CS60021: Scalable Data Mining

Sourangshu Bhattacharya

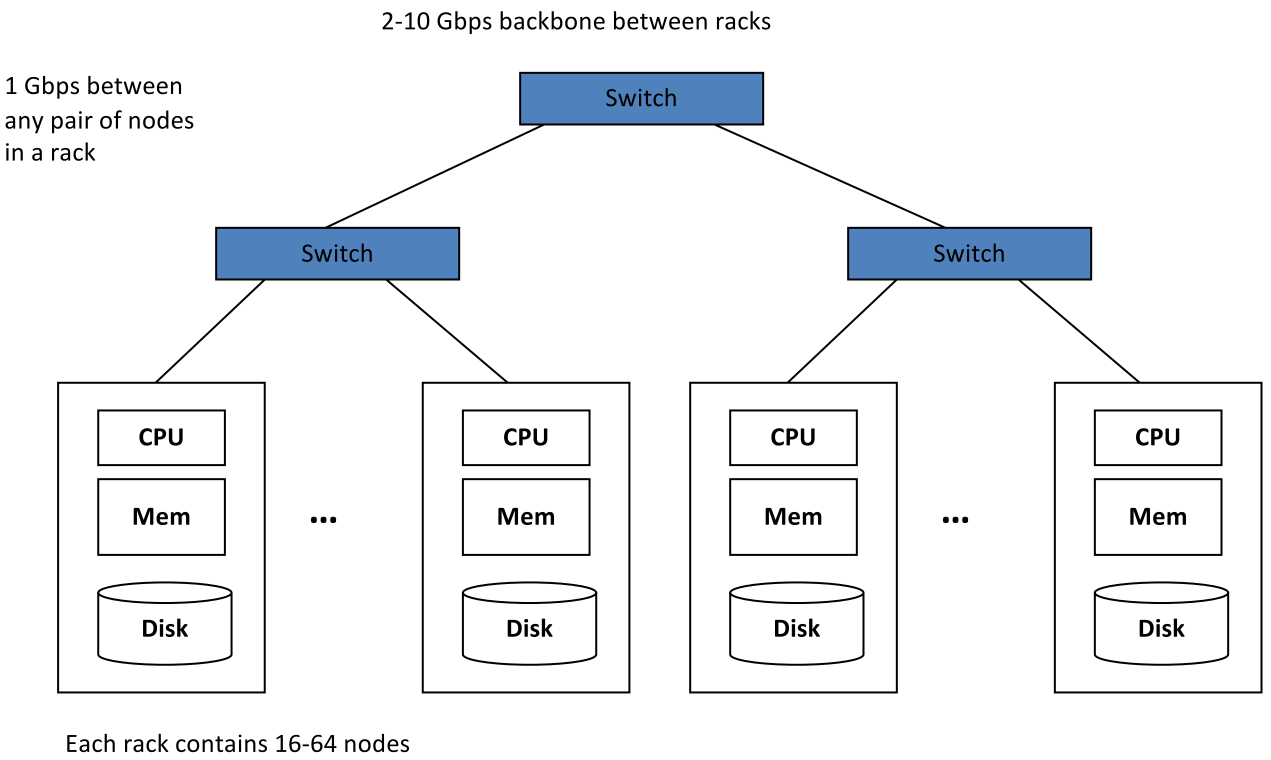
In this Lecture:

- Outline:
  - What is Big Data?
  - Issues with Big Data
  - What is Hadoop ?
  - What is Map Reduce ?
  - Example Map Reduce program.

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the data
- ~ 400 hard drives to store the data
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

Cluster Architecture



Large-scale Computing

- **Large-scale computing** for **data mining** problems on **commodity hardware**

- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - People estimated Google has ~1M machines
      - 1,000 machines fail every day!

Big Data Challenges

- Scalability: processing should scale with increase in data.
- Fault Tolerance: function in presence of hardware failure
- Cost Effective: should run on commodity hardware
- Ease of use: programmers do not write additional code for communication, fault tolerance, etc.
- Flexibility: able to process unstructured data
- Solution: Map Reduce !

Idea and Solution

- **Issue: Copying data over a network takes time**
- **Ideas:**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Map-reduce** addresses these problems
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce

What is Hadoop ?

- A scalable fault-tolerant distributed system for data storage and processing.

- Core Hadoop:
  - Hadoop Distributed File System (HDFS)
  - Hadoop YARN: Job Scheduling and Cluster Resource Management
  - Hadoop Map Reduce: Framework for distributed data processing.

- Open Source system with large community support.
  https://hadoop.apache.org/

What is Map Reduce ?

- Method for distributing a task across multiple servers.
- Proposed by Dean and Ghemawat, 2004.
- Consists of two developer created phases:
  - Map
  - Reduce
- In between Map and Reduce is the Shuffle and Sort phase.
- User is responsible for casting the problem into map – reduce framework.
- Multiple map-reduce jobs can be “chained”.

Programming Model: MapReduce

Warm-up task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:**
  - Analyze web server logs to find popular URLs

Task: Word Count

Case 1:

– File too large for memory, but all <word, count> pairs fit in memory

Case 2:

- Count occurrences of words:
  – `words(doc.txt) | sort | uniq -c`
    - where **`words`** takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of **MapReduce**
  – Great thing is that it is naturally parallelizable

MapReduce: Overview

- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** → <k’, v’>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k’, <v’>*)** → <k’, v’’>*
    - **All values *v’* with same key *k’* are reduced together and processed in *v’* order**
    - There is one Reduce function call per unique key *k’*

MapReduce: Word Counting

Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the
   document
   for each word w in value:
    emit(w, 1)

 reduce(key, values):
 // key: a word; value: an iterator over
 counts
      result = 0
      for each count v in values:
           result += v
      emit(key, result)
```

HADOOP

Map Phase

- User writes the mapper method.

- Input is an unstructured record:
  - E.g. A row of RDBMS table,
  - A line of a text file, etc

- Output is a set of records of the form: <key, value>
  - Both key and value can be anything, e.g. text, number, etc.
  - E.g. for row of RDBMS table: <column id, value>
  - Line of text file: <word, count>

Shuffle/Sort phase

- Shuffle phase ensures that all the mapper output records with the same key value, goes to the same reducer.
- Sort ensures that among the records received at each reducer, records with same key arrives together.

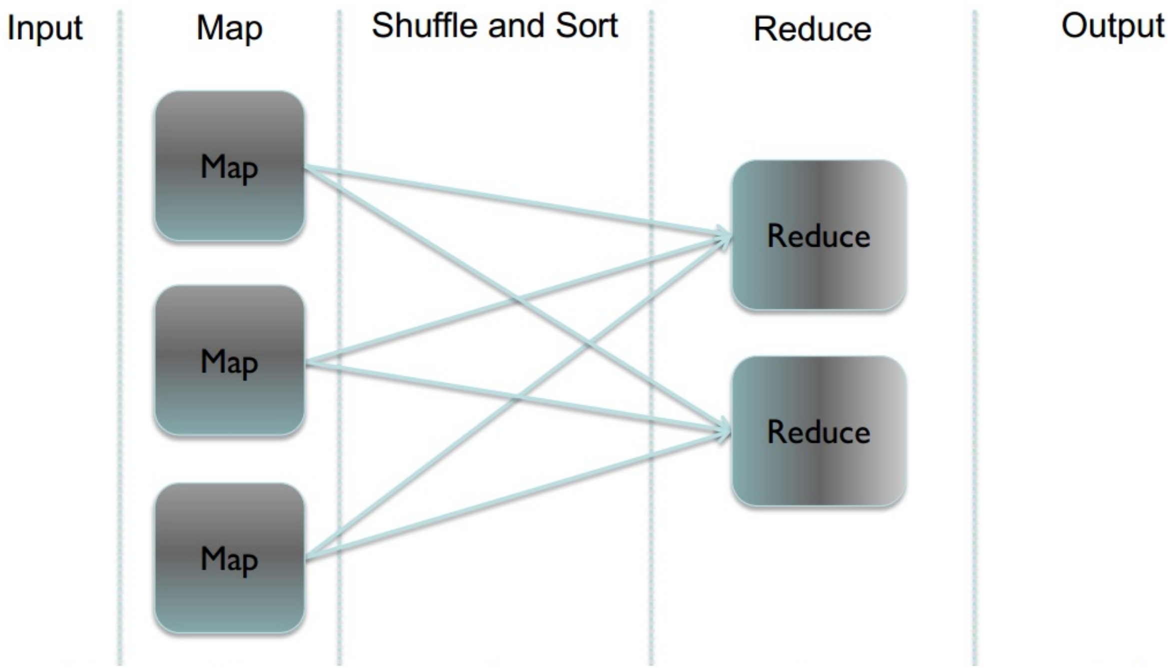
Reduce phase

- Reducer is a user defined function which processes mapper output records with some of the keys output by mapper.

- Input is of the form <key, value>
  - All records having same key arrive together.

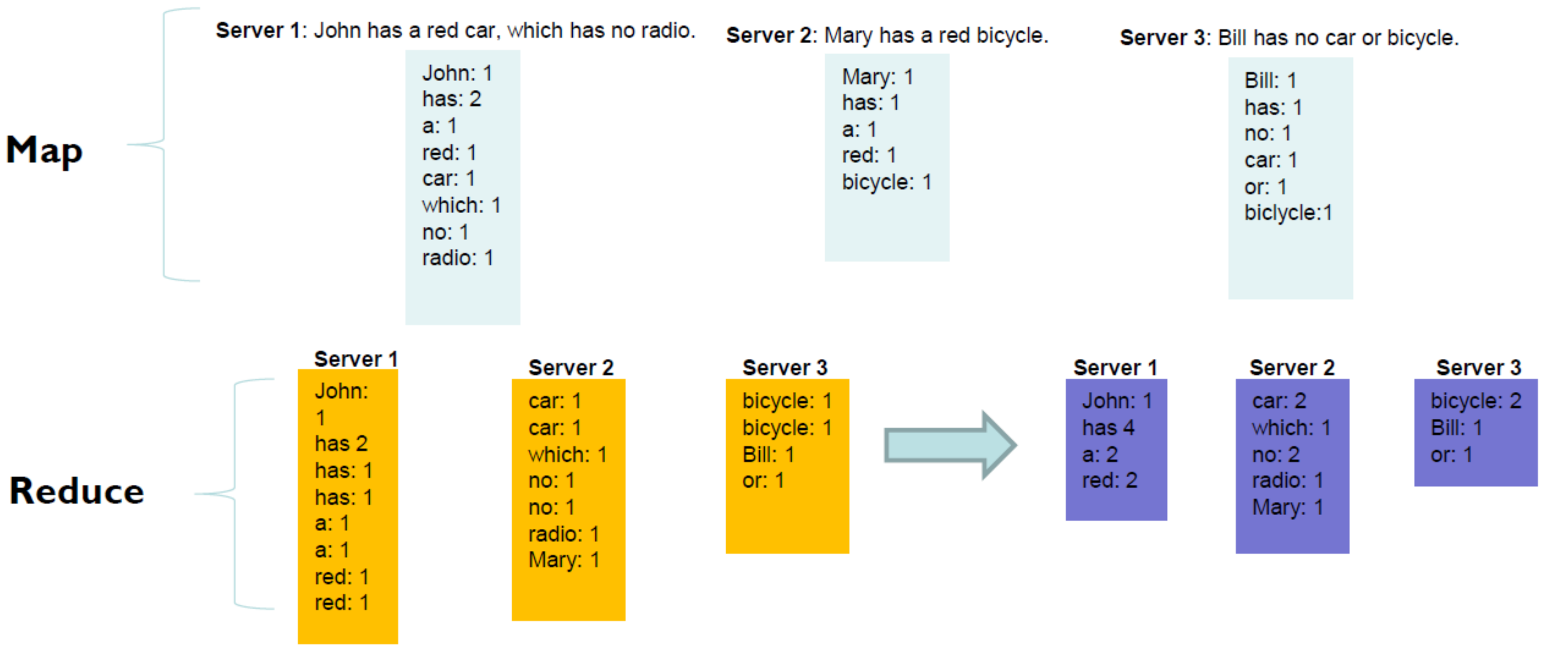
- Output is a set of records of the form <key, value>
  - Key is not important

Parallel picture

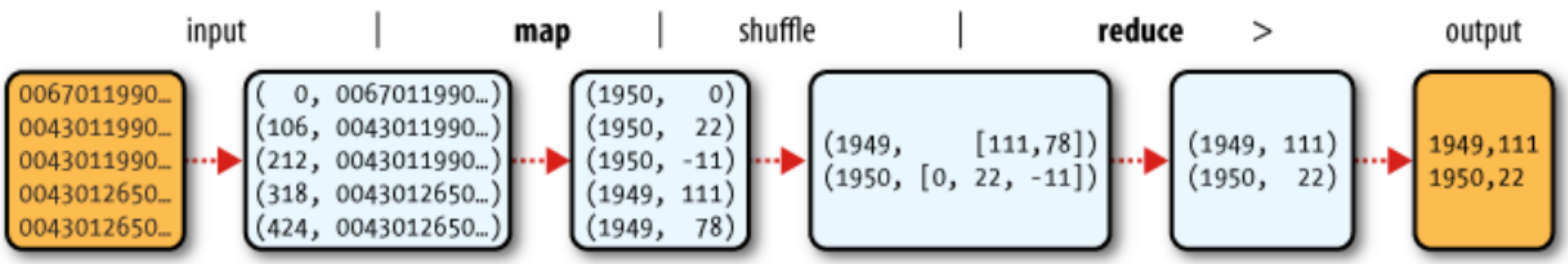


Example

Word Count: Count the total no. of occurrences of each word



Map Reduce - Example



What was the max/min temperature for the last century ?

Hadoop Map Reduce

- Provides:
  - Automatic parallelization and Distribution
  - Fault Tolerance
  - Methods for interfacing with HDFS for colocation of computation and storage of output.
  - Status and Monitoring tools
  - API in Java
  - Ability to define the mapper and reducer in many languages through Hadoop streaming.

HDFS

What’s HDFS

- HDFS is a distributed file system that is fault tolerant, scalable and extremely easy to expand.
- HDFS is the primary distributed storage for Hadoop applications.
- HDFS provides interfaces for applications to move themselves closer to data.
- HDFS is designed to ‘just work’, however a working knowledge helps in diagnostics and improvements.

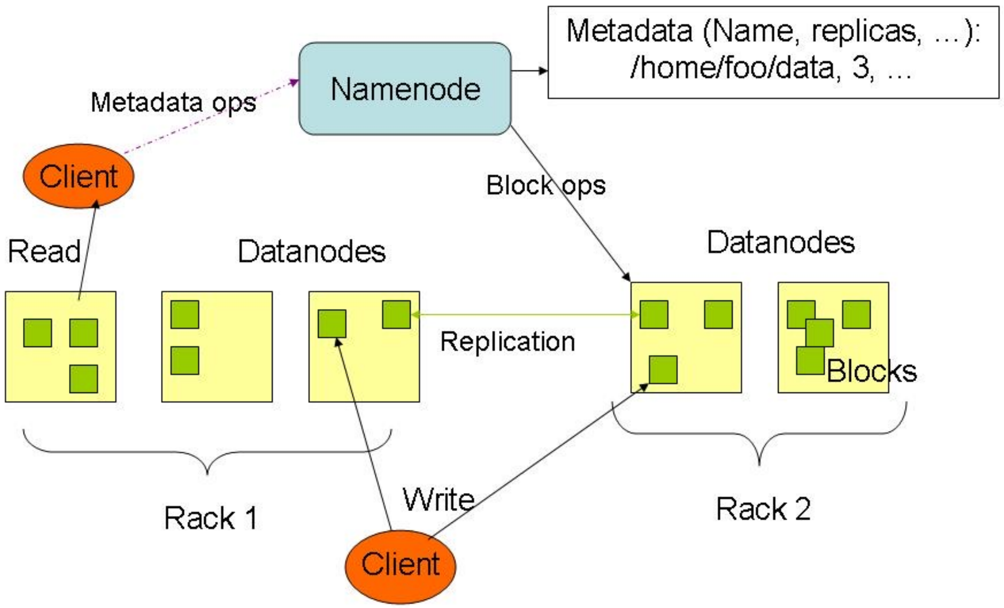
Components of HDFS

There are two (*and a half*) types of machines in a HDFS cluster

- NameNode :– is the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g; what blocks make up a file, and on which datanodes those blocks are stored.

- DataNode :- where HDFS stores the actual data, there are usually quite a few of these.

HDFS Architecture



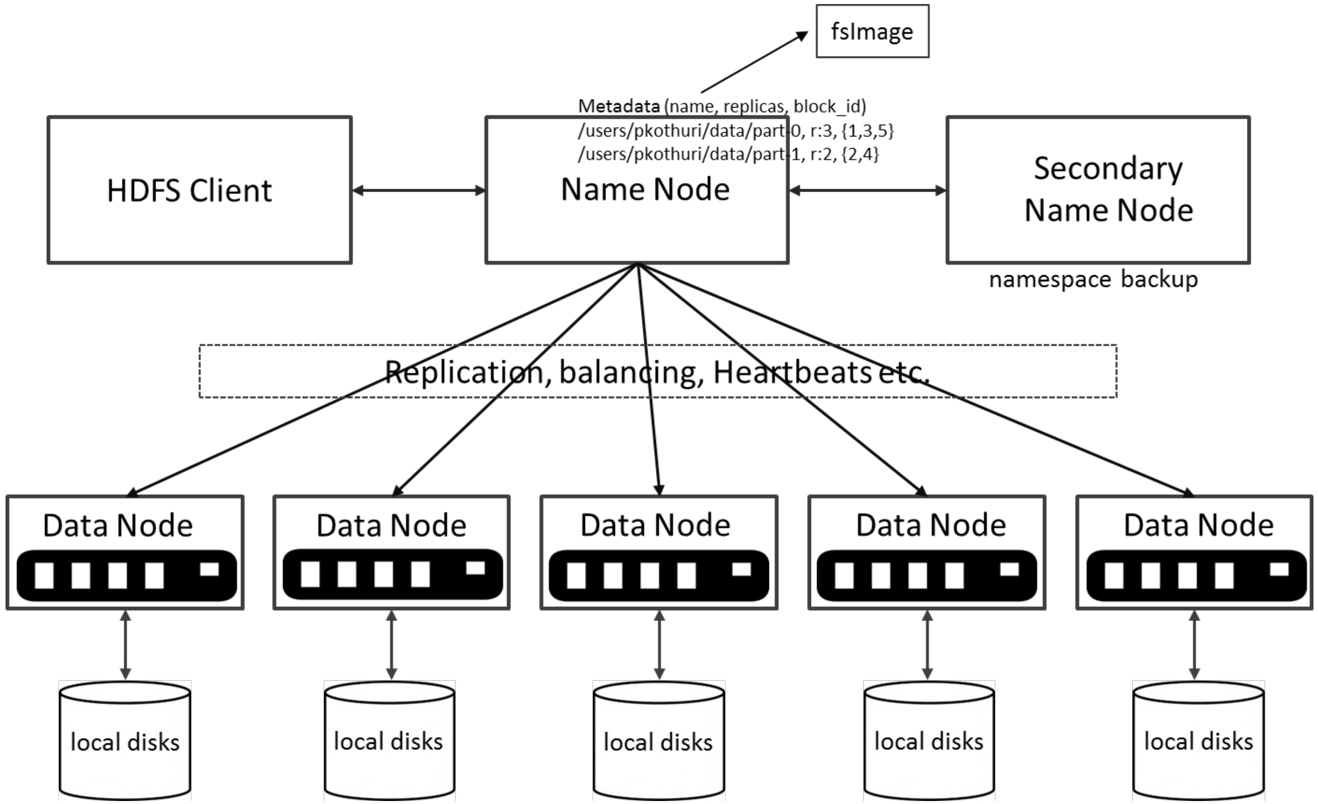
HDFS

- Design Assumptions
  - Hardware failure is the norm.
  - Streaming data access.
  - Write once, read many times.
  - High throughput, not low latency.
  - Large files.
- Characteristics:
  - Performs best with modest number of large files
  - Optimized for streaming reads
  - Layer on top of native file system.

HDFS

- Data is organized into file and directories.
- Files are divided into blocks and distributed to nodes.
- Block placement is known at the time of read
  - Computation moved to same node.
- Replication is used for:
  - Speed
  - Fault tolerance
  - Self healing.

HDFS Architecture



DataNode

- **A Block Server**
  – Stores data in the local file system (e.g. ext3)
  – Stores meta-data of a block (e.g. CRC)
  – Serves data and meta-data to Clients
- **Block Report**
  – Periodically sends a report of all existing blocks to the NameNode
- **Facilitates Pipelining of Data**
  – Forwards data to other specified DataNodes

NameNode Metadata

- **Meta-data in Memory**
  - The entire metadata is in main memory
  - No demand paging of meta-data
- **Types of Metadata**
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g creation time, replication factor
- **A Transaction Log**
  - Records file creations, file deletions. etc

HDFS – User Commands (dfs)

List directory contents

```
hdfs dfs -ls
hdfs dfs -ls /
hdfs dfs -ls -R /var
```

Display the disk space used by files

```
hdfs dfs -du /hbase/data/hbase/namespace/
hdfs dfs -du -h /hbase/data/hbase/namespace/
hdfs dfs -du -s /hbase/data/hbase/namespace/
```

HDFS – User Commands (dfs)

Copy data to HDFS

```
hdfs dfs -mkdir tdata
hdfs dfs -ls
hdfs dfs -copyFromLocal tutorials/data/geneva.csv tdata
hdfs dfs -ls -R
```

Copy the file back to local filesystem

```
cd tutorials/data/
hdfs dfs -copyToLocal tdata/geneva.csv geneva.csv.hdfs
md5sum geneva.csv geneva.csv.hdfs
```

HDFS – User Commands (acls)

List acl for a file

```
hdfs dfs -getfacl tdata/geneva.csv
```

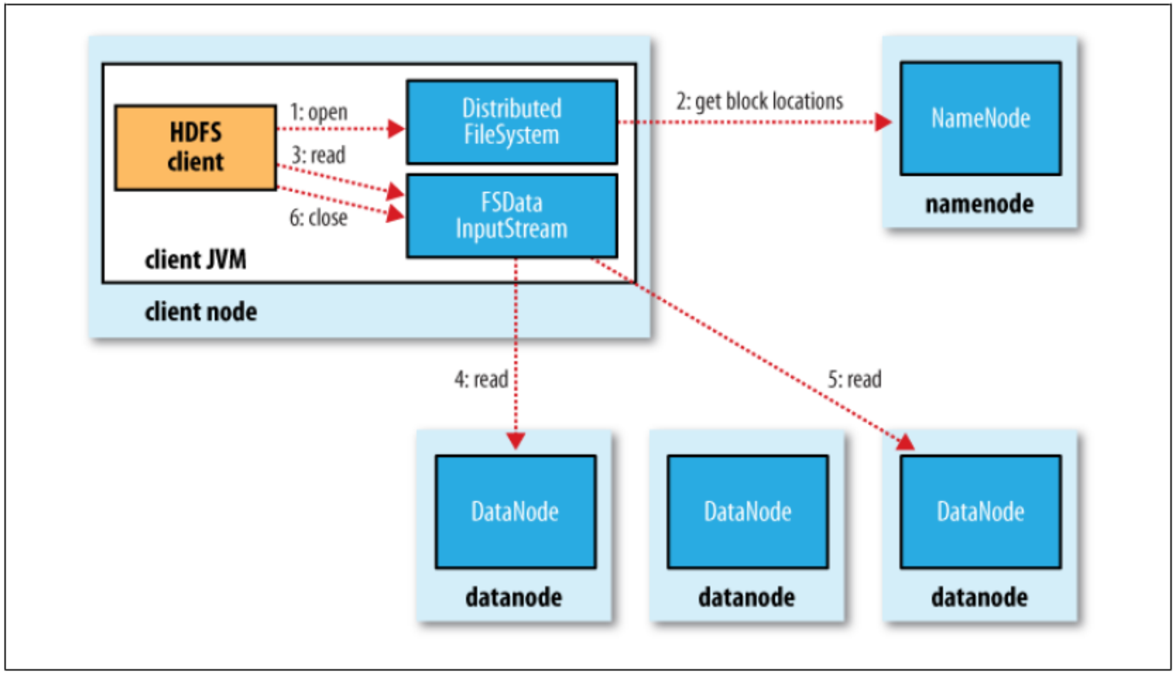
List the file statistics – (%r – replication factor)

```
hdfs dfs -stat "%r" tdata/geneva.csv
```

Write to hdfs reading from stdin

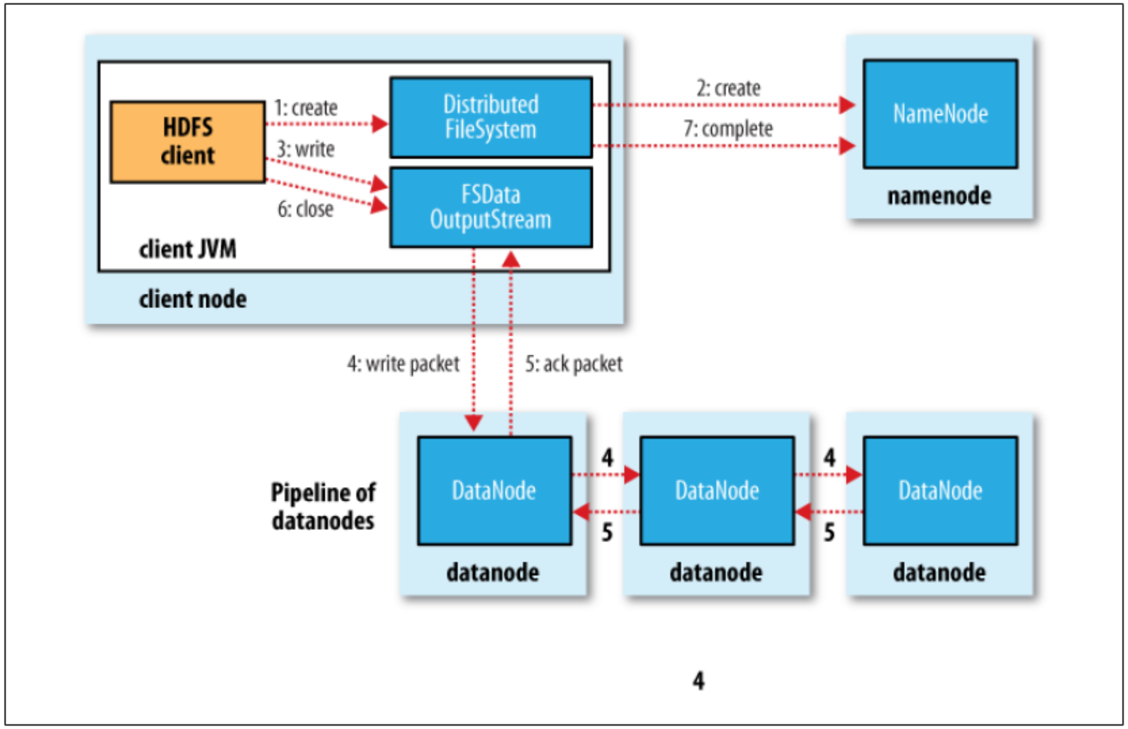
```
echo "blah blah blah" | hdfs dfs -put - tdataset/tfile.txt
hdfs dfs -ls -R
hdfs dfs -cat tdataset/tfile.txt
```

HDFS read client



Source: Hadoop: The Definitive Guide

HDFS write Client



Source: Hadoop: The Definitive Guide

Block Placement

- **Current Strategy**

  -- One replica on local node

  -- Second replica on a remote rack

  -- Third replica on same remote rack

  -- Additional replicas are randomly placed

- **Clients read from nearest replica**

- **Would like to make this policy pluggable**

NameNode Failure

- **A single point of failure**
- **Transaction Log stored in multiple directories**
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)

Data Pipelining

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
- The first DataNode forwards the data to the next DataNode in the Pipeline
- Usually, when all replicas are written, the Client moves on to write the next block in file

Conclusion:

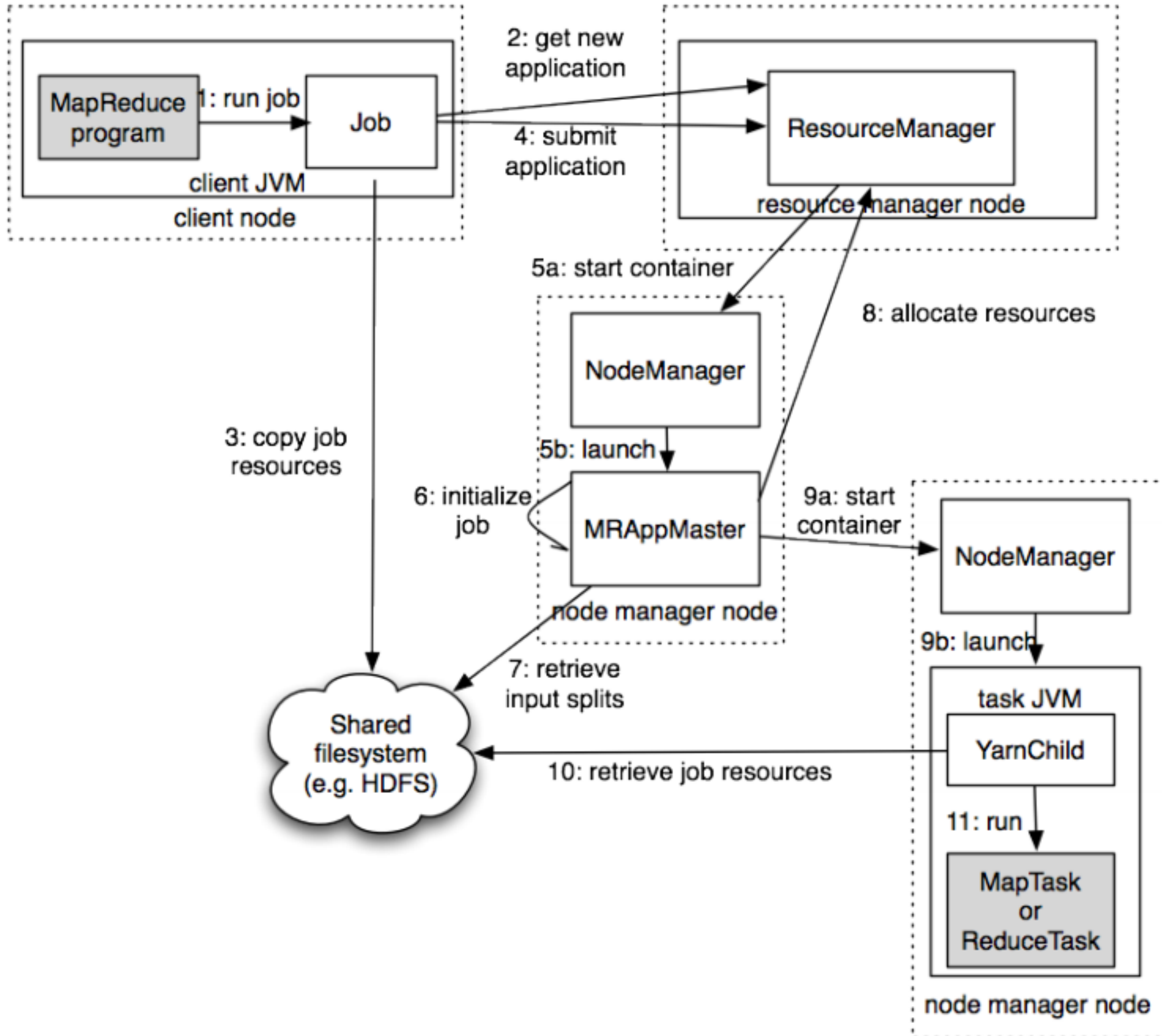
- We have seen:
  - The structure of HDFS.
  - The shell commands.
  - The architecture of HDFS system.
  - Internal functioning of HDFS.

MAPREDUCE INTERNALS

Hadoop Map Reduce

- Provides:
  - Automatic parallelization and Distribution
  - Fault Tolerance
  - Methods for interfacing with HDFS for colocation of computation and storage of output.
  - Status and Monitoring tools
  - API in Java
  - Ability to define the mapper and reducer in many languages through Hadoop streaming.

Hadoop(v2) MR job



Source: Hadoop: The Definitive Guide

Wordcount program

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

Wordcount program - Main

```
public class WordCount {

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
} }
```

Wordcount program - Mapper

```
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, one);
   }
}
}
```

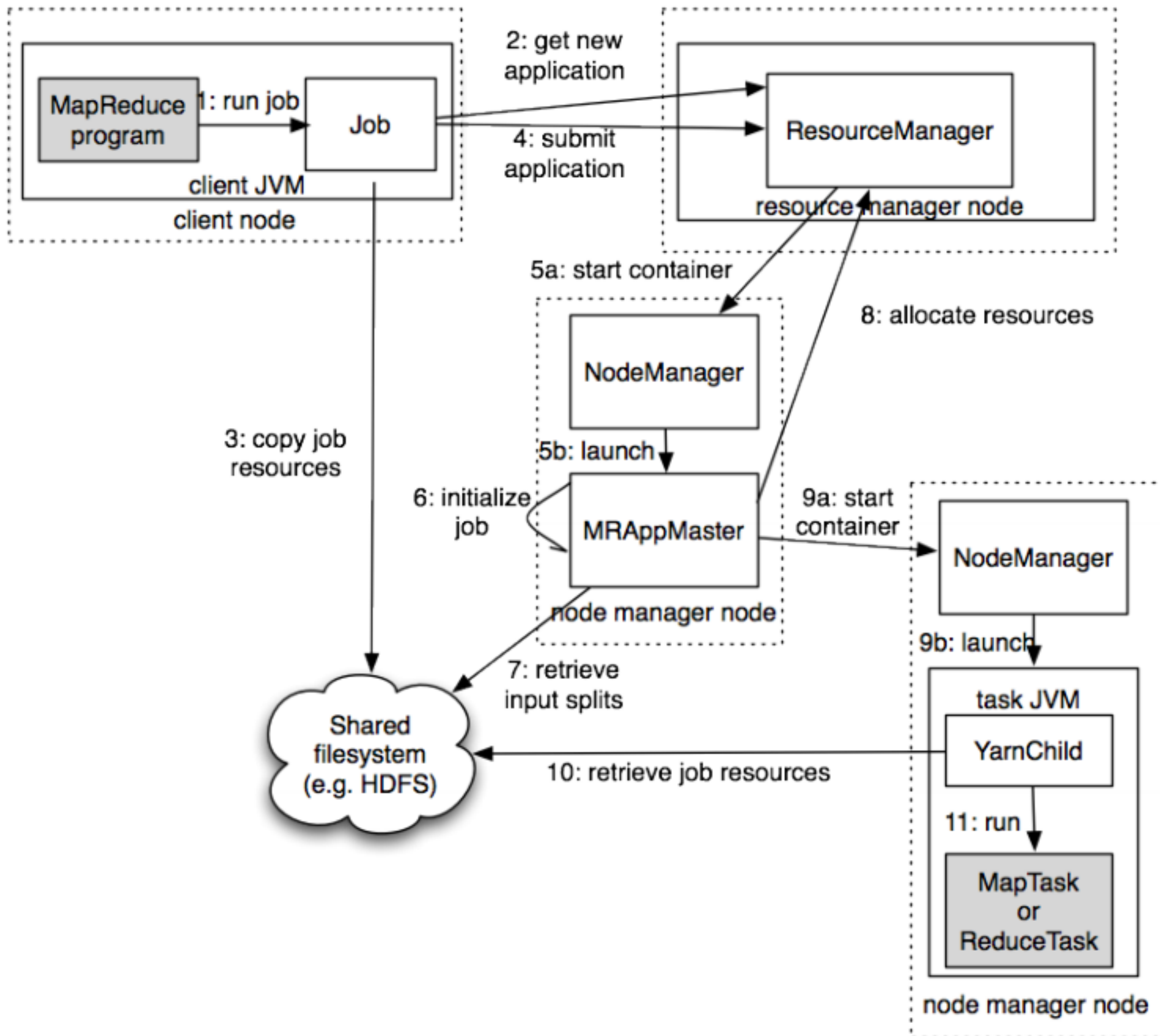
Wordcount program - Reducer

```
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);
}
}
```

Wordcount program - running

```
export JAVA_HOME=[ Java home directory ]

bin/hadoop com.sun.tools.javac.Main WordCount.java

jar cf wc.jar WordCount*.class

bin/hadoop jar wc.jar WordCount [Input path] [Output path]
```

Hadoop(v2) MR job



Source: Hadoop: The Definitive Guide

Wordcount in python

Mapper.py

```
#!/usr/bin/env python

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)
```

Wordcount in python

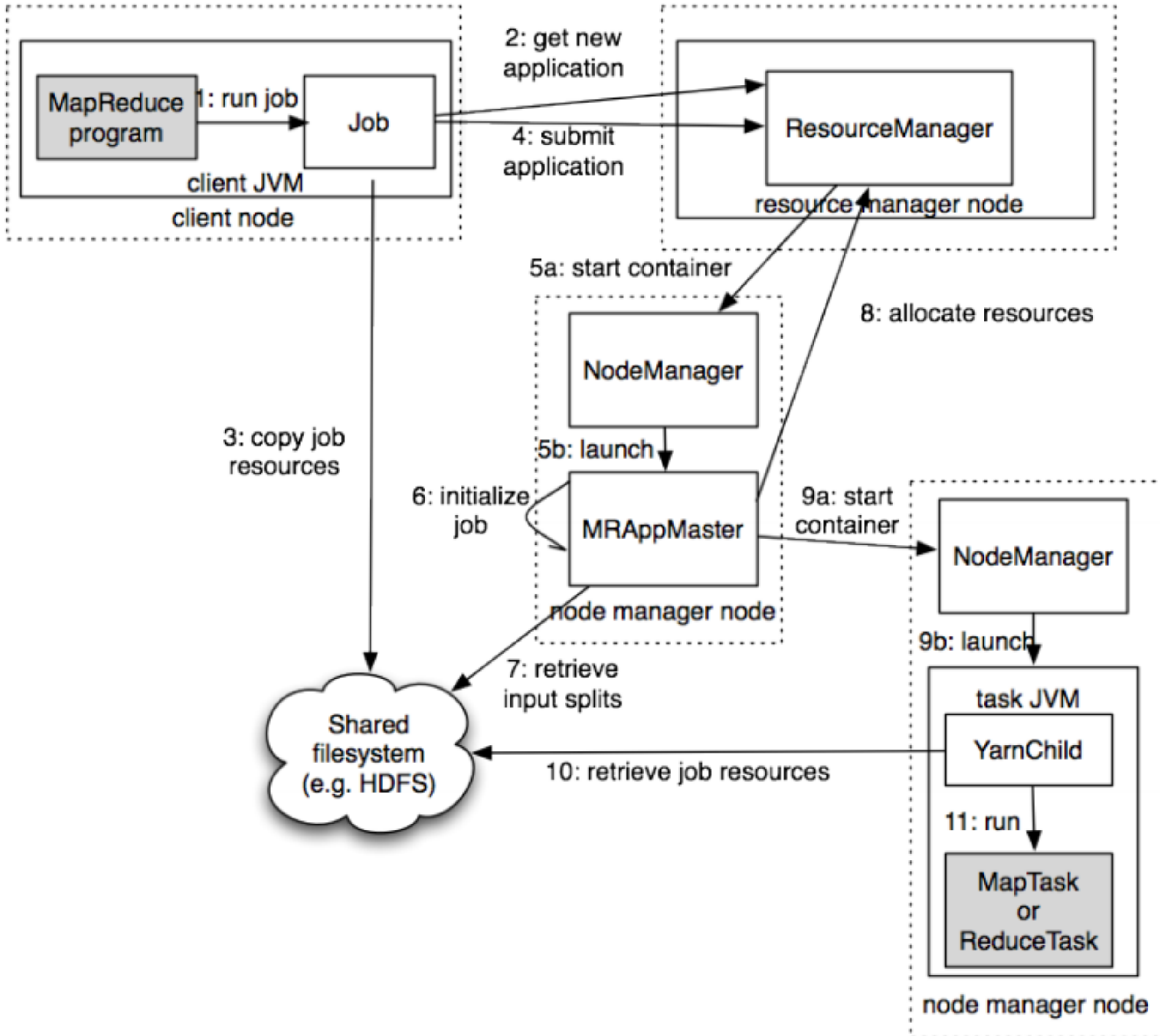
Reducer.py

```
#!/usr/bin/env python

from operator import itemgetter
import sys

# maps words to their counts
word2count = {}

# 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)
        word2count[word] = word2count.get(word, 0) + count
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        pass

# sort the words lexigraphically;
#
# this step is NOT required, we just do it so that our
# final output will look more like the official Hadoop
# word count examples
sorted_word2count = sorted(word2count.items(), key=itemgetter(0))

# write the results to STDOUT (standard output)
for word, count in sorted_word2count:
    print '%s\t%s'% (word, count)
```

Execution code

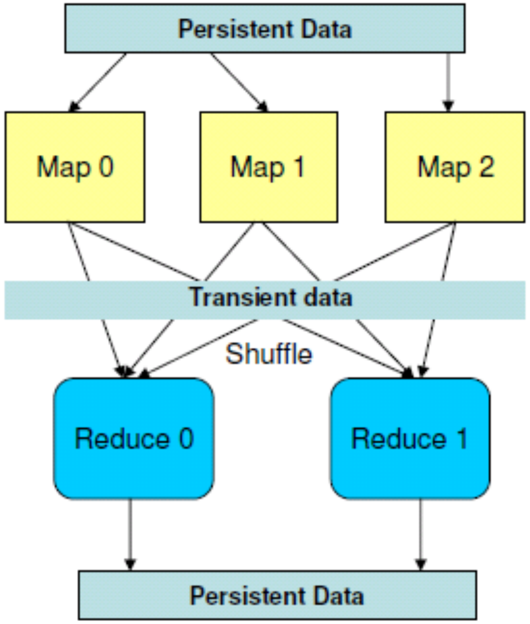
```
bin/hadoop dfs -ls

bin/hadoop dfs -copyFromLocal example example

bin/hadoop jar contrib/streaming/hadoop-0.19.2-streaming.jar -file
wordcount-py.example/mapper.py -mapper wordcount-
py.example/mapper.py -file wordcount-py.example/reducer.py -reducer
wordcount-py.example/reducer.py -input example -output java-output

bin/hadoop dfs -cat java-output/part-00000

bin/hadoop dfs -copyToLocal java-output/part-00000 java-output-local
```

Hadoop(v2) MR job

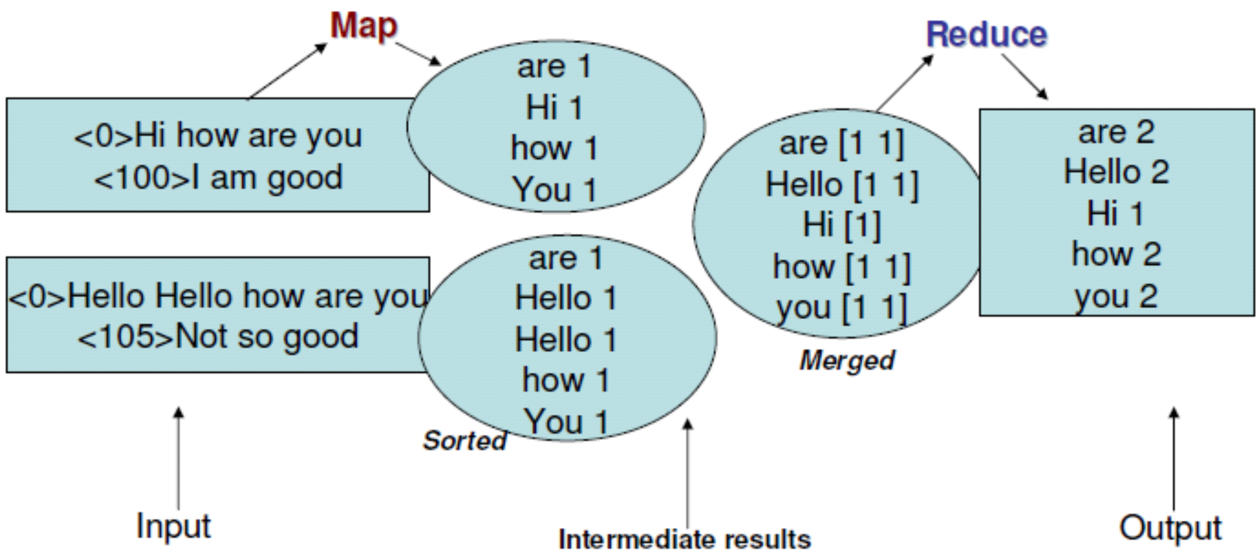


Source: Hadoop: The Definitive Guide

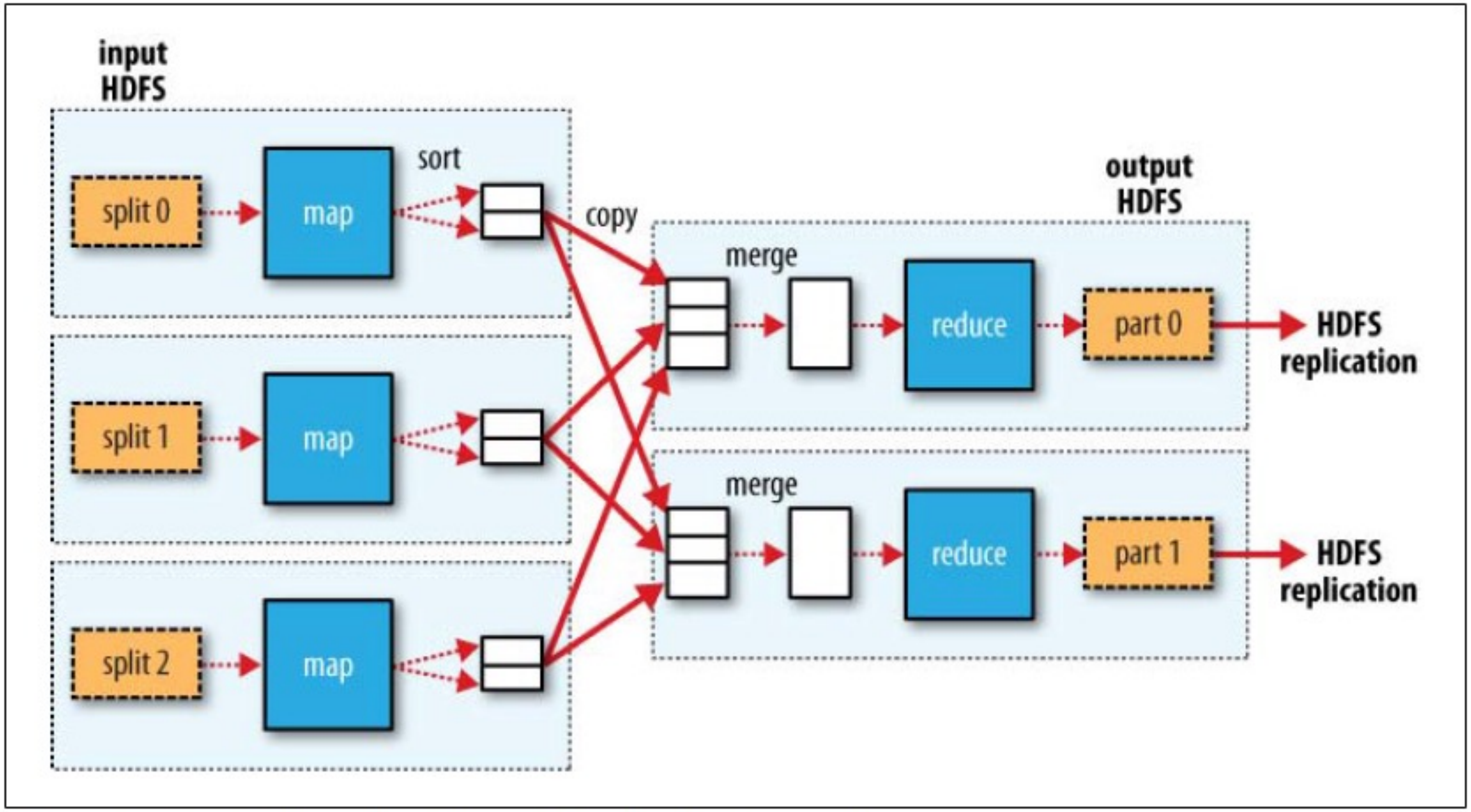
Map Reduce Data Flow



Data: Stream of keys and values

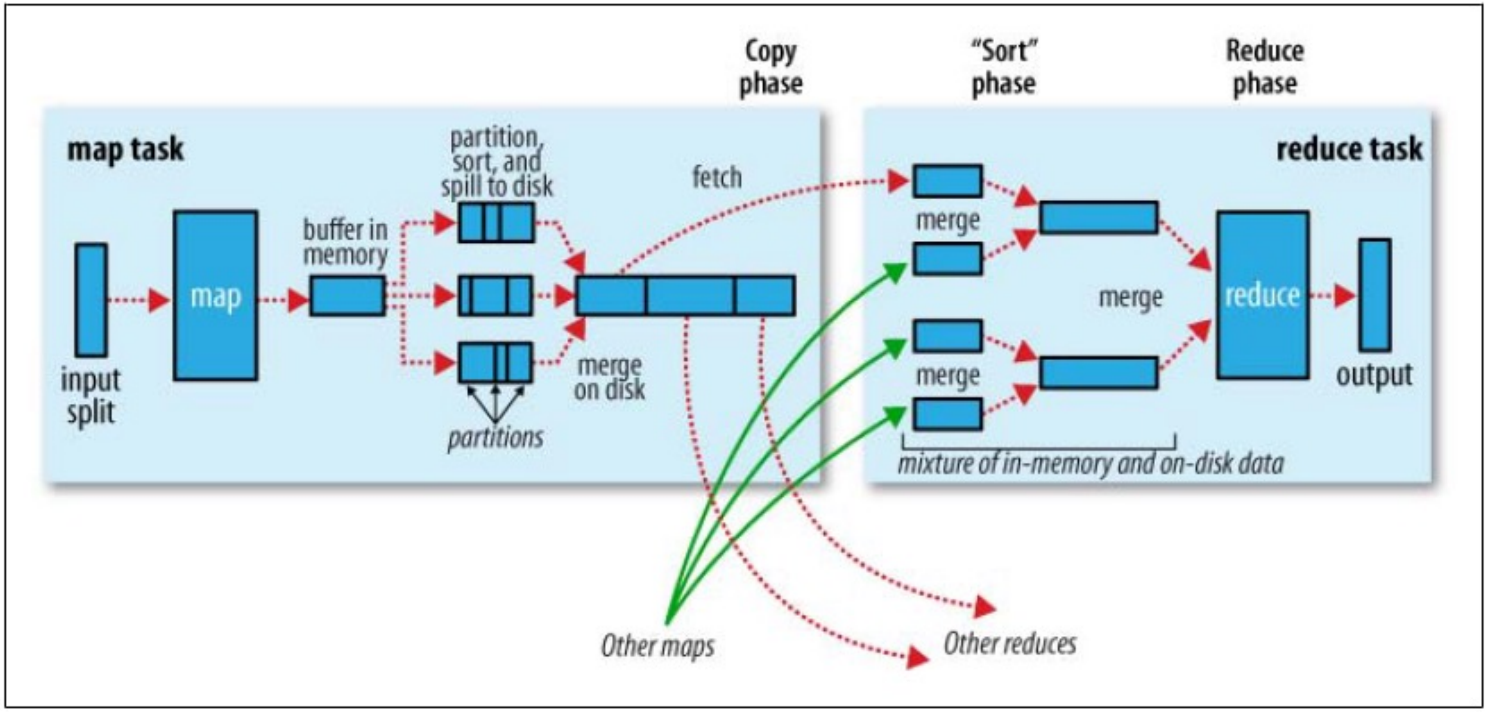


Hadoop MR Data Flow



Source: Hadoop: The Definitive Guide

Shuffle and sort

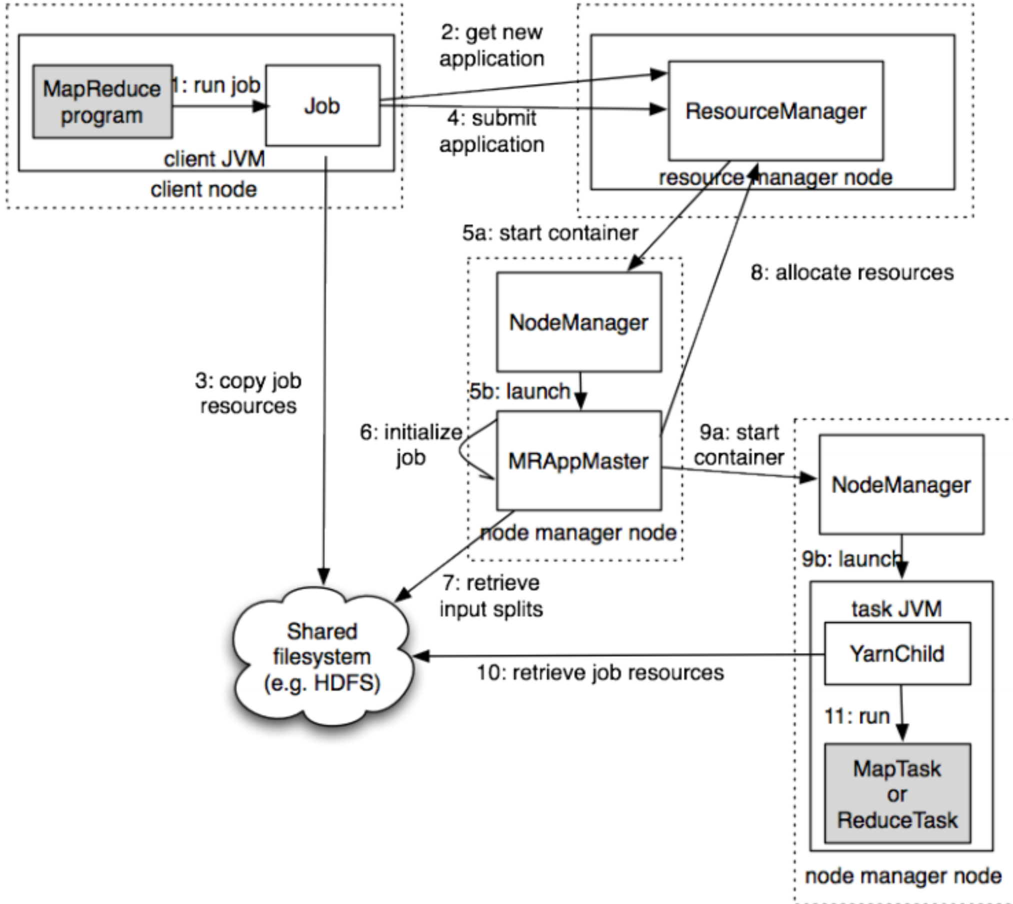


Source: Hadoop: The Definitive Guide

Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- **Intermediate results are stored on local FS of Map workers.**
- **Output** of **Reduce** workers are stored on a **distributed file system**.
- **Output is often input to another MapReduce task**

Hadoop(v2) MR job



Source: Hadoop: The Definitive Guide

Fault tolerance

- Comes from scalability and cost effectiveness
- HDFS:
  - Replication
- Map Reduce
  - Restarting failed tasks: map and reduce
  - Writing map output to FS
  - Minimizes re-computation

Coordination: Master

- **Master node takes care of coordination:**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Failures

- Task failure
  - Task has failed – report error to node manager, appmaster, client.
  - Task not responsive, JVM failure – Node manager restarts tasks.
- Application Master failure
  - Application master sends heartbeats to resource manager.
  - If not received, the resource manager retrieves job history of the run tasks.
- Node manager failure

Dealing with Failures

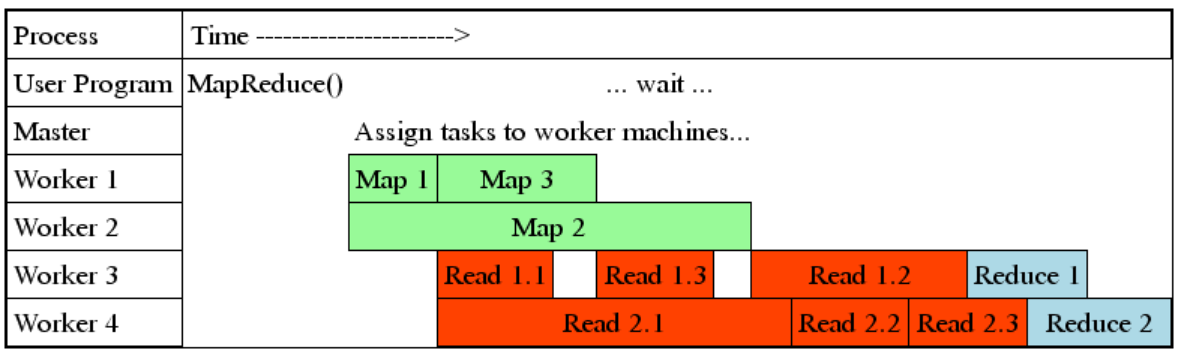
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing

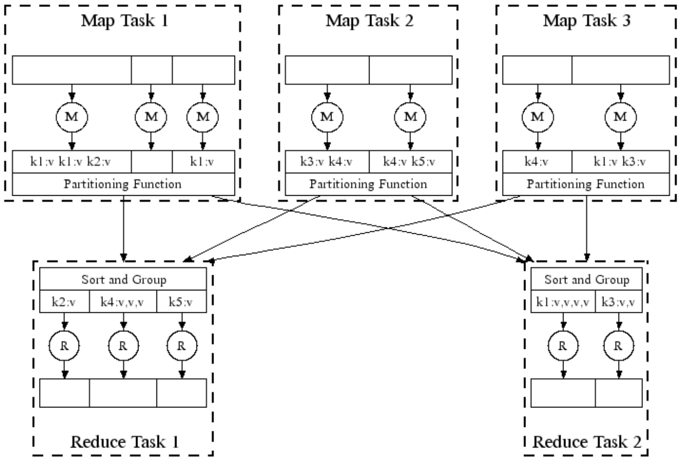


Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”
- **Effect**
  - Dramatically shortens job completion time

Refinement: Combiners

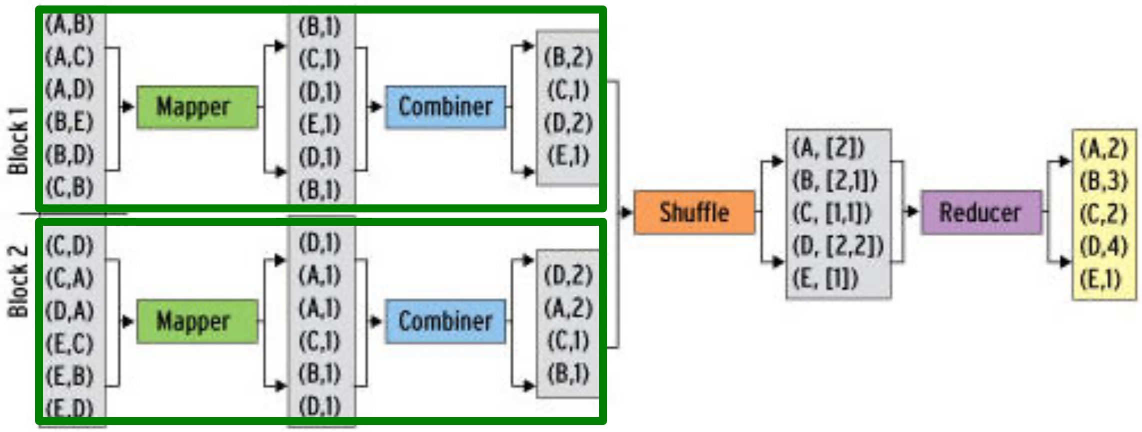
- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2)$, … for the same key k
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function



- Works only if reduce function is commutative and associative

Refinement: Combiners

- **Back to our word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Partition Function

- **Want to control how keys get partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker

- **System uses a default partition function:**
  - **hash(key) mod *R***

- **Sometimes useful to override the hash function:**
  - E.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

References:

- Jure Leskovec, Anand Rajaraman, Jeff Ullman. **Mining of Massive Datasets.** *2nd edition. - Cambridge University Press.* *http://www.mmds.org/*

- Tom White. **Hadoop: The definitive Guide.** Oreilly Press.