HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

Chapter 2

Hadoop Ecosystem

Content

- Apache Hadoop
- Hadoop File System (HDFS) •

MapReduce data processing

paradigm • Other components in the Hadoop ecosystem

Goals of Hadoop

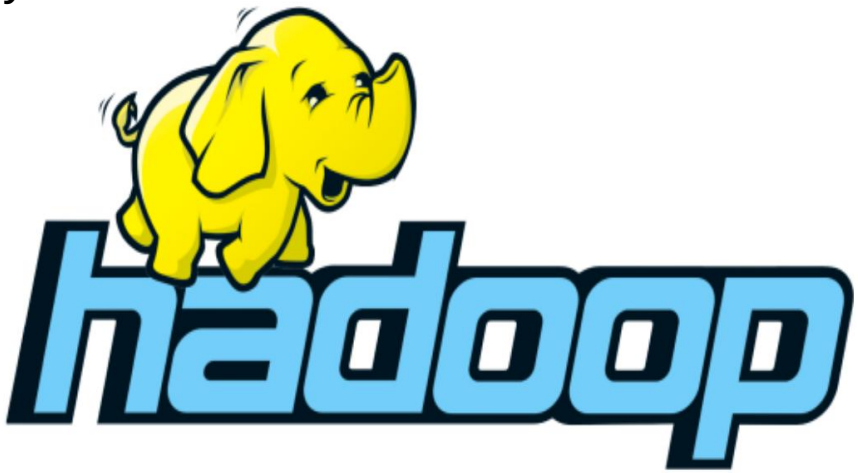
- Main objectives •

  Open, reliable data storage • Powerful

  data processing • Efficient visualization

- With challenge

  - Slow storage devices, unreliable computers, and distributed parallel programming are not easy



Introduction to Apache Hadoop

- **Open, cost-effective data storage and processing**
  - Distributed data processing with simpler, more user-friendly programming models like

  MapReduce • Hadoop is designed to scale through scale-out techniques, increasing the number

  of servers • Designed to operate on common hardware, with the ability to withstand hardware failures
- Inspired by Google's data architecture

Main components of Hadoop

- Data storage: Hadoop distributed file system (HDFS) • Data

processing: MapReduce framework • System

utilities: • Hadoop Common:

Common utilities that support Hadoop components.
• Hadoop YARN: A

framework for resource management and scheduling in Hadoop clusters.

Hadoop solves the scalability problem

- Design for “dispersion” from the start
  - Hadoop is designed by default to be deployed on clusters
    Servers
- The servers that participate in a cluster are called
  Nodes
  - Each node participates in both storage and computation roles.
    maths
- Hadoop scaled using scale-out techniques
  - Can scale Hadoop clusters to tens of thousands of nodes

Hadoop solves the fault tolerance problem

- With deployment on common server clusters
  - Hardware failure is a daily occurrence, not an exception.

    rate

  - Hadoop is fault tolerant through “redundancy” techniques
- Files in HDFS are fragmented and replicated across nodes in the cluster
  - If a node fails, the

    data associated with that node is replicated across other nodes

- Data processing work is fragmented into independent tasks • Each task

  processes a portion

  of the input data • Tasks are executed in

  parallel with other tasks • Failed tasks will be rescheduled to

  execute on another node • Hadoop system is designed so

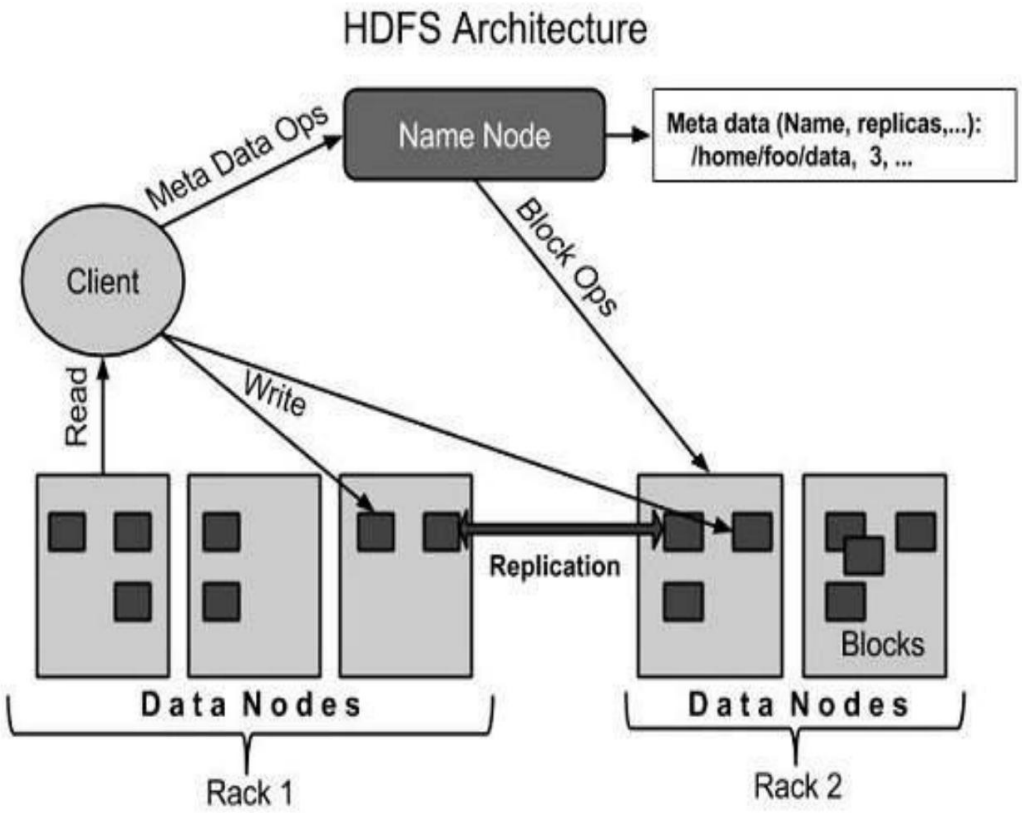
that errors occurring in the system are handled automatically, without affecting the applications above

HDFS Overview

- HDFS provides reliable and cost-effective storage for large volumes of data
- Optimized for large files (from several hundred MB up to several TB)
- HDFS has a hierarchical directory tree like UNIX (eg, /hust/soict/hello.txt)
  - Supports user control and authorization mechanisms like UNIX
- Different from UNIX file system
  - Only supports append data to the end of the file (APPEND) operation
  - Write once and read many times

Architecture of HDFS

- Master/Slave architecture
- HDFS master: named node
  - Namespace management and metadata that maps files to chunks locations
  - Monitor data nodes
- HDFS slave: data node
  - Direct I/O operations on chunks



Core design principles of HDFS

- I/O pattern
  - Only append ÿ reduce concurrency control costs
- Data distribution •

  File is divided into large chunks (64 MB)

  ÿ Reduce metadata size

  ÿ Reduce data transmission costs
- Data replication
  - Each chunk is normally copied into 3 copies
- Fault tolerance

  mechanism • Data node: uses replication

  mechanism • Name node
    - Use Secondary Name Node • SNN

    asks data nodes at startup instead of having to perform synchronization mechanism complex with primary NN

MapReduce data processing model

- MapReduce is the default data processing model in Hadoop
- MapReduce is not a programming language, proposed by Google •

Characteristics of MapReduce •

Simplicity

- Flexibility
- Scalability

A MR job = {Isolated Tasks}n

- Each MapReduce program is a job that is decomposed into many independent tasks and these tasks are distributed on different nodes of the cluster for execution
  - Each task is

executed independently of other tasks to achieve scalability

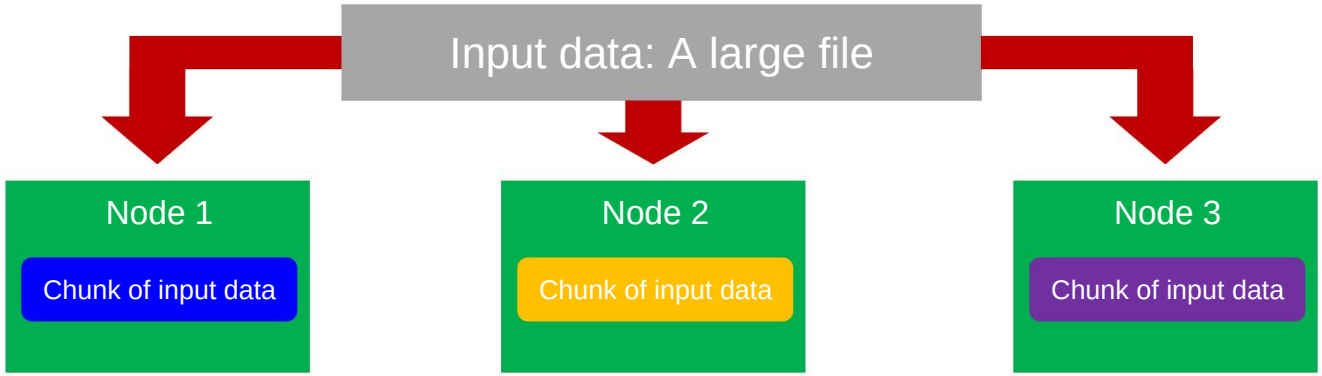
  - Reduce communication

  between server nodes • Avoid having to

  implement synchronization mechanisms between tasks

Data for MapReduce

- MapReduce in Hadoop environment typically works with data that is available on HDFS
- When executed, the MapReduce program code is sent to nodes that already have the corresponding data.

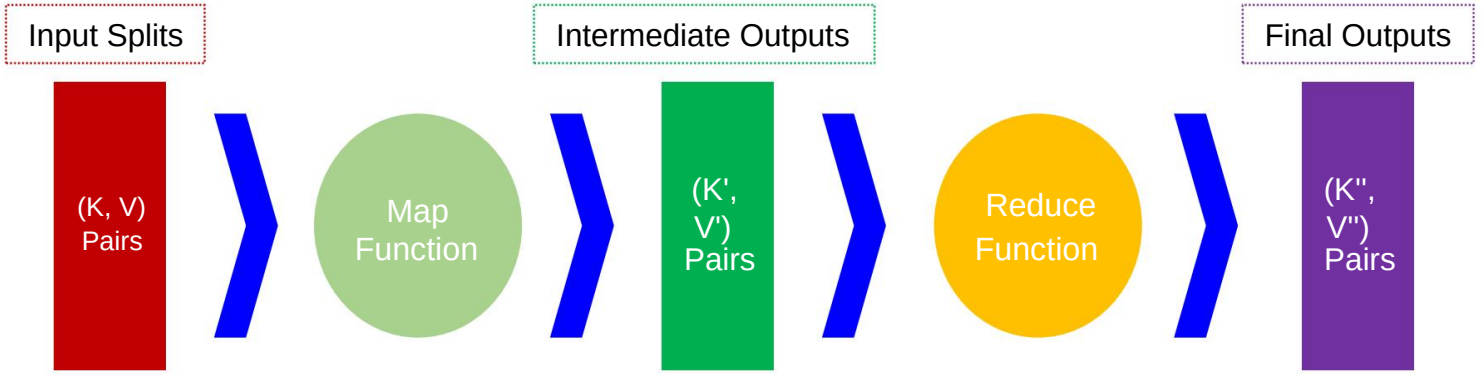


MapReduce Program

- Programming with MapReduce requires installing 2 functions Map and Reduce
  - These two functions are executed by the Mapper and Reducer processes respectively.

response

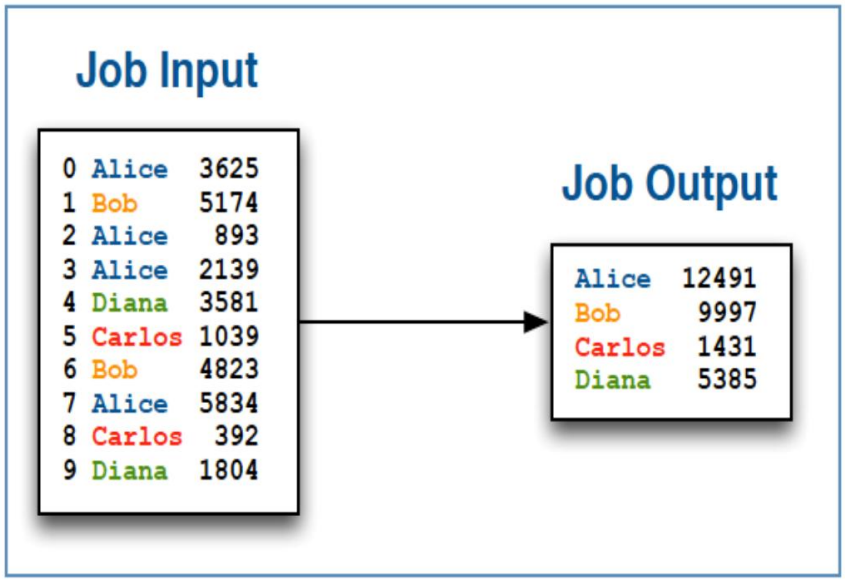
- In MapReduce programs, data is viewed as pairs of
  key – value
- The Map and Reduce functions take input and return output pairs (key – value)



MapReduce Example

- Input: text file containing information about order ID, employee name, and sale

amount • Output: Sales by employee

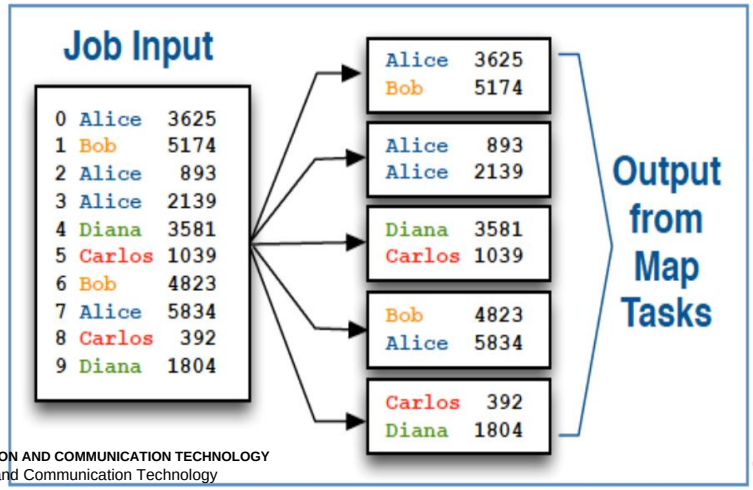


Map Step

- Input data is processed by multiple independent Mapping tasks
  - The number of Mapping tasks is determined by the amount of input data (~ number of chunks)
  - Each Mapping task processes a chunk of the original data block • For each Mapping

task, the Mapper processes each input record in turn

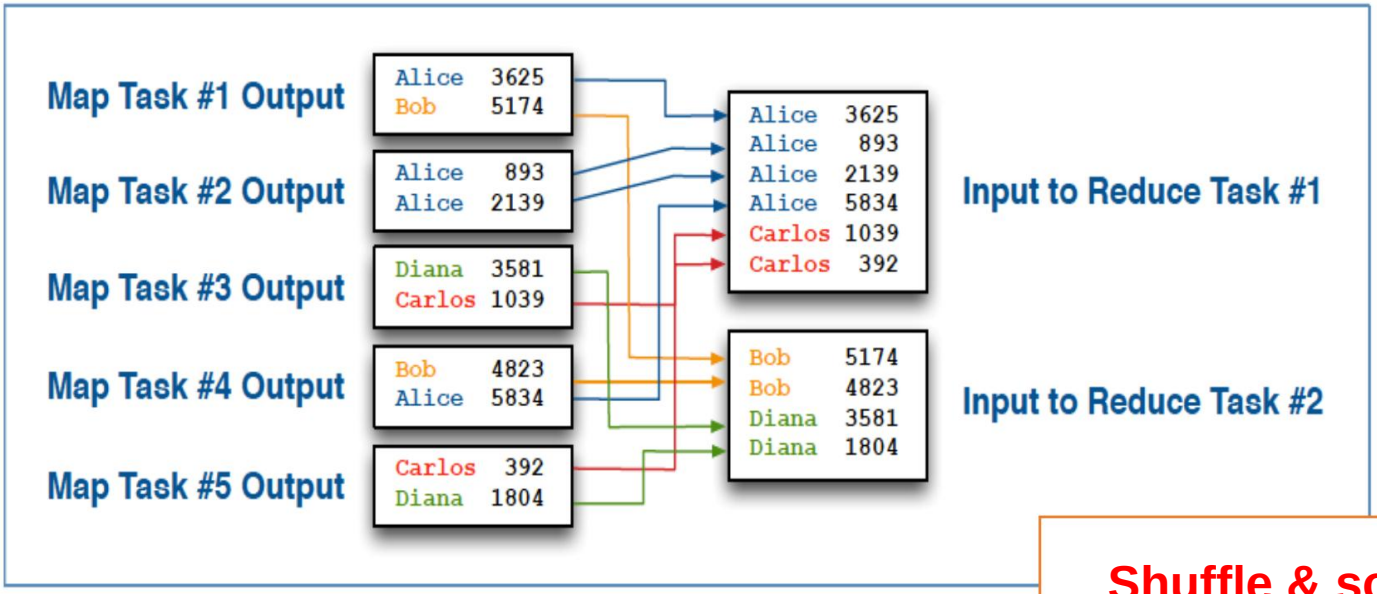
  - For each input record (key-value), Mapper produces 0 or more output records (intermediate key – value)
- In this example, the Mapping task simply reads each line of text and outputs the employee name and corresponding sales



Map phase

Shuffle & sort step

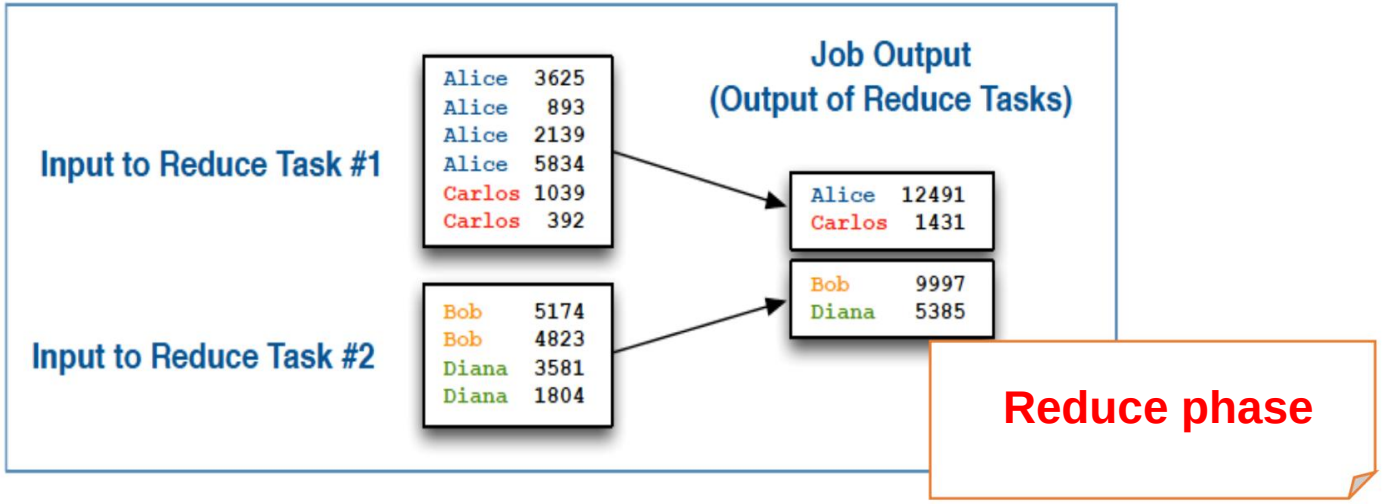
- Hadoop automatically sorts and aggregates the output of Mappers by partitions • Each partition is an input to a Reducer



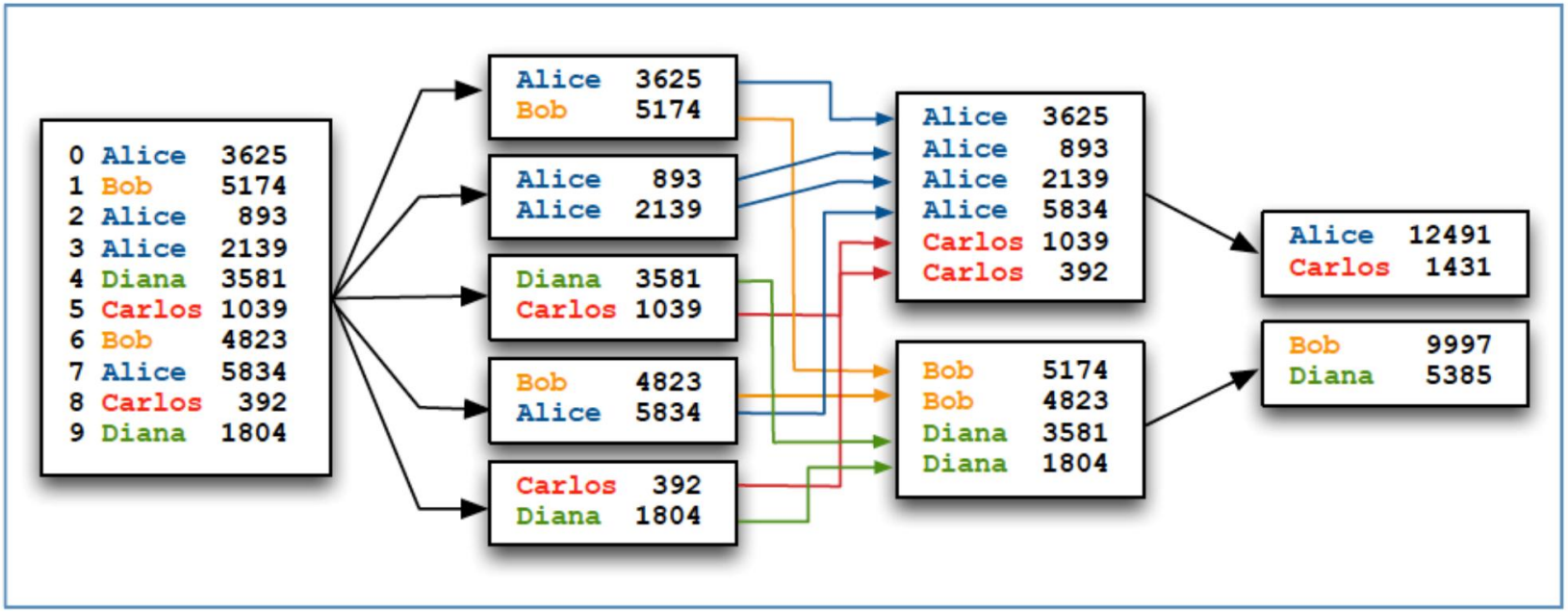
Shuffle & sort phase

Reduce Step

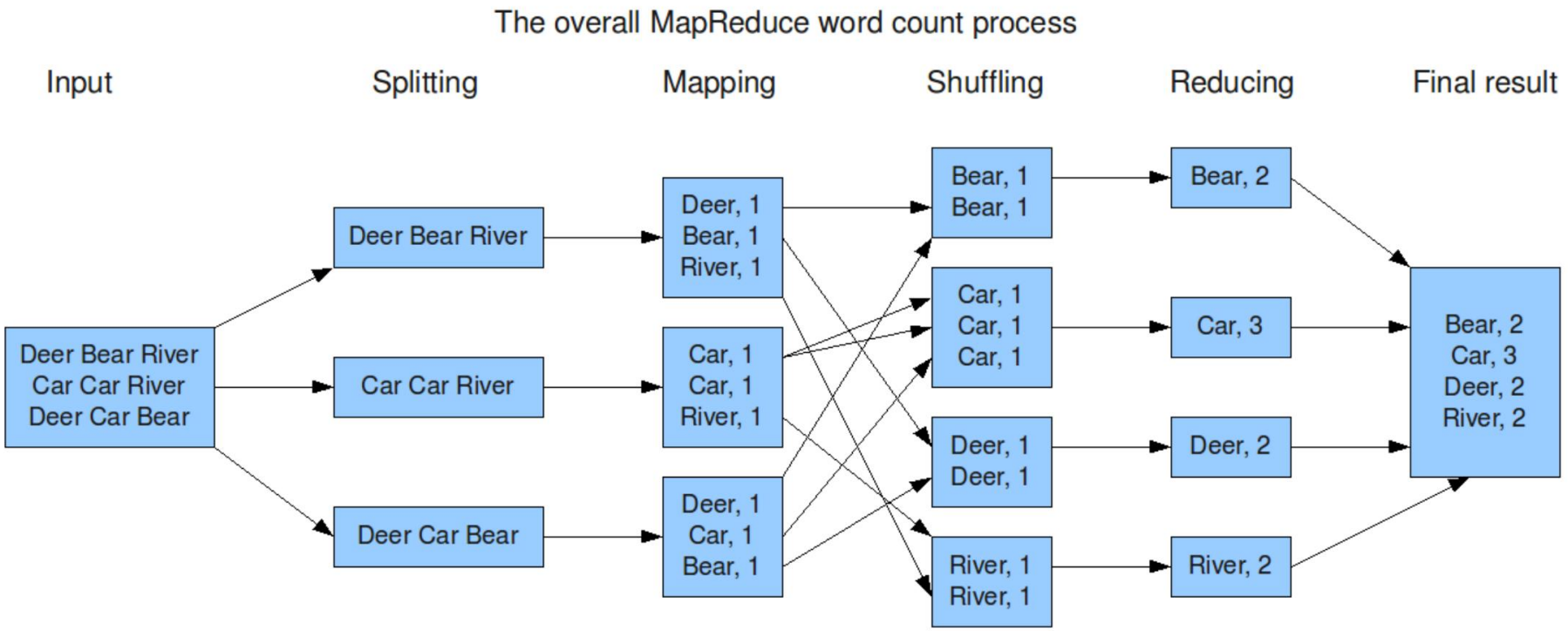
- Reducer receives input data from shuffle & sort step
  - All key – value records corresponding to a key are processed by a single Reducer
  - Similar to the Map step, the Reducer processes each key in turn, each time with all corresponding values
- In the example, the reduce function simply calculates the total sales for each employee, the output is the key – value pairs corresponding to the employee name – total sales



Data Stream



Data flow with Word Count problem



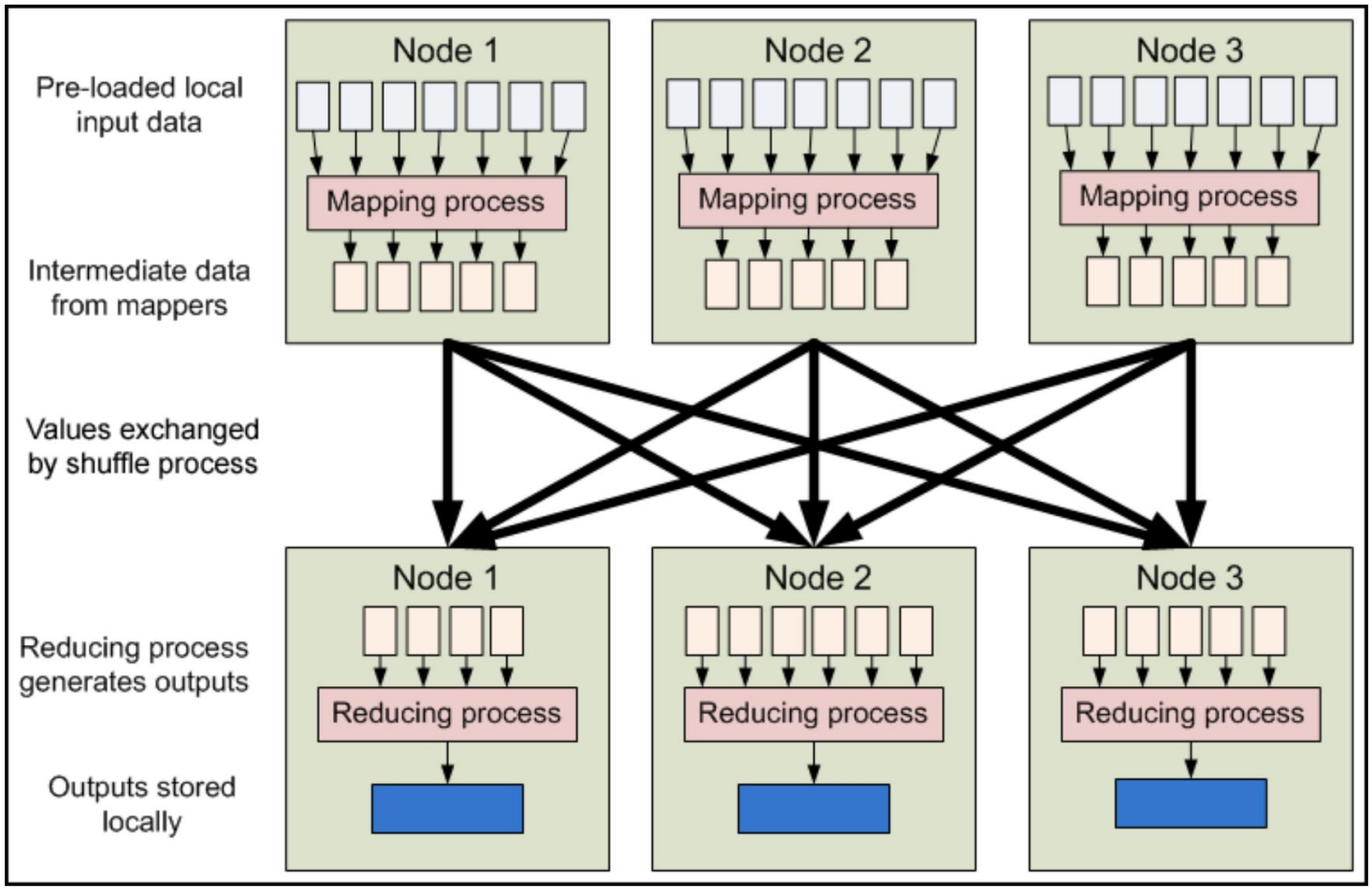
Practical Word Count Program (1)

```
9  import org.apache.hadoop.mapreduce.Job;
10 import org.apache.hadoop.mapreduce.Mapper;
11 import org.apache.hadoop.mapreduce.Reducer;
12 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.util.GenericOptionsParser;
15
16
17
18
19 public class WordCount {
20 public static void main(String [] args) throws Exception
21 {
22 Configuration c=new Configuration();
23 String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
24 Path input=new Path(files[0]);
25 Path output=new Path(files[1]);
26 Job j=new Job(c,"wordcount");
27 j.setJarByClass(WordCount.class);
28 j.setMapperClass(MapForWordCount.class);
29 j.setReducerClass(ReduceForWordCount.class);
30 j.setOutputKeyClass(Text.class);
31 j.setOutputValueClass(IntWritable.class);
32 FileInputFormat.addInputPath(j, input);
33 FileOutputFormat.setOutputPath(j, output);
34 System.exit(j.waitForCompletion(true)?0:1);
35 }
```

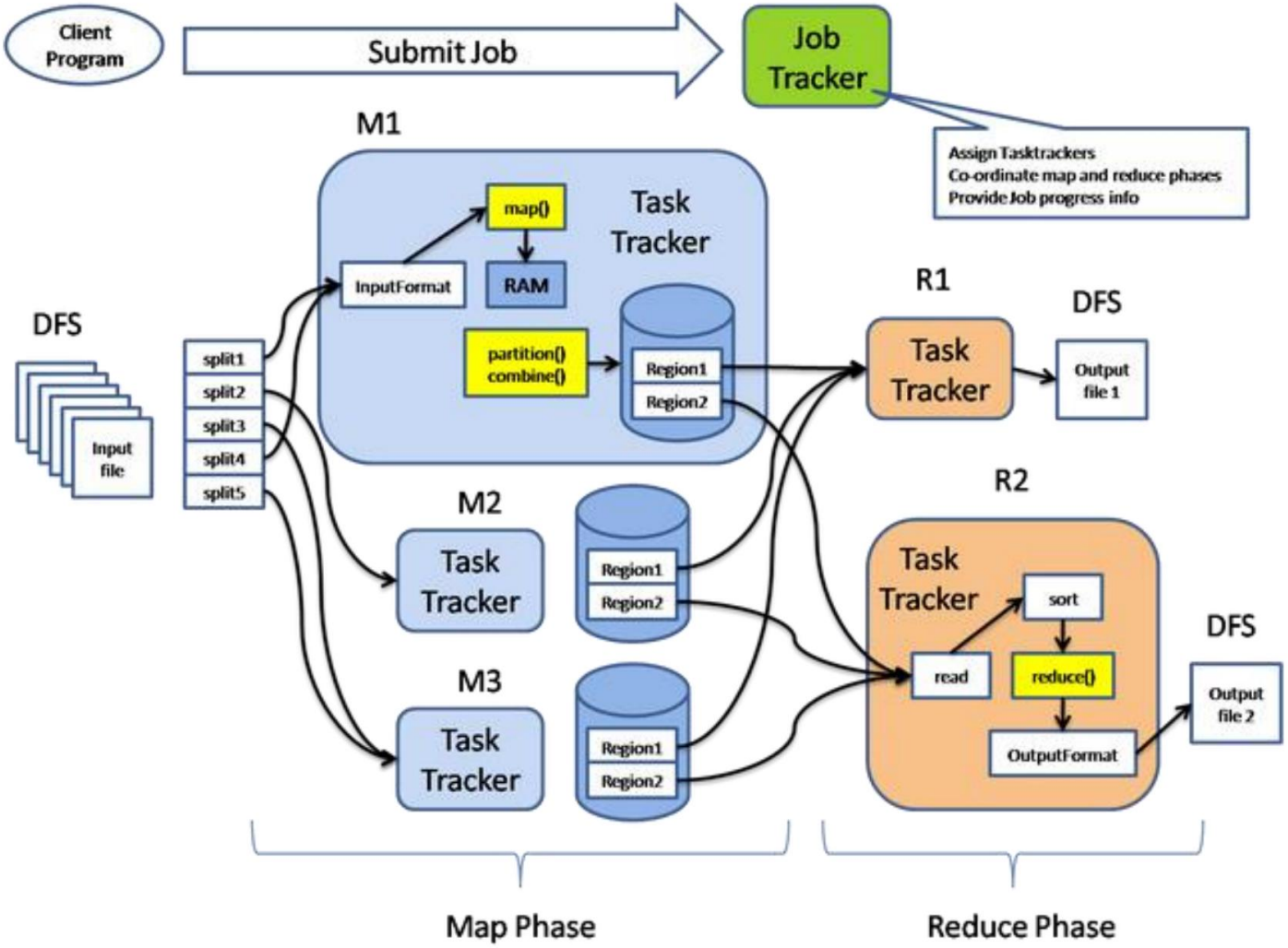
Practical Word Count Program (2)

```
36 public static class MapForWordCount extends Mapper<LongWritable, Text, Text, IntWritable>{
37 public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException
38 {
39 String line = value.toString();
40 String[] words=line.split(",");
41 for(String word: words )
42 {
43       Text outputKey = new Text(word.toUpperCase().trim());
44   IntWritable outputValue = new IntWritable(1);
45   con.write(outputKey, outputValue);
46 }
47 }
48 }
49
50 public static class ReduceForWordCount extends Reducer<Text, IntWritable, Text, IntWritable>
51 {
52 public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, InterruptedE
53 {
54 int sum = 0;
55    for(IntWritable value : values)
56    {
57    sum += value.get();
58    }
59    con.write(word, new IntWritable(sum));
60 }
```

MapReduce in a Distributed Environment



The role of Job tracker and Task tracker



Other components in the Hadoop ecosystem

- In addition to HDFS and MapReduce, the Hadoop ecosystem has many other systems and components serving •
  - Data analysis
  - • Data integration •
  - Flow management •
  - Etc.
- These components are not 'core Hadoop' but are part of the Hadoop ecosystem
  - • Mostly open source on Apache

Apache Pig

- Apache Pig provides a high-level data processing interface
  - Pig is especially good for Join and Transformation operations
- Pig compiler runs on the client machine
  - Convert PigLatin scripts into MapReduce jobs • Submit these jobs to the cluster

```
people = LOAD '/user/training/customers' AS (cust_id, name);
orders = LOAD '/user/training/orders' AS (ord_id, cust_id, cost);
groups = GROUP orders BY cust_id;
totals = FOREACH groups GENERATE group, SUM(orders.cost) AS t;
result = JOIN totals BY group, people BY cust_id;
DUMP result;
```

Apache Hive

- Also a high-level abstraction layer of MapReduce • Reduces development time • Provides HiveQL language: SQL-like language
- Hive compiler running on client machine
  - Convert HiveQL scripts into MapReduce jobs
  - Submit these jobs to the compute cluster


```
SELECT customers.cust_id, SUM(cost) AS total
  FROM customers
  JOIN orders
    ON customers.cust_id = orders.cust_id
GROUP BY customers.cust_id
ORDER BY total DESC;
```

Apache Hbase

- HBase is a distributed, extensible columnar database that stores data on HDFS
  - Considered as Hadoop's database management system
- Data is logically organized as tables, which contain a large number of rows and

  columns • Table size can be up to Terabytes, Petabytes
  - Tables can have thousands of

columns • Highly scalable, accommodating high-speed data write bandwidth •

  Supports hundreds of thousands of INSERT operations per second (/s)

- However, the functions are still very limited when compared with the system. Traditional QTCSDL • Is

  NoSQL: does not have a high-level query language like SQL • Must

  use API to scan/ put/ get/ data by key

Apache Sqoop

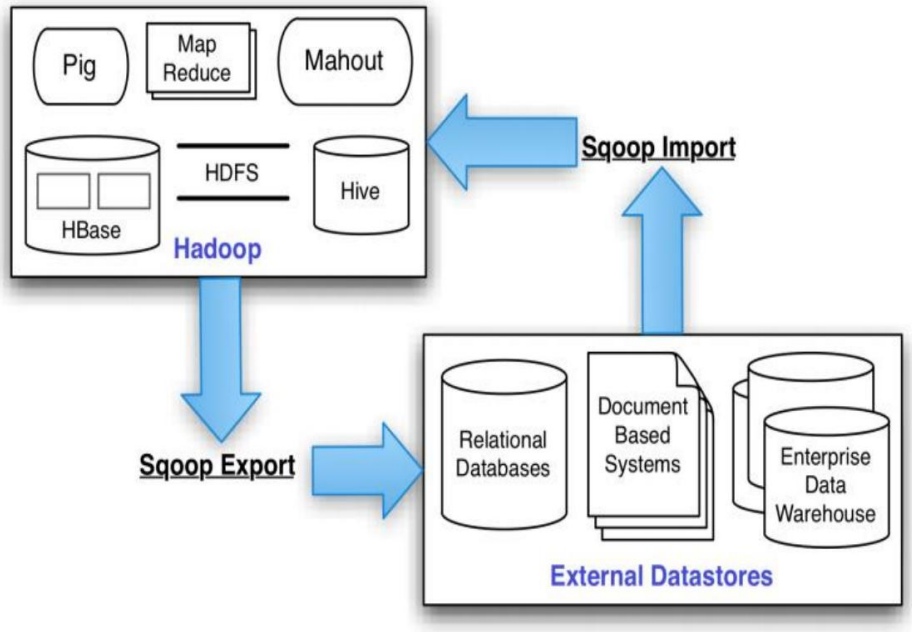
• Sqoop is a tool that allows for the
transfer of block-based data from
Apache Hadoop and structured
databases such as
Relational

database • Supports importing all tables,
one table or part of a table into HDFS •
Via Map only

or
MapReduce job

• The result is a folder in
HDFS contains text files that
separate fields by delimiter
characters (e.g. , or \t) •

Supports reverse data export
back from Hadoop to outside

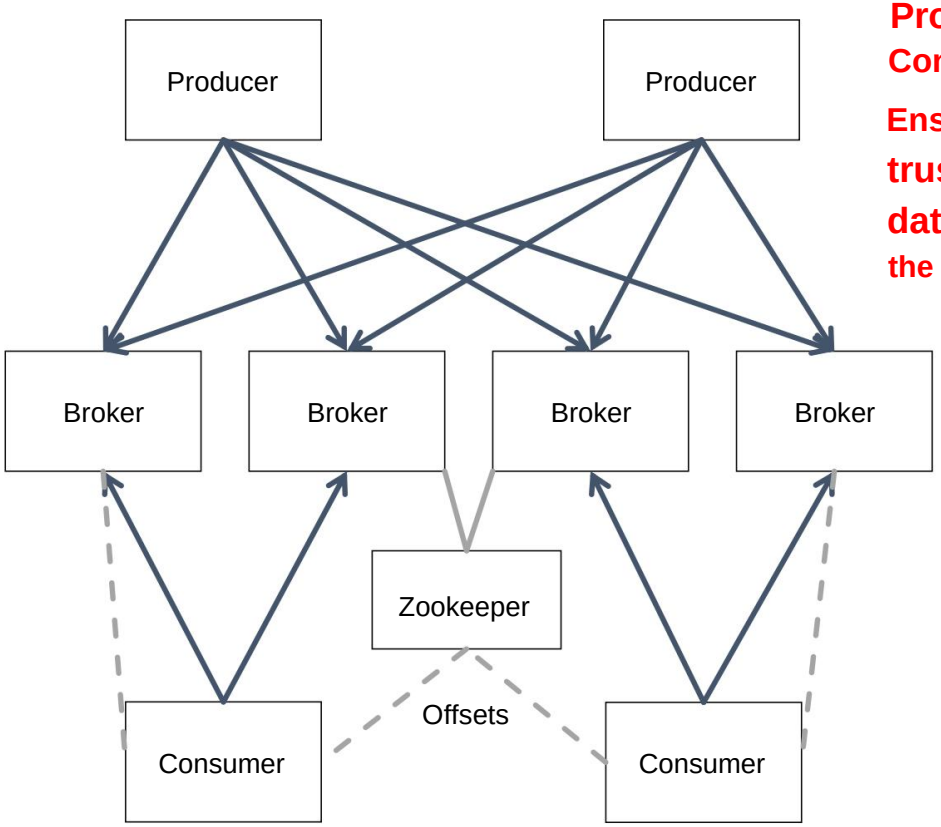


Apache Kafka

Producers

Kafka Cluster

Consumers



Producers don't need to know Consumers

Ensure flexibility and trust in the process data transfer between the parties

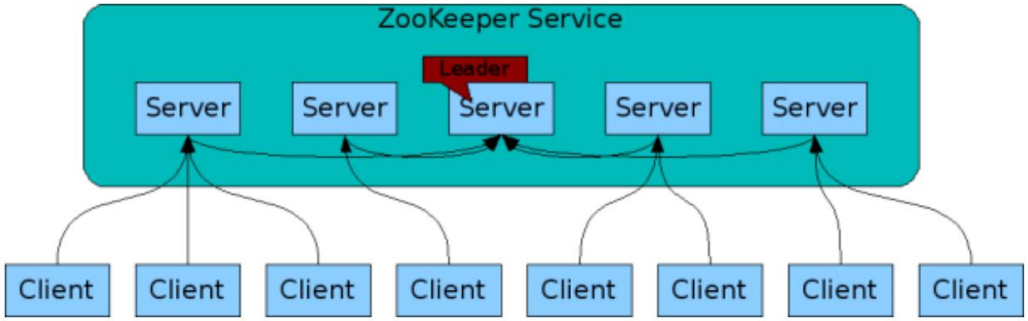
Kafka enables clean separation of components participating in a data stream

Apache Oozie

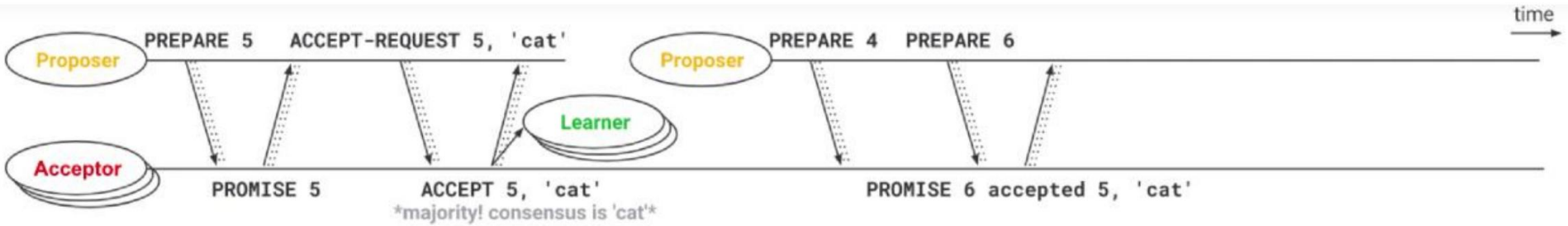
- Oozie is a workflow scheduling system to manage jobs running on the cluster Hadoop
- Oozie's workflow is a circular graph with Directed Acyclical Graphs (DAGs) of Work blocks
- Oozie supports a variety of jobs
  - Execute MapReduce jobs
  - Execute Pig or Hive scripts • Execute Java or Shell programs
  - Interact with data on HDFS • Run remote programs via SSH • Send and receive emails

Apache Zookeeper

- Apache ZooKeeper is a service that provides highly reliable distributed coordination functions • Server group membership management • Leader election • Dynamic configuration information management • System state monitoring
- These are core services, critical in distributed systems.



PAXOS algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE** IDp to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE IDp accepted IDa, value**.
No -> Reply with **PROMISE IDp**.

1 **If a majority of acceptors promise, no ID<IDp can make it through.**

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends **ACCEPT-REQUEST IDp, VALUE** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

⇨ **Acceptor** receives an **ACCEPT-REQUEST** message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT IDp, value**. Also send it to all **Learners**.

2 **If a majority of acceptors accept IDp, value, consensus is reached. Consensus is and will always be on value (not necessarily IDp).**

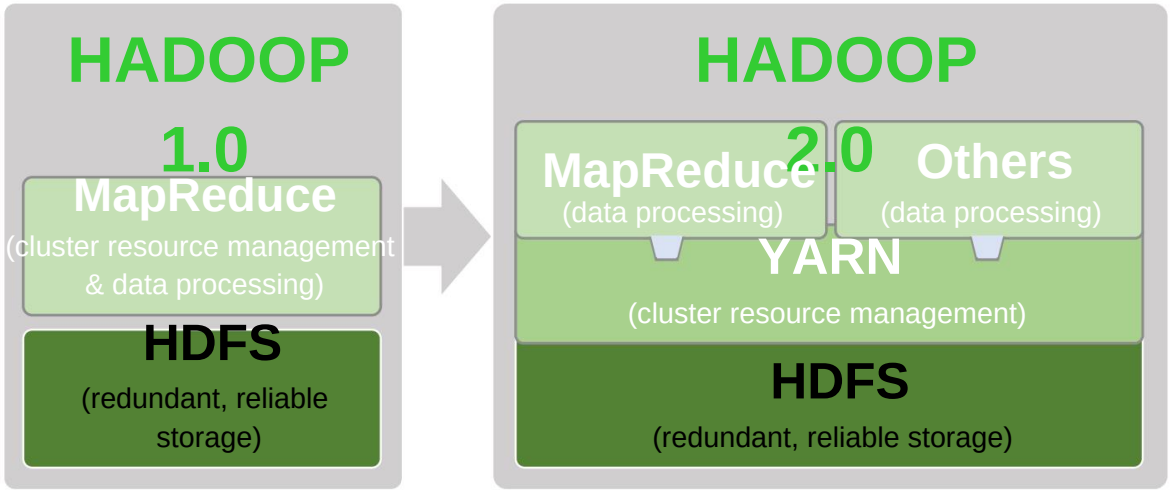
⇨ Proposer or Learner get **ACCEPT** messages for IDp, value:

3 **If a proposer/learner gets majority of accept for a specific IDp, they know that consensus has been reached on value (not IDp).**

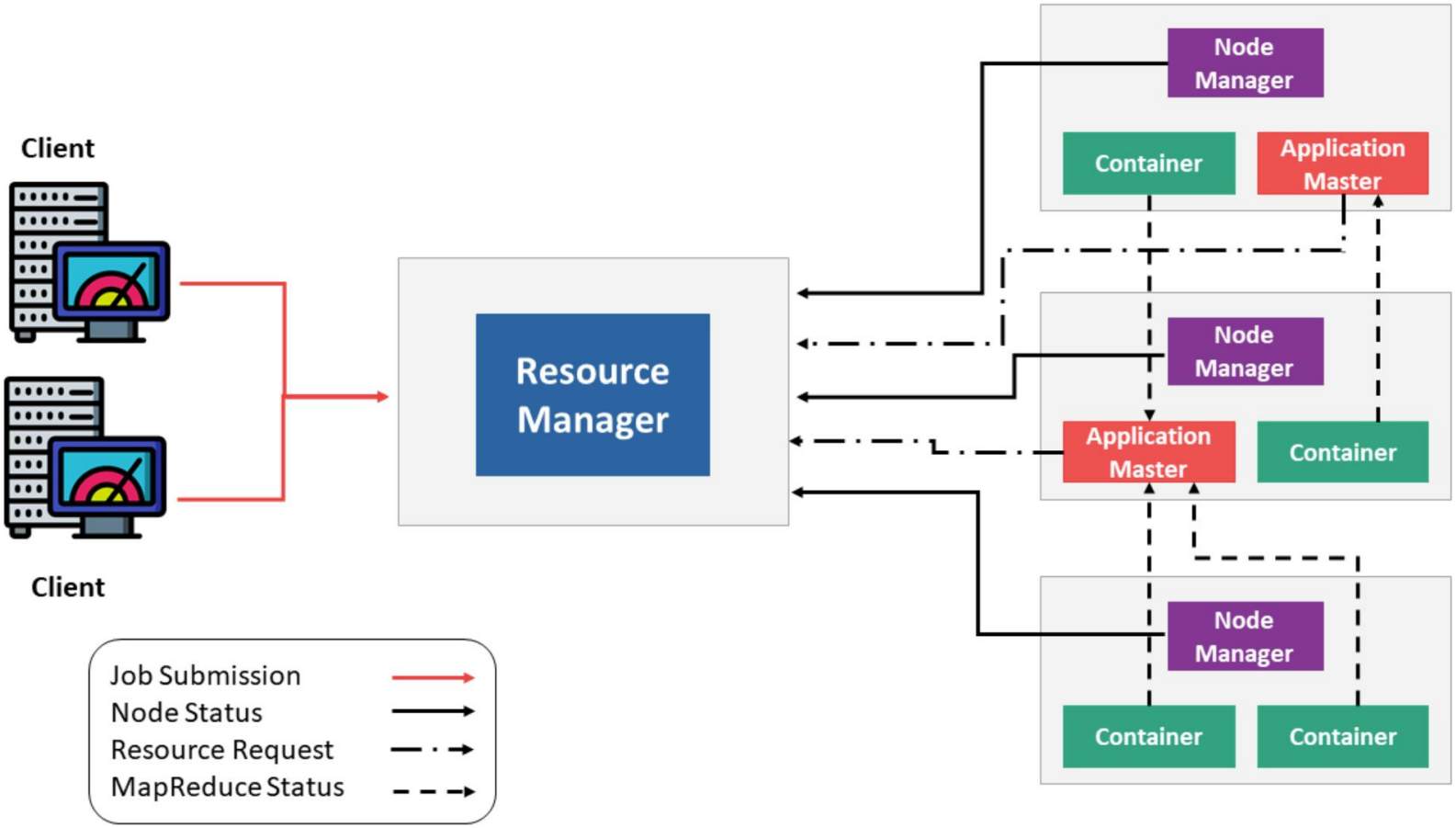
https://www.youtube.com/watch?v=d7nAGI_NZPk

YARN – Yet Another Resource Negotiator

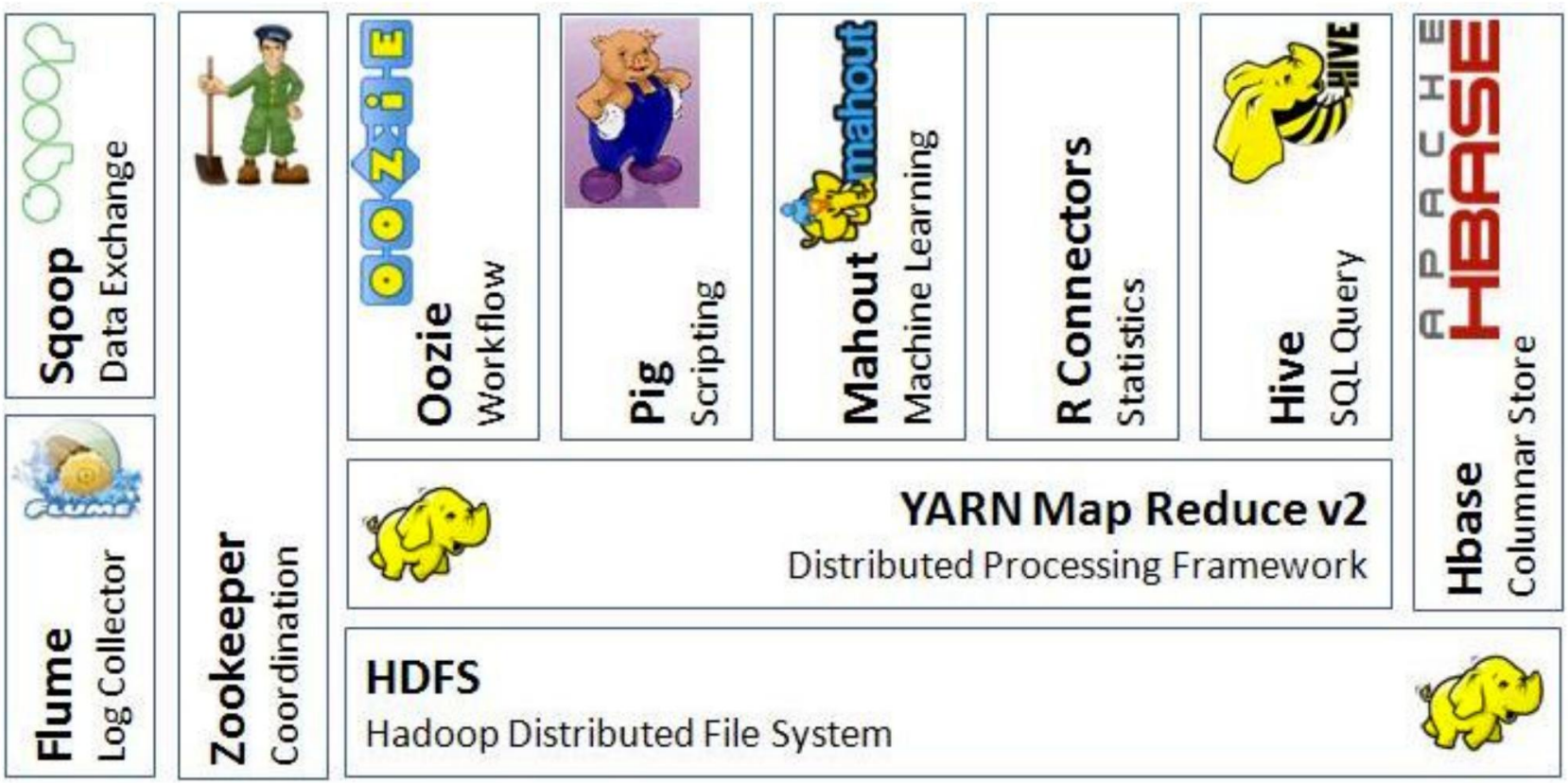
- Nodes have resources – memory and CPU cores
- YARN plays a role in allocating the appropriate amount of resources to applications when required
- YARN was introduced in Hadoop 2.0
  - Allows MapReduce and non-MapReduce to run together on the same cluster Hadoop
  - With MapReduce jobs, the role of the job tracker is performed by the application tracker



Example of allocation on YARN



The Big Picture of the Hadoop Ecosystem



Big data management platforms