Hadoop – An Introduction

minute, and every second.

1. **Every day:**
   (a) NYSE (New York Stock Exchange) generates 1.5 billion shares and trade data.
   (b) Facebook stores 2.7 billion comments and Likes.
   (c) Google processes about 24 petabytes of data.
2. **Every minute:**
   (a) Facebook users share nearly 2.5 million pieces of content.
   (b) Twitter users tweet nearly 300,000 times.
   (c) Instagram users post nearly 220,000 new photos.
   (d) YouTube users upload 72 hours of new video content.
   (e) Apple users download nearly 50,000 apps.
   (f) Email users send over 200 million messages.
   (g) Amazon generates over $80,000 in online sales.
   (h) Google receives over 4 million search queries.
3. **Every second:**
   (a) Banking applications process more than 10,000 credit card transactions.

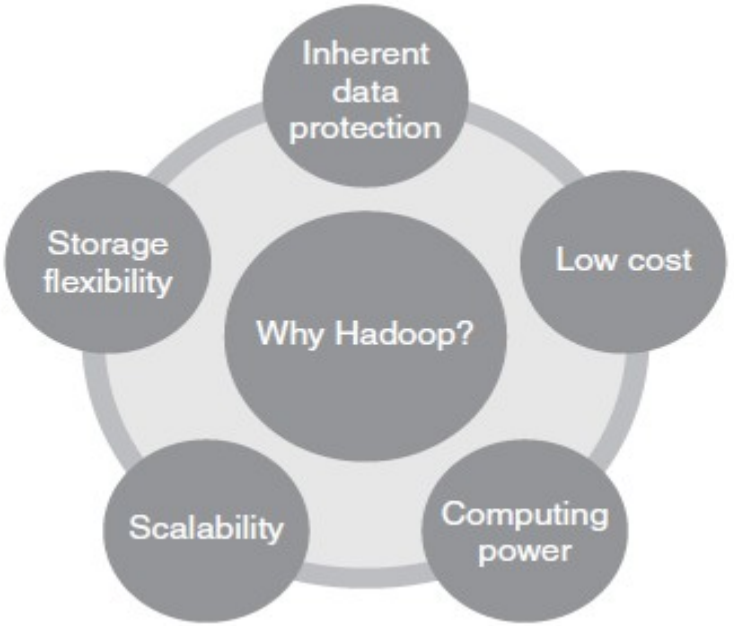
Hadoop

Ever wondered why Hadoop has been and is one of the most wanted technologies!!

The key consideration (the rationale behind its huge popularity) is:

Its capability to handle massive amounts of data, different categories of data – fairly quickly.

The other considerations are :



Hadoop makes use of commodity hardware, distributed file system, and distributed computing as shown in Figure 5.3. In this new design, groups of machine are gathered together; it is known as a **Cluster**.

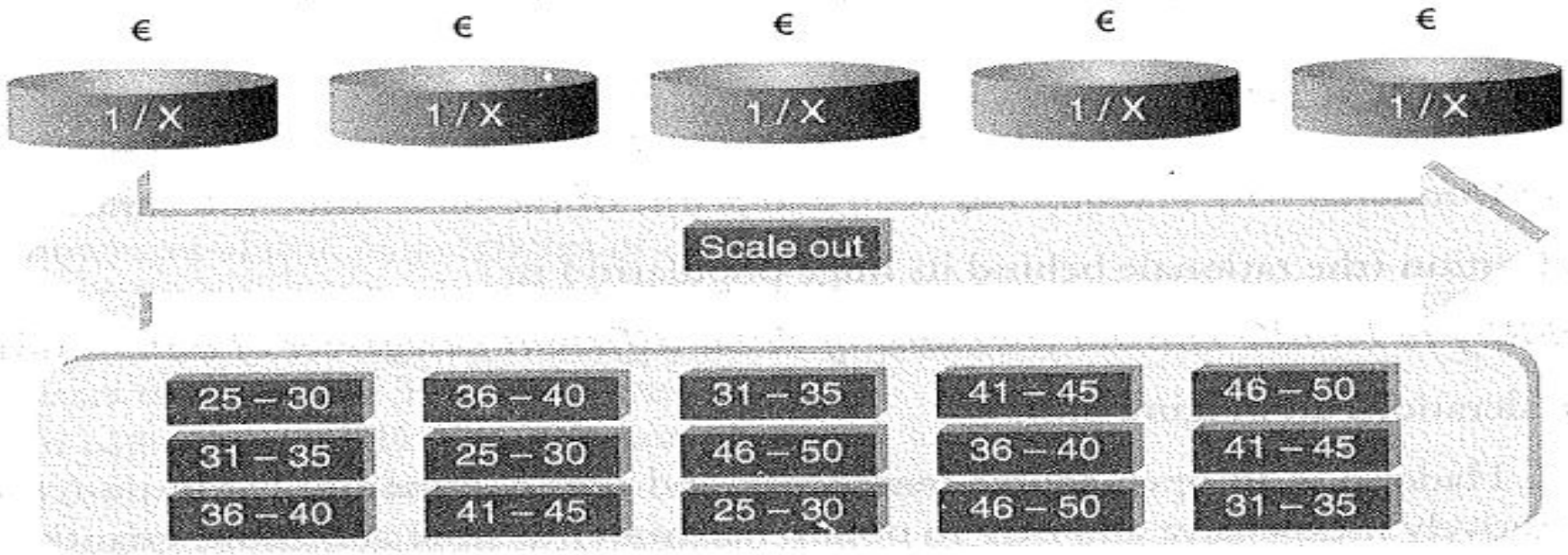


Figure 5.3 Hadoop framework (distributed file system, commodity hardware).

With this new paradigm, the data can be managed with **Hadoop** as follows:

1. Distributes the data and duplicates chunks of each data file across several nodes, for example, 25–30 is one chunk of data as shown in Figure 5.3.
2. Locally available compute resource is used to process each chunk of data in parallel.
3. Hadoop Framework handles failover smartly and automatically.

RDBMS versus HADOOP

RDBMS versus HADOOP

PARAMETERS	RDBMS	HADOOP
System	Relational Database Management System.	Node Based Flat Structure.
Data	Suitable for structured data.	Suitable for structured, unstructured data. Supports variety of data formats in real time such as XML, JSON, text based flat file formats, etc.
Processing	OLTP	Analytical, Big Data Processing
Choice	When the data needs consistent relationship.	Big Data processing, which does not require any consistent relationships between data.
Processor	Needs expensive hardware or high-end processors to store huge volumes of data.	In a Hadoop Cluster, a node requires only a processor, a network card, and few hard drives.
Cost	Cost around $10,000 to $14,000 per terabytes of storage.	Cost around $4,000 per terabytes of storage.

Distributed Computing Challenges

Distributed Computing Ch

- Hardware Failure

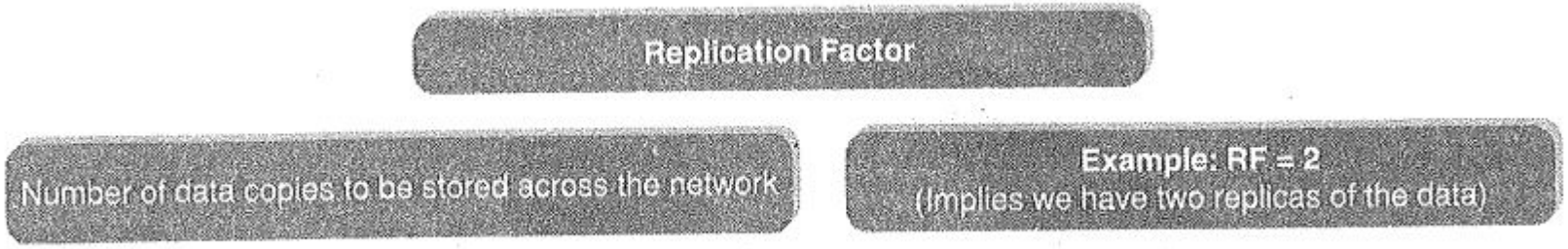


Figure 5.5 Replication factor.

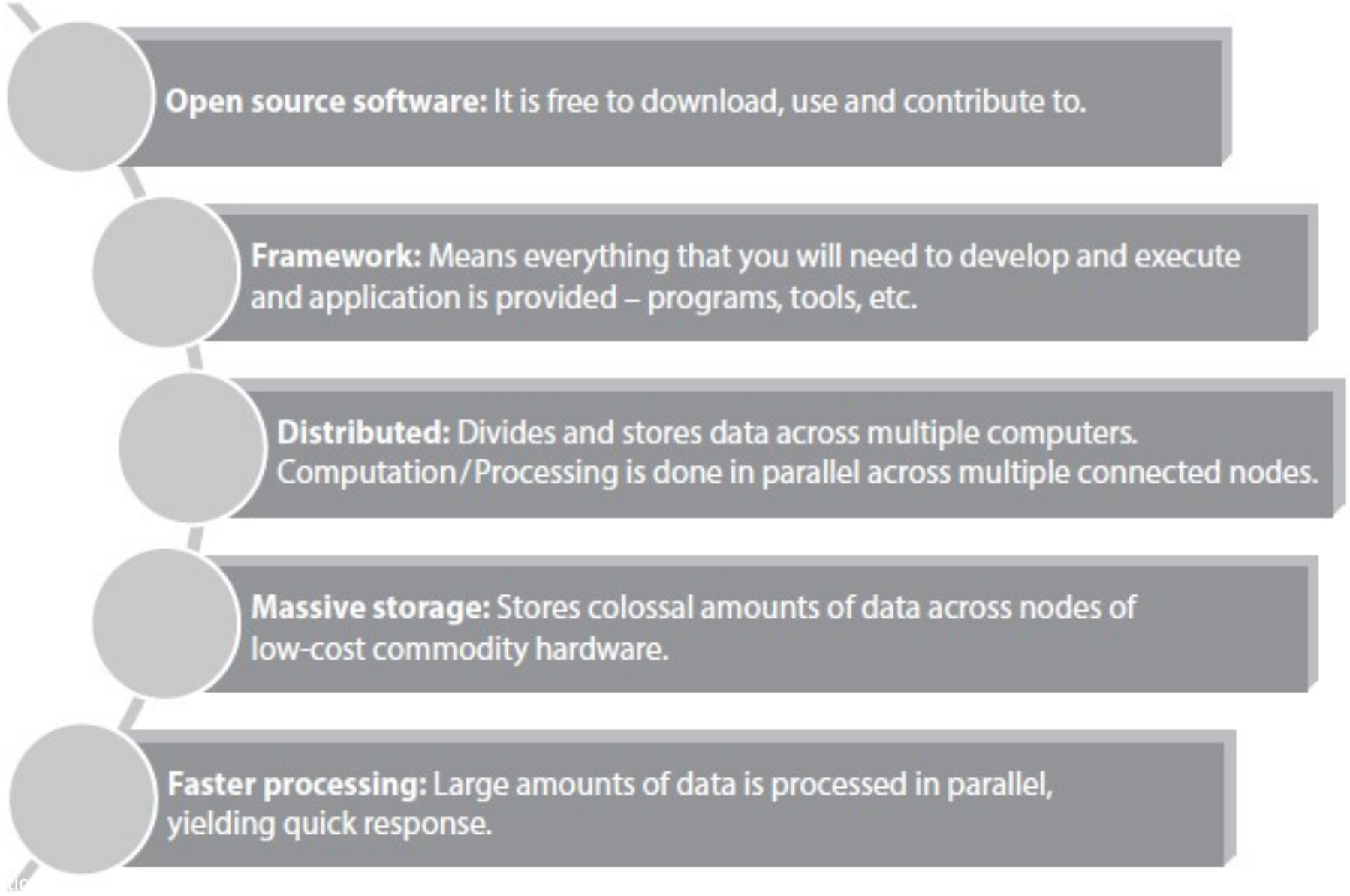
- **How to Process This Gigantic Store of Data?**

In a distributed system, the data is spread across the network on several machines. A key challenge here is to integrate the data available on several machines prior to processing it.

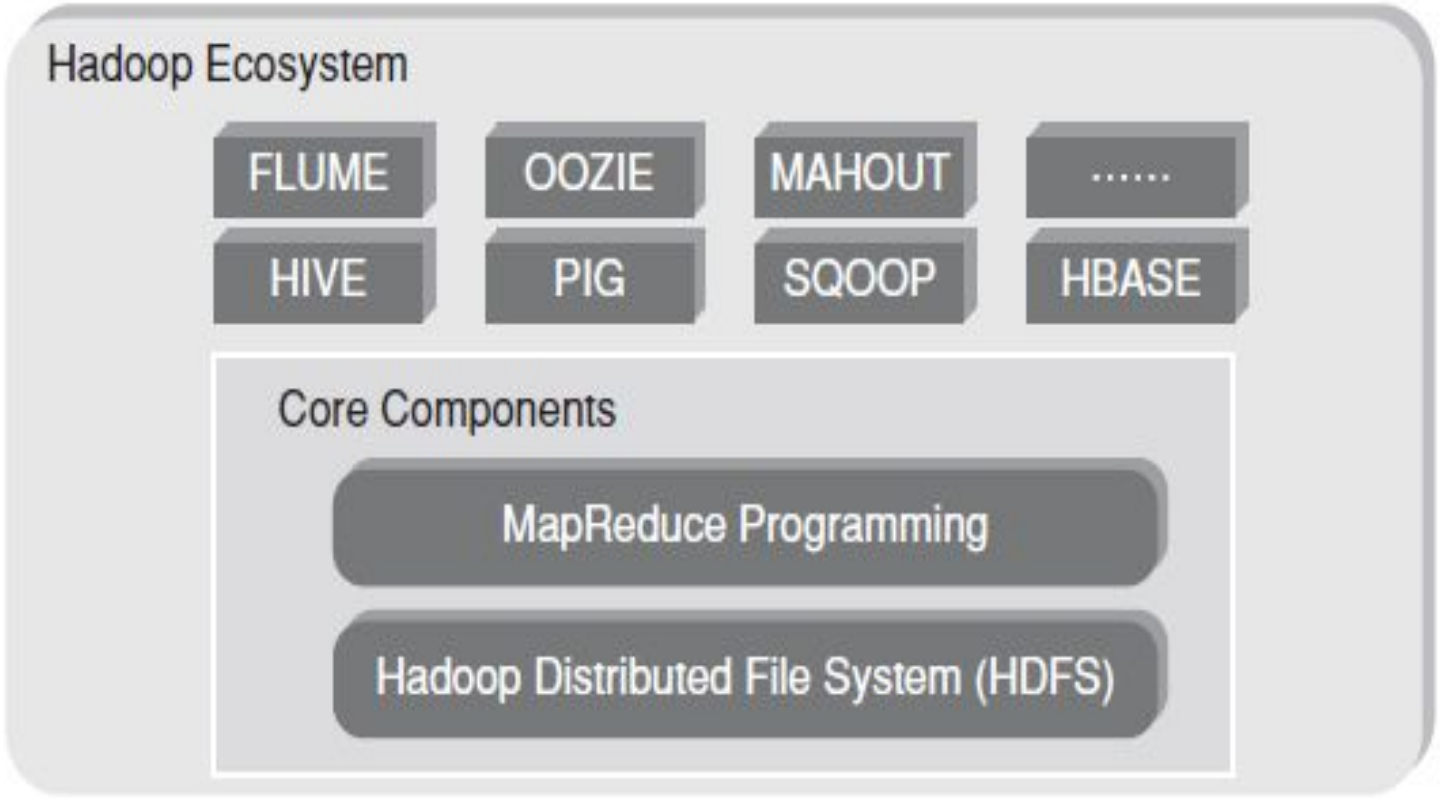
Hadoop solves this problem by using **MapReduce** Programming. It is a programming model to process the data (MapReduce programming will be discussed a little later).

Hadoop Overview

Key Aspects of Hadoop



Hadoop Components



Hadoop Components

Hadoop Core Components:

HDFS:

(a) Storage component.
(b) Distributes data across several nodes.
(c) Natively redundant.

MapReduce:

(a) Computational framework.
(b) Splits a task across multiple nodes.
(c) Processes data in parallel.

Hadoop Ecosystem

Support projects to **enhance the functionality of Hadoop Core Componen**ts

1. Hive
2. Pig
3. Scoop
4. Hbase
5. Flume
6. Oozie
7. Mahout

Hadoop Conceptual Layer

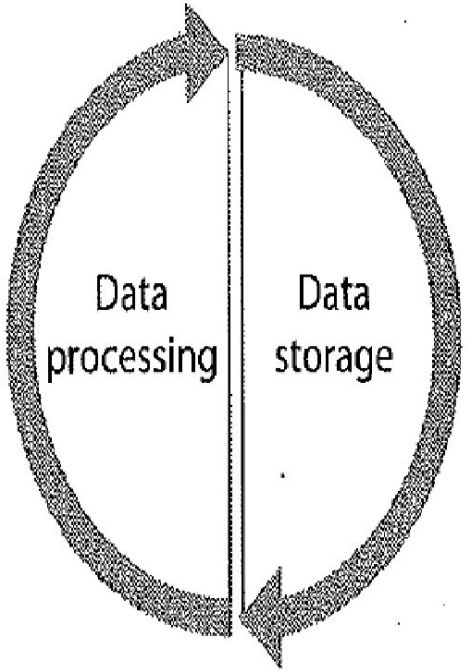


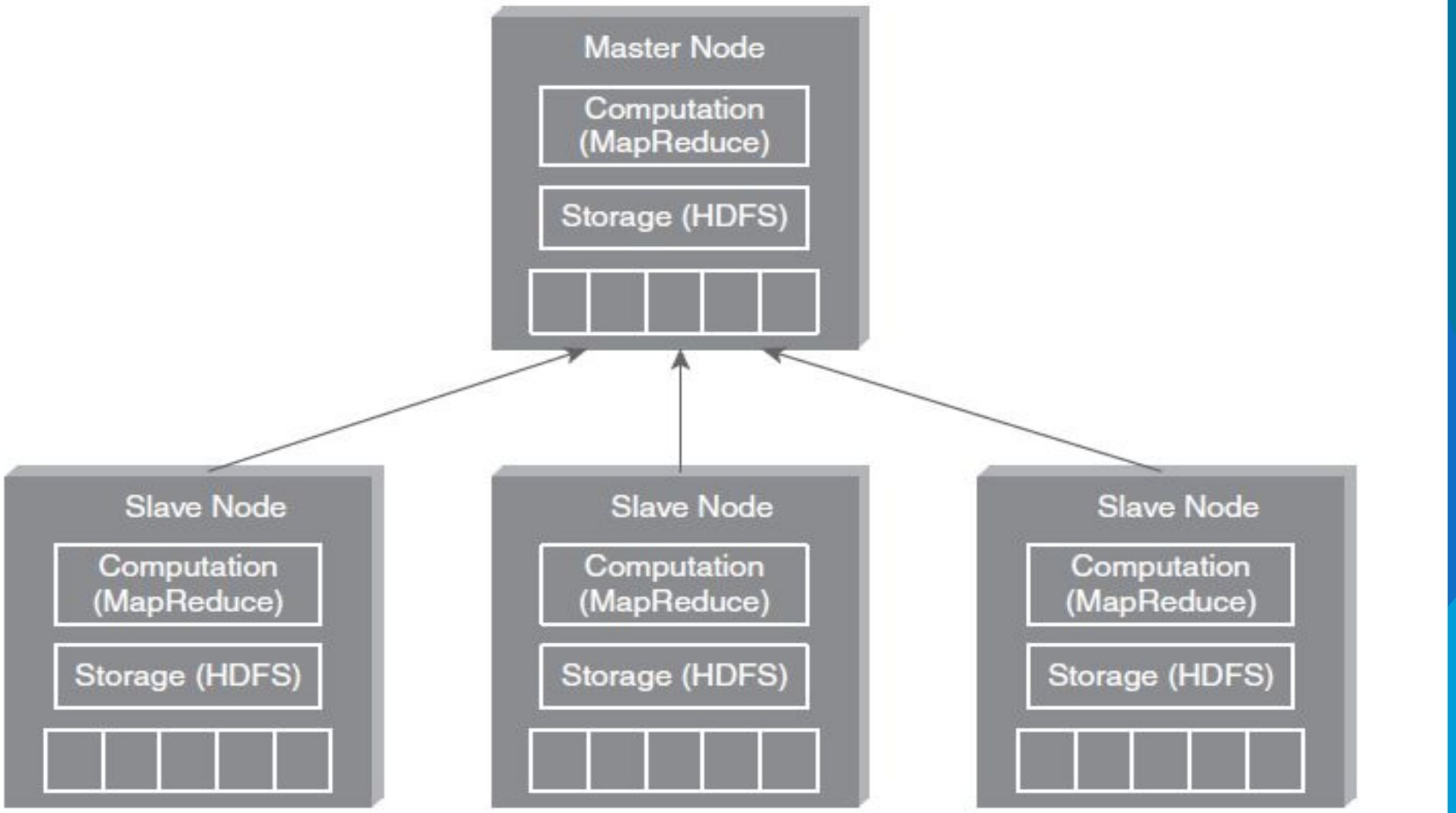
Figure 5.9 Hadoop conceptual layer.

Hadoop is conceptually divided into

1. Data storage layer **which stores huge volume of data** and

2. Data processing layer which **processes data in parallel** to extract **richer and meaningful insights from data**

Hadoop High Level Architecture



Hadoop is a distributed master slave architecture.

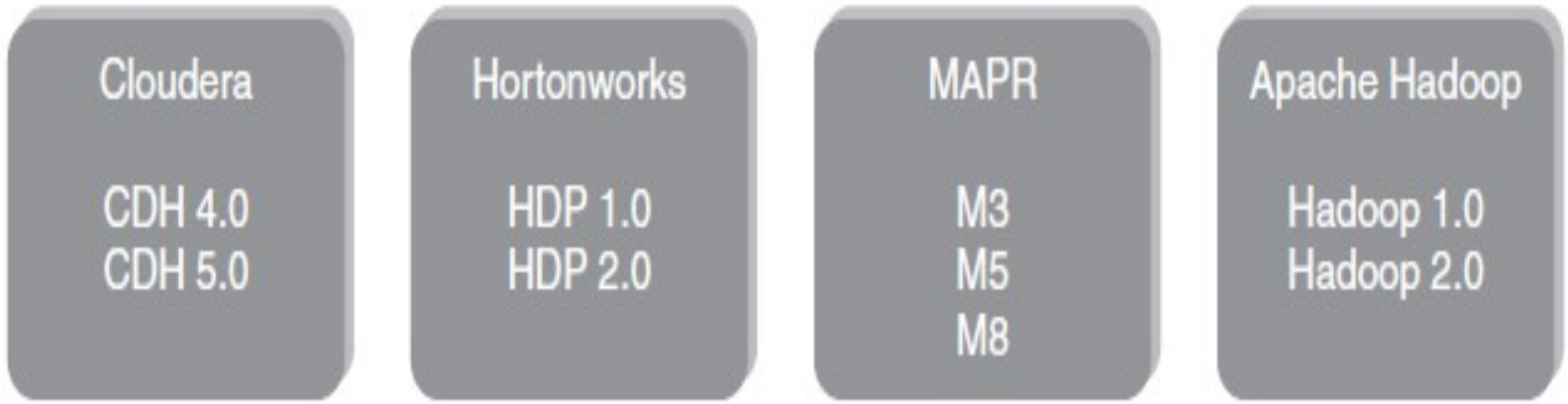
Master node- Name node
Slave node- Data node

Master HDFS- Its Main responsibility is **partitioning data storage across the slave node. It also keeps track of location of data on data nodes**

Master Mapreduce- It **decides and schedules computation task on slave node**

Hadoop Distributors

Hadoop Distributors



HDFS
(HADOOP DISTRIBUTED FILE SYSTEM)

HDFS is a **distributed file system (DFS) that runs on large clusters and provides high-throughput access to data.**

HDFS is a **highly fault-tolerant system** and is designed to work with **commodity hardware**.

HDFS stores **each file as a sequence of blocks**.

The **blocks of each file are replicated on multiple machines** in a cluster to provide fault tolerance

Let us look at the characteristics of HDFS:

- **Scalable Storage for Large Files**:

HDFS has been designed to store large files .

Large files are broken into chunks or blocks and each block is replicated across multiple machines in the cluster.

HDFS has been designed to scale to clusters comprising of thousands of nodes.

- **Replication:**

HDFS **replicates data blocks to multiple machines i**n a cluster which makes the system reliable and fault-tolerant.

The **default block size used is 64MB and the default replication factor is 3**.

- **Streaming Data Access:**

HDFS has been **designed for streaming data access patterns and provides high throughput streaming reads and writes.**

The HDFS make it **suitable for batch operations thus trading off interactive access capability.**

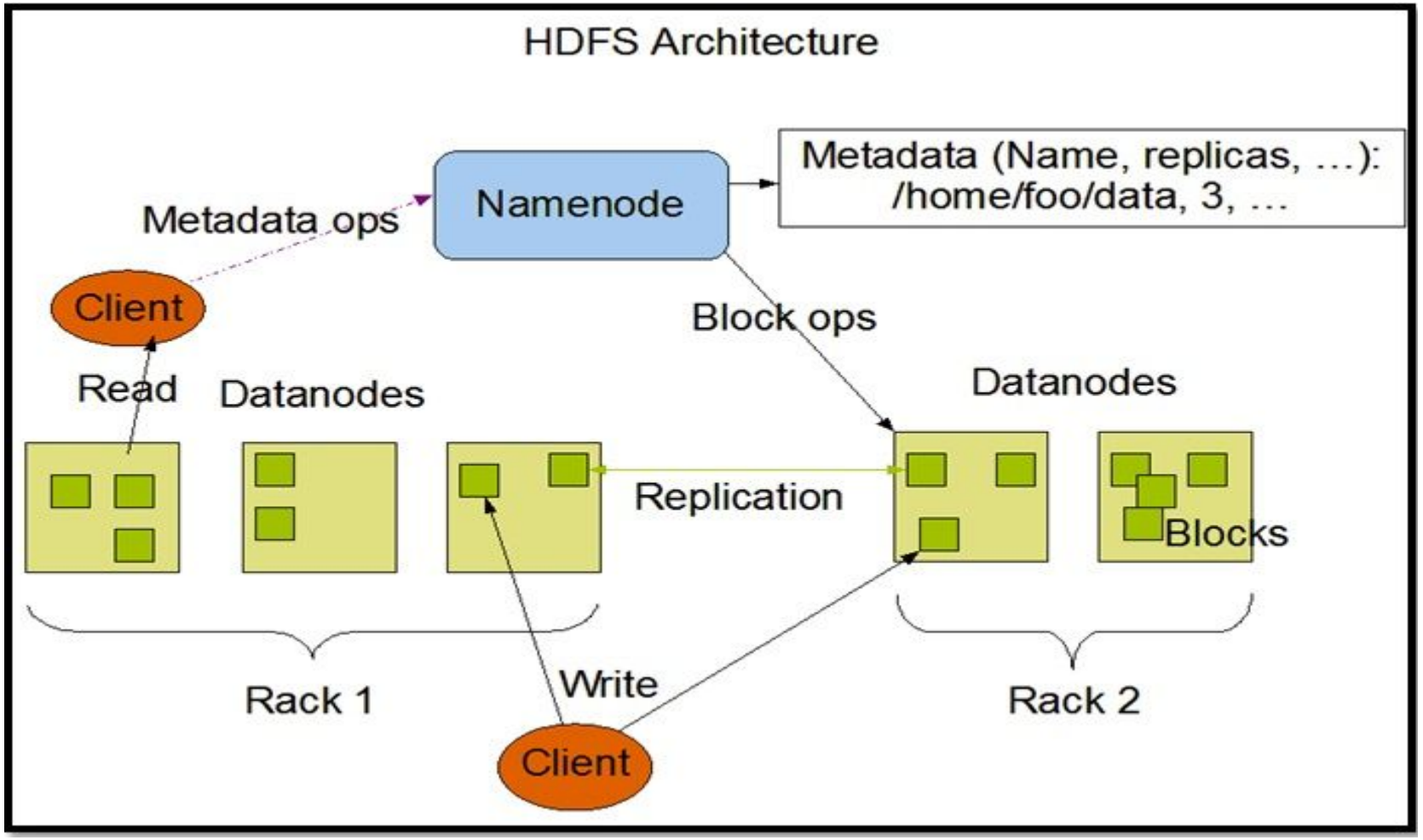
This **design choice has been made to meet the requirements of applications that involve write-once, read many times** data access patterns.

- **File Appends:**

Recent versions of HDFS have introduced the append capability

HDFS Architecture

Figure shows the architecture of HDFS. HDFS has two types of nodes: **Namenode and Datanode**.



Namenode

Namenode **manages the filesystem namespace**.

All the **filesystem meta-data is stored on the Namenode**.

While **Namenode is responsible for executing operations such as opening and closing of files, no data actually flows through the Namenode**.

Namenode **executes the read and write operations while the data is transferred directly to/from the Datanodes.**

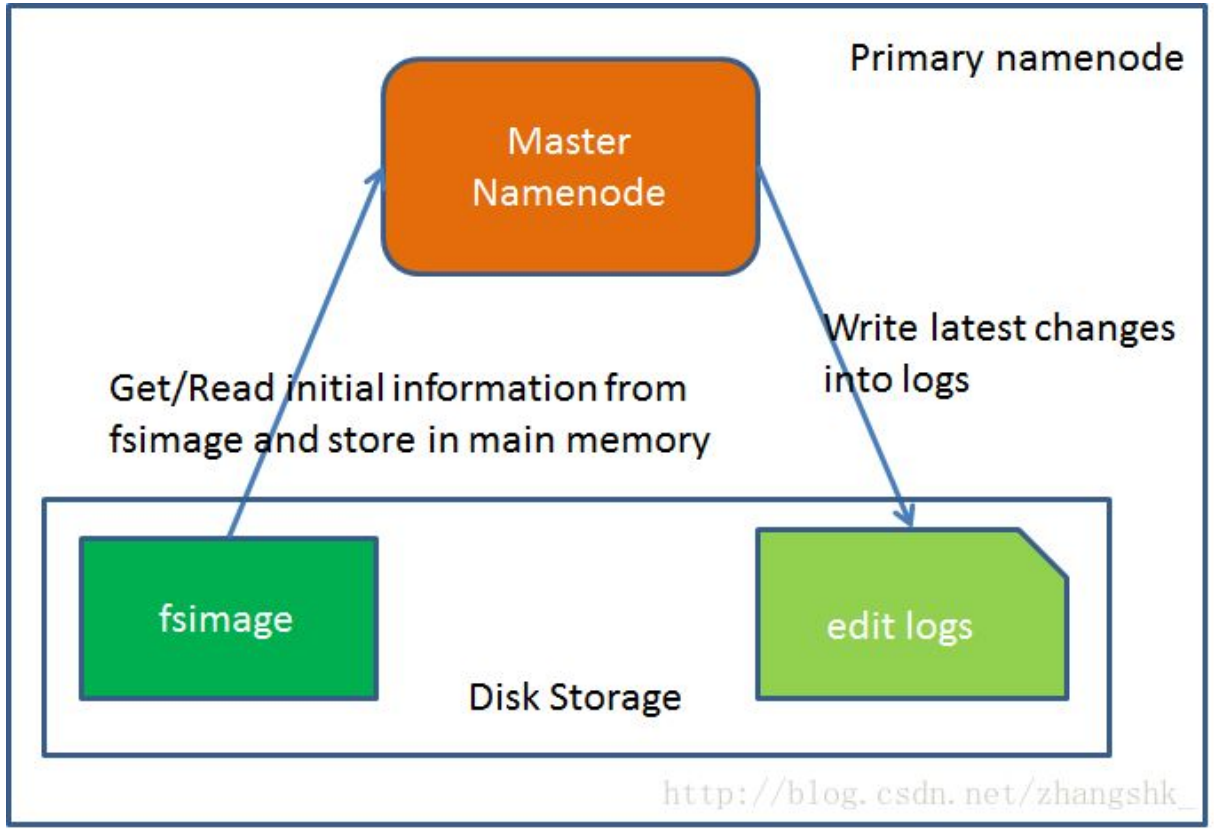
HDFS **splits files into blocks, and the blocks are** stored on the Datanodes.

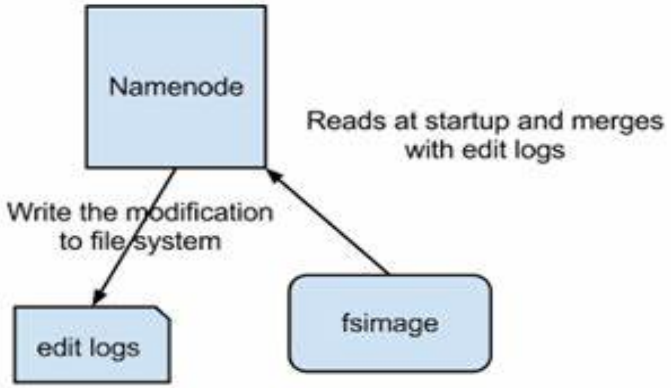
For each block, **multiple replicas are kept**.

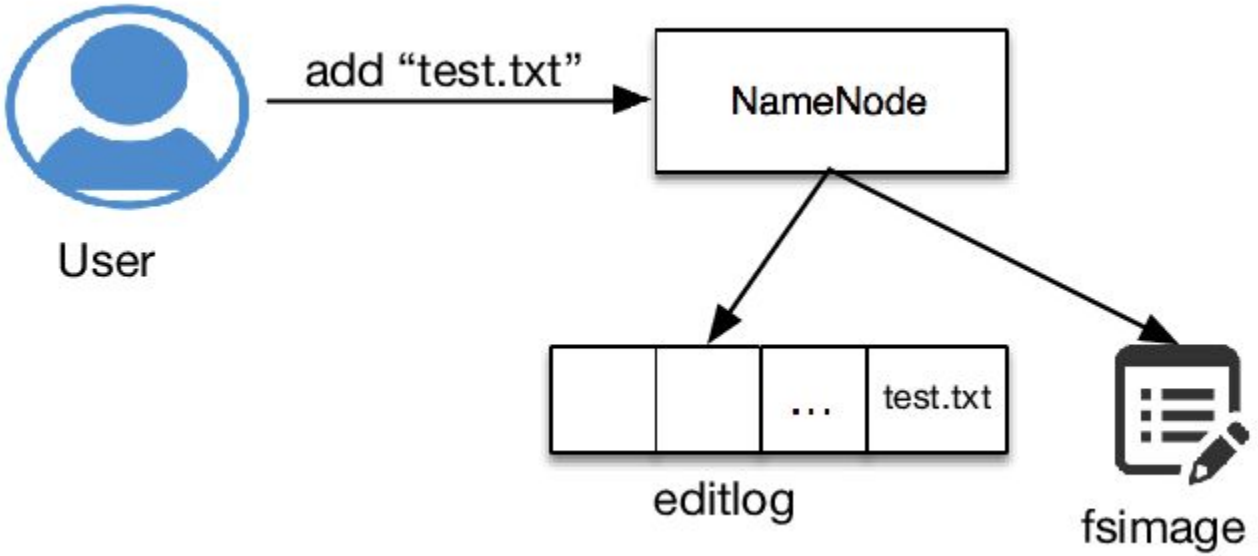
Namenode **persistently stores the filesystem meta-data and the mappings of the blocks to the datanodes, on the disk as two files**:

- fsimage and
- edits files.

.







The **fsimage contains a complete snapshot of the filesystem meta-data**.

The **edits file stores the incremental updates to the meta-data**.

When the **Namenode starts, it loads the fsimage file into the memory and applies the edits file to bring the in-memory view of the filesystem up-to-date**.

Namenode **then writes a new fsimage file to the disk**

Secondary Namenode

The **edits file keeps growing in size, over time, as the incremental updates are stored.**

The **responsibility of applying the updates to the fsimage file is delegated to the Secondary Namenode**, as the Namenode may not have enough resources available, as it is performing other operations.

This process is called **checkpointing**.

The **checkpointing process is done either periodically** (default 1 hour) or **after a certain number of uncheck pointed transactions have been reached on the Namenode.**

The **new fsimage is uploaded by the Secondary Namenode to the Namenode**

When the **checkpointing process begins, the Secondary Namenode downloads the fsimage and edits files from the Namenode to the checkpoint directory on the Secondary Namenode.**

The **Secondary Namenode then applies the edits on the fsimage file and creates a new fsimage file**.

Datanode
While the Namenode stores the filesystem meta-data, the Datanodes store the data blocks and serve the read and write requests.

Datanodes periodically send heartbeat messages and blockreports to the Namenode.

While the heartbeat messages tell the Namenode that a Datanode is alive, the block reports contain information on the blocks on a Datanode.

Data Blocks & Replication

Blocks are replicated on the Datanodes and by default **three replicas** are created.

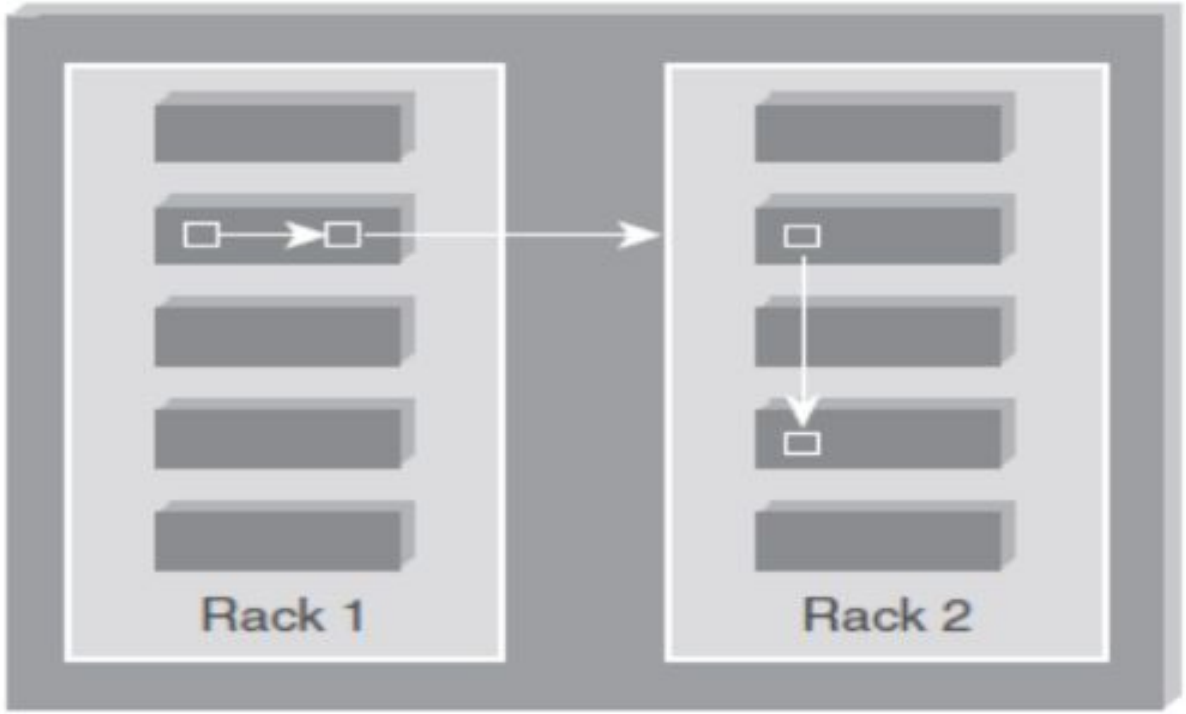
The **placement of replicas on the Datanodes is determined by a rack-aware placement policy.**

This placement policy ensures reliability and availability of the blocks.

For a replication factor of three, **one replica is placed on a node on a local rack, the second replica is placed on a different node on a remote rack and the third replica is placed on a different node on the same remote rack.**

This ensures that **even if the rack becomes unavailable, at least one replica will remain available.**

Placement of **replicas on different nodes in the same rack minimizes the network traffic between the racks.**



HDFS Read Path

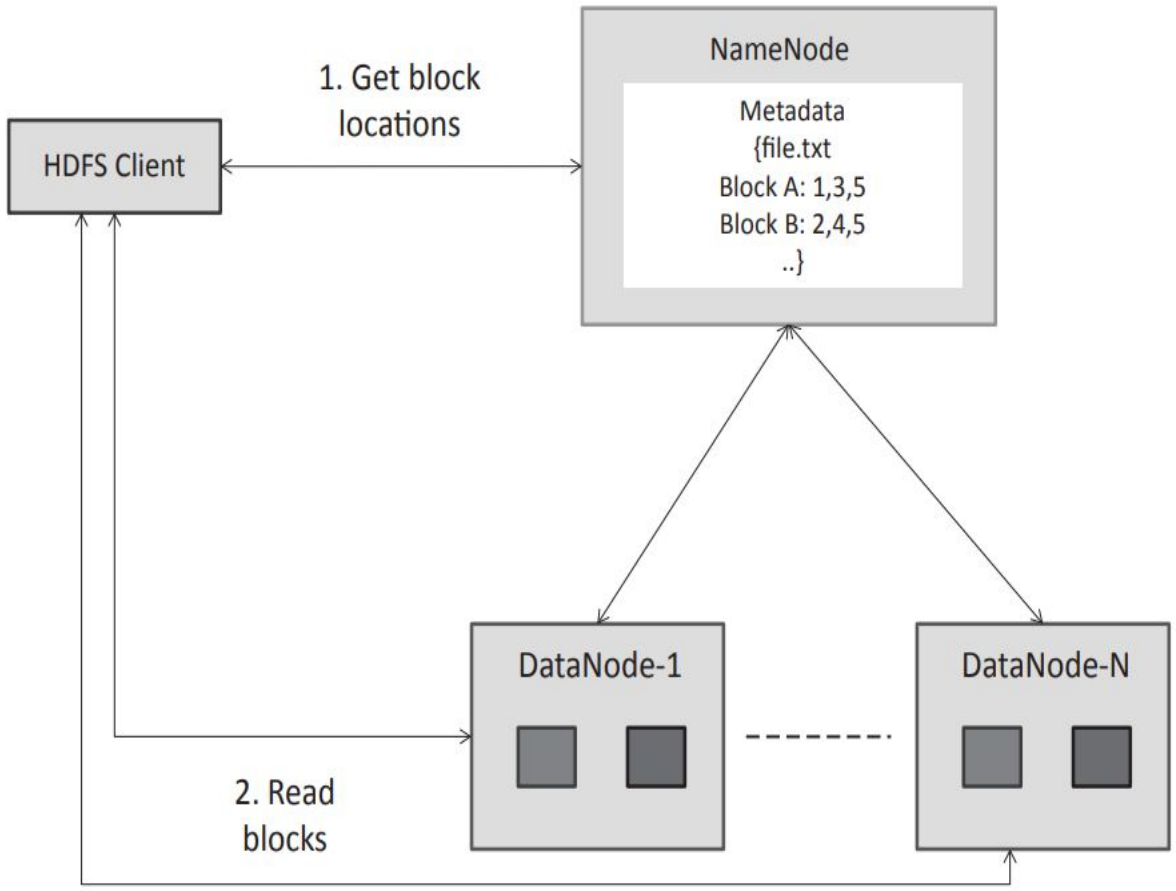


Figure 6.2: HDFS read path

Figure shows the HDFS read path.

The read process begins with the **client sending a request to the Namenode to obtain the locations of the data blocks for a file.**

The **Namenode checks if the file exists and whether the client has sufficient permissions to read the file.**

The **Namenode responds with the data block locations sorted by the distance to the client.**

This helps in **minimizing the traffic between the nodes as the client can read the blocks from the nearest node**.

For example, if the **client is on the same node as a data block, it can read the data block locally.**

The **client reads the data blocks directly from the Datanodes in order, till all the blocks have been read.**

The **Datanodes stream the data to the client**.

During the read process, **if a replica becomes unavailable, the client can read another replica on a different Datanode**.

HDFS Write Path

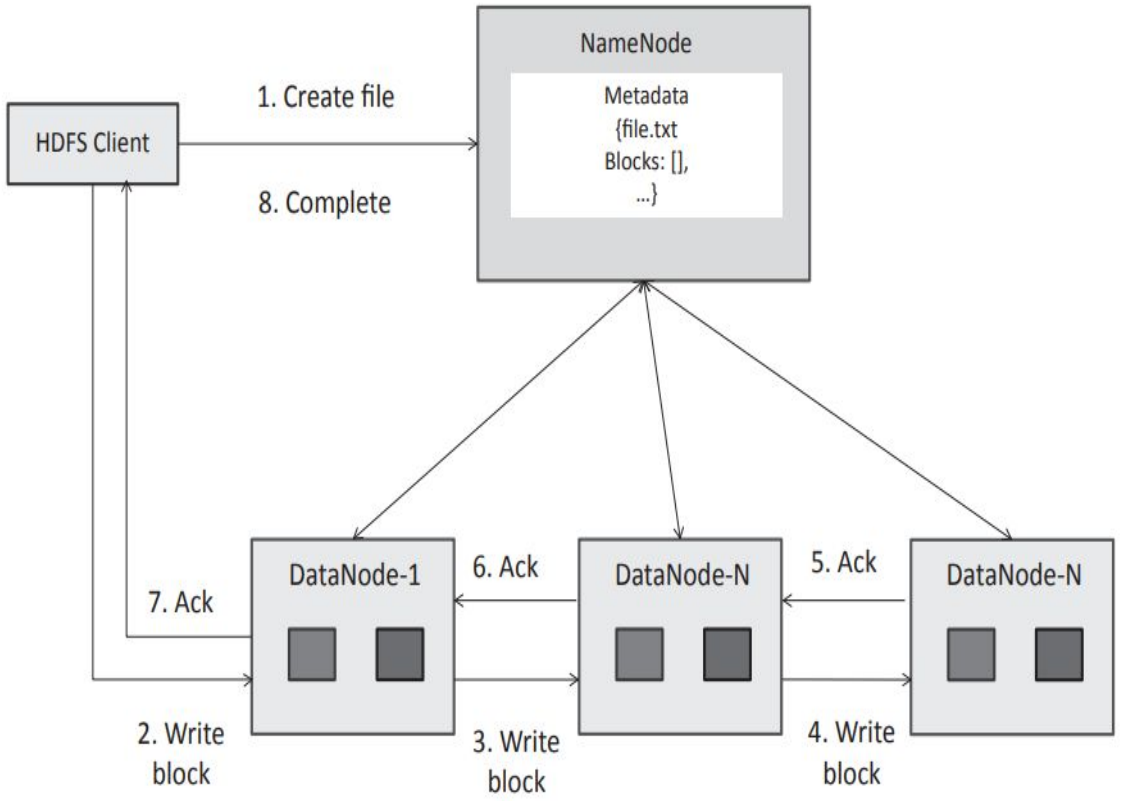


Figure 6.3: HDFS write path

Figure shows the HDFS write path.

The write process begins with the **client sending a request to the Namenode to create a new file in the filesystem namespace.**

The **Namenode checks if the user has sufficient permissions to create the file** and whether the file doesn't already exist in the filesystem.

The **Namenode responds to the client with an output stream object.**

The **client writes data to the output stream object which splits the data into packets and enqueues them into a data queue**.

The **packets are consumed from the data queue in a separate thread, which requests the Namenode to allocate new blocks on the Datanodes to which the data should be written.**

Namenode responds with the locations of the blocks on the Datanodes.

The client **then establishes direct connections to the Datanodes on which the blocks are to be replicated forming a replication pipeline.**

The **data packets consumed from the data queue are written to the first Datanode on the replication pipeline, which writes data to the second Datanode in the pipeline and so on**

Once the packets are successfully written, each Datanode in the pipeline sends an acknowledgment.

The **client keeps a track of which all packets are acknowledged by the Datanodes**. The **process of writing data packets to the Datanodes proceeds till the block size is reached.**

Upon reaching the block size, the client again requests the Namenode to return a set of new blocks on the Datanodes.

The **client then streams the packets to the Datanodes.** This **process repeats till all the data packets are written and acknowledged**.

Finally, the **client closes the output stream and sends a request to the Namenode to close the file**

HDFS Commands

Summary

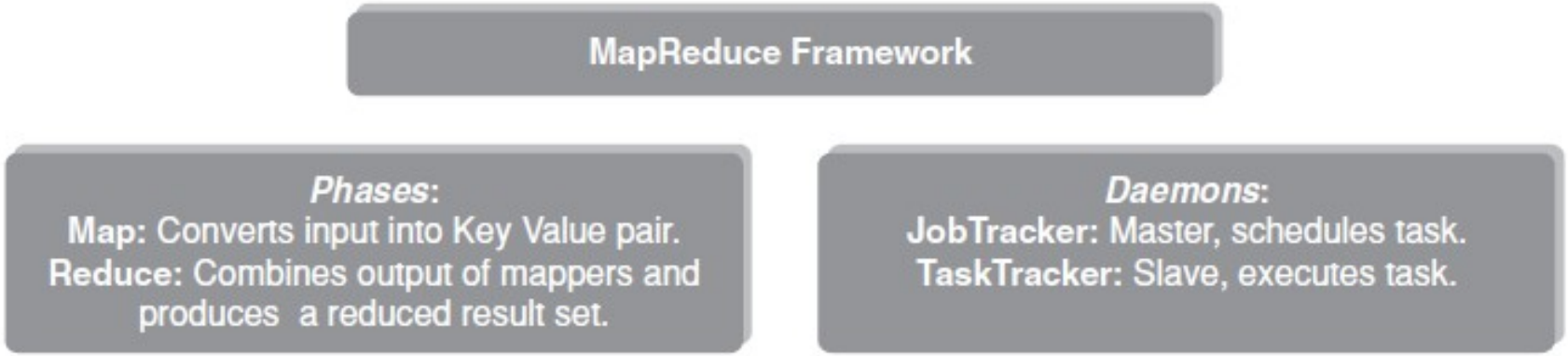
HDFS is a distributed file system that runs on large clusters and provides high-throughput access to data. HDFS provides scalable storage for large files which are broken into blocks. The blocks are replicated to make the system reliable and fault-tolerant. The HDFS Namenode stores the filesystem meta-data and is responsible for executing operations such as opening and closing of files. The Secondary Namenode helps in the checkpointing process by applying the updates in the edits file to the fsimage file which contains a complete snapshot of the filesystem meta-data. Datanodes store the data blocks which are replicated. The placement of replicas on the Datanodes is determined by a rack-aware placement policy. We described examples of accessing HDFS using the command line tools, a Python library for HDFS and the HDFS web interface.

Hadoop and MapReduce

Apache **Hadoop is an open source framework for distributed batch processing of big data**.

Similarly, **MapReduce is a parallel programming model i.e, a software framework suitable for analysis of big data.**

MapReduce algorithms **allow large-scale computations to be automatically parallelized across a large cluster of servers.**



MapReduce Programming Model

MapReduce is a **parallel data processing model for processing and analysis of massive scale data .**

MapReduce model has **two phases**: **Map and Reduce**.

MapReduce programs are **written in a functional programming style to create Map and Reduce functions.**

The **input data to the map and reduce phases is in the form of key-value pairs.**

Run-time systems for MapReduce are typically large clusters built of commodity hardware.

The MapReduce **run-time systems take care of tasks such partitioning the data, scheduling of jobs and communication between nodes in the cluster**.

This makes it easier for **programmers to analyze massive scale data without worrying about tasks such as data partitioning and scheduling**.

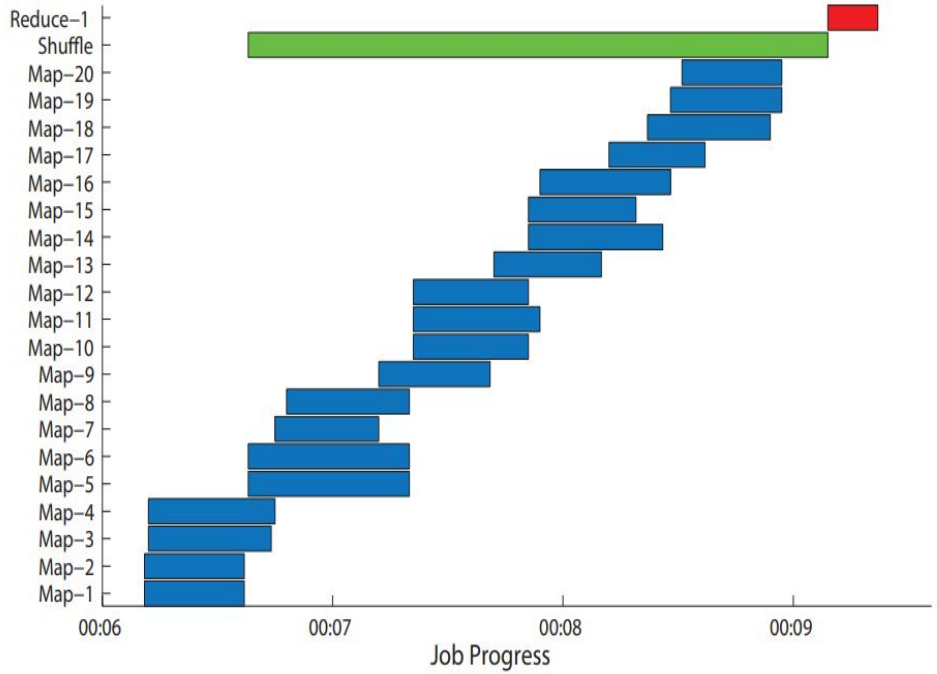
In the Map phase, **data is read from a distributed file system, partitioned among a set of computing nodes in the cluster, and sent to the nodes as a set of key-value pairs.**

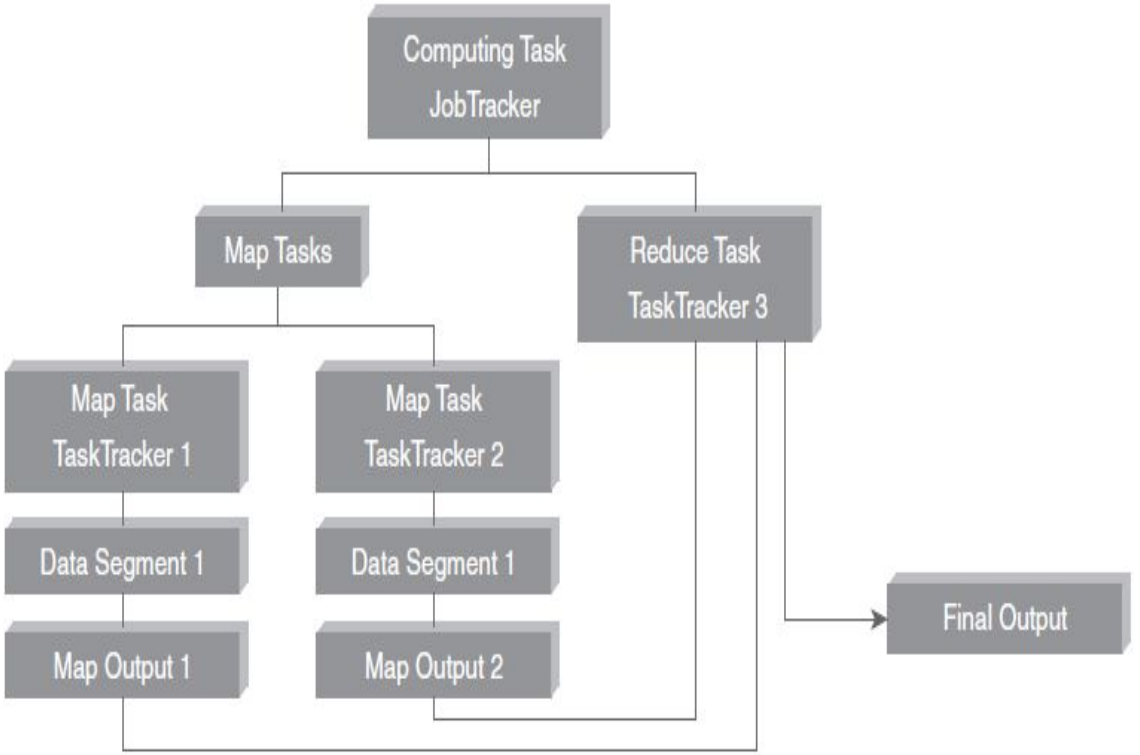
The **Map tasks process the input records independently of each other and produce intermediate results as key-value pairs.**

The **intermediate results are stored on the local disk of the node running the Map task.**

When all the **Map tasks are completed, the Reduce phase begins in which the intermediate data with the same key is aggregated.**

An **optional Combine task can be used to perform data aggregation on the intermediate data of the same key for the output of the mapper before transferring the output to the Reduce task.**





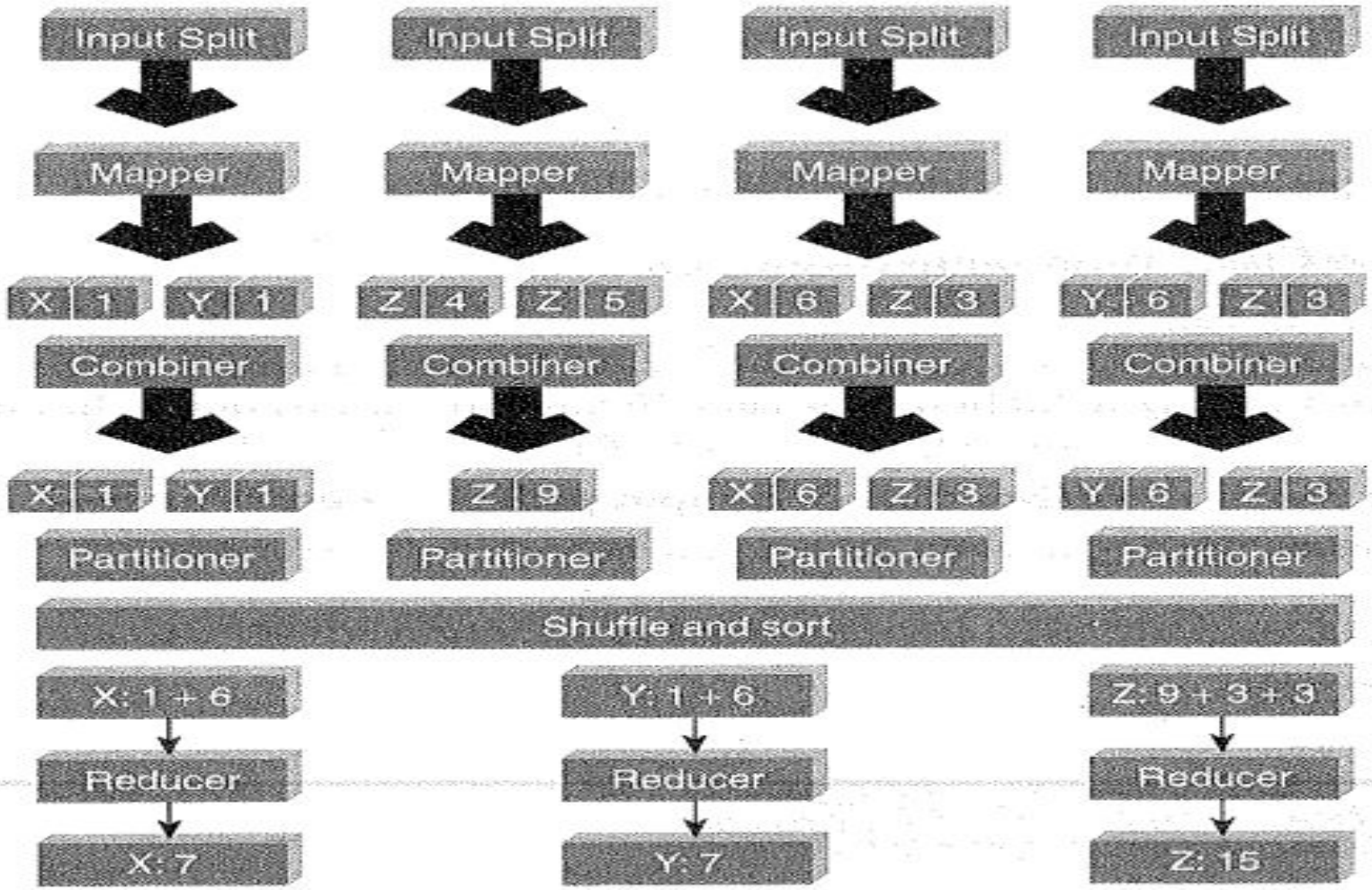
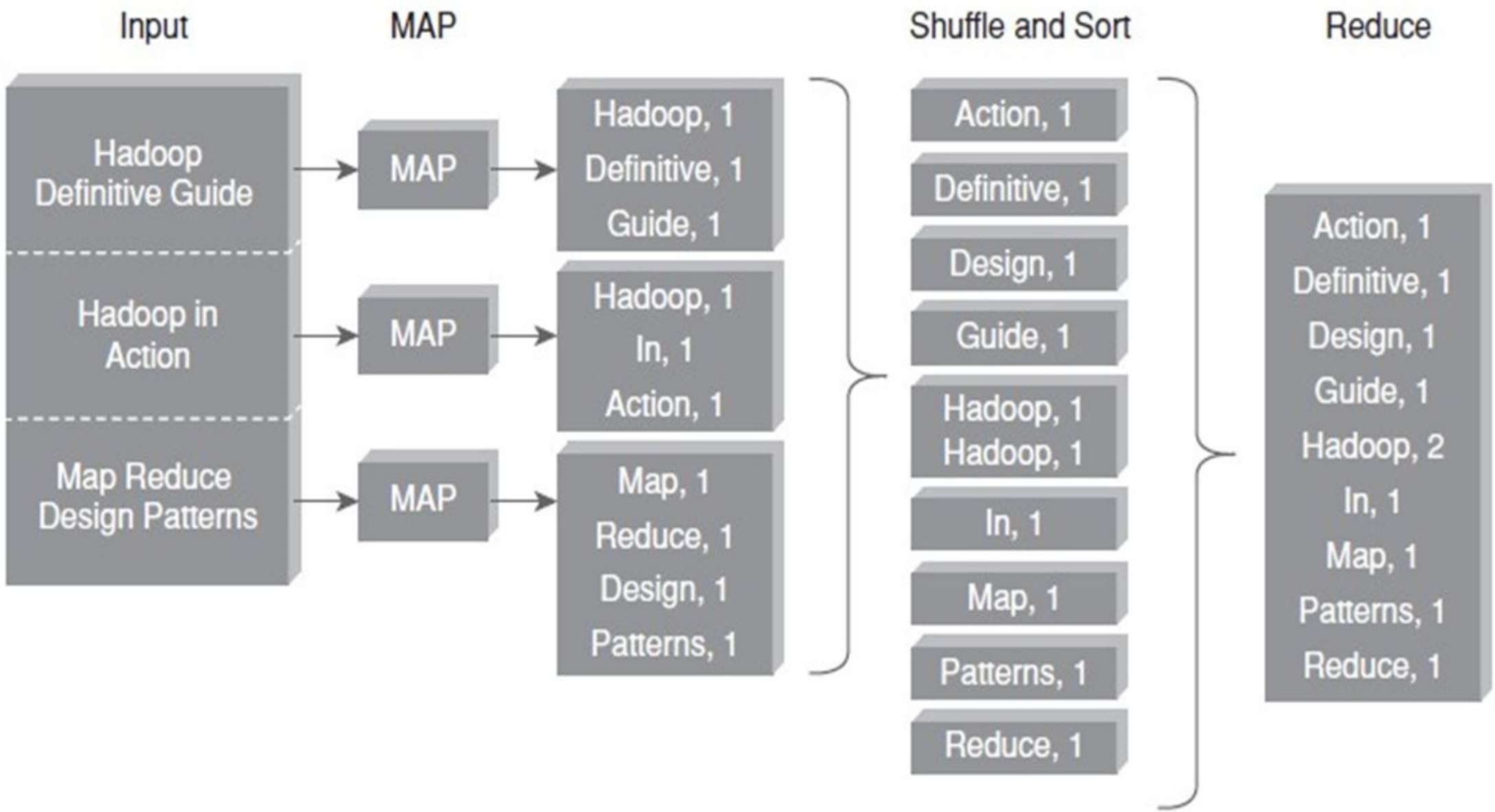


Figure 8.1 The chores of Mapper, Combiner, Partitioner, and Reducer.


Input
MAP
Shuffle and Sort
Reduce
Hadoop Definitive Guide
Hadoop in Action
Map Reduce Design Patterns
MAP
MAP
MAP
Hadoop, 1
Definitive, 1
Guide, 1
Hadoop, 1
In, 1
Action, 1
Map, 1
Reduce, 1
Design, 1
Patterns, 1
Action, 1
Definitive, 1
Design, 1
Guide, 1
Hadoop, 1
Hadoop, 1
In, 1
Map, 1
Patterns, 1
Reduce, 1
Action, 1
Definitive, 1
Design, 1
Guide, 1
Hadoop, 2
In, 1
Map, 1
Patterns, 1
Reduce, 1

Drawback of Hadoop 1.0

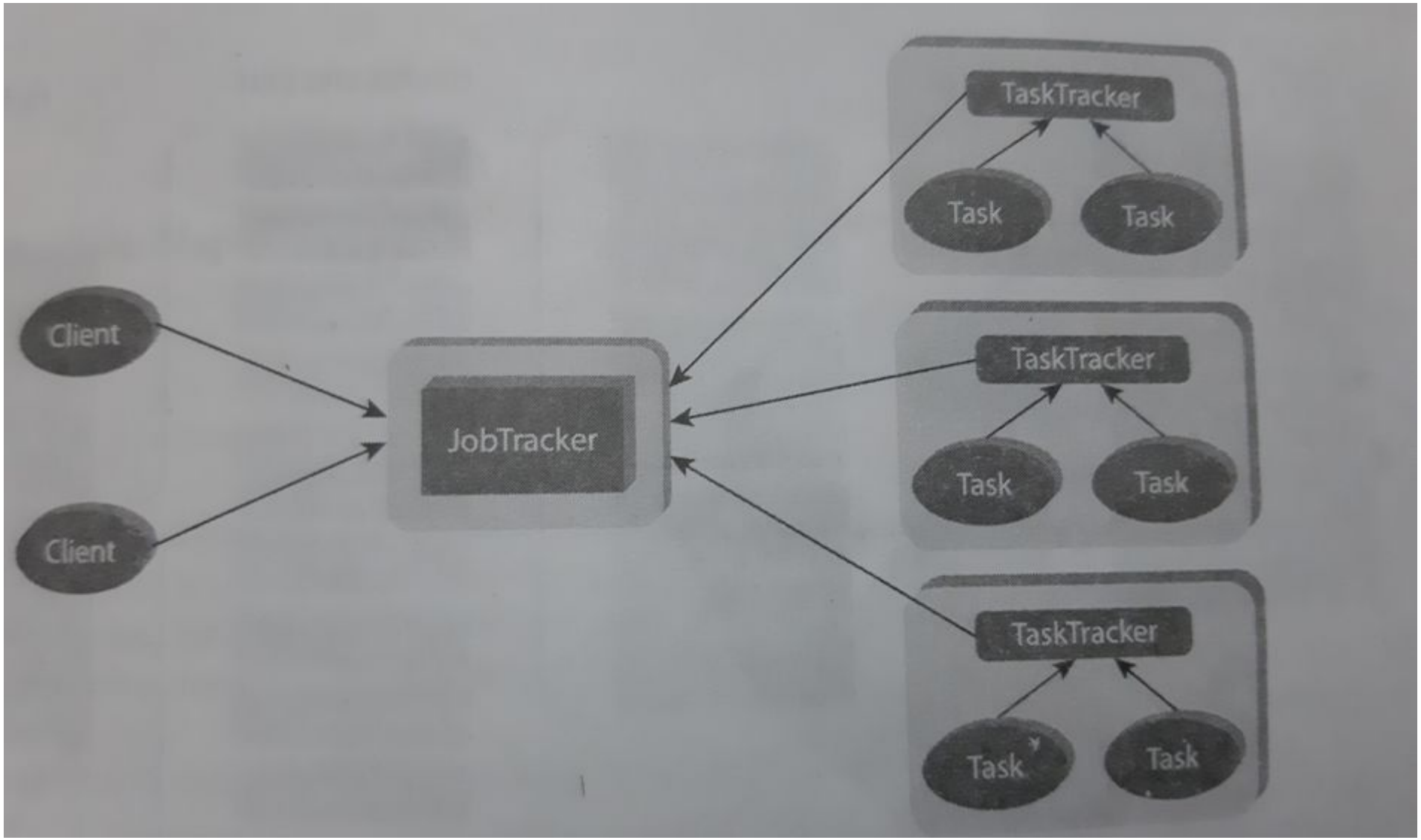
- It is **only suitable for Batch Processing** of Huge amount of Data

- It is **not suitable for Real-time** Data Processing.

- It is **not suitable for Data Streaming**.
- It **supports upto 4000 Nodes** per Cluster.

- It **has a single component : JobTracker** to perform many activities like **Resource Management, Job Scheduling, Job Monitoring, Re-scheduling Jobs etc**.

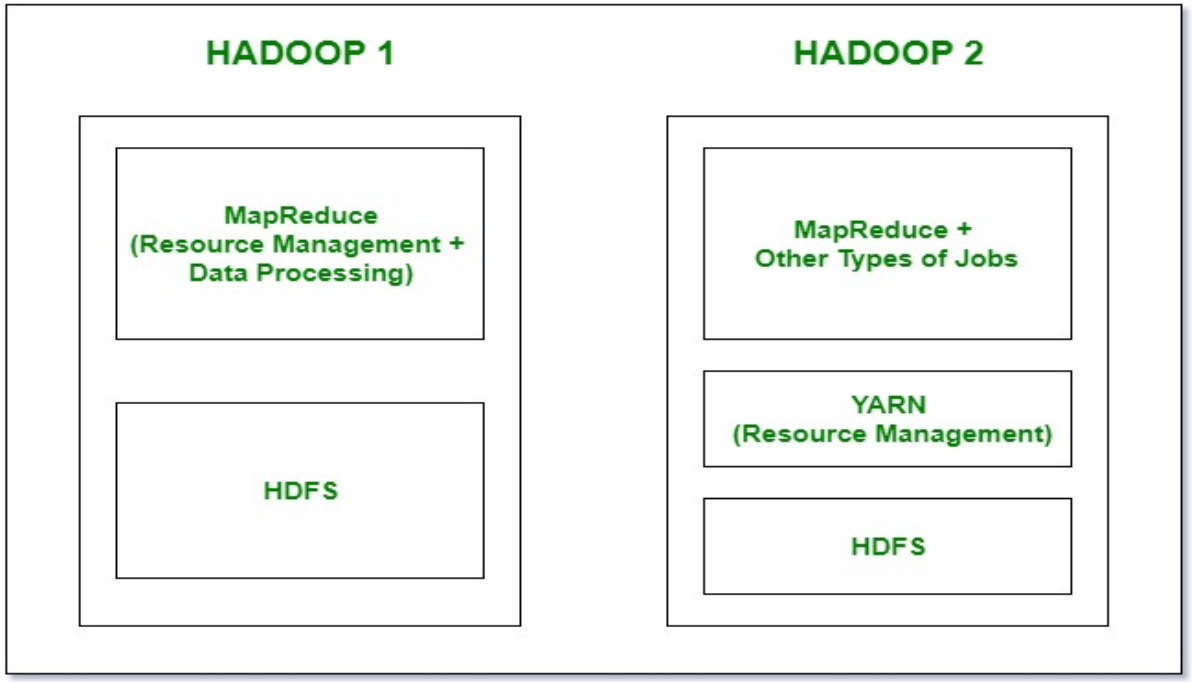
- JobTracker is the **single point of failure**.
- It supports **only one Name Node and One Namespace per Cluster**.
- It **runs only Map/Reduce j**obs


Client
Client
JobTracker
TaskTracker
Task
Task
TaskTracker
Task
Task
TaskTracker
Task
Task

Hadoop 2 YARN- Taking Hadoop beyond Batch

Hadoop YARN is **the next generation architecture of Hadoop** (version 2.x).

In the YARN architecture, **the original processing engine of Hadoop (MapReduce) has been separated from the resource management component (which is now part of YARN) as shown**



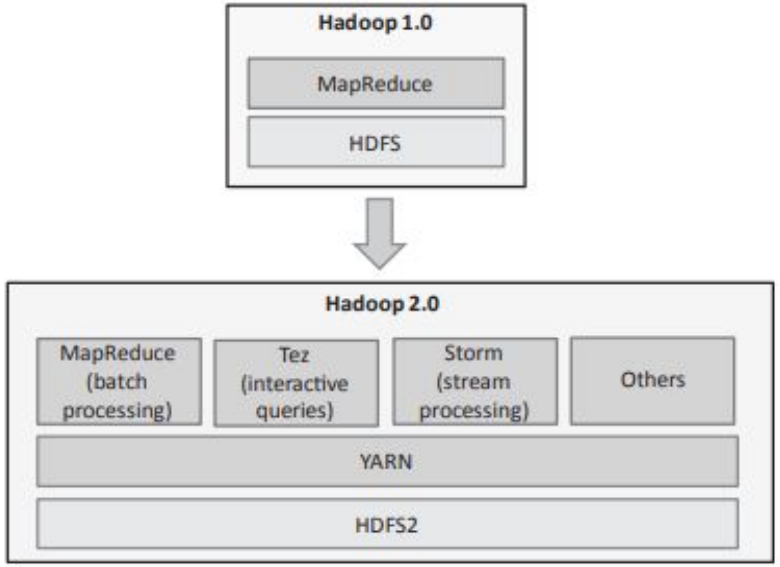


Figure 7.3: Comparison of Hadoop 1.x and 2.x architectures

Figure shows the **MapReduce job execution workflow for the next generation Hadoop MapReduce framework (MR2).**

The next-generation MapReduce architecture **divides the two major functions of the JobTracker** in Hadoop 1.x – **resource management and job life-cycle management** – into separate components – **ResourceManager and ApplicationMaster**.

The key components of YARN are described as follows:

• Resource Manager (RM): RM **manages the global assignment of compute resources to applications.**

RM consists of **two main services**:

– Scheduler: Scheduler is a **pluggable service that manages and enforces the resource scheduling policy** in the **cluster**.

– Applications Manager (AsM): AsM **manages the running Application Masters in the cluster.**

AsM is **responsible for starting application masters and for monitoring and restarting them on different nodes in case of failures**.

- Application Master (AM): A **per-application AM manages the application's life cycle.**
  AM is **responsible for negotiating resources from the RM and working with the NMs to execute and monitor the tasks.**

- Node Manager (NM): A **per-machine NM manages the user processes on that machine.**

- Containers: Container is a **bundle of resources allocated by RM** (memory, CPU and network).

- A container is a **conceptual entity that grants an application the privilege to use a certain amount of resources on a given machine to run a task**.

- Each node has an **NM that spawns multiple containers based on the resource allocations made by the RM**

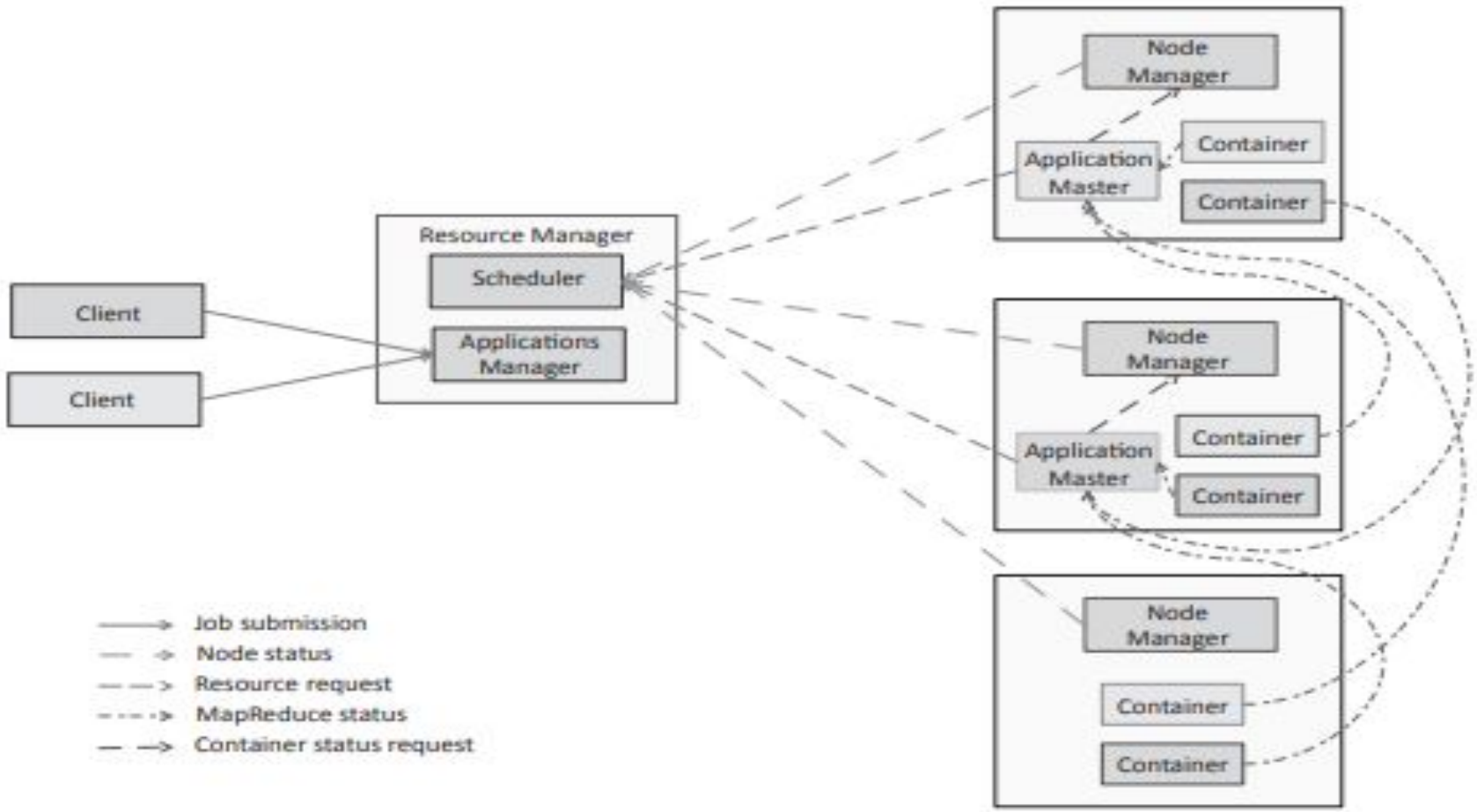


Figure 7.4: Hadoop MapReduce Next Generation (YARN) job execution

Figure 7.4 shows a **YARN cluster with a Resource Manager node and three Node Manager nodes**.

There are **as many Application Masters running as there are applications** (jobs).

Each **application's AM manages the application tasks such as starting, monitoring and restarting tasks in case of failures.**

Each application has multiple tasks. Each task runs in a separate container.

Containers in **YARN architecture are similar to task slots in Hadoop MapReduce 1.x** (MR1).

However, **unlike MR1 which differentiates between map and reduce slots, each container in YARN can be used for both map and reduce tasks.**

The **resource allocation model in MR1 consists of a predefined number of map slots and reduce slots**.

This static allocation of slots **results in low cluster utilization**.

The **resource allocation model of YARN is more flexible with the introduction of resource containers which improve cluster utilization.**

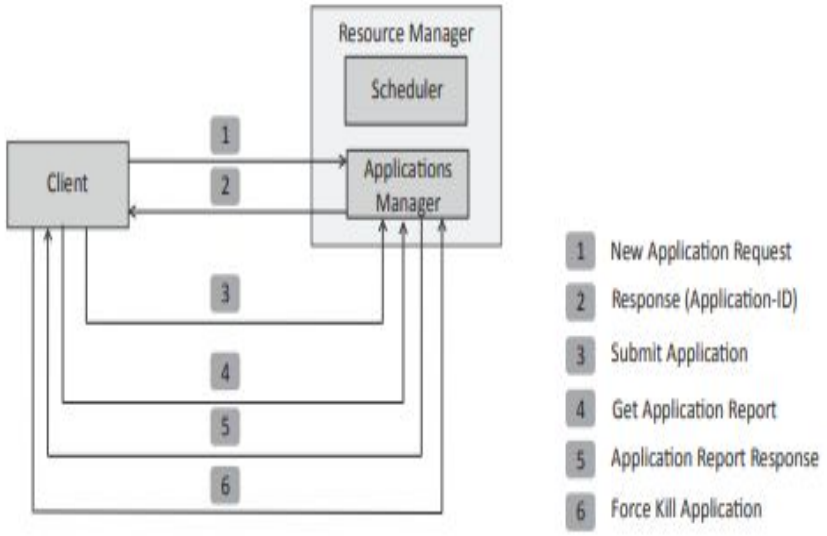


Figure 7.5: Client - Resource Manager interaction

To better understand the **YARN job execution workflow let us analyze the interactions between the main components on YARN.**

Figure 7.5 shows the **interactions between a Client and Resource Manager**.

Job execution begins with the submission of a new application request by the client to the RM.

The R**M then responds with a unique application ID and information about cluster resource capabilities that the client will need in requesting resources for running the application's AM**.

Using the information received from the RM, **the client constructs and submits an Application Submission Context which contains information such as scheduler queue, priority and user information.**

The **Application Submission Context also contains a Container Launch Context which contains the application's jar, job files, security tokens and any resource requirements**.

The **client can query the RM for application reports**.
The **client can also "force kill" an application** by sending a request to the RM

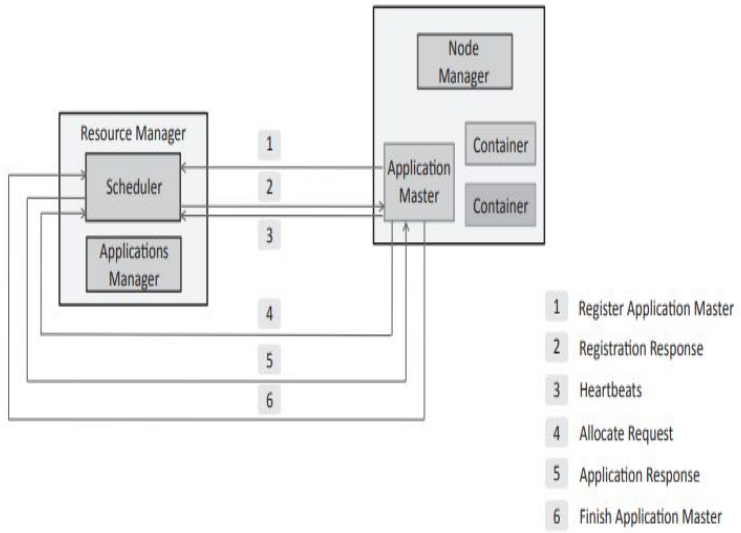


Figure 7.6: Resource Manager - Application Master interaction

Figure 7.6 shows the **interactions between Resource Manager and Application Master**.

Upon **receiving an application submission context from a client, the RM finds an available container meeting the resource requirements for running the AM for the application.**

On finding a suitable container, the RM contacts the NM for the container to start the AM process on its node. When the AM is launched it registers itself with the RM.

The **registration process consists of handshaking that conveys information such as the RPC port that the AM will be listening on, the tracking URL for monitoring the application's status and progress, etc.**

The **registration response from the RM contains information for the AM that is used in calculating and requesting any resource requests for the application's individual tasks** (such as minimum and maximum resource capabilities for the cluster).

The **AM relays heartbeat and progress information to the RM. The AM sends resource allocation requests to the RM that contains a list of requested containers, and may also contain a list of released containers by the AM**.

Upon **receiving the allocation request, the scheduler component of the RM computes a list of containers that satisfy the request and sends back an allocation response.**

Upon **receiving the resource list, the AM contacts the associated NMs for starting the containers. When the job finishes, the AM sends a Finish Application message to the RM**

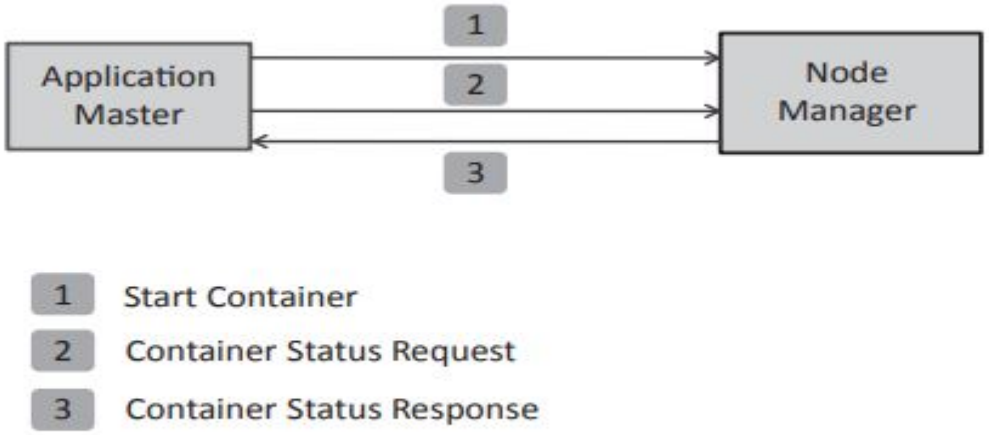


Figure 7.7: Application Master - Node Manager interaction

Figure 7.7 shows the **interactions between the an Application Master and the Node Manager.**

Based on the **resource list received from the RM, the AM requests the hosting NM for each container to start the container.**

The **AM can request and receive a container status report from the Node Manager**.

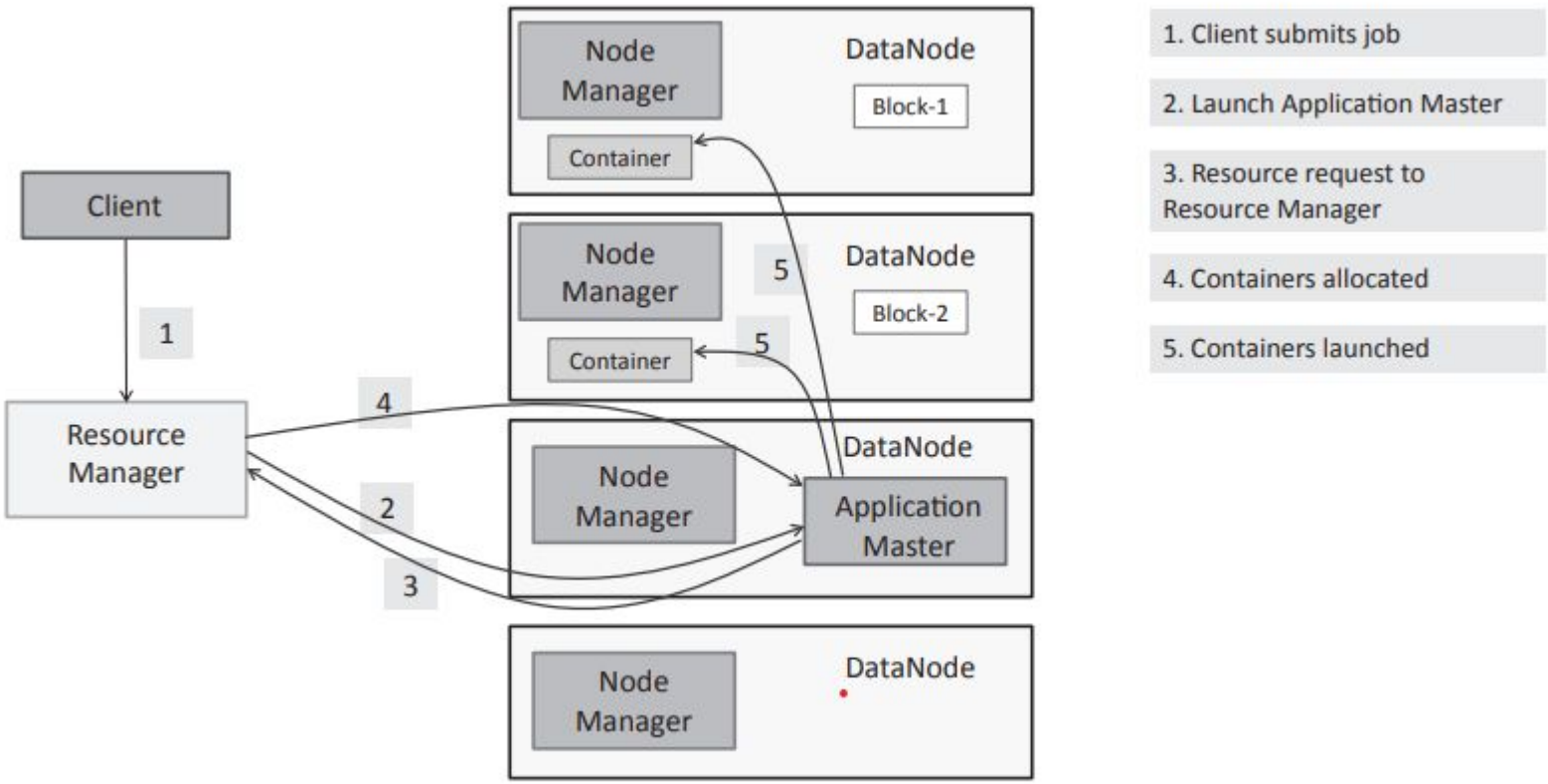


Figure 7.8: MapReduce job execution within a YARN cluster

Figure 7.8 shows the **MapReduce job execution within a YARN cluster.**

Hadoop Schedulers

The scheduler is **a pluggable component in Hadoop that allows it to support different scheduling algorithms.**

The pluggable scheduler framework provides the flexibility to support a variety of workloads with varying priority and performance constraints.

The Hadoop scheduling algorithms are described as follows

FIFO

FIFO scheduler maintains a work queue in which the jobs are queued.

The scheduler pulls jobs in first-in first-out manner (oldest job first) for scheduling.

There is no concept of priority or size of the job in FIFO scheduler.

Fair Scheduler

The Fair Scheduler **allocates resources evenly between multiple jobs and also provides capacity guarantees.**

Fair Scheduler **assigns resources to jobs such that each job gets an equal share of the available resources on average over time.**

Fair Scheduler lets **short jobs finish in reasonable time while not starving long jobs.**

Tasks slots that are free are assigned to the new jobs, so that each job gets roughly the same amount of CPU time.

The Fair Scheduler **maintains a set of pools into which jobs are placed. Each pool has a guaranteed capacity.**

When there are **multiple jobs in the pools, each pool gets at least as many task slots as guaranteed.**

Each **pool receives at least the minimum share**. When a pool does not require the guaranteed share the excess capacity is split between other jobs.

This lets the scheduler guarantee capacity for pools while utilizing resources efficiently when these pools don't contain jobs.

The Fair Scheduler keeps track of the compute time received by each job.

The scheduler computes periodically the difference between the computing time received by each job and the time it should have received in ideal scheduling.

The job which has the highest deficit of the compute time received is scheduled next.

This ensures that over time, each job gets its fair share of compute time.
Fair scheduler is useful when a small or large Hadoop cluster is shared between multiple groups of users in an organization.

Capacity Scheduler.

In Capacity Scheduler, **multiple named queues are defined, each with a configurable number of map and reduce slots.**

Each queue is also assigned a guaranteed capacity.

The **Capacity Scheduler gives each queue its capacity when it contains jobs, and shares any unused capacity between the queues.**

Within each queue **FIFO scheduling with priority is used**.

For fairness, it is **possible to place a limit on the percentage of running tasks per user, so that users share a cluster equally.**

A **wait time for each queue can be configured.**

When a queue is not scheduled for more than the wait time, it can preempt tasks of other queues to get its fair share.

When a TaskTracker has free slots, the Capacity Scheduler picks a queue for which the ratio of number of running slots to capacity is the lowest.

The scheduler then picks a job from the selected queue to run. Jobs are sorted based on when they're submitted and their priorities.

Jobs are considered in order, and a job is selected if its user is within the user-quota for the queue, i.e., the user is not already using queue resources above the defined limit.

The capacity scheduler is useful when a large Hadoop cluster is shared between with multiple clients and different types and priorities of jobs.

Though the capacity scheduler ensures fairness by maintaining a set of queues and providing guaranteed capacity to each queue, it does not provide any timing guarantees and, therefore, it may be ill-equipped for real-time jobs.

Word Count:
Map step: mapper.py

```
#!/usr/bin/env python
"""mapper.py"""

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Reduce step: reducer.py

```
from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Reference: Writing An Hadoop MapReduce Program In Python (michael-noll.com)

Batch Analysis of Sensor Data

Figure 7.9 shows a Hadoop MapReduce workflow for batch analysis of weather data.

Batch analysis is done to **aggregate data** (such as computing mean, maximum, and minimum) on various timescales.

For this example, we will **assume that we have a data collector which retrieves the sensor data collected in the cloud database and creates a raw data file in a form suitable for processing by Hadoop.**

The raw data file consists of the **raw sensor readings along with the timestamps** as shown below:

"2015-04-29 10:15:32",38,42,34,5
:
"2015-04-30 10:15:32",87,48,21,4

Box 7.1 shows the map program for the batch analysis of sensor data.

The **map program reads the data from standard input (stdin) and splits the data into the timestamp and individual sensor readings.**

The **map program emits key-value pairs where the key is a portion of the timestamp** (that depends on the timescale on which the data is to be aggregated), and the **value is a comma separated string of sensor readings**

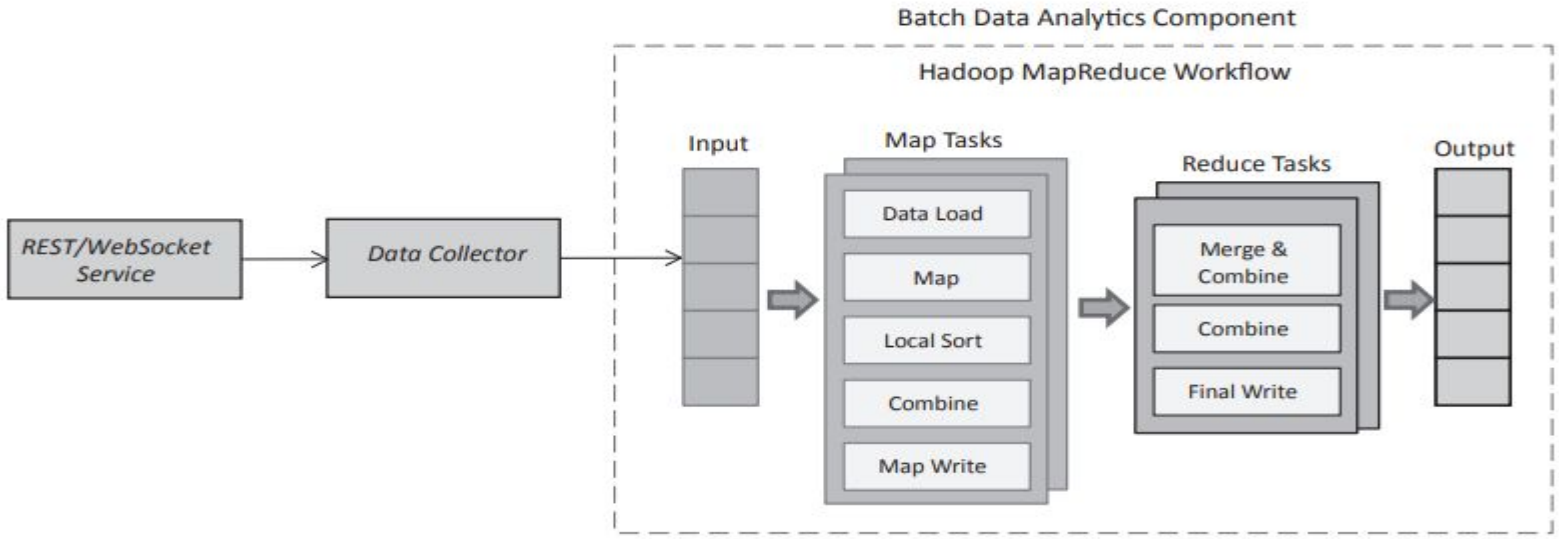


Figure 7.9: Using Hadoop MapReduce for batch analysis of sensor data

Box 7.1: Map program - mapper.py

```
#!/usr/bin/env python
import sys
#Calculates mean temperature, humidity, light and CO2
# Input data format:
#"2014-04-29 10:15:32",37,44,31,6
#Output:
#"2014-04-29 10:15 [48.75, 31.25, 29.0, 16.5]"
#Input comes from STDIN (standard input)

for line in sys.stdin:
# remove leading and trailing whitespace
line = line.strip()
data = line.split(‘,’)
l=len(data)
#For aggregation by minute
key=str(data[0][0:17])
value=data[1]+‘,’+data[2]+‘,’+data[3]+‘,’+data[4]
print ‘%s \t%s’ % (key, value)
```

Box 7.2 shows the reduce program for the batch analysis of sensor data.

The key-value pairs emitted by the map program are **shuffled to the reducer and grouped by the key.**

The **reducer reads the key-value pairs grouped by the same key from standard input and computes the means of temperature, humidity, light and CO readings**.

Box 7.2: Reduce program - reducer.py

```
#!/usr/bin/env python
from operator import itemgetter
import sys
import numpy as np
current_key = None
current_vals_list = []
word = None
#Input comes from STDIN
for line in sys.stdin:
# remove leading and trailing whitespace
line = line.strip()

#Parse the input from mapper
key, values = line.split('\t', 1)
list_of_values = values.split(',')

#Convert to list of strings to list of int
list_of_values = [int(i) for i in list_of_values]
if current_key == key:
current_vals_list.append(list_of_values)
else:
if current_key:
l = len(current_vals_list)+ 1
b = np.array(current_vals_list)
meanval = [np.mean(b[0:l,0]),np.mean(b[0:l,1]),
np.mean(b[0:l,2]), np.mean(b[0:l,3])]
print '%s\t%s' % (current_key, str(meanval))

current_vals_list = []
current_vals_list.append(list_of_values)
current_key = key

#Output the last key if needed
if current_key == key:
l = len(current_vals_list)+ 1
b = np.array(current_vals_list)
meanval = [np.mean(b[0:l,0]),np.mean(b[0:l,1]),
np.mean(b[0:l,2]), np.mean(b[0:l,3])]
print '%s\t%s' % (current_key, str(meanval))
```

Batch Analysis of N-Gram Dataset

Let us look at another example of MapReduce to analyze Google N-Gram dataset , which is a freely-available collection of n-grams (fixed size tuples of words) extracted from the Google Books corpus.

The n specifies the number of elements in the tuple, so for example, a 5-gram contains five words.

The n-grams in this dataset were produced by passing a sliding window over the text of books and outputting a record for each new token.
For example, for the line – 'Python is a
high level language', The 2-grams (or bigrams) will be:
(Python, is)
(is, a)
(a, high)
(high, level)
(level, language)
Each row of data contains:
1) n-gram itself
2) year in which the n-gram appeared
3) number of times the n-gram appeared in the books from the corresponding year (count)
4) number of pages on which the n-gram appeared in this year (page-count)
5) number of distinct books in which the n-gram appeared in this year (book count

Example (5-gram): analysis is often described as 1991 1 1 1

Interpretation of the 5-gram: In 1991, the phrase "analysis is often described as" occurred one time (that's the first 1), and on one page (the second 1), and in one book (the third 1).

Box 7.4 shows MapReduce program that calculates the most popular bigram (2-gram) of all time in the dataset.

This example uses the MRJob Python library which lets you write MapReduce jobs in Python and run them on several platforms including local machine, Hadoop cluster and Amazon Elastic MapReduce (EMR).
MRJob can be installed as follow

```
 #Installing MRJob
sudo apt-get install git
git clone https://github.com/Yelp/mrjob.git
cd mrjob
python setup.py install
```

MapReduce program that calculates the most popular bigram of all

```
time - mr.py
from mrjob.job import MRJob
class MyMRJob(MRJob):
def mapper(self, _, line):
data=line.split('\t')
ngram = data[0].strip()
year = data[1].strip()
count = data[2].strip()
pages = data[3].strip()
books = data[4].strip()
#Emit key-value pairs where key is ngram+year and value is
count
yield ngram+year, int(count)
def reducer(self, key, list_of_values):
# Send all (count, ngram+year) pairs to the same
reducer.
# So we can easily use Python's max() function.
yield None, (sum(list_of_values),key)
def reducer2(self, _, list_of_values):
# Reducer-2 get input tuples as follows:
# None, [(212, cloud computing 2006), (156,
mobile phones 2003)]
# max function will yield tuple with max value of
the count
yield max(list_of_values)
def steps(self):
return [self.mr(mapper=self.mapper,
reducer=self.reducer),
self.mr(reducer=self.reducer2)]
if __name__ == '__main__':
MyMRJob.run()
```

The example in Box 7.4 implements a class MyMRJob that defines mapper and reducer functions.

In this example, we have one map-reduce pair and another reduce function which is chained to the output of the first reducer. When the program is run, the mapper function is invoked for each line of the input file

Find top-N words with MapReduc

```
from mrjob.job import MRJob
class MyMRJob(MRJob):
def mapper(self, _, line):
line = line.strip()
words = line.split()
for word in words:
yield (word, 1)
def reducer(self, key, list_of_values):
word = key
total_count = sum(list_of_values)
yield None, (total_count, word)
def reducer2(self, _, list_of_values):
N = 3
list_of_values = sorted(list(list_of_values), reverse=True)
return list_of_values[:N]
def steps(self):
return [self.mr(mapper=self.mapper,
reducer=self.reducer), self.mr(reducer=self.reducer2)]
if __name__ == '__main__':
MyMRJob.run()
```