Introduction to Map-Reduce

CMM705 - Big Data Programming
Lecture 4

Overview

- Introduction to Mapreduce & Hadoop
  - History of Map-Reduce
  - Hadoop Ecosystem
  - Hadoop Architecture
- Map-Reduce Concepts

Challenges

Exponential growth of data size (in Exabyte, Zetabytes, etc. . .)

- Storage
- Processing (Computing)
- Integrating different data sets from different resources
- Making sense of data (find knowledge, patterns, etc. . .)

MapReduce for Rescue !

History of MapReduce

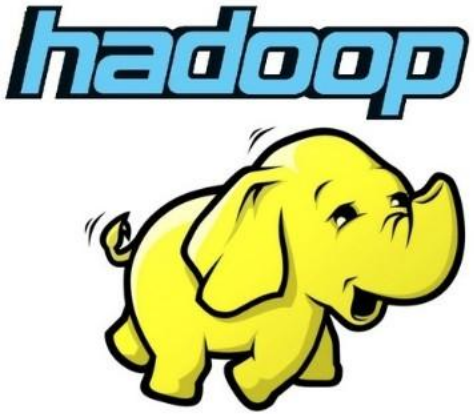
- The actual origins of MapReduce are arguable.
- But the Google presented a paper “MapReduce: Simplified Data Processing on Large Clusters” by Jeffrey Dean and Sanjay Ghemawat in 2004.
  - http://research.google.com/archive/mapreduce.html
- This paper described how Google split, processed, and aggregated their data set of mind-boggling size.
- This is the most cited paper that started the journey on MapReduce.

History of MapReduce ...

- Shortly after that, free and open source software pioneer Doug Cutting started working on a MapReduce implementation to solve scalability in a project called Nutch, an open source search engine.
- Nutch got funded by Yahoo!
- Then Hadoop split out as its own project and eventually became a top-level Apache Foundation project.
- Today, many organizations contribute to Hadoop.
- Later Spark was implemented by running MapReduce in-memory, providing 10X to 100X faster processing speed.

Hadoop

- Hadoop provide storage and computation capabilities
- It is open source and provide reliable, scalable and distributed computing services



- Fault-Tolerant
- Flexible (manage structured or unstructured data)

Hadoop …

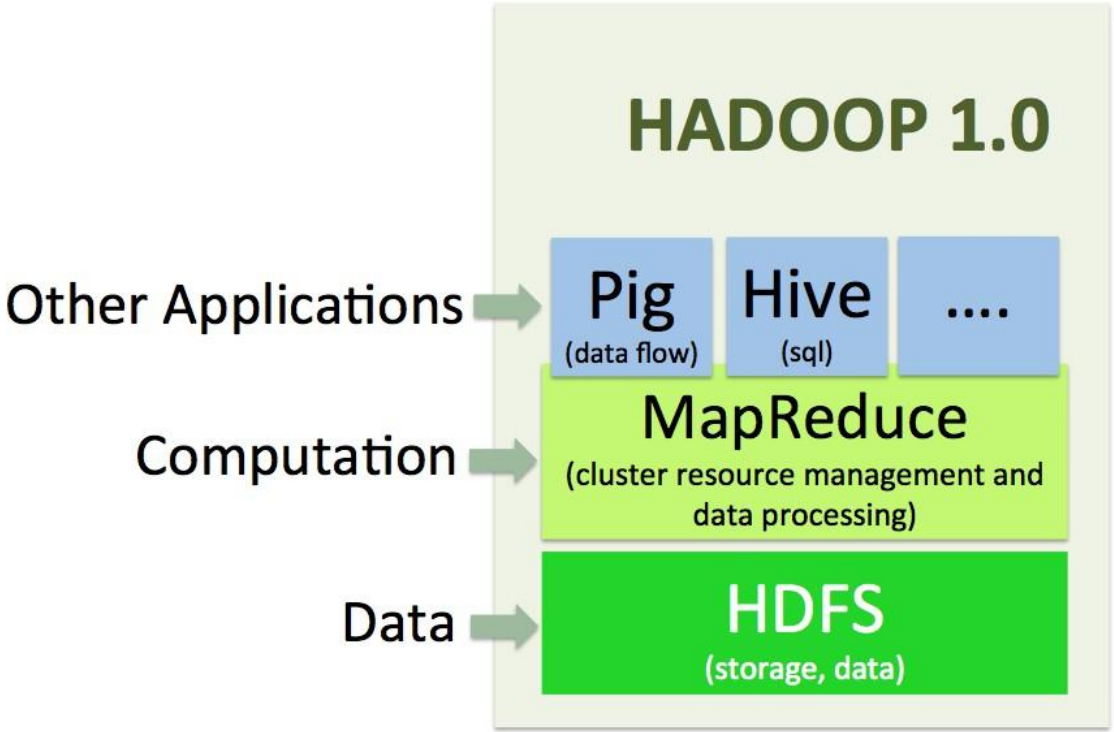
Hadoop has two main components (Modules):

1. Data Storage and Management
2. Processing and Computation

Hadoop Framework = HDFS + MapReduce

- MapReduce: A model for processing large scale data in a distributed manner
- HDFS : Hadoop Distributed File System

Hadoop Components



Hadoop Components : MapReduce

- MapReduce is concerned with the processing/computation of the data.
- Moving Computation is Cheaper than Moving Data.
- It consists of two steps:
  1. **Map**: Divide the problem into smaller and independent sub-problems (sub-tasks) and distribute them to the slave nodes (workers)
  2. **Reduce**: Process data (subtasks) from the slave nodes and perform the Reduce task to perform the final output

MapReduce Engine

The MapReduce engine enables the execution of map and reduce tasks across the cluster and report results.

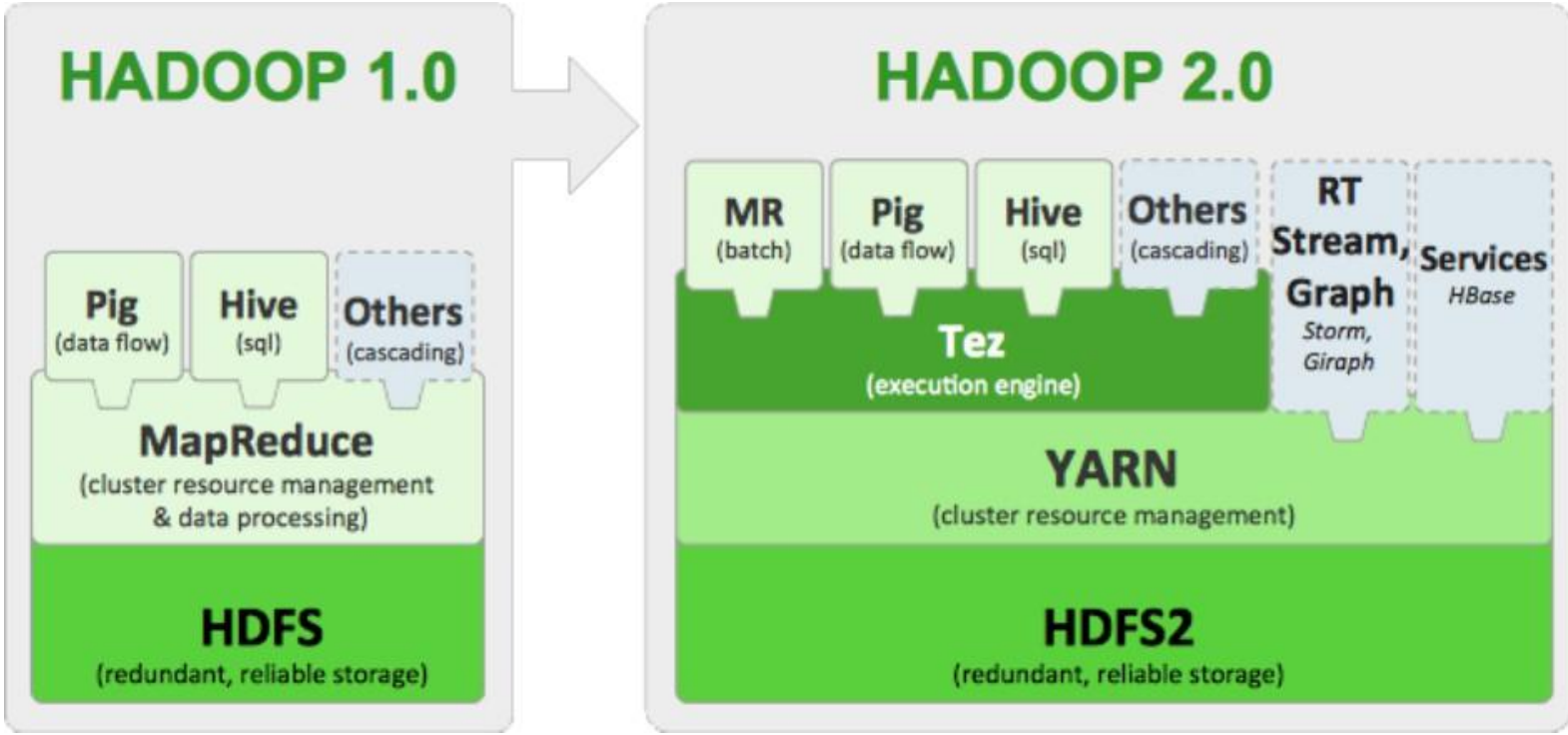
The engine is composed of two main components:

- Job Tracker
  - Divides job into smaller tasks (Map and Reduce)
  - Assigns tasks to TaskTrackers on available DataNodes
- Task Tracker
  - Execute Map and Reduce tasks
  - Monitor the progress of tasks
  - Send task status reports to JobTracker
  - Re-execute failed tasks if necessary

MapReduce Engine (MRv1)

- A client makes a request of a Hadoop cluster, then the request is managed by the JobTracker which allocates the work as closely as possible to the data on which it will work.
- The JobTracker pushes the map and reduce tasks into available slots at one or more TaskTrackers.
- TaskTracker work to execute map and reduce tasks on data from the DataNode.
- When the map and reduce tasks are complete, the TaskTracker notifies the JobTracker, which keeps tracks of all tasks.

Hadoop Stack Transition



YARN (MRv2)

- The fundamental idea of MRv2 is to split up the two major functionalities of the JobTracker into separate daemons:
  - Resource management - Global ResourceManager (RM)
  - Job scheduling/monitoring - Per-application ApplicationMaster (AM)
- YARN enhances the power of a Hadoop compute cluster while keeping it MapReduce compatible
- Rich sources of applications on top of YARN
- It also support other framework such as Tez and SPARK

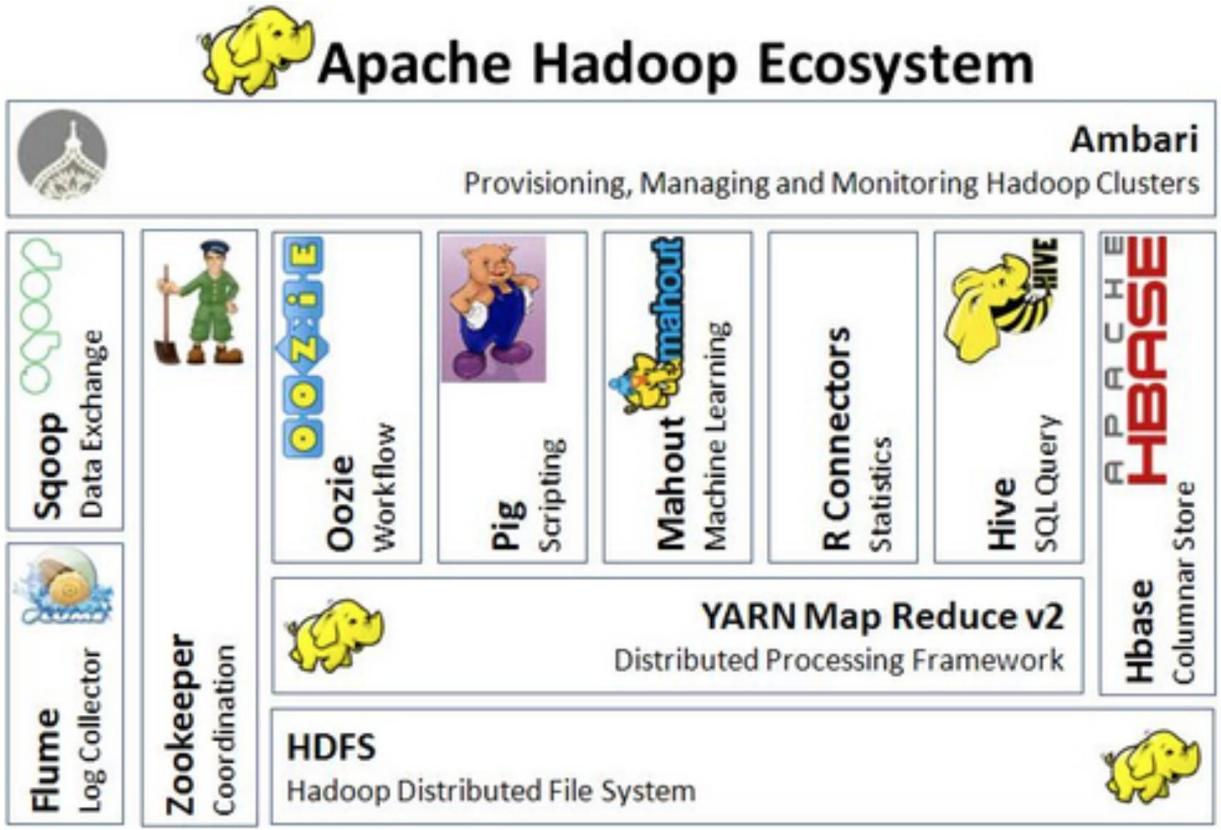
Hadoop Ecosystem

Hadoop Ecosystem is growing very fast, applications and tools are also becoming more popular and aims at allowing interaction with the data (i.e. process, analyse, visualise data) :

- Pig - Platform for analysing large data sets
- Hive - SQL or Hive Query Language (HQL)
- HBase - Columnar Store
- Mahout - Creating scalable machine learning applications
- ZooKeeper - Distributed coordinator
- ...

Hadoop is becoming the kernel of a distributed data processing system.

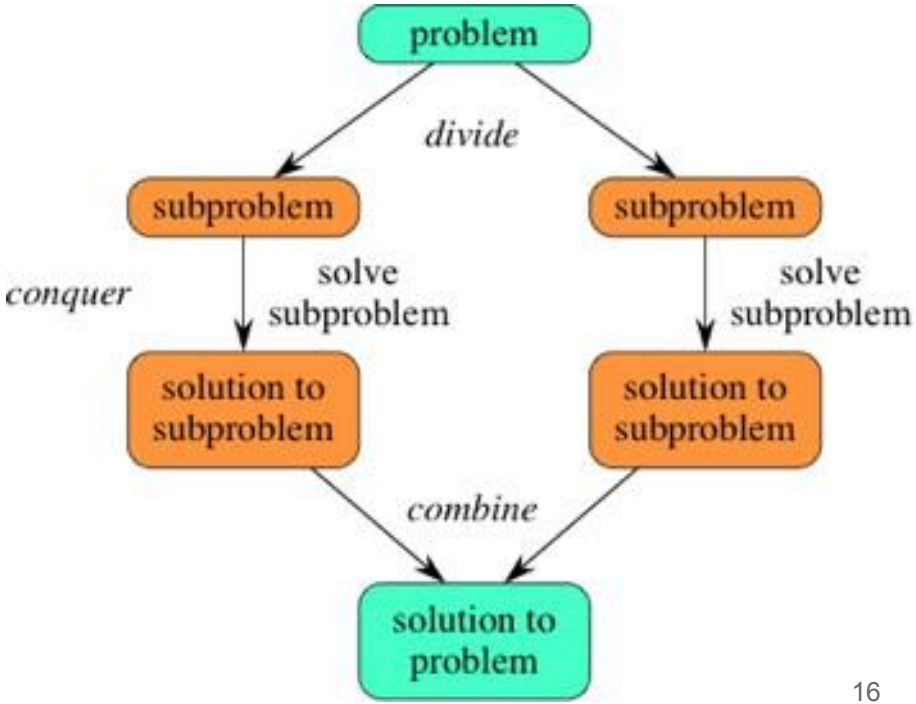
Apache Hadoop Ecosystem ...



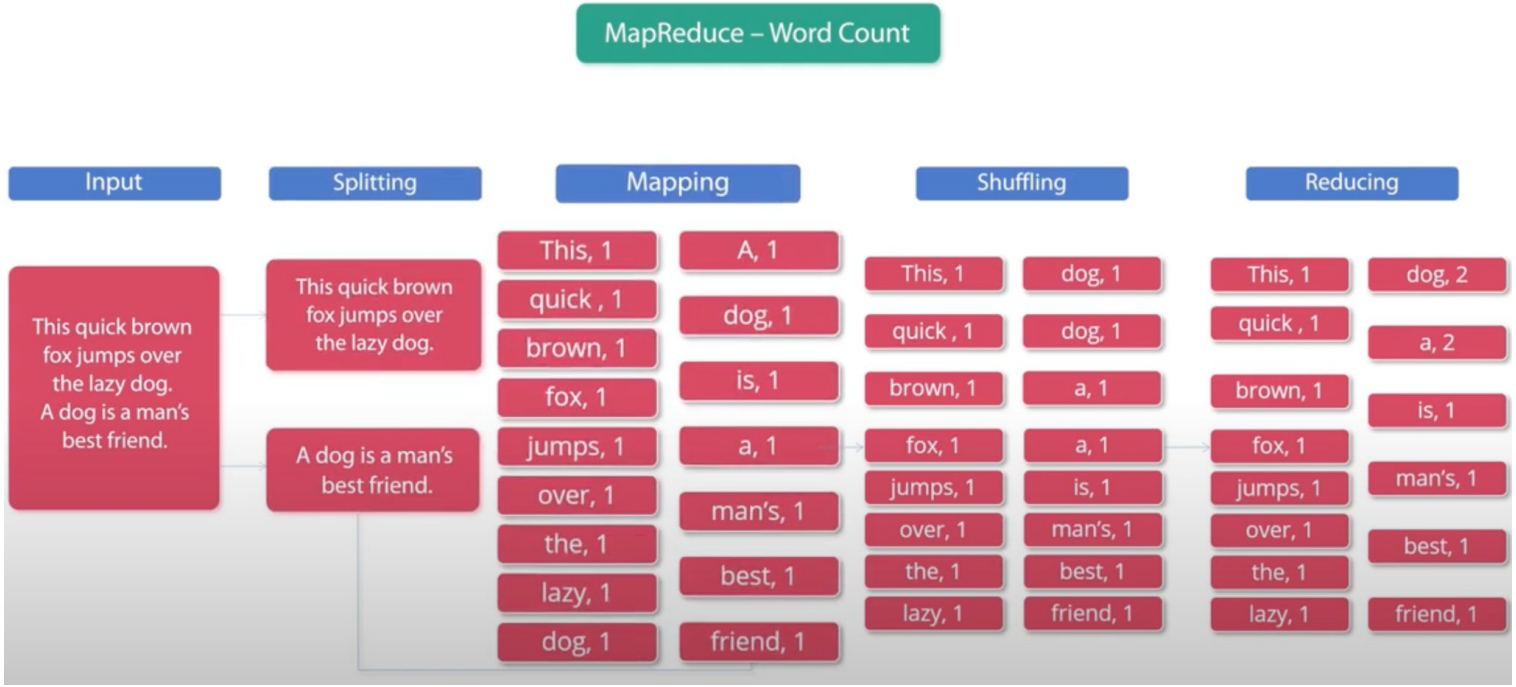
MapReduce: Concepts

Automatic distribution and parallelization of data analytics that reads and processes a lot of data

- map
  - extract something about from each record
  - shuffle and sort
- Reduce
  - aggregate, summarize, filter, or transform
  - creates results
- I/O scheduling
  - load balancing
  - network and data transfer optimization
- fault tolerance
  - handling of machine failures

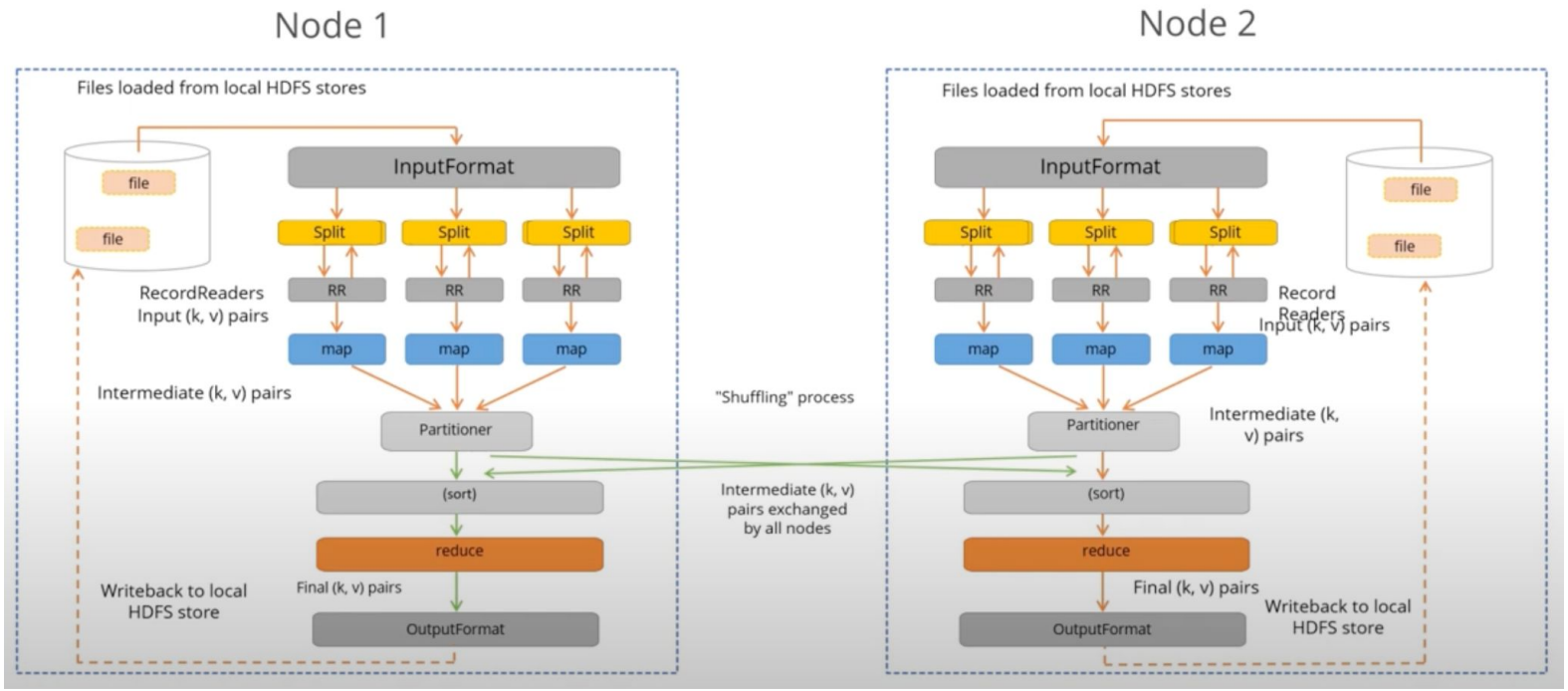


Work Count Example

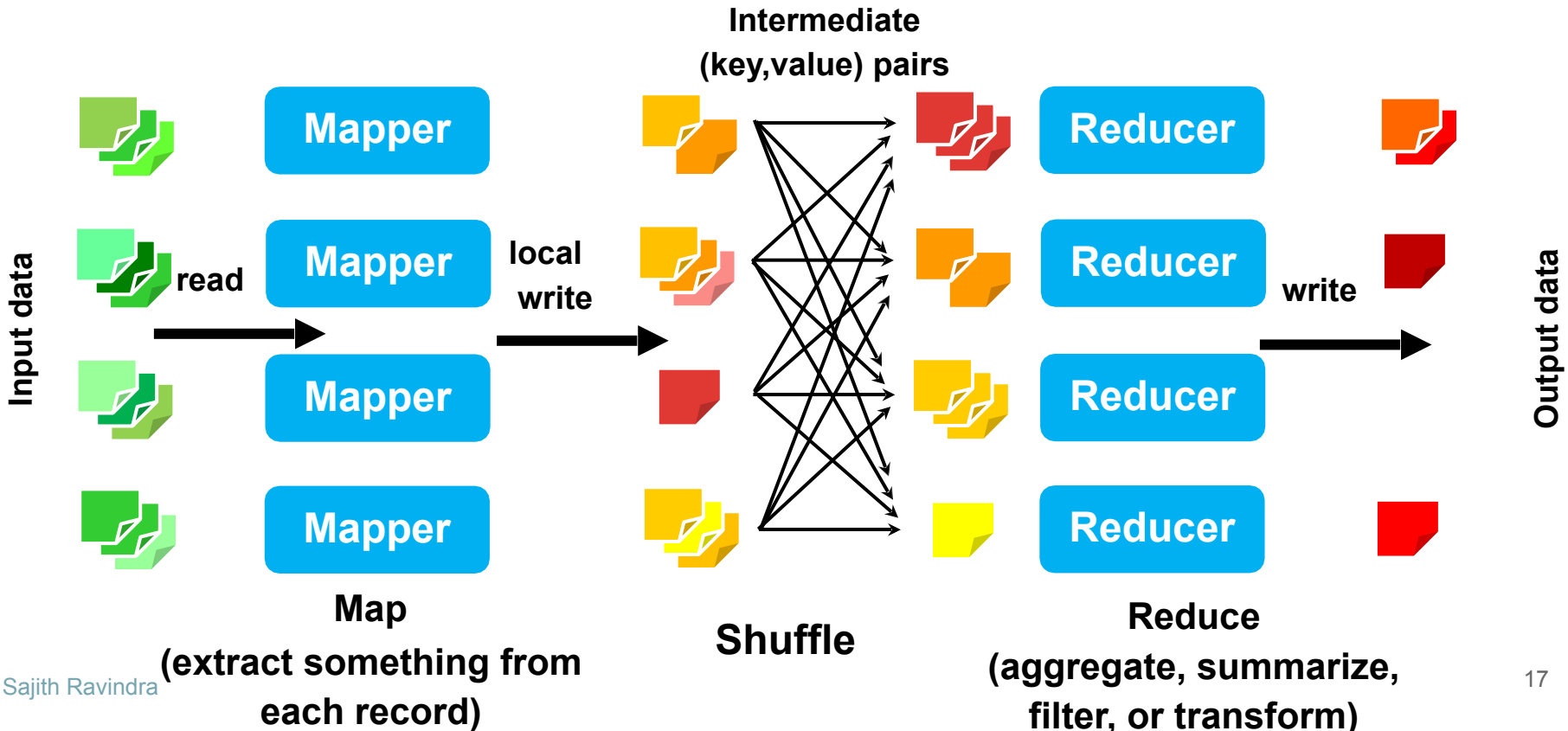


Map Execution Phases

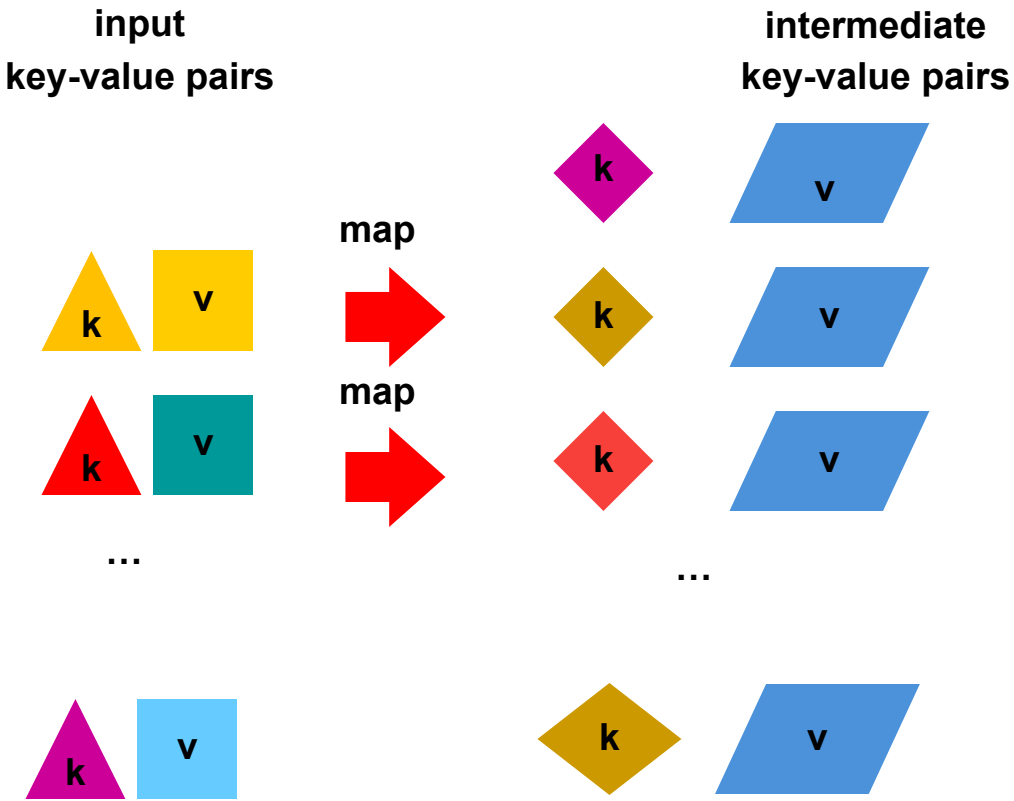
Distributed two node environment



MapReduce: Data and Workflow



MapReduce: Map Abstraction



MapReduce: Map Abstraction

```
The sun rises in the east
The sun sets in the west
```

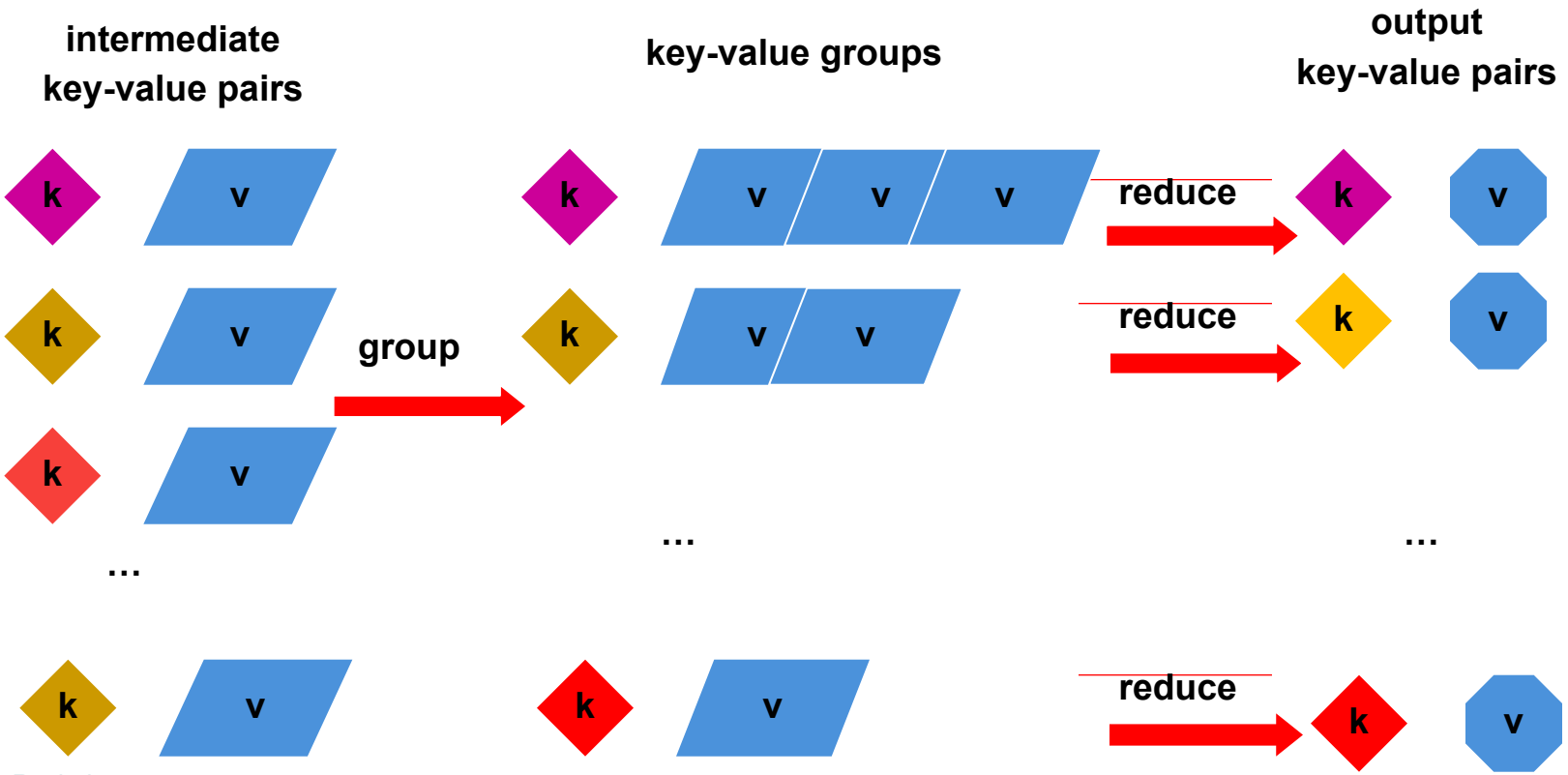
- input = list of key-value pairs
  - Key = input data item or reference to the input data item
  - Value = the data item
- Evaluation
  - Processes a key-value pair to generate intermediate key-value pairs
  - Applies to every value in the input list
- output = new list of key-value pairs
  - it can be different type from the input key-value pair

```
Key: 0, Value: "The sun rises in the east"
Key: 33, Value: "The sun sets in the west"
```

```
Key: "The", Value: 1
Key: "sun", Value: 1
Key: "rises", Value: 1
Key: "in", Value: 1
Key: "the", Value: 1
Key: "east", Value: 1
```

```
def map(key, value):
    list = []
    for x in value:
        if test:
            list.append( (key, x) )
    return list
```

MapReduce: Reduce Abstraction



MapReduce: Reduce Abstraction

- input = large number of intermediate key-value pairs
  - one key-value pair for each data item in all files being greped (including multiple entries for the same data item)
  - starting pairs are sorted by keys where iterator supplies the values for a given key to the Reduce function
  - a given reduce worker processes inputs of the same key.
- Evaluation
  - merges all intermediate values associated with the same key
- outputs = very few finalized key-value pairs
  - one key-value for each unique key across all the files with the
  - number of instances summed into this entry

```
Key: "The", Value: 2
Key: "sun", Value: 2
Key: "rises", Value: 1
Key: "in", Value: 2
Key: "the", Value: 1
```

```
def reduce(key, listOfValues):
    result = 0
    for x in listOfValues:
        result += x
    return (key, result)
```

Word Count

Problem: Given a list of user's comments, count word occurrences.

Run :
yarn jar <jar path> <main class> <input> <output>

I.e.

yarn jar mapreduce-design-patterns/target/mapreduce-design-patterns-1.0-SNAPSHOT.jar week4.count.WordCount stackapps.com/comments/ /output/

Hadoop Map-Reduce

- MapReduce jobs are divided into a set of
  - Map tasks - load, parse, transform, and filter data
  - Reduce tasks - group and aggregate data

- The input to a MapReduce job is a set of files in the data store that are spread out over the Hadoop Distributed File System (HDFS).

- In Hadoop, these files are split with an input format, which defines how to separate a file into input splits.

Hadoop Map-Reduce ...

- Each map task in Hadoop is broken into the following phases: ***record reader***, ***mapper***, ***combiner***, and ***partitioner***
- The output of the map tasks, called the **intermediate keys** and values, are sent to the reducers.
- The reduce tasks are broken into the following phases: ***shuffle***, ***sort***, ***reducer***, and ***output format***.
- The map tasks are run on the nodes in which the data rests.
    - Data typically does not have to move over the network and can be computed on the local machine.

Record reader

- The record reader translates an “input split” generated by “input format” into records.
- The purpose of the record reader is to parse the data into records,
  - But NOT parsing the record itself.
- It passes the data to the mapper in the form of a key/value pair.
  - Key in this context is positional information or a reference to the value
  - The value is the chunk of data that composes a record.

Map

- User-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value pairs, called the ***intermediate pairs***.
- The decision of what is the key and value is very important to accomplish expected results in MapReduce.
  - The key is what the data will be grouped on
  - The value is the information pertinent to the analysis in the reducer.

Combiner

- An optional localized reducer, can group data in the map phase.
- It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper.
- This significantly reduces the amount of data that has to move over the network.
    - E.g. The count of an aggregation is the sum of the counts of each part, we can produce an intermediate count and then sum those intermediate counts for the final result.
      I.e. Sending (hello world, 3) requires fewer bytes than sending (hello world, 1) three times over the network.

Partitioner

- This takes the intermediate key/value pairs from the mapper (or combiner if it is being used) and splits them up into shards, one shard per reducer.
- By default, the partitioner interrogates the object for its hash code (md5sum) and performs a modulus operation by the number of reducers:

  key.hashCode() % (number of reducers)

  - This randomly distributes the keyspace evenly over the reducers, but still ensures that keys with the same value in different mappers end up at the same reducer.
- The partitioned data is written to the local file system for each map task and waits to be pulled by its respective reducer.

Design Mapreduce Applications

MapReduce: Programming Model Implementation

- MapReduce implementations are available in
  - C
  - Java
  - Python
- Hadoop MapReduce is in Java
  - users can construct their own pseudo-distributed Hadoop
  - users can analyze their problem and format their data flow in <key,value> style
  - users must override the Mapper.class and Reducer.class with their own class
  - important parameters
    - *mapred.map.tasks*
    - *mapred.reduce.tasks*
    - *mapred.tasktracker.map.tasks.maximum*
    - *mapred.tasktracker.reduce.tasks.maximum*
    - *mapred.child.java.opts*
    - *Mapred.task.timeout*

MapReduce: Design (1)

Challenge:

- Maintaining fewer computation partitions than data partitions
  - all data is accessible via a distributed file system with replication
  - master is responsible for scheduling, keeping all nodes busy
  - master knows how many data partitions there are, which have completed – atomic commits to disk
  - worker nodes produce data in key order to make it easy to merge

Decision 1: defining map and reduce tasks

- map can do something to each individual key-value pair, but it can't look at other key-value pairs
  - example: filtering out key-value pairs we don't need
- map can emit more than one intermediate key-value pair for each incoming key-value pair
  - example: incoming data is text, map produces (word,1) for each word
- reduce can aggregate data looking at multiple values, as long as map has mapped them to the same (intermediate) key
  - example: count the number of words adding up their total number

MapReduce: Design (2)

Decision 2: intermediate data format

- if reduce needs to look at several values together, map must emit them using the same key

Decision 3: file system

- distributed file system that must store the inputs, outputs but local storage is used to store temporary results

Decision 4: driver program (Client machine)

- driver program runs on a single node
  - specifying where to find the inputs, where to transfer the outputs
  - specifying what mapper and reducer to use
  - customizing behavior of the execution

Decision 5: runtime system

- runtime system supervises the execution of tasks (JobTracker)

MapReduce: Design (3)

Decision 6: locality

- as much as possible map task must be scheduled on machine that already has the input data

Decision 7: task granularity

- define task granularity considering the number of map and reduce tasks

Decision 8: dealing with stragglers

- Long running tasks
- schedule some backup tasks to avoid long tail problem

Decision 9: saving bandwidth

- use combiner tasks to optimize bandwidth

Decision 10: handling bad records

- manage “last gasp" packet with current sequence number

MapReduce: Design Mistakes

Mistake 1: do not use static variables

- Mapper and Reducer should be stateless, i.e. after Mapper + Reducer return they should remember nothing about the processed data
  - reason: no guarantees about which key-value pairs will be processed by which workers

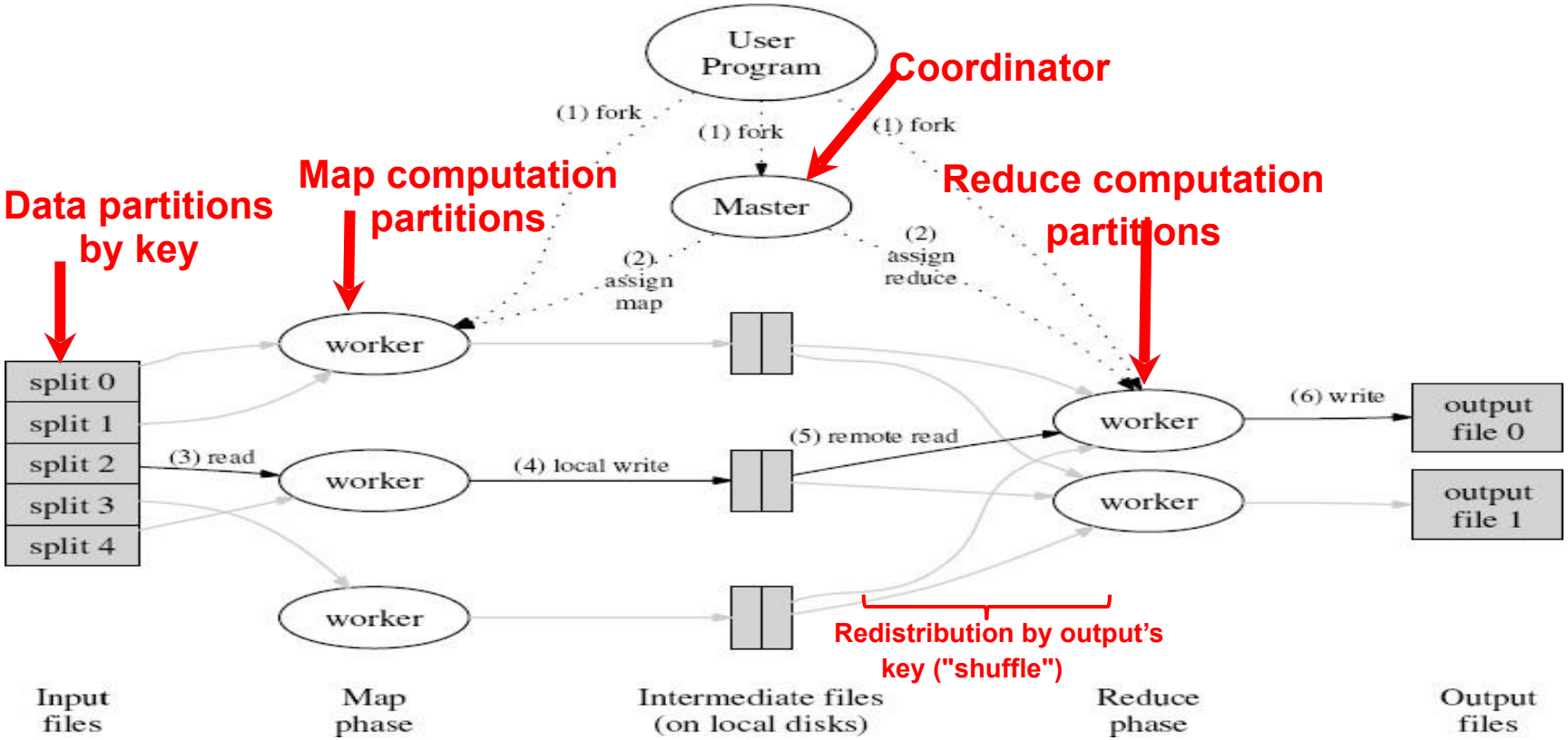
Mistake 2: do not try to do your own I/O

- don't try to read from or write to files in the file system
  - reason: MapReduce framework does all the I/O for you.All the incoming data will be fed as arguments to Mapper and Reducer any data functions produce should be output via emit

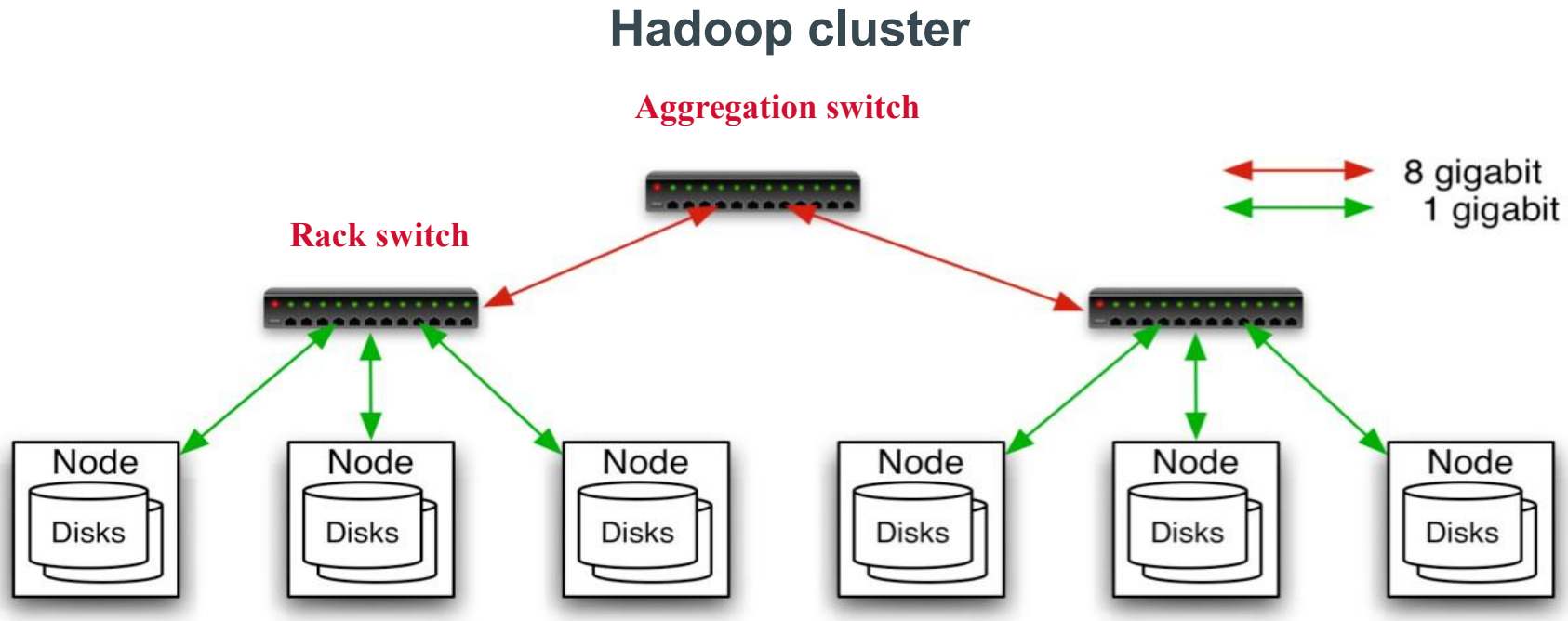
Mistake 3: Mapper must not map too much data to the same key

- don't map everything to the same key otherwise the Reducer will be overwhelmed It's okay if some reduce workers have more work than others

MapReduce: Execution (1)



MapReduce: Execution (2)



MapReduce: Developing Process

User to do list:

- Specify
  - Input/output files
  - M: number of map workers
  - R: number of reduce workers
  - W: number of nodes
- Write *map* and *reduce* functions
- Write master program

MapReduce: Word Count Example (1)

Mapper

- input: value: lines of text of input
- output: key: word, value: 1

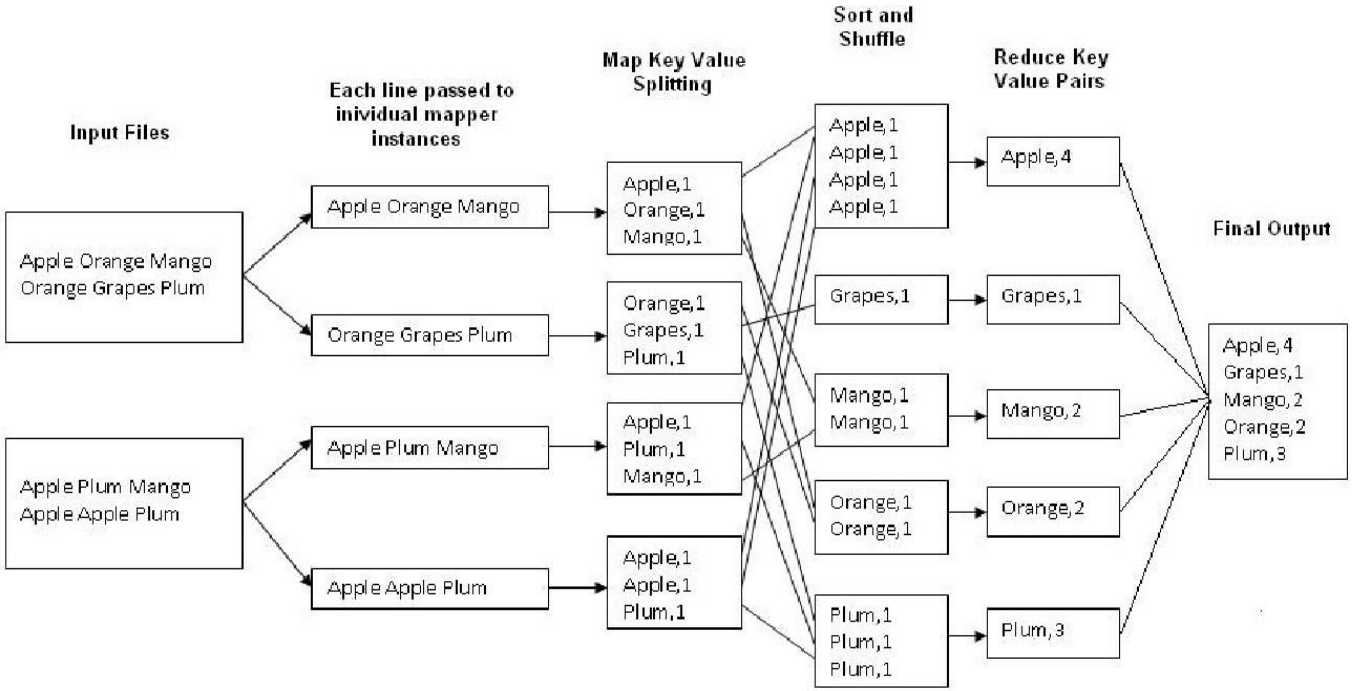
Reducer

- input: key: word, value: set of counts
- output: key: word, value: sum

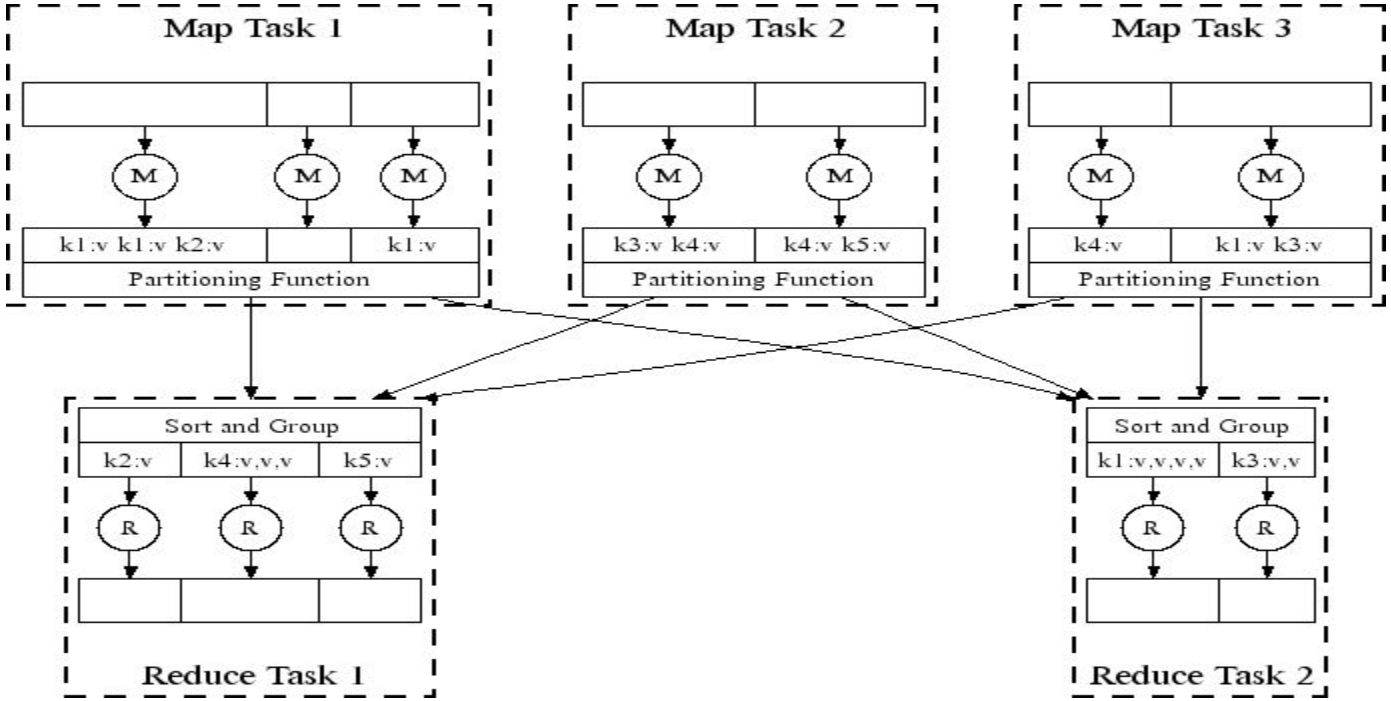
Master program

- forks workers and assigns them map and reduce tasks
- submits workers to the Hadoop cluster

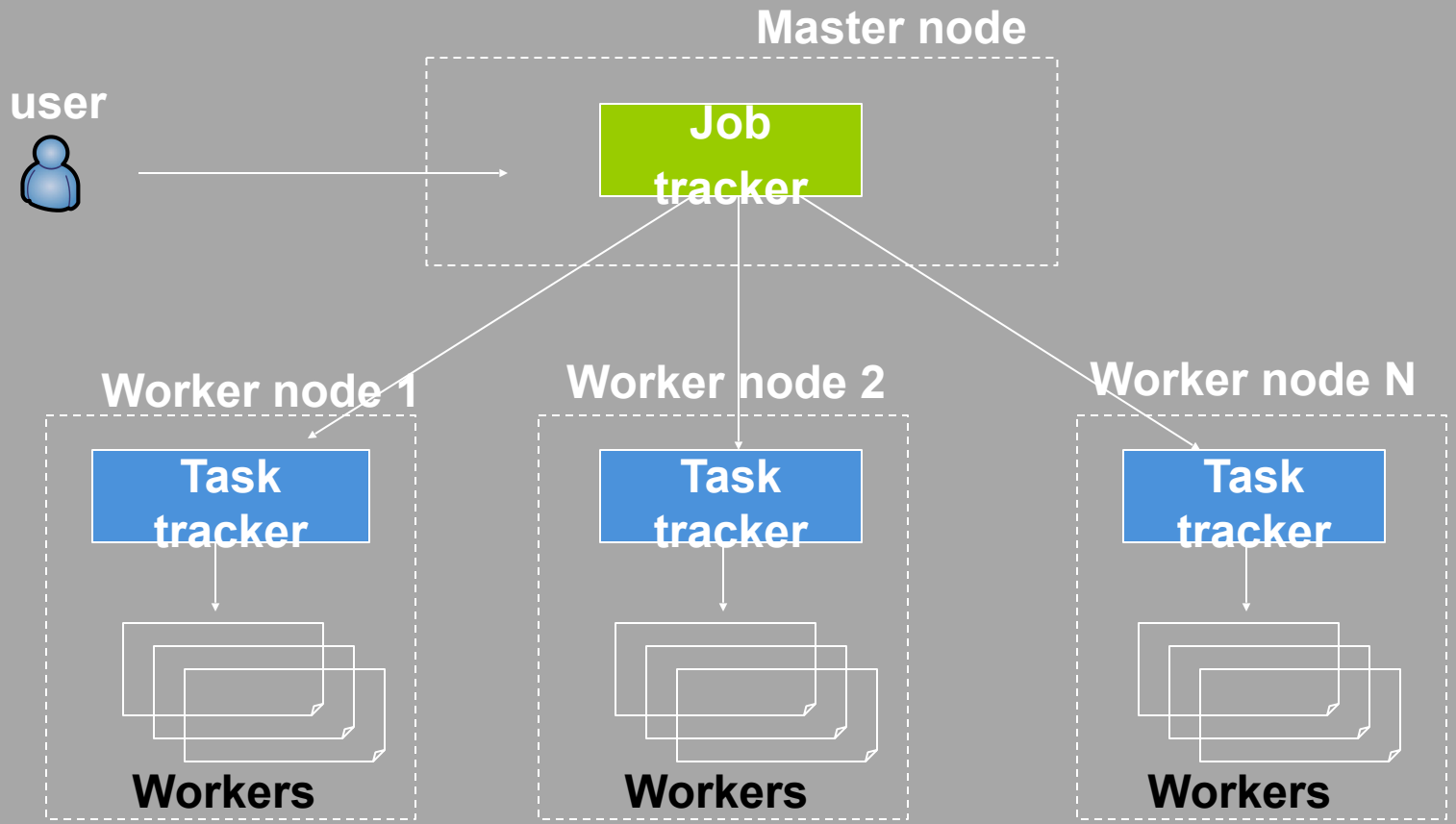
MapReduce: Word Count Example (2)



MapReduce: Word Count Example (3)



MapReduce: Word Count Example (4)



MapReduce: Word Count - Mapper

```
public static class Map extends MapReduceBase
             implements Mapper<LongWritable,Text,Text,IntWritable> {
  private static final IntWritable one = new IntWritable(1);
  private Text word  = new Text();

  public static void map(LongWritable key, Text value,
                         OutputCollector<Text,IntWritable> output,
                         Reporter reporter)
                         throws IOException {
       String line = value.toString();
       StringTokenizer = new StringTokenizer(line);
       while(tokenizer.hasNext()) {
          word.set(tokenizer.nextToken());
          output.collect(word,one);
       }
     }
  }
```

MapReduce: Word Count - Reducer

```
public static class Reduce extends MapReduceBase
            implements Reducer<Text,IntWritable,Text,IntWritable> {

 public static void map(Text key, Iterator<IntWritable> values,
            OutputCollector<Text,IntWritable> output, Reporter reporter)
            throws IOException {
      int sum = 0;
      while(values.hasNext()) {
        sum += values.next().get();
      }
       output.collect(key, new IntWritable(sum));
    }
  }
```

MapReduce: Word Count - Configure

Configuring map and reduce workers

- jobs are controlled by configuring *JobConfs* that are maps from attribute names to string values
- framework defines attributes to control how the job is executed

```
conf.set("mapred.job.name", "MyApp");
```

- applications can add arbitrary values to the *JobConf*

```
conf.set("my.string", "foo");
conf.set("my.integer", 12);
```

- *JobConf* is available to all tasks

Launching and controlling the Word Count application

- create a main() program for the application configuring:
  - *Mapper* and *Reducer task implementations*
  - output key-value types (input types are inferred from the *InputFormat*)
  - locations for the input and output

- launching program submits the job and typically waits for its completion

MapReduce: Word Count Example - main()

```
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");
// configuring output key-value pair
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
// specifing Mapper, Combiner and Reducer
conf.setMapperClass(Map.class);
conf.setCombinerClass(Reduce.class);
conf.setReducer(Reduce.class);
// defining input + output data format
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
// launch the MapReduce application
JobClient.runJob(conf);
```

Lab Session

StackOverflow DataSet

Comments :

<row **Id**="6" **PostId**="12" **Score**="0" **Text**="Just as an addition, if you are going to test your library, don't use a key (that is what no key is for). However, make sure the final product demands and API key."

CreationDate="2010-05-19T23:48:05.680" **UserId**="23" />

Setup

- Clone https://github.com/ramindu-msc/iit
  - or
    - `cd iit/`
    - `git checkout main`
    - `git pull origin main`
- Install java and maven
  - `./files/install-java.sh`
- Navigate to mapreducer module and build
  - `cd lab3_4_map_reduce/mapreduce-design-patterns/`
  - `mvn clean install`
  - `cp target/mapreduce-design-patterns-1.0-SNAPSHOT.jar src/main/resources/`
- Execute the following from `mapreduce-design-intro`
  - `sudo docker compose -f ../lab1/docker-compose.yaml up -d`
- check the setup is running using (replace your VM external IP address)
  - http://34.173.176.232:9870/dfshealth.html#tab-overview
- Check container logs using
  - `sudo docker logs -f namenode`
  - `sudo docker logs -f datanode1` and `sudo docker logs -f datanode2`

Run Map Reduce

- Add files to HDFS
  - `sudo docker exec -it namenode bash`
  - `hdfs dfs -mkdir -p /user/test/`
  - `hdfs dfs -put /opt/hadoop/resources/stackapps.com /user/test/stackapps.com`
  - `hdfs dfs -ls /user/test/`
- Run Hadoop
  - Incase if you have run the command before
    - `hadoop fs -rm -r /user/test/output/wordCount`
  - `yarn jar /opt/hadoop/resources/mapreduce-design-patterns-1.0-SNAPSHOT.jar` **`org.iit.mapreduce.patterns.count.WordCount`** `/user/test/stackapps.com/comments/ /user/test/output/wordCount`
- Check output
  - `INFO output.FileOutputCommitter: Saved output of task 'attempt_local262476617_0001_r_000000_0' to hdfs://namenode:9000/user/test/output/wordCount`
  - `hadoop fs -ls /user/test/output/wordCount`
  - `hadoop fs -cat /user/test/output/wordCount/part-r-00000`
  - `hadoop fsck /user/test/output/wordCount/part-r-00000 -files -blocks -locations`
  -