Database for Enterprise

Module 4 – Distributed Databases

Introduction to Hadoop

Hadoop/MapReduce Computing Paradigm

Jixue Liu

In reference to slides by Mohamed Eltabakh

Learning Objectives

- Understand (at the conceptual level)
  - How data is distributed in Hadoop
  - How performance and fault tolerance are achieved
  - What type of data does it support
  - What is MapReduce and how it works
  - Advantages and disadvantages
- Apply map and reduce operations to word/color count applications on paper if map() and reduce() functions are given

Reading

- There are many readings. Below is one of them https://blog.matthewrathbone.com/2013/04/17/what-is-hadoop.html
- Book (not free): **Hadoop: The Definitive Guide by Tom White**
  https://www.amazon.com/gp/product/1449311520/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1449311520&linkCode=as2&tag=matratsblo-20

Large-Scale Data Analytics

- MapReduce computing paradigm (E.g., Hadoop) vs. Traditional database systems

vs.

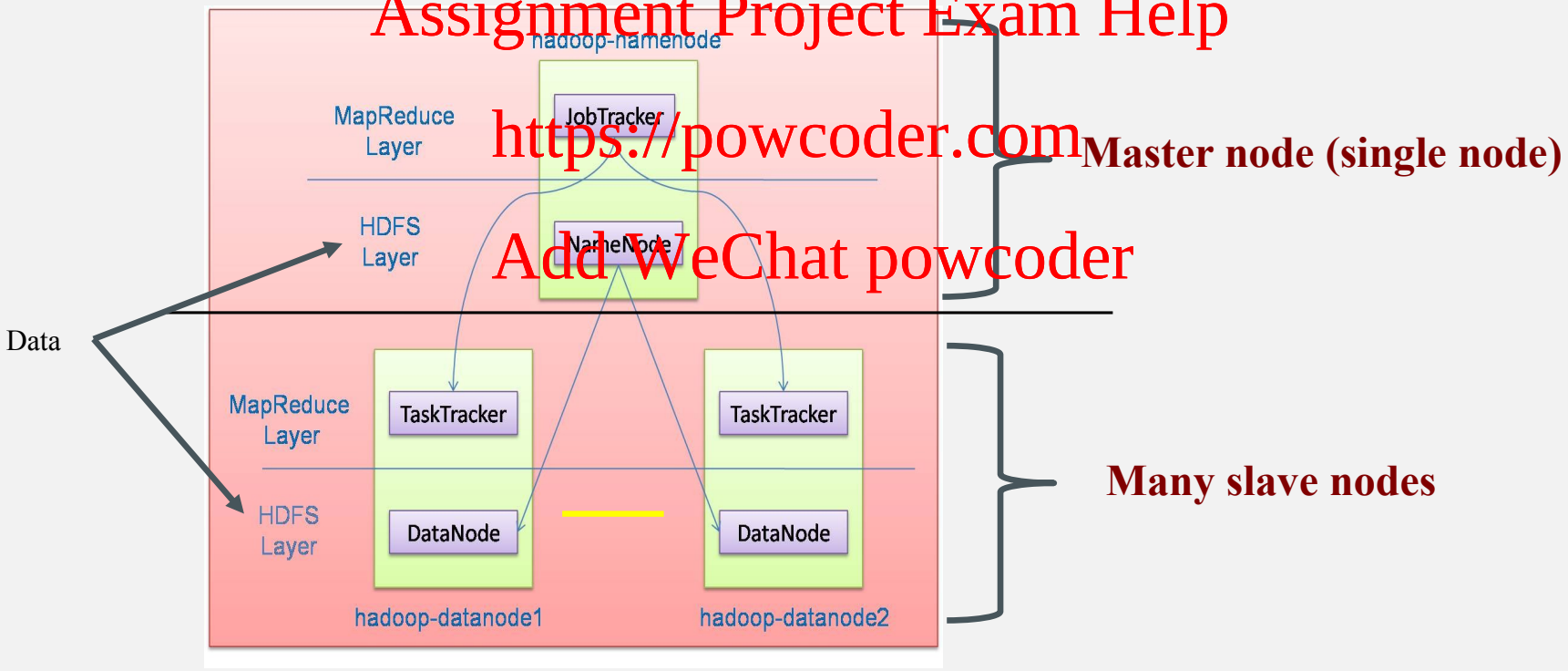
- **Many enterprises are turning to Hadoop**
  - Especially applications generating ***big data***
  - Web applications, social networks, scientific applications

What is Hadoop

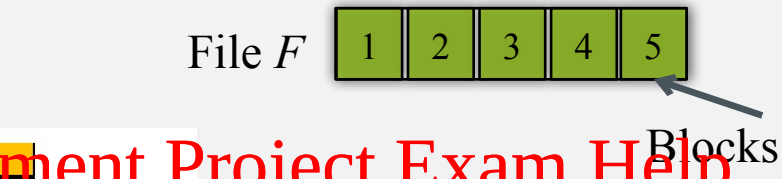
- Hadoop is a software framework for *distributed processing* of *large datasets* across *large clusters* of computers
  - ***Large datasets*** → Terabytes or petabytes of data
  - ***Large clusters*** → hundreds or thousands of nodes
- Hadoop is open-source implementation for Google ***MapReduce***
- Hadoop is based on a simple programming model called *MapReduce*
- Hadoop is based on a simple data model, *any data will fit*

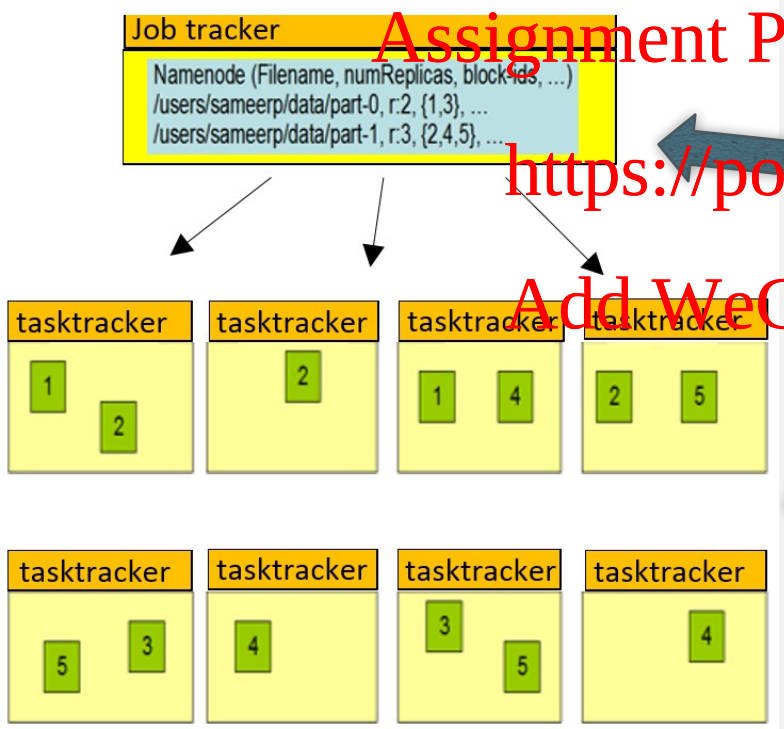
Hadoop Master/Slave Architecture

- Hadoop is designed as a *master-slave shared-nothing* architecture



Hadoop Distributed File System (HDFS)





Centralized namenode
- Maintains metadata info about files

Many datanode (1000s)
- Store the actual data
- Files are divided into blocks
- Each block is replicated *N* times (Default = 3)

Design Principles of Hadoop

- Need to process big data
- Need to parallelize computation across thousands of nodes
- **Commodity hardware**
  - Large number of low-end cheap machines working in parallel to solve a computing problem
- This is in contrast to **Parallel DBs**
  - Small number of high-end expensive machines

This cheap is not in normal sense.

Design Principles of Hadoop

- **Automatic parallelization & distribution**
  - Hidden from the end-user

Some distributed DB transparencies are supported automatically with costs

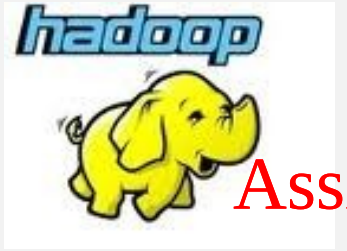
- **Fault tolerance and automatic recovery**
  - Nodes/tasks will fail and will recover automatically

Replication is supported automatically

- **Clean and simple programming abstraction**
  - Users only provide two functions “map” and “reduce”

The cost is that search by attributes is difficult if possible.

Why Hadoop is able to compete?



vs.

Scalability (petabytes of data, thousands of machines)

Flexibility in accepting all data formats (no schema)

Efficient and simple fault-tolerant mechanism

Commodity inexpensive hardware

Performance (tons of indexing, tuning, data organization tech.)

Features:
- Transaction management
- Provenance tracking
- ….

Hadoop is for big data

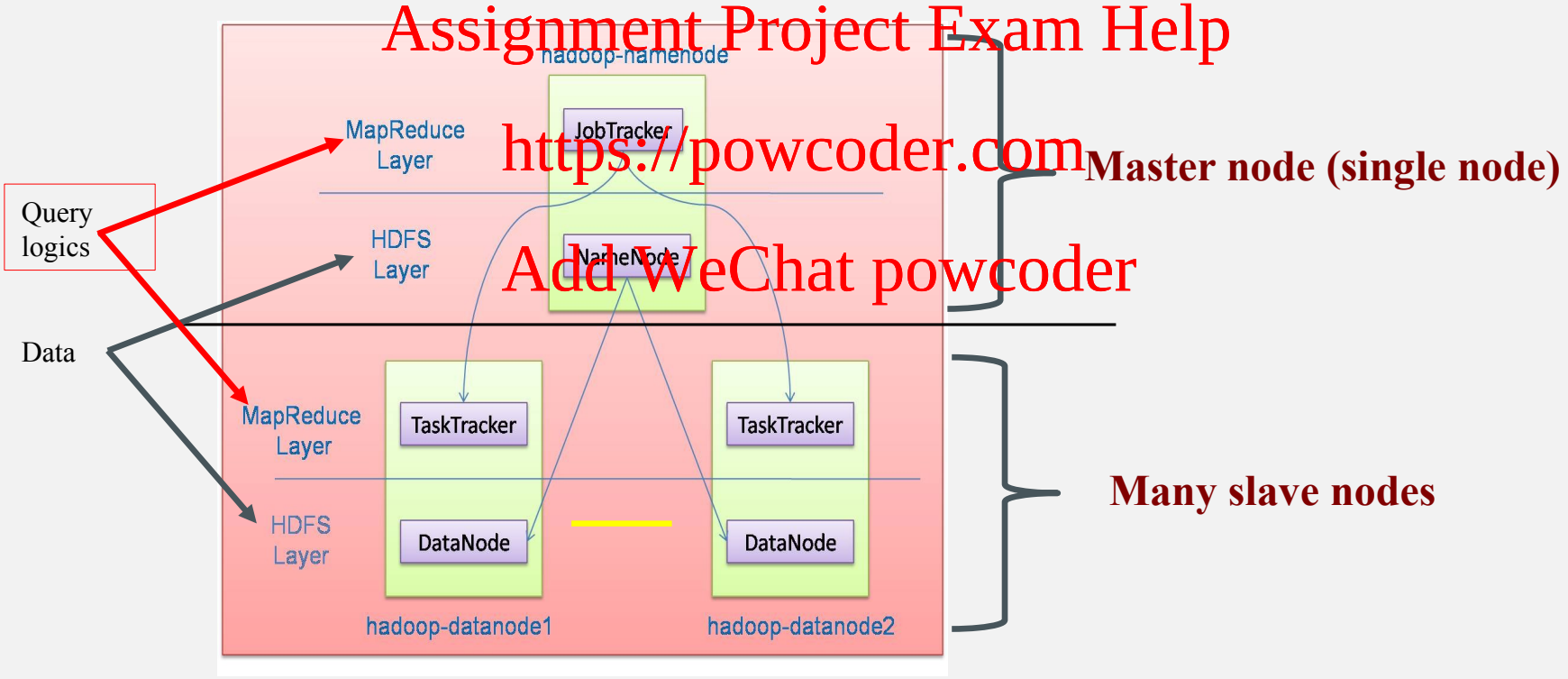
- Volume
- Variety
- Velocity
- Variability

Who Uses MapReduce/Hadoop

- Google: Inventors of MapReduce computing paradigm
- Yahoo: Developing Hadoop open-source of MapReduce
- IBM, Microsoft, Oracle
- Facebook, Amazon, AOL, NetFlex
- Many others + universities and research labs

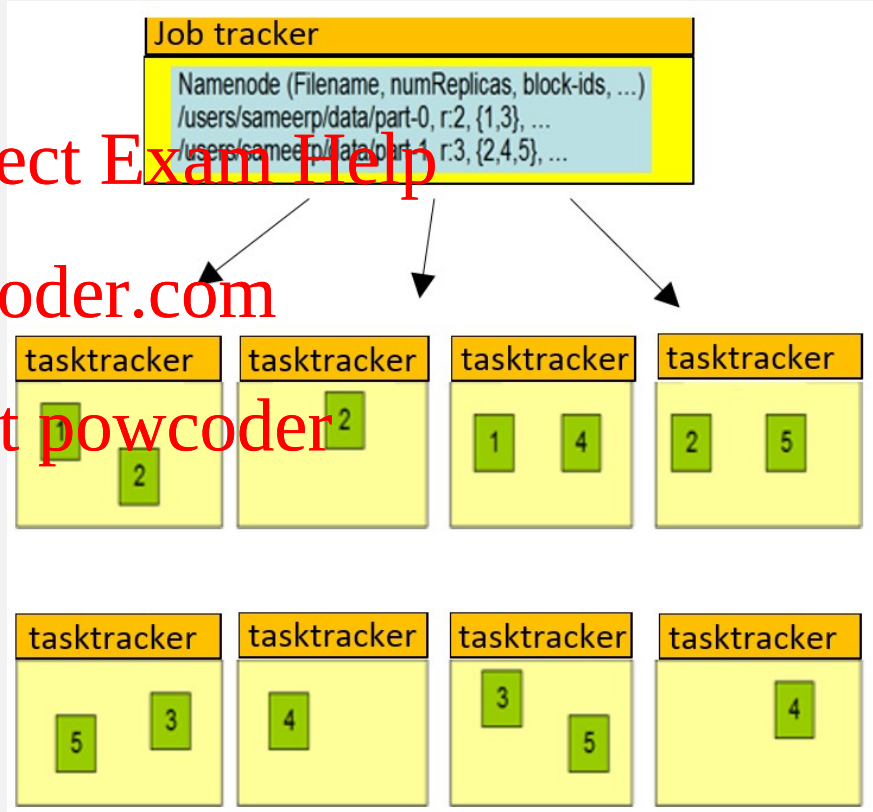
Hadoop Master/Slave Architecture

- Hadoop is designed as a *master-slave shared-nothing* architecture



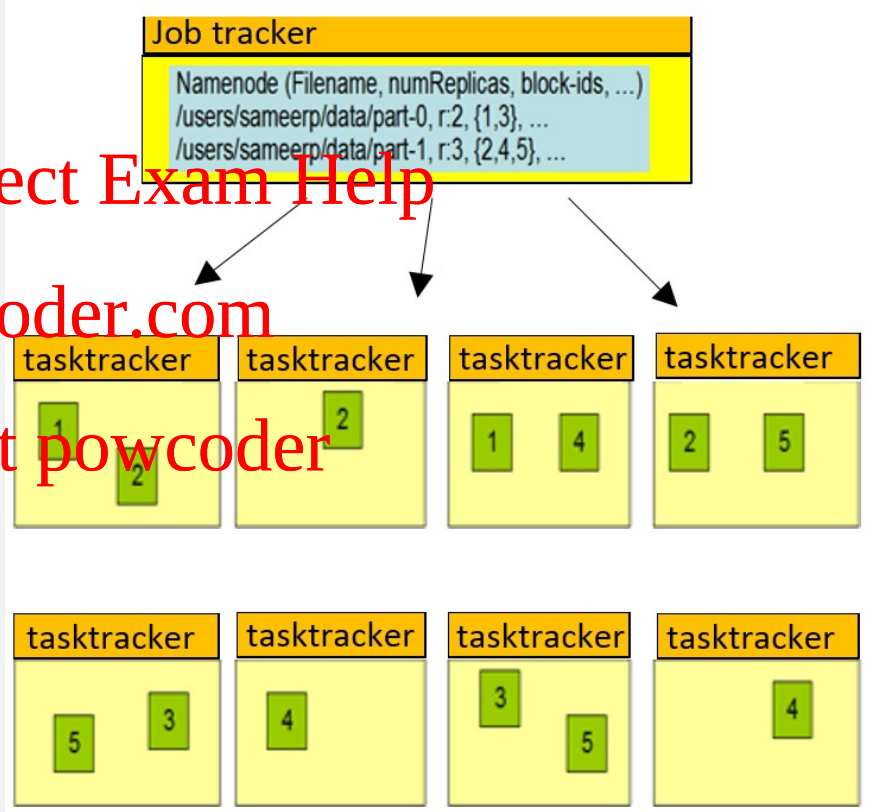
Properties of MapReduce Engine

- **Job Tracker is the master node (runs with the namenode)**
  - Receives the user's job
  - Decides on how many tasks will run (number of mappers)
  - Decides on where to run each mapper (concept of locality)



Properties of MapReduce Engine (Cont'd)

- **Task Tracker is the slave node (runs on each datanode)**
  - Receives the task from Job Tracker
  - Runs the task until completion (either map or reduce task)
  - Always in communication with the Job Tracker reporting progress



Hadoop Processing Logics: map and map( )

- Map: a set of (key, value) pairs – a data structure.
  - A dictionary is a map: the following dictionary has three (key, value) pairs:

    (apple: the round fruit of a tree of the rose family)

    (do: to perform (an act, duty, role, etc.))

    (make: to bring into existence by shaping or changing material)
  - Of course, value can be a number.

- Map( ) is a process in program code . It is called a mapper and
  - takes an input block and
  - produces a map (some (key, value) pairs).

Hadoop Processing Logics: reduce( )

- Reduce( ) is a process to combine multiple maps into one map. It is called a reducer and
  - takes some maps
  - produces a map (some (key, value) pairs).
- Example: for the two sets of maps:

  (apple: the round fruit of a tree of the rose family)

  (do: to perform (an act, duty, role, etc.))

  and

  (apple: often with red colour)

  (make: to bring into existence by shaping or changing material)

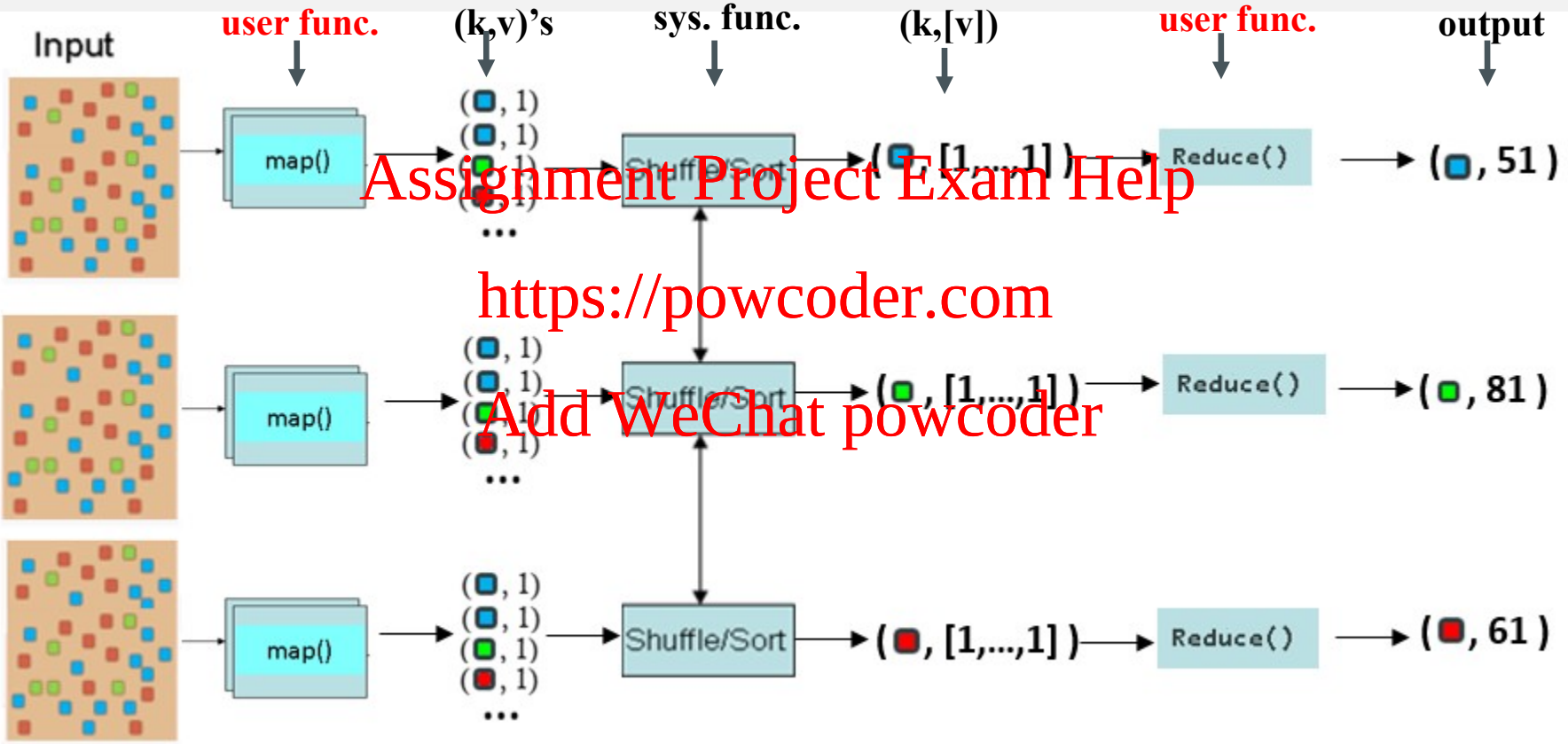
  the reduced result is

  (apple: the round fruit of a tree of the rose family; often with red colour)

  (do: to perform (an act, duty, role, etc.))

  (make: to bring into existence by shaping or changing material)

MapReduce Phases



Deciding on what will be the key and what will be the value ➔ developer's responsibility

Key-Value Pairs

- Mappers and Reducers are users' code (provided functions)
- Just need to obey the Key-Value pairs interface
- **Mappers:**
  - Data blocks/records
  - Produce <key, value> pairs

← input
← output

- **Reducers:**
  - Consume <key, <list of values>>
  - Produce <key, value>

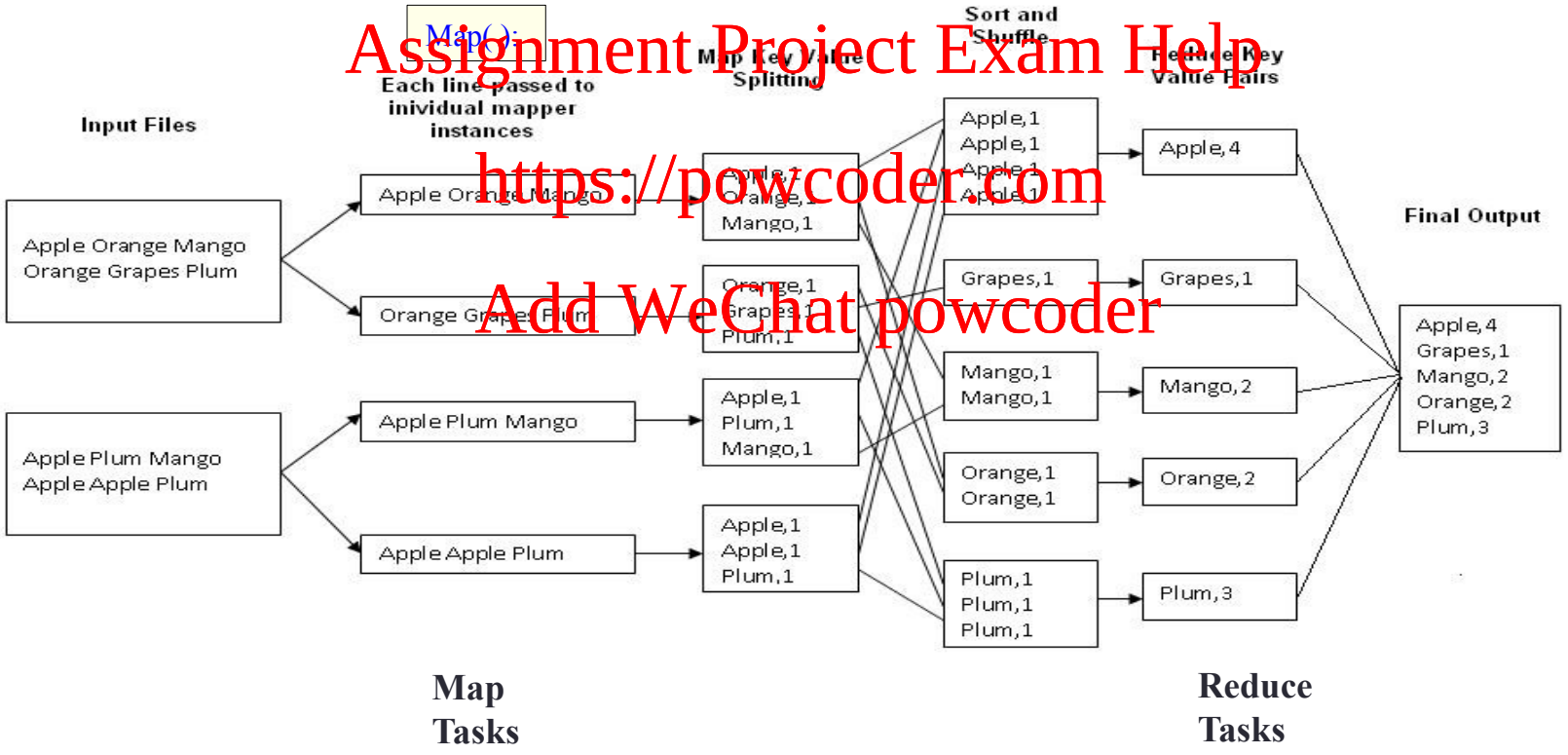
← input
← output

- **Shuffling and Sorting:**
  - Hidden phase between mappers and reducers run automatically by the engine.
  - **Group**s all similar keys from all mappers, **sort**s and passes them to a certain reducer in the form of <key, <list of values>>
  - Input: <key, value> pairs
  - Output: some <key, [value,value,…]> pairs

← input
← output

Example: Word Count

- **Job: Count the occurrences of each word in a data set**

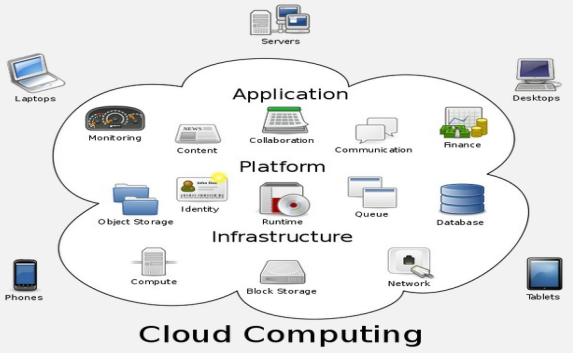


Map( ) is user-defined. You may define it differently. For example, you may produce a map for each file instead for each record of a file.

Bigger Picture: Hadoop vs. Other Systems

	Distributed Databases	Hadoop
Computing Model	- Notion of transactions - Transaction is the unit of work - ACID properties, Concurrency control	- Notion of jobs - Job is the unit of work - No concurrency control
Data Model	- Structured data with known schema - Read/Write mode	- Any data will fit in any format - (un)(semi)structured - ReadOnly mode
Cost Model	- Expensive servers	- Cheap commodity machines
Fault Tolerance	- Failures are rare - Recovery mechanisms	- Failures are common over thousands of machines - Simple yet efficient fault tolerance
Key Characteristics	- Efficiency, optimizations, fine-tuning	- Scalability, flexibility, fault tolerance

- ***Cloud Computing***
  - A computing model where any computing infrastructure can run on the cloud
  - Hardware & Software are provided as remote services
  - Elastic: grows and shrinks based on the user's demand
  - Example: Amazon EC2



Details: Word Count Example – input files

- In the folder ‘input’, put the following two files:

- file01:

  Hello World input 1 for Hadoop

- file02:

  Hello Hadoop Goodbye Hadoop

When these two files are uploaded to Hadoop HDFS, they will be stored on different data nodes.

Details: Word Count Example – the map( ) function

```
public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(Object key, Text value, Context context )
        throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word,
    }
  }
}
```

not used here

count number as object

input text

handle to the bank1 where (k,v) pairs go

Output of mapper for file01:
(Hello, 1)
(World,1)
(input,1)
(1,1)
(for,1)
(Hadoop,1)

Output of mapper for file02:
(Hello, 1)
(Hadoop,1) (Goodbye,1)
(Hadoop,1)

bank1 is input to the Shuffler

Details: Word Count Example – the reduce( ) function

```
public static class IntSumReducer    extends
      Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();
  public void reduce(Text key,
          Iterable<IntWritable> values, Context context )

          throws IOException, InterruptedException      {
     int sum = 0;
     for (IntWritable val : values) {  sum += val.get();    }
     result.set(sum);
     context.write(key, result);
    }
}
```

key k

count number as object

input collection [v1,v2,…]

handle to the bank2 where (k,v) pairs go

Input from shuffler:
(Hello, [1,1])
(World,[1])
(input,[1])
(1,[1])
(for,[1])
(Hadoop,[1,1,1])
(Goodbye,[1])

Output from reducer
(Hello, 2)
(World,1)
(input,1)
(1,1)
(for,1)
(Hadoop,3)
(Goodbye,1)