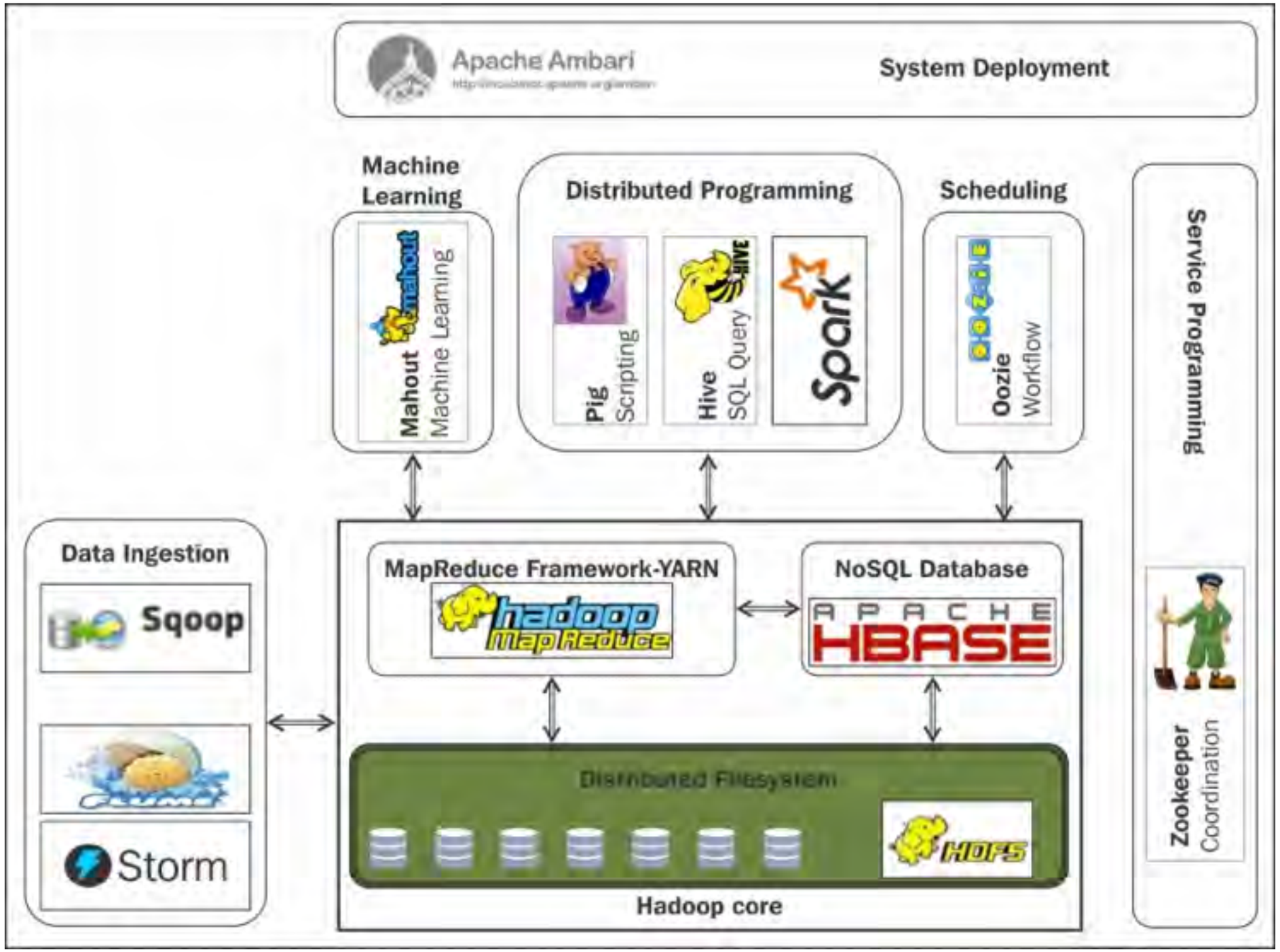
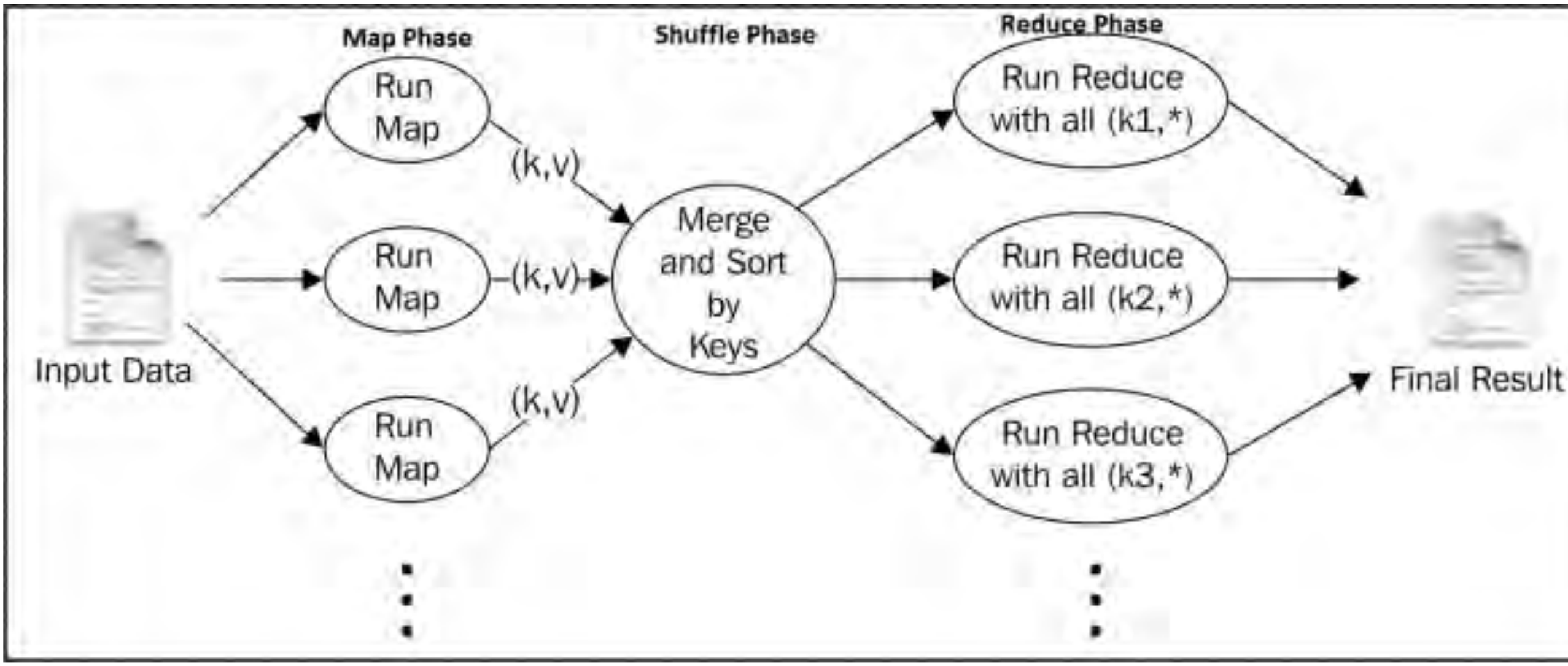
How MapReduce Works

Hadoop


Apache Ambari
System Deployment
Machine Learning
Mahout
Machine Learning
Distributed Programming
Pig
Scripting
Hive
SQL Query
Spark
Scheduling
Oozie
Workflow
Service Programming
Data Ingestion
Sqoop
Flume
Storm
MapReduce Framework-YARN
hadoop MapReduce
NoSQL Database
APACHE HBASE
Distributed Filesystem
HDFS
Hadoop core
Zookeeper
Coordination

MapReduce

- Batch-oriented Parallel data processing model for processing framework
  - Enables developers to process massive amount of data in a distributed fashion across large number of nodes
  - Programs used by MapReduce framework automatically parallelized
- Two phases:
  - Map
  - Reduce

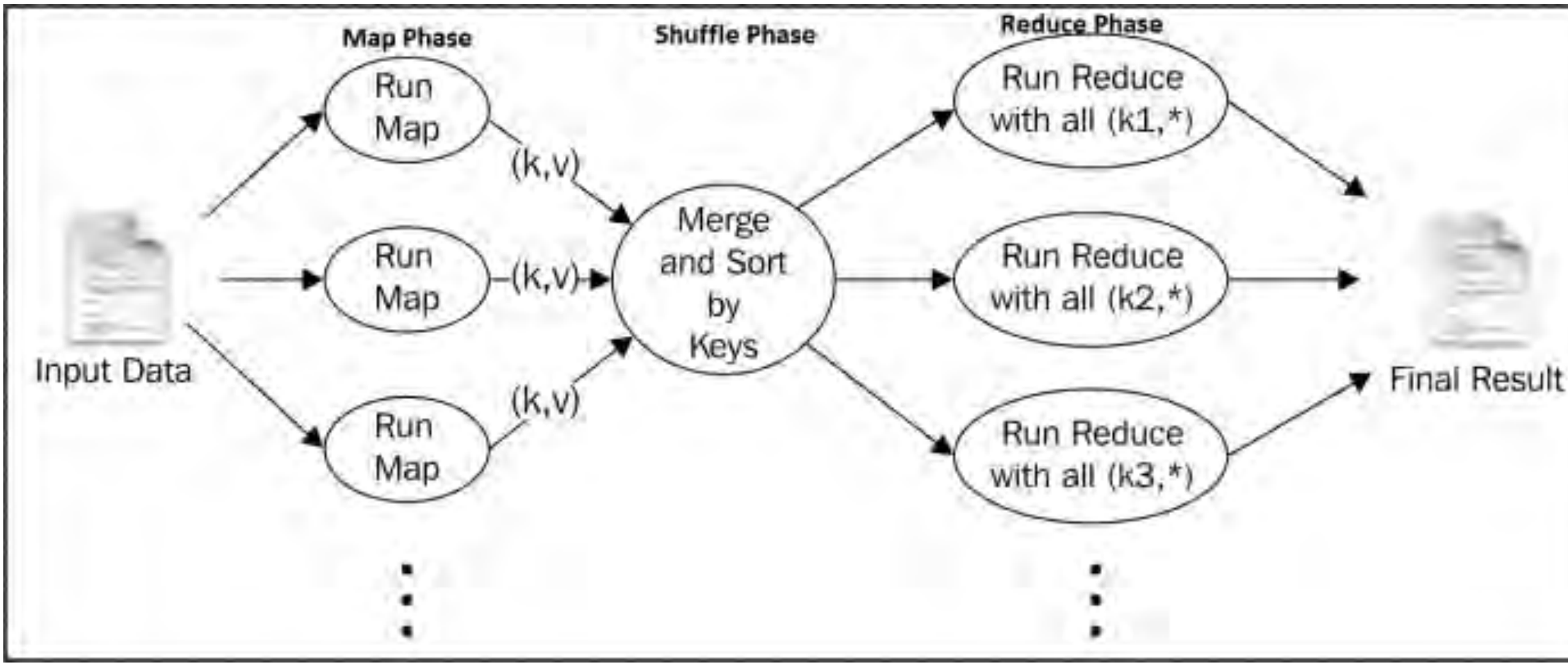

Map Phase
Shuffle Phase
Reduce Phase
Run Map
Run Map
Run Map
(k,v)
(k,v)
(k,v)
Merge and Sort by Keys
Run Reduce with all (k1,*)
Run Reduce with all (k2,*)
Run Reduce with all (k3,*)
Input Data
Final Result

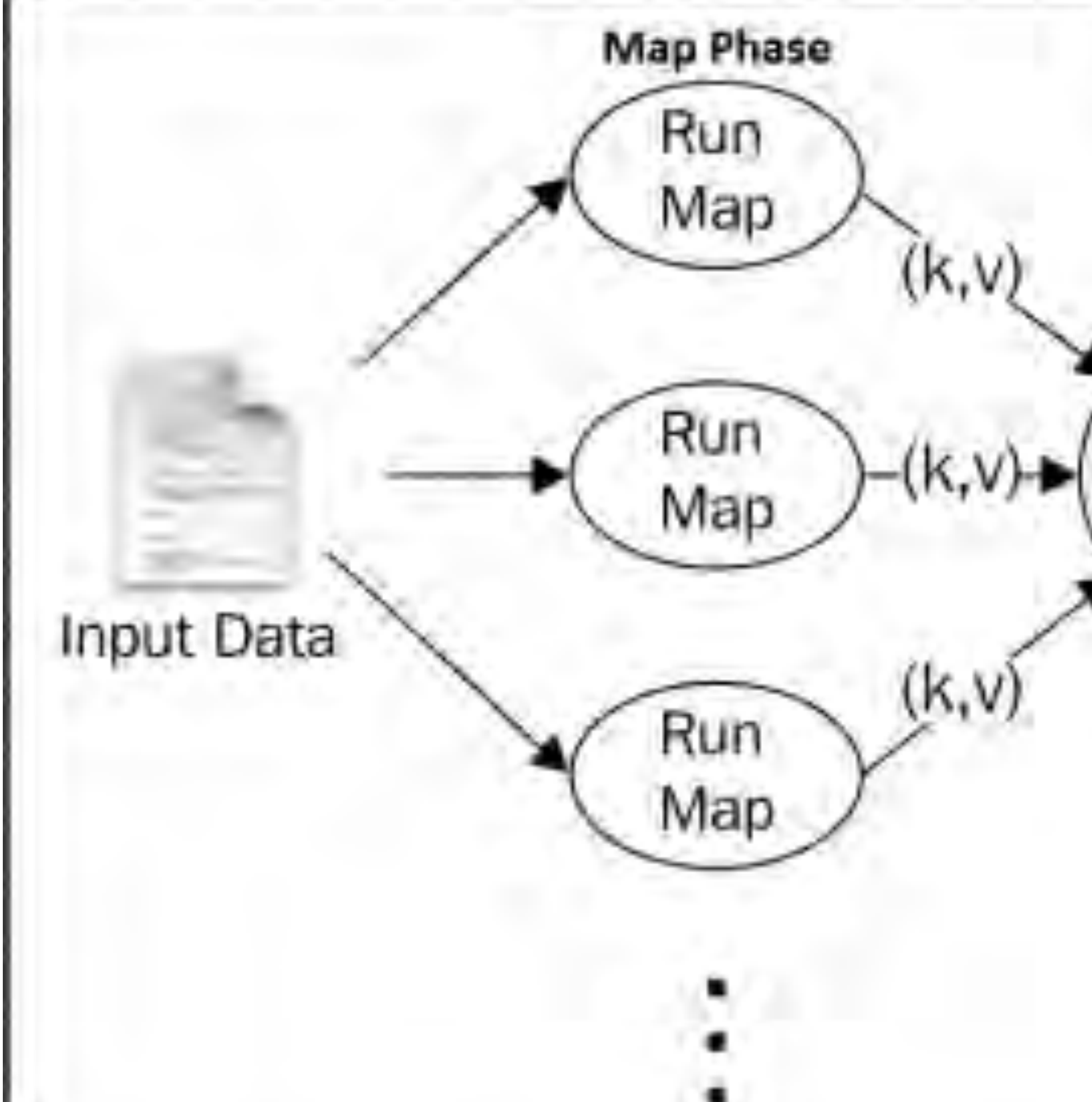
MapReduce

- Map function receives each record of the input data as key-value pairs
- Each Map function invocation is independent of each other
  - Divide and conquer to execute the computation in parallel
    - Hadoop creates a single Map task instance for each HDFS data block of the input data
    - The number of Map function invocations inside the Map task instance is equal to number of data records in the input data block

MapReduce

- Hadoop MapReduce groups the output key-value records of all the Map tasks
  - Distributes them to the Reduce tasks
  - This distribution and transmission of data to reduce tasks is the Shuffle phase
    - Input data to each reduce task is sorted and grouped by the key
    - Reduce function is invoked for each key and group of values for that key is sorted
- User implements Map and Reduce functions
  - Hadoop takes care of the scheduling and executing them


Map Phase
Shuffle Phase
Reduce Phase
Run Map
Run Map
Run Map
(k,v)
(k,v)
(k,v)
Merge and Sort by Keys
Run Reduce with all (k1,*)
Run Reduce with all (k2,*)
Run Reduce with all (k3,*)
Input Data
Final Result


Map Phase
Run Map
Run Map
Run Map
(k,v)
(k,v)
(k,v)
Input Data

Map Phase

Run Map

Run Map

Run Map

ut Data

Shuffle Phase

(k,v)

(k,v)

(k,v)

Merge and Sort by Keys

Reduce Phase

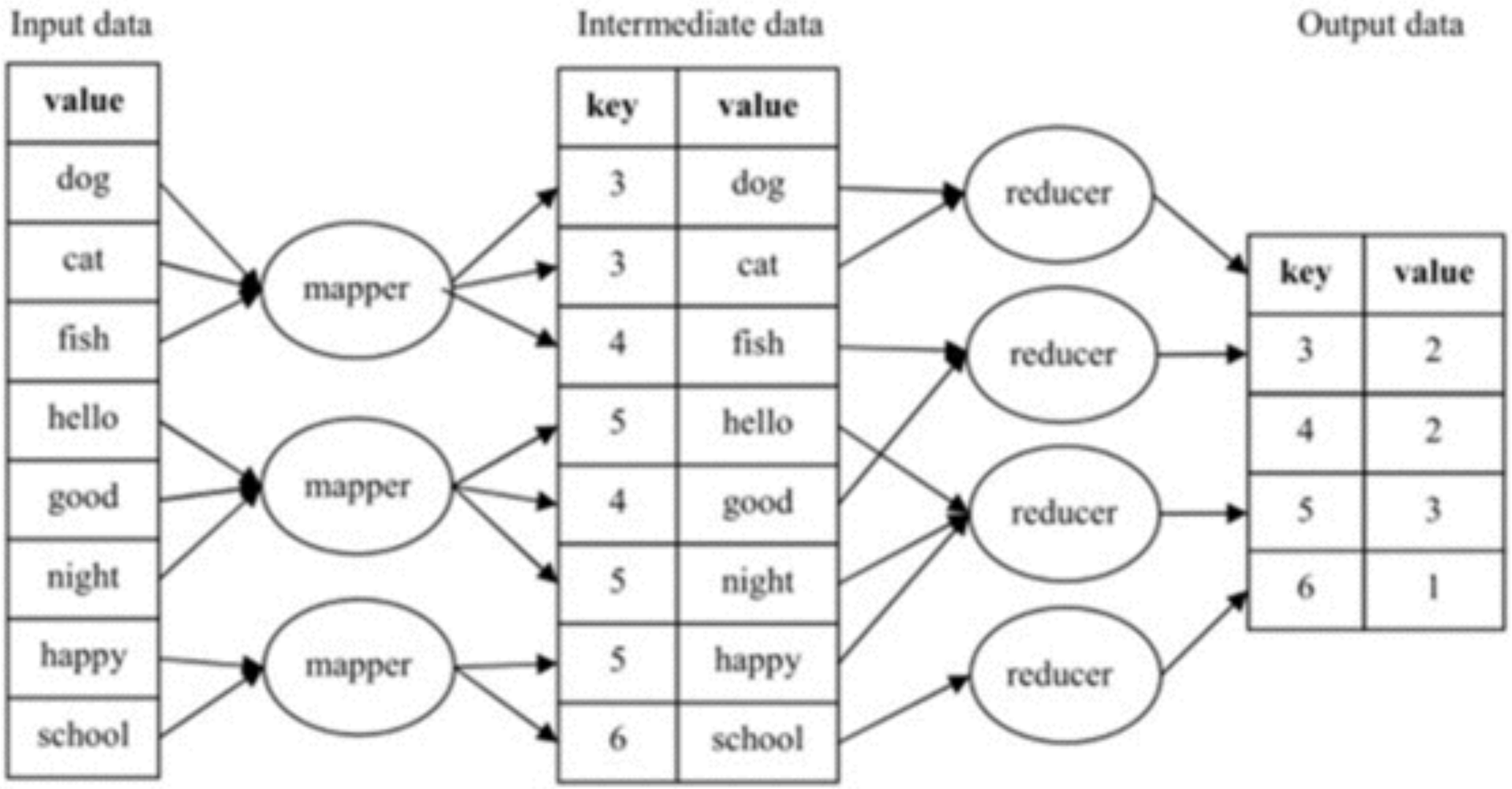
Run Reduce with all (k1,*)

Run Reduce with all (k2,*)

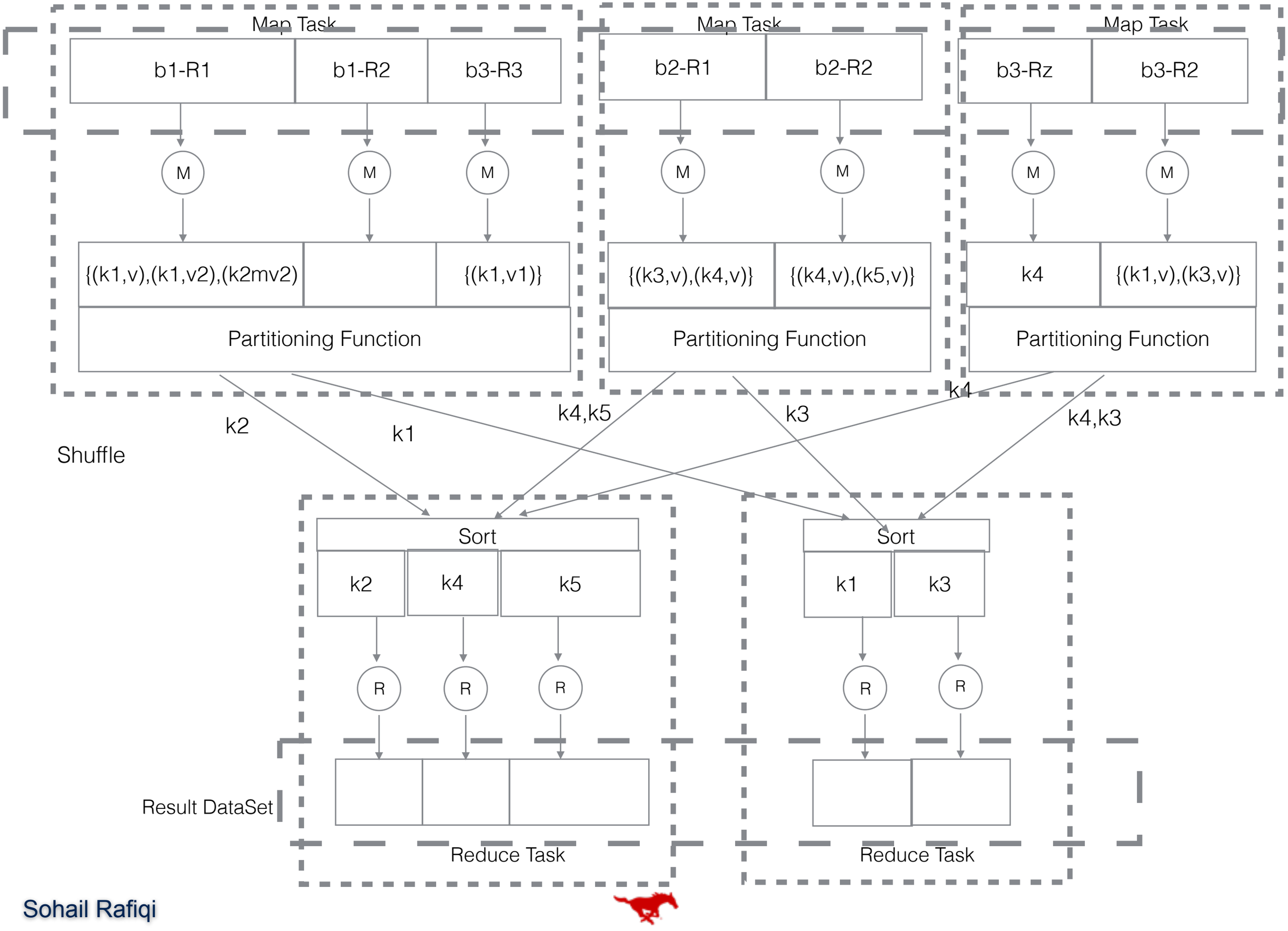
Run Reduce with all (k3,*)

Final Result

Example



- Counting the frequency of all words with different length
- Using length as the key
- For word “Hello” mapper would generate key/value pair of “5/Hello”


Map Task
b1-R1
b1-R2
b3-R3
M
M
M
{(k1,v),(k1,v2),(k2mv2)
{(k1,v1)}
Partitioning Function
Map Task
b2-R1
b2-R2
M
M
{(k3,v),(k4,v)}
{(k4,v),(k5,v)}
Partitioning Function
Map Task
b3-Rz
b3-R2
M
M
k4
{(k1,v),(k3,v)}
Partitioning Function
k2
k1
k4,k5
k3
k1
k4,k3
Shuffle
Sort
k2
k4
k5
Sort
k1
k3
R
R
R
R
R
Result DataSet
Reduce Task
Reduce Task

Map

- The Map phase is a true shared nothing processing stage.
  - Does not share anything with other Map tasks
- Each Map task iterates parallel
  - Every record (key-value pair) is processed only once
  - Each map function receives one key-value pair and outputs one or more key-value pairs
    - Intermediate values
- User writes a map program

Partition Function

- Ensures each key is passed to only one reducer task
- Most common implementation is a hash partitioner
  - Creates a hash for the key and divides the hash key space into n partitions

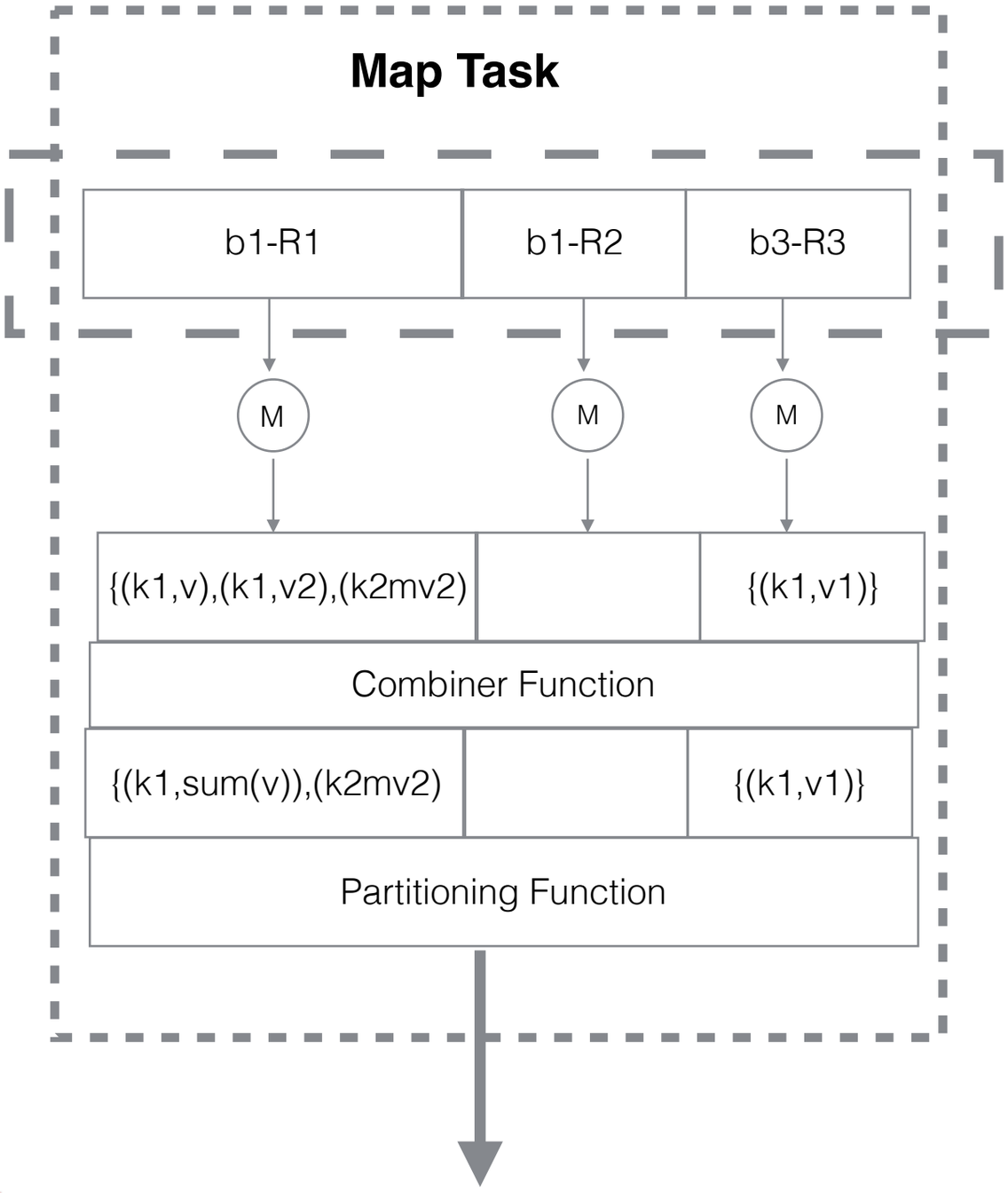
Reduce

- Each reducer tasks executes reduce() function
  - For each intermediates key & values
  - Output can be input to other Map functions
- Simple reduce function example

```
reduce (k, list <v>) =
        sum = 0
        for int i in list <v> :
          sum + = i
        emit (k, sum)
```


Local Combiner

- Local combiner function can be called
  - As long as running the function does not impact the output
    - E.g., Sum or Count
    - Not average



Asymmetry

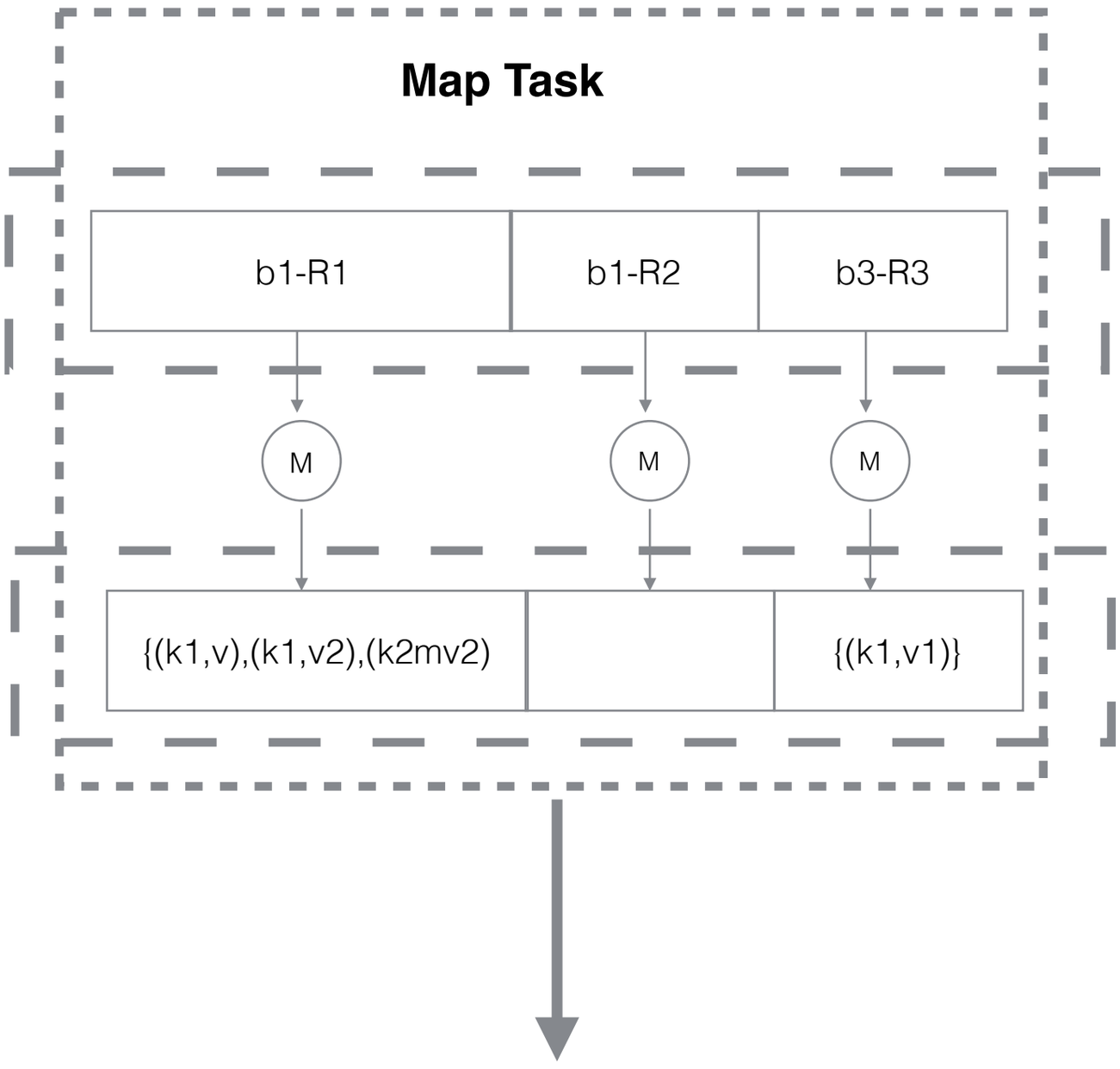
- Map & Reduce functions are asymmetrical in nature
- One Map task may do more than other Map tasks
  - More relevant keys existing in one input block vs other
  - One Reduce function having more processing to do
  - Results in some Map/Reduce tasks slower than other
- May become a bottleneck

Speculative Execution

- Can configure “Speculative Execution”
  - Looks for configurable, tolerated differences in tasks progress
  - If tasks falls outside of the configured tolerance
    - Hadoop would create a duplicate task to run simultaneously
    - The result of first task to complete will be used

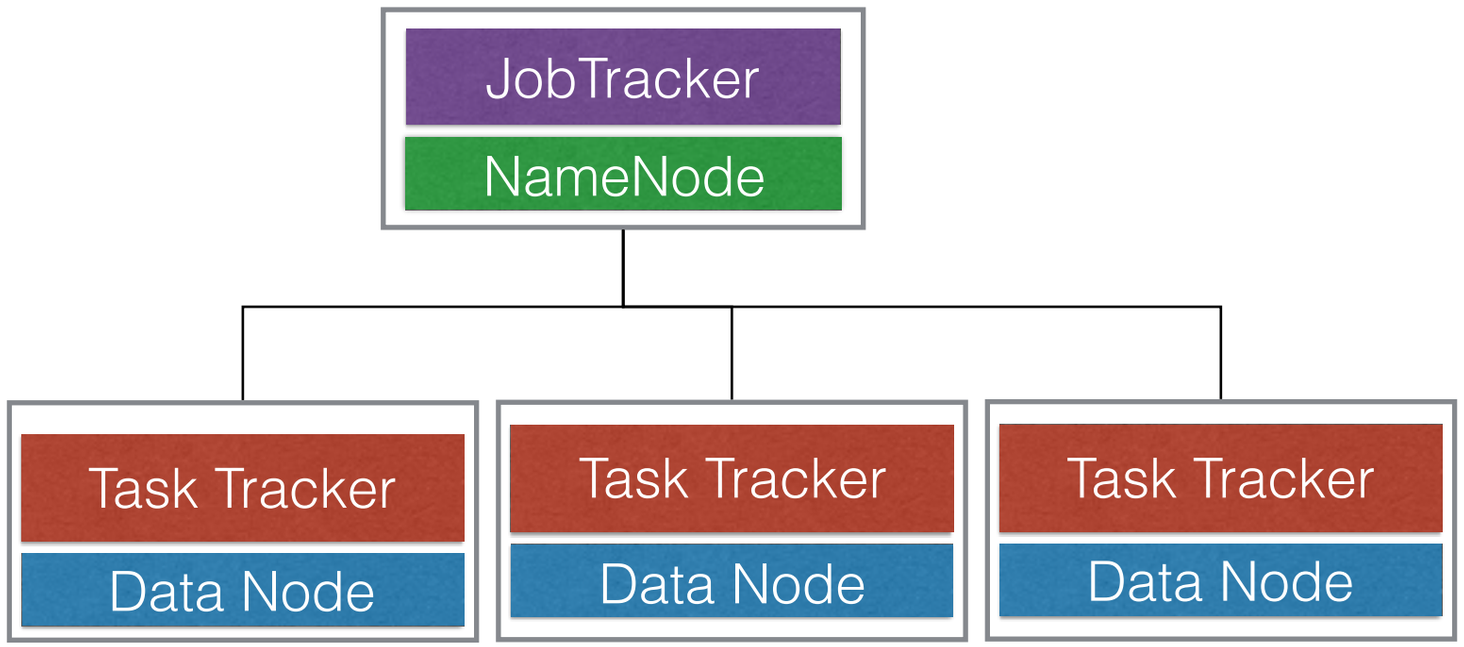
Map only

- Some application may not require Reduce function
- E.g., ETL
  - No reason to summarize data
- No partition function
- Output of the Map Task is final output



MapReduce Architecture

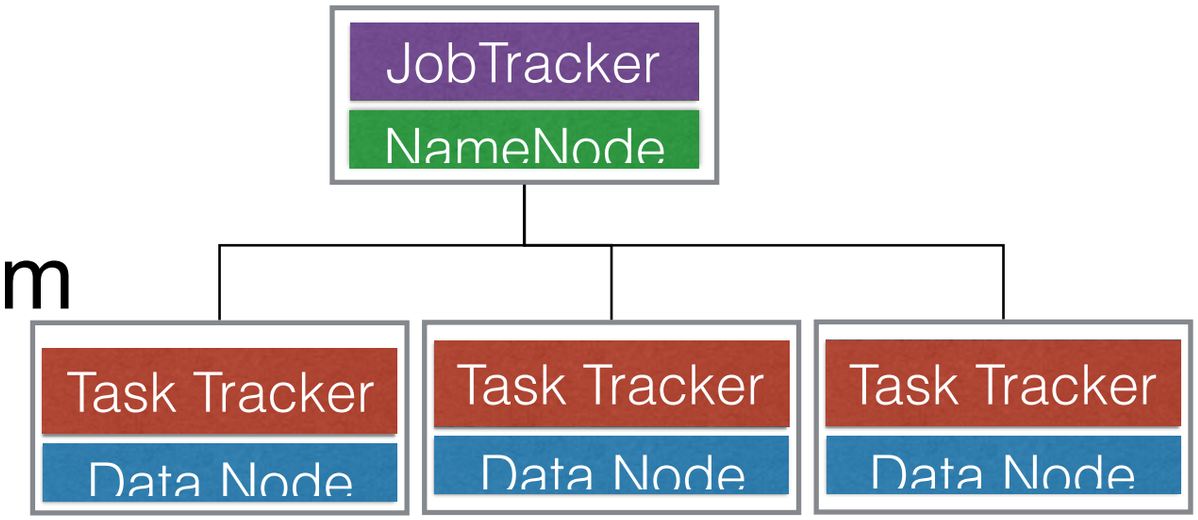
MapReduce Architecture



- Based on master (JobTracker) and slaves (TaskTracker) architecture
- JobTracker is a daemon running on the master node
- Task tracker running on slave nodes

MapReduce Architecture

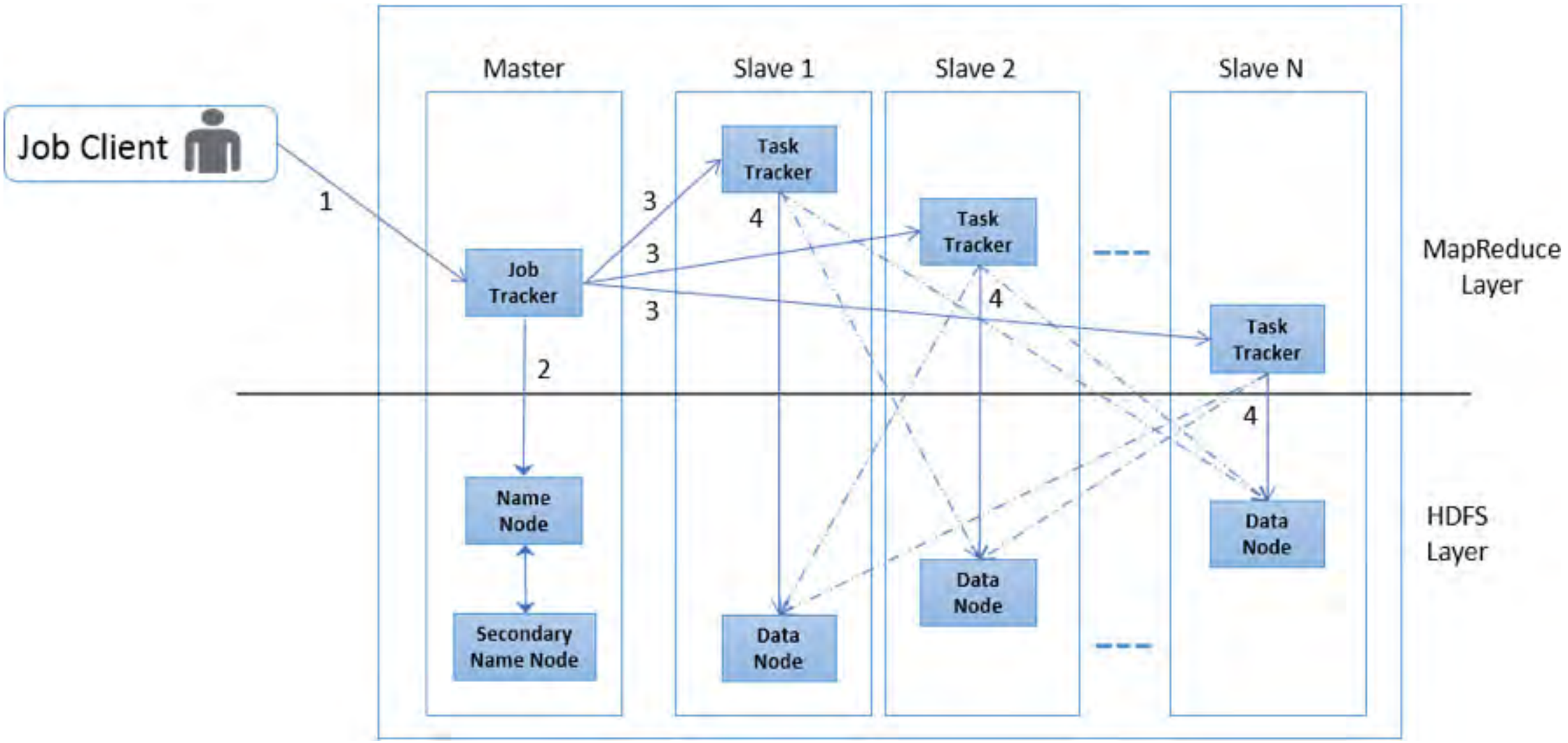
- Each TaskTracker spawns off JVM for the task
  - Keeps task isolated from TaskTracker itself
    - Monitors JVM and informs JobTracker upon completion
- TaskTracker periodically sends heartbeat to JobTracker



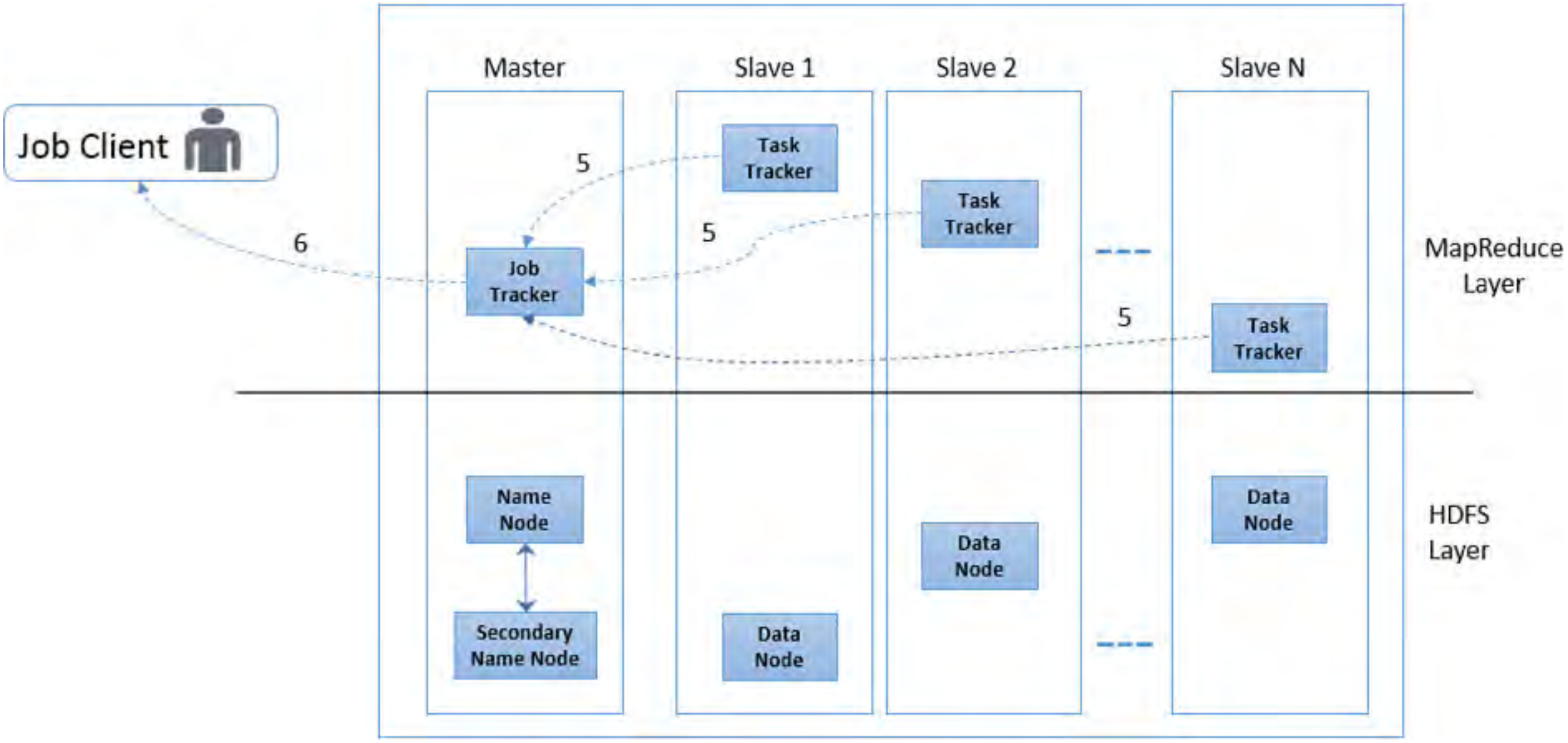
MapReduce Architecture

- Each TaskTracker has a define amount of computation tasks that it can accept
  - Map & Reduce — Default for both is set to 2
  - E.g., In 5 nodes system in a cluster — run 10 mapper and 10 reducers each
- When assigning job JobTracker tries to find TaskTracker with open slot on the DataNode containing the desired data

Job Flow — Request



Job Flow — Response



Job Flow Components

- InputFormat
- OutputFormat
- Mapper, Partitioner, Combiner
- Driver
- Tool Interface
- Context Object

Job Flow Components — InputFormat

- Define how to read data from a file
- InputSplit interface — Splits the input and spawns one map task for each InputSplit
  - Assigned to an individual mapper
  - Contains reference to actual data
- RecordReader — Reads record from InputSplits and submit key-value pairs to the Mapper
- Input Classes available for data reading:
  - Text, Binary file, read data from relational DB — also write custom

DistributedCache

- Enables user to share static data globally among all nodes in the cluster
  - Unlike reading from HDFS in which data from a single file is read from a single mapper
- Can be private or public
  - Private -- cached in local directory private to the user
  - Public — Visible to all users

MapReduce Joins

- The MapReduce framework enables you to join two or more datasets to produce a final output.
- Writing these types of MapReduce is complex
- Easier to use a high-level framework such as Hive or Pig to write queries that involve joining several datasets.
  - Few lines of HiveQL would translate into hundreds of lines of code in a MapReduce job.

YARN

Hadoop 1.0

- Limitations of Hadoop 1.0 with respect to MapReduce processing
  - Single point of failure — JobTracker like NameNode
  - Lack of support for interactive queries
    - Was designed for batch mode processing
    - No true support for interactive and streaming queries
    - Only MapReduce can access data in HDFS
      - ✓ Other apps either go through MapReduce or store data somewhere other than HDFS

Hadoop 1.0

- Limitations of Hadoop 1.0 with respect to MapReduce processing
  - Low computing resource utilization
    - Resources divided as Map slots and Reduce slots in a hard partition manner
      - ✓ If more Mappers are running and few Reducers are running
      - ✓ Create inefficiencies
- JobTracker too busy
  - Responsible for managing both resources and the application life cycle

Hadoop 2.0

MapReduce
(Batch Processing)

YARN — Yet Another Resource Negotiator
(Resource Management and Job Scheduling/Monitoring)

HDFS 2

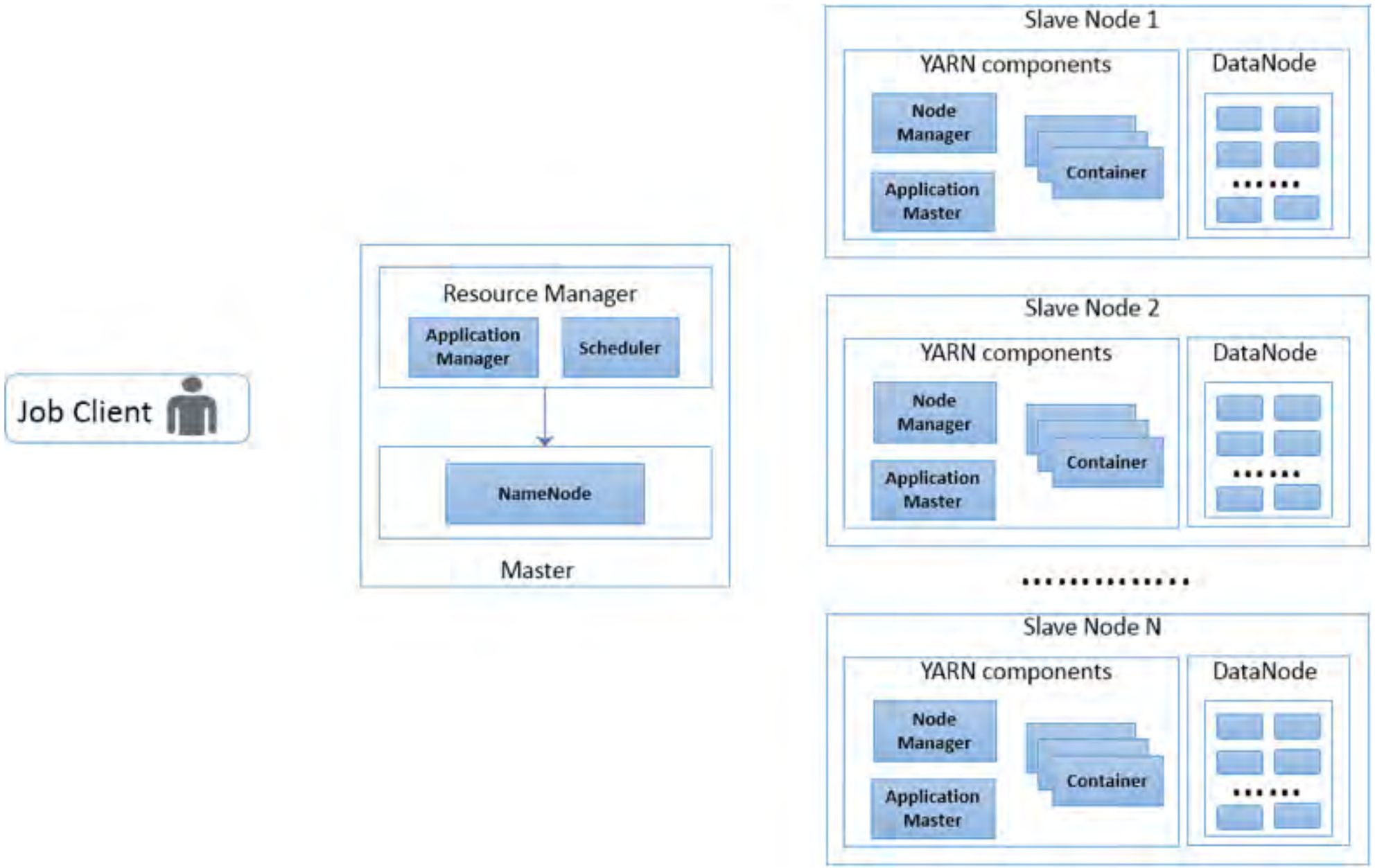
Hadoop 2.0

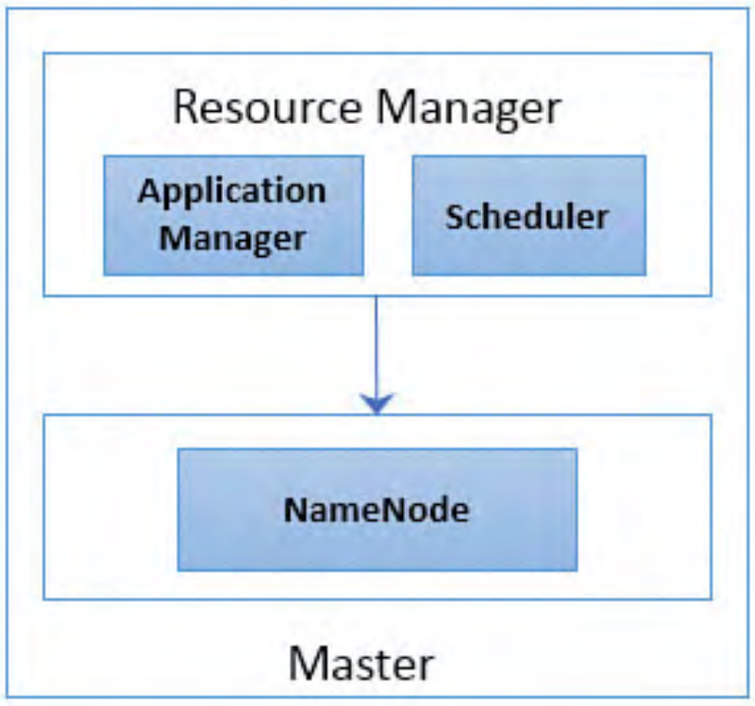
- YARN acts like an OS for the Hadoop cluster
  - Providing resource management and application life-cycle management
- Separation of Duties
  - MapReduce can focus on batch mode data processing
  - YARN focuses on resource management

Hadoop 2.0

- Multitenancy
  - Other applications can connect to HDFS through YARN
  - YARN can assign cluster capacity to the different applications
    - Can handle more than just the MapReduce batch jobs
- Scalability and Performance
  - YARN now has global Resource Manager and per-machine Node Manager to manage the resources of the cluster
  - Application Master than negotiates resources from Resource Manager and manages application life cycle

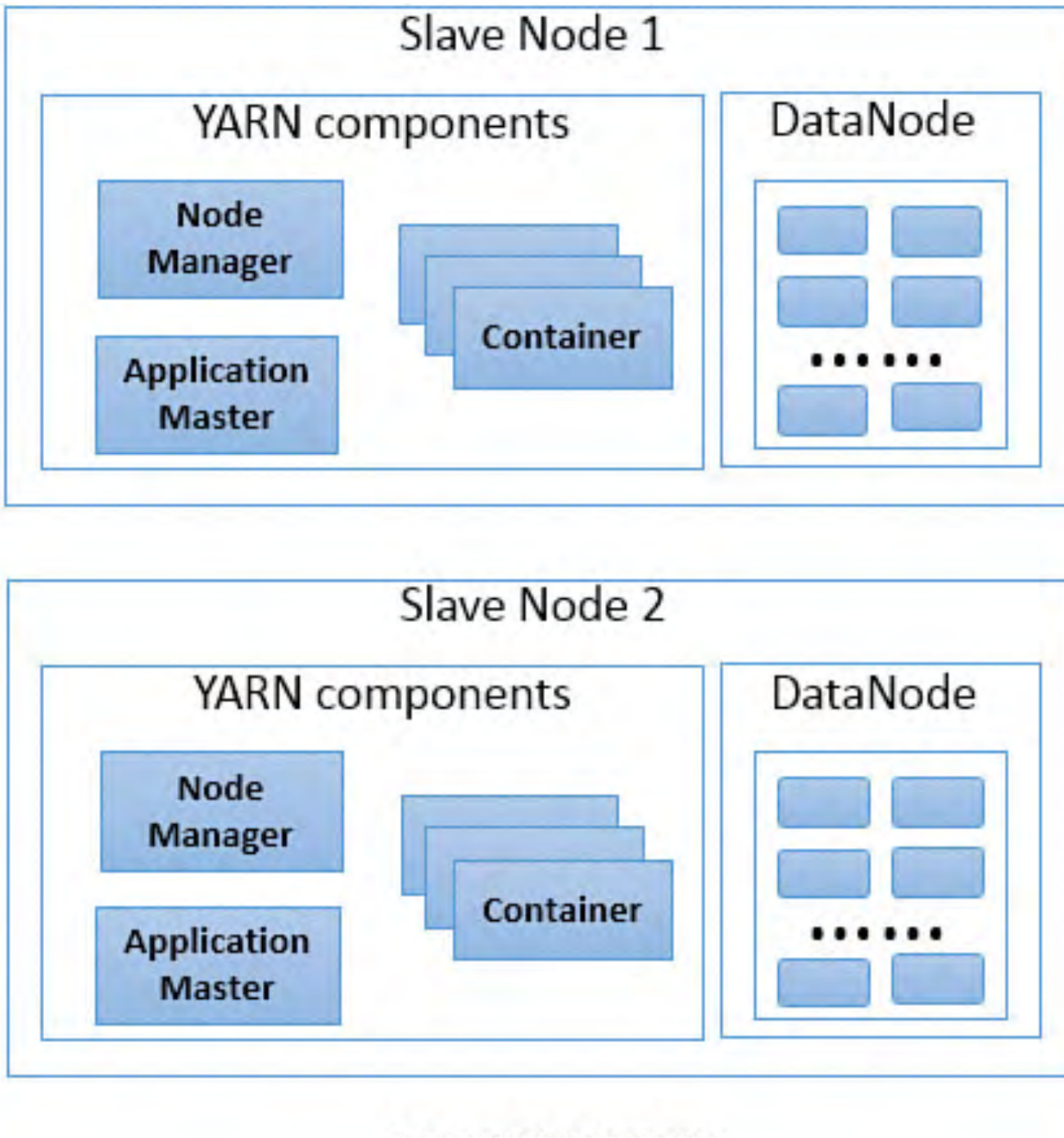
YARN


Job Client
Resource Manager
Application Manager
Scheduler
NameNode
Master
Slave Node 1
YARN components
Node Manager
Application Master
Container
DataNode
Slave Node 2
YARN components
Node Manager
Application Master
Container
DataNode
Slave Node N
YARN components
Node Manager
Application Master
Container
DataNode

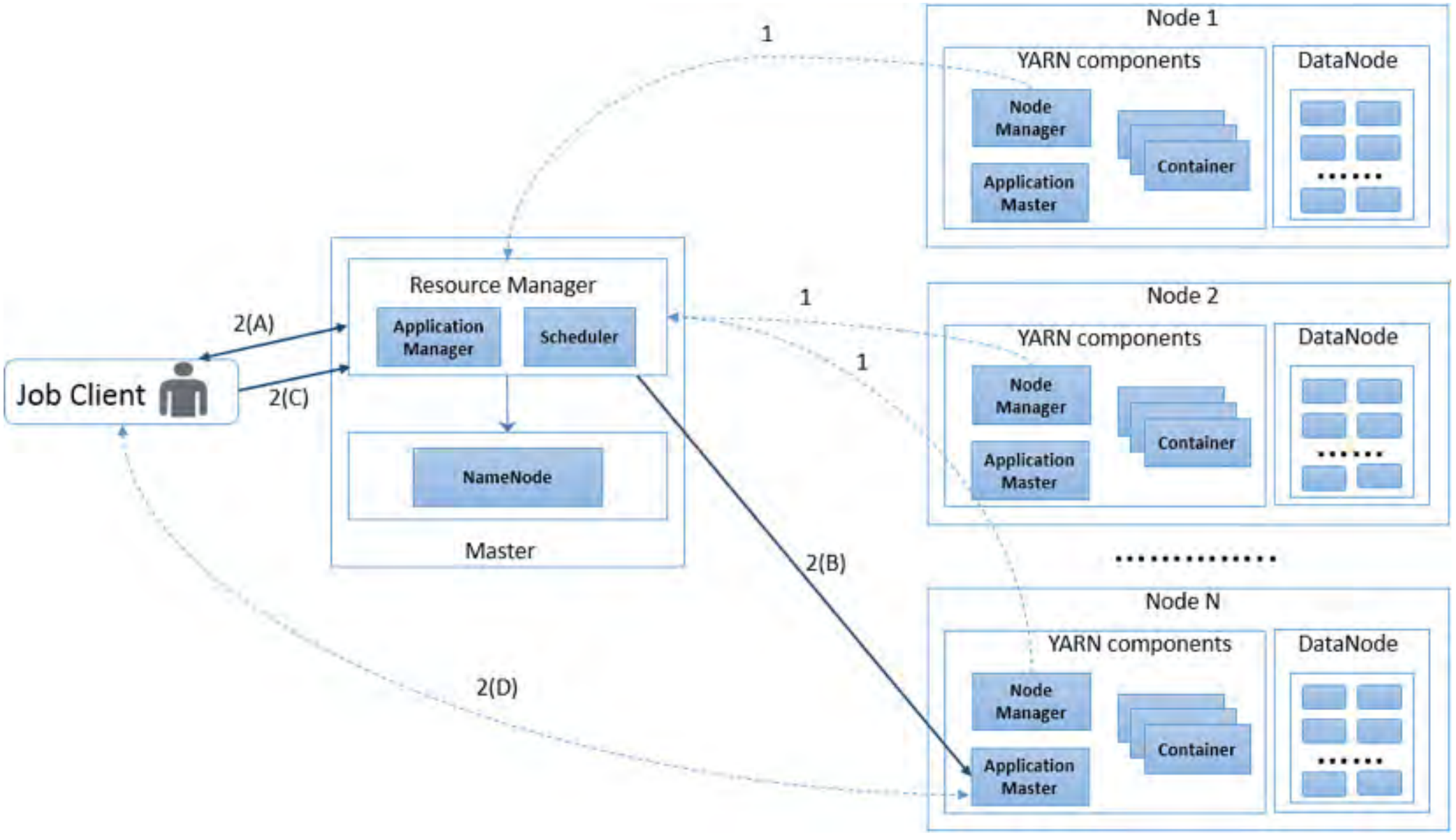


- Job client initiates talk with Application Manager
  - Handle job submission and negotiates resources to execute Application Master
- Scheduler — schedules and allocates cluster resources
-

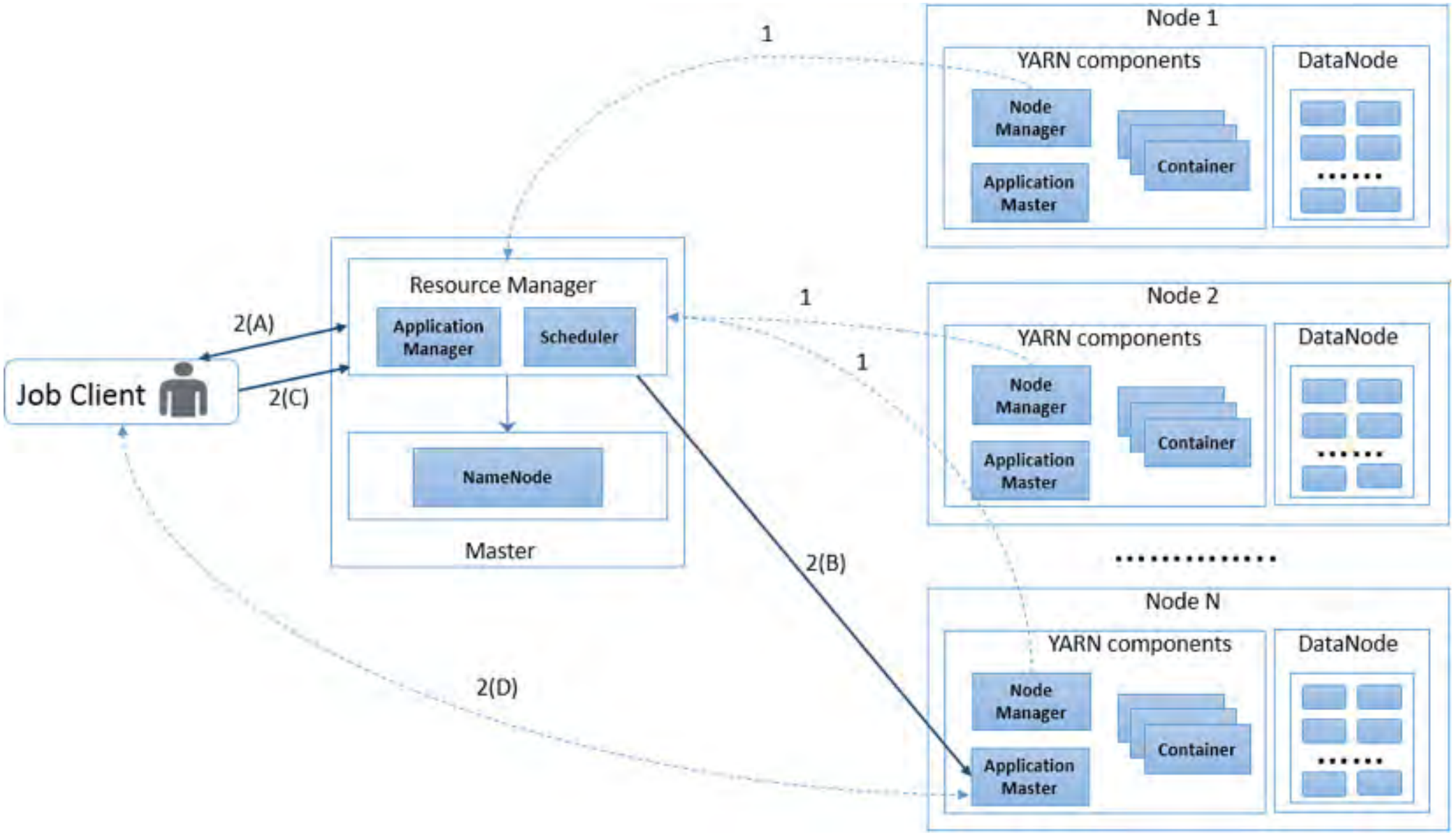
- Hadoop 2.0 added multi tenancy
  - Multiple applications can simultaneously utilize the Hadoop cluster
- Each application can implement Application Master
  - Request resources from Resource Manager
- Node Manager is similar to TaskTrakcer
  - Runs on each slave node in the cluster and launches tasks
- Container
  - Encapsulation of cluster resources (CPU, memory, etc.)
- Now container can be of variable size unlike Hadoop 1.0



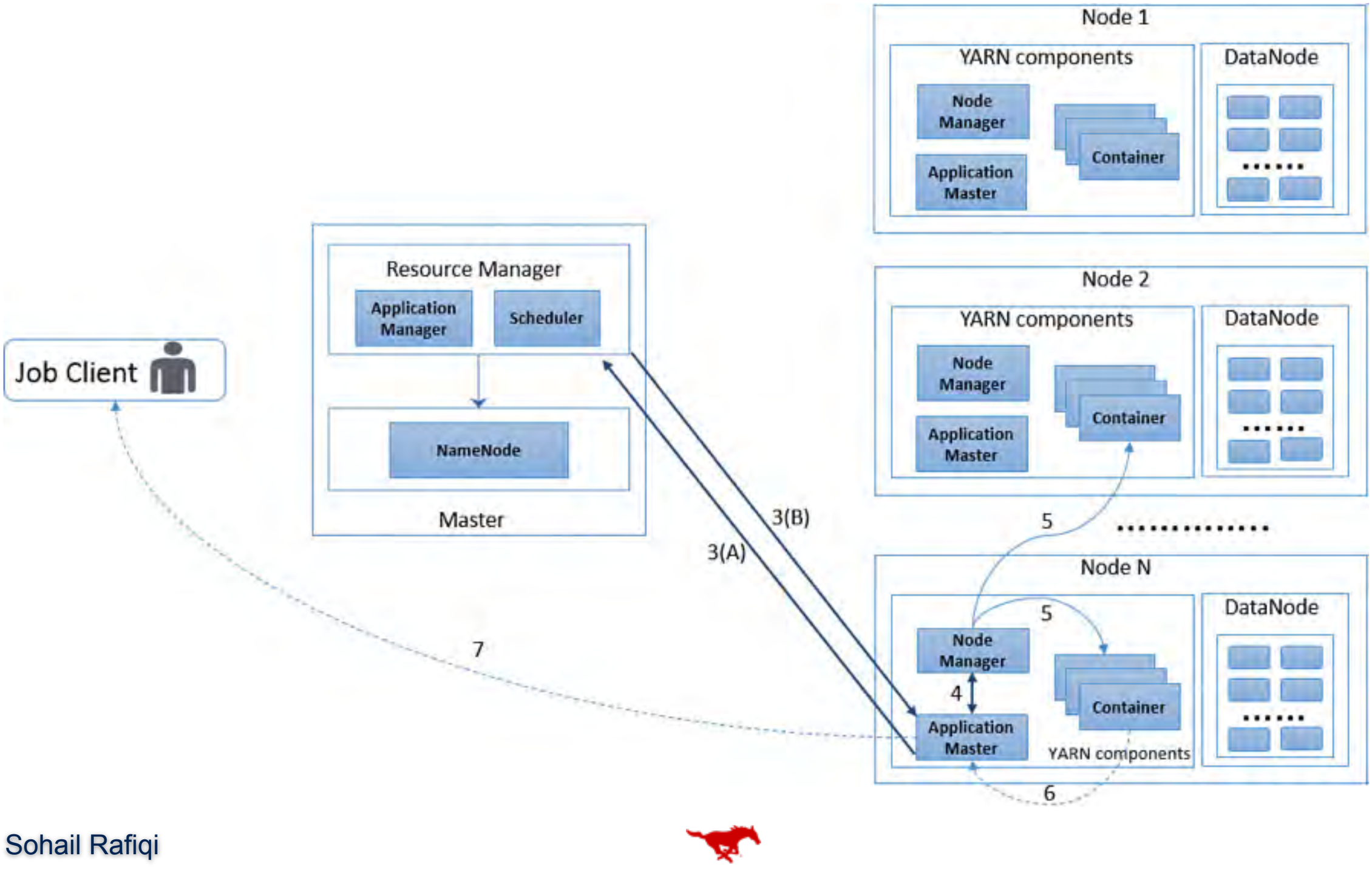
YARN Job Flow



YARN Job Flow



YARN Job Flow



YARN Job Flow