Big Data Analytics

C. Distributed Computing Emvironments / C.1 Map Reduce

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute for Computer Science
University of Hildesheim, Germany

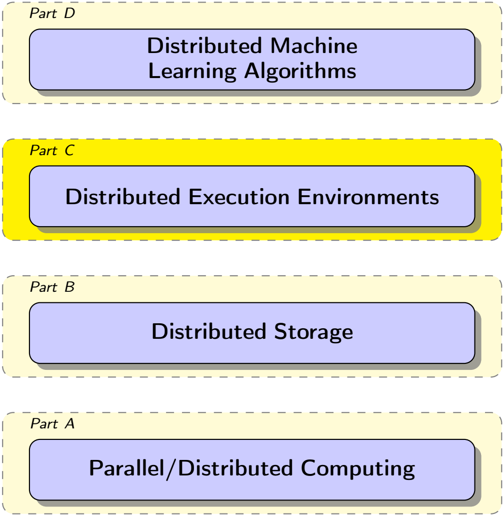
Syllabus

Tue.	9.4.	(1)	0. Introduction
			A. Parallel Computing
Tue.	16.4.	(2)	A.1 Threads
Tue.	23.4.	(3)	A.2 Message Passing Interface (MPI)
Tue.	30.4.	(4)	A.3 Graphical Processing Units (GPUs)
			B. Distributed Storage
Tue.	7.5.	(5)	B.1 Distributed File Systems
Tue.	14.5.	(6)	B.2 Partioning of Relational Databases
Tue.	21.5.	(7)	B.3 NoSQL Databases
			C. Distributed Computing Environments
Tue.	28.5.	(8)	C.1 Map-Reduce
Tue.	4.6.	—	— *Pentecoste Break* —
Tue.	11.6.	(9)	C.2 Resilient Distributed Datasets (Spark)
Tue.	18.6.	(10)	C.3 Computational Graphs (TensorFlow)
			D. Distributed Machine Learning Algorithms
Tue.	25.6.	(11)	D.1 Distributed Stochastic Gradient Descent
Tue.	2.7.	(12)	D.2 Distributed Matrix Factorization
Tue.	9.7.	(13)	Questions and Answers

Outline

Outline

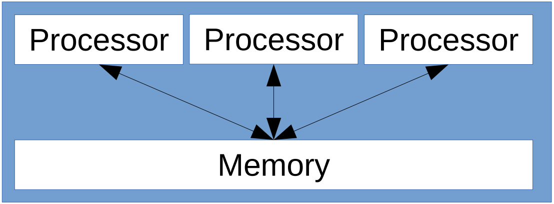
Technology Stack



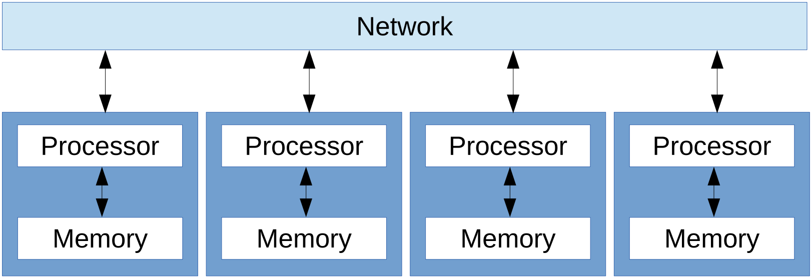
Why do we need a Computational Model?

- Our data is nicely stored in a distributed infrastructure
- We have a number of computers at our disposal
- We want our analytics software to take advantage of all this computing power
- When programming we want to focus on understanding our data and not our infrastructure

Shared Memory Infrastructure



Distributed Infrastructure

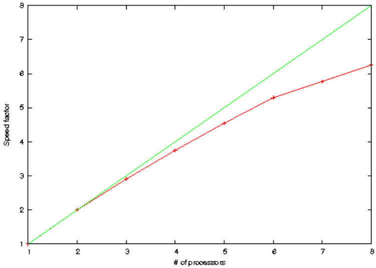


Outline

Parallel Computing / Speedup

- We have p processors available to execute a task T
- Ideally: the more processors the faster a task is executed
- Reality: synchronisation and communication costs
- Speedup $s(T, p)$ of a task T by using p processors:
  - Be $t(T, p)$ the time needed to execute T using p processors
  - **Speedup** is given by:

$$s(T, p) = \frac{t(T, 1)}{t(T, p)}$$



Parallel Computing / Efficiency

- We have p processors available to execute a task T
- **Efficiency** $e(T, p)$ of a task T by using p processors:

$$e(T, p) = \frac{t(T, 1)}{p \cdot t(T, p)}$$

Considerations

- It is not worth using a lot of processors for solving small problems
- Algorithms should increase efficiency with problem size

Outline

Word Count Example

- Given a corpus of text documents

$$D := \{d_1, \ldots, d_n\}$$

each containing a sequence of words:

$$(w_1, \ldots, w_m)$$

from a set W of possible words.

- the task is to generate word counts for each word in the corpus

Paradigms — Shared Memory

- All the processors have access to all counters
- Counters can be overwritten
- Processors need to lock counters before using them

Shared vector for word counts: $c \in \mathbb{N}^{|W|}$
$c \leftarrow \{0\}^{|W|}$

Each processor:

1. access a document $d \in D$
2. for each word w in document d:
    2.1 lock(c_w)
    2.2 $c_w \leftarrow c_w + 1$
    2.3 unlock(c_w)

Paradigms — Shared Memory

- inefficient due to waiting times for the locks
  - the more processors, the less efficient
- in a distributed scenario even worse
  due to communication overhead for acquiring/releasing the locks

Paradigms — Message passing

- Each processor sees only one part of the data

$$\pi(D, p) := \{d_{p\frac{n}{P}}, \ldots, d_{(p+1)\frac{n}{P}-1}\}$$

- Each processor works on its partition

- Results are exchanged between processors (message passing)

For each processor p:

1. For each $d \in \pi(D, p)$
   1.1 process(d)
2. Communicate results

Word Count — Message passing

We need to define two types of processes:

1. **slave**
   - counts the words on a subset of documents and informs the master
2. **master**
   - gathers counts from the slaves and sums them up

Word Count — Message passing

Slave:
Local memory:

subset of documents: $\pi(D, p) := \{d_{p\frac{n}{P}}, \ldots, d_{(p+1)\frac{n}{P}-1}\}$

address of the master: addr_master

local word counts: $c \in \mathbb{R}^{|W|}$

1. $c \leftarrow \{0\}^{|W|}$
2. for each document $d \in \pi(D, p)$
   for each word w in document d:
   $c_w \leftarrow c_w + 1$
3. **Send message** send(addr_master, c)

Word Count — Message passing

Master:

Local memory:

1. **Global word counts:** $c^{\text{global}} \in \mathbb{R}^{|W|}$
2. **List of slaves:** S

$c^{\text{global}} \leftarrow \{0\}^{|W|}$

$s \leftarrow \{0\}^{|S|}$

For each received message (p, c^p)

1. $c^{\text{global}} \leftarrow c^{\text{global}} + c^p$
2. $s_p \leftarrow 1$
3. if $||s||_1 = |S|$ return c^{global}

Paradigms — Message passing

- We need to manually assign master and slave roles for each processor
- Partition of the data needs to be done manually
- Implementations like OpenMPI only provide services to exchange messages

Outline

Map-Reduce

- distributed computing environment
  - introduced 2004 by Google
  - open source reference implementation: **Hadoop** (since 2006)
  - meanwhile supported by many distributed programming environments
    - e.g., in document databases such as **MongoDB**
- builds on a **job scheduler**
  - for Hadoop: yarn
- considers **data is partitioned** over nodes
  - for Hadoop: blocks of a file in a distributed filesystem
- high level abstraction
  - programmer only specifies a **map** and a **reduce** function

Key-Value input data

- Map-Reduce requires the data to be stored in a key-value format
- Examples:

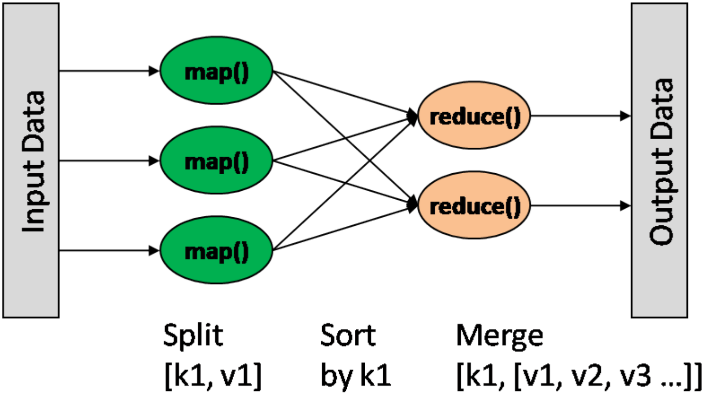
Key	Value
document	array of words
document	word
user	movies
user	friends
user	tweet

Map-Reduce / Idea

- represent input and output data as key/value pairs.
- break down computation into three phases:
  1. **map phase**
     - apply a function **map** to each input key/value pair
     - the result is represented also as key/value pairs
     - execute map on each data node for its data partition (**data locality**)
  2. **shuffle phase**
     - group all intermediate key/value pairs by key to key/valueset pairs
     - repartition the intermediate data by key
  3. **reduce phase**
     - apply a function **reduce** to each intermediate key/valueset pair
     - execute reduce on each node for its intermediate data partition

Note: The **shuffle phase** is also called **sort** or **merge** phase.

Map-Reduce


Input Data
map()
map()
map()
reduce()
reduce()
Output Data
Split
[k1, v1]
Sort
by k1
Merge
[k1, [v1, v2, v3 ...]]

The Paradigm - Formally

Let I be a set called **input keys**,
O be a set called **output keys**,
X be a space called **input values**,
V be a space called **intermediate values**,
Y be a space called **output values**

A function

$$m : I \times X \rightarrow (O \times V)^*$$

is called **map**,
a function

$$r : O \times (V^*) \rightarrow O \times Y$$

reducer.

Note: As always, * denotes sequences.

Map-Reduce Driver Algorithm

map-reduce$(m, r, p, (D_w)_{w \in W})$:
- in parallel on workers $w \in W$:
  - $E := \emptyset$
  - for $(i, x) \in D_w$:
    - $E := E \cup m(i, x)$
  - $E' := \text{dict}(\text{default} = \emptyset)$
  - for $(o, v) \in E$:
    - $E'[o] := E'[o] \cup \{v\}$
  - synchronize all workers
  - for $(o, vs) \in E'$:
    - send (o, vs) to worker $p(o)$
  - $F := \text{dict}(\text{default} = \emptyset)$
  - for all (o, vs) received:
    - $F(o) := F(o) \cup vs$
  - synchronize all workers
  - $G_w := \emptyset$
  - for all $(o, \text{vs}) \in F$:
    - $G_w := G_w \cup r(o, \text{vs})$

where

- m a mapper,
- r a reducer,
- $p : O \rightarrow W$ a partition function for the output keys,
- $(D_w)_{w \in W}$ a dataset partition stored on worker w

result:

- dataset G, stored distributed on workers W

Word Count Example

Map:

- Input: document-word list pairs
- Output: word-count pairs

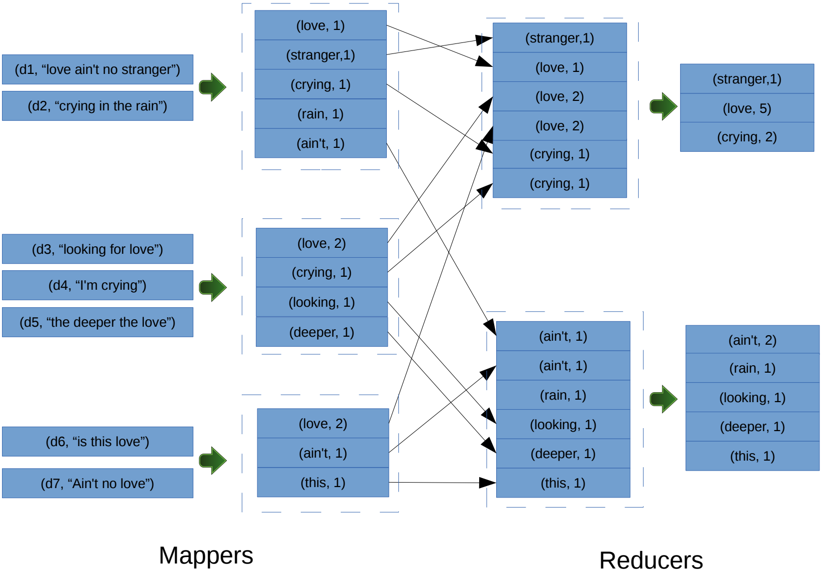
$$(d_n, (w_1, \ldots, w_M)) \mapsto ((w, c_w)_{w \in W : c_w > 0})$$

Reduce:

- Input: word-(count list) pairs
- Output: word-count pairs

$$(w, (c_1, \ldots, c_K)) \mapsto (w, \sum_{k=1}^{K} c_k)$$

Word Count Example



Hadoop Example / Map

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

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter)
        throws IOException {

        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```

Hadoop Example / Reduce

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

      public void reduce(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));
      }
}
```

Hadoop Example / Main

```
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}
```

Execution

- mappers are executed in parallel
- reducers are executed in parallel
  - started after all mappers have completed
- high-level abstraction:
  - No need to worry about how many processors are available
  - No need to specify which ones will be mappers and which ones will be reducers

Fault Tolerance

- **map failure**
  - re-execute map
    - preferably on another node
  - speculative execution
    - execute two mappers on each data segment in parallel each
    - keep results from the first
    - kill the slower one, once the other completed.
- **node failure**
  - re-execute completed and in-progress map()
  - re-execute in-progress reduce tasks
- **particular key-value pairs** that cause mappers to crash
  - skip just the problematic pairs

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T,1) = wD$
- Time for producing intermediate data after the *map* phase: $t(T^{\text{inter}},1) = \sigma D$
- Overheads:
  - intermediate data per mapper: $\frac{\sigma D}{p}$
  - each of the p reducers needs to read one p-th of the data written by each of the p mappers:
$$\frac{\sigma D}{p}\frac{1}{p}p = \frac{\sigma D}{p}$$
- Time for performing the task with Map-reduce:

$$t_{MR}(T,p) = \frac{wD}{p} + 2K\frac{\sigma D}{p}$$

Note: K represents the overhead of IO operations (reading and writing data to disk)

Parallel Efficiency of Map-Reduce

- Time for performing the task in one processor: wD
- Time for performing the task with p processors on Map-reduce:

$$t_{MR}(T,p) = \frac{wD}{p} + 2K\frac{\sigma D}{p}$$

- Efficiency of Map-Reduce:

$$\begin{aligned} e_{MR}(T,p) &= \frac{t(T,1)}{p \cdot t(T,p)} \\ &= \frac{wD}{p(\frac{wD}{p} + 2K\frac{\sigma D}{p})} \\ &= \frac{1}{1 + 2K\frac{\sigma}{w}} \end{aligned}$$

Parallel Efficiency of Map-Reduce

$$e_{MR}(T, p) = \frac{1}{1 + 2K\frac{\sigma}{w}}$$

- Apparently the efficiency is independent of p
- High speedups can be achieved with large number of processors
- If σ is high (too much intermediate data) the efficiency deteriorates
- In many cases σ depends on p

Summary

- Map Reduce is a **distributed computing framework**.
- Map Reduce represents input and output **data as key/value pairs**.
- Map Reduce decomposes computation into three phases:
  1. **map**: applying a function to **each input key/value pair**.
  2. **shuffle**:
     - **grouping intermediate data** into key/valueset pairs
     - **repartitioning intermediate data** by key over nodes
  3. **reduce**: applying a function to **each intermediate key/valueset pair**.
- Mappers are executed **data local**
- For a program, the **map** and **reduce** functions have to be specified.
  - the shuffle step is fixed.
- The size of the intermediate data is crucial for efficiency as it has to be repartioned.

Further Readings

- original Map Reduce framework by Google:
  - Dean and Ghemawat [2004, 2008, 2010]
- MapReduce reference implementation in Hadoop:
  - [White, 2015, ch. 2, 7]
  - also ch. 6, 8 and 9.
- MapReduce in a document database, MongoDB:
  - [Chodorow, 2013, ch. 7]

Outline

Mappers

- extend baseclass **Mapper<KI,VI,KO,VO>** (org.apache.hadoop.mapreduce)
  - types: KI = input key, VI = input value,
    KO = output key, VO = output value.

Mappers

- extend baseclass **Mapper<KI,VI,KO,VO>** (org.apache.hadoop.mapreduce)
  - types: KI = input key, VI = input value,
    KO = output key, VO = output value.

- overwrite **map(KI key, VI value, Context ctxt)**
  - **Context** is an inner class of **Mapper<KI,VI,KO,VO>**
    - **write(KO,VO)**: write next output pair

Mappers

- extend baseclass **Mapper<KI,VI,KO,VO>** (org.apache.hadoop.mapreduce)
  - types: KI = input key, VI = input value,
    KO = output key, VO = output value.

- overwrite **map(KI key, VI value, Context ctxt)**
  - **Context** is an inner class of **Mapper<KI,VI,KO,VO>**
    - **write(KO,VO)**: write next output pair

- optionally,
  setup(Context ctxt): set up mapper
  - called once before first call to **map**

- **cleanup(Context ctxt)**: clean up mapper
  - called once after last call to **map**

Key and Value Types

Requirements for key and value types:

- serializable — **Writable** (org.apache.hadoop.io)
  - To store the output of mappers, combiners and reducers in files

Key and Value Types

Requirements for key and value types:

- serializable — **Writable** (org.apache.hadoop.io)
  - To store the output of mappers, combiners and reducers in files

Additional requirements for key types:

- comparable — **Comparable** (java.lang)
  - To sort the output of mappers and combiners by key

Key and Value Types

Requirements for key and value types:

- serializable — **Writable** (org.apache.hadoop.io)
  - To store the output of mappers, combiners and reducers in files

Additional requirements for key types:

- comparable — **Comparable** (java.lang)
  - To sort the output of mappers and combiners by key
- stable **hashcode** across different JVM instances
  - To partition the output of combiners by key.
  - The default implementation in **Object** is not stable!

Key and Value Types

Requirements for key and value types:

- serializable — **Writable** (org.apache.hadoop.io)
  - To store the output of mappers, combiners and reducers in files

Additional requirements for key types:

- comparable — **Comparable** (java.lang)
  - To sort the output of mappers and combiners by key
- stable **hashcode** across different JVM instances
  - To partition the output of combiners by key.
  - The default implementation in **Object** is not stable!
- pooled in interface **WritableComparable** (org.apache.hadoop.io)

Serialization

To store the output of mappers, combiners and reducers in files, keys and values have to be **serialized**:

- hadoop requires all keys and values of a step to be of the same class.
  $\rightsquigarrow$ the class of an object to deserialize is known in advance.

- standard Java serialization (**java.lang.Serializable**) serializes class information, thus is more verbose and complex and therefore not used in hadoop.

- new interface **Writable** (org.apache.hadoop.io):
  - **void write(DataOutput out) throws IOException**:
    write object to a data output.
  - **void readFields(DataInput in) throws IOException**:
    set member variables (fields) of an object to values read from a data input.
  - standard **DataInput** and **DataOutput** (java.io)

Serialization (2/2)

- **Writable** wrappers for elementary data types:

BooleanWritable	boolean
ByteWritable	byte
DoubleWritable	double
FloatWritable	float
IntWritable	int
LongWritable	long
ShortWritable	short
Text	String

  (all in org.apache.hadoop.io)

  - these are not subclasses of the default wrappers **Integer**, **Double** etc. (as the latter are final)

Serialization (2/2)

- **Writable** wrappers for elementary data types:

BooleanWritable	boolean
ByteWritable	byte
DoubleWritable	double
FloatWritable	float
IntWritable	int
LongWritable	long
ShortWritable	short
Text	String

  (all in org.apache.hadoop.io)
  - these are not subclasses of the default wrappers **Integer**, **Double** etc. (as the latter are final)

- If one needs to pass more complex objects between steps, implement custom **Writable** (in terms of these elementary **Writable**s).

Example 1 / Mapper (in principle)

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

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TokenizerMapperS
    extends Mapper<Object, Text, Text, IntWritable>{

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens())
        context.write(new Text(itr.nextToken()), new IntWritable(1));
    }
}
```

Example 1 / Mapper

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

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public 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);
        }
    }
}
```

Job Configuration

Configuration (org.apache.hadoop.conf):

- default constructor:
  read default configuration from files
  - core-default.xml and
  - core-site.xml

  (to be found in the classpath).
- **addResource(Path file)**:
  update configuration from another configuration file.
- **String get(String name)**:
  get the value of a configuration option.
- **set(String name, String value)**:
  set the value of a configuration option.

Job

Job (org.apache.hadoop.conf):

- constructor **Job(Configuration conf, String name)**:
  create a new job.
- **setMapperClass(Class cls)**, **setCombinerClass(Class cls)**, **setReducerClass(Class cls)**:
  set the class for mappers, combiners and reducers
- **setOutputKeyClass(Class cls)**, **setOutputValueClass(Class cls)**:
  set the class for output keys and values.
- **boolean waitForCompletion(boolean verbose)**:
  submit job and wait until it completes.

Input and Output Paths

FileInputFormat (org.apache.hadoop.mapreduce.lib.input):

- **addInputPath(Job job, Path path)**:
  add input paths

FileOutputFormat (org.apache.hadoop.mapreduce.lib.output):

- **setOutputPath(Job job, Path path)**:
  set output paths

Example 1 / Job Runner (Mapper Only)

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MRJobStarter1 {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setMapperClass(TokenizerMapper.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);
  }
}
```

Compiling and Running Map Reduce Classes

1. Set paths

```
export PATH=PATH : /home/lst/system/hadoop/binexportHADOOP_C LASSPATH =(JAVA_HOME)/lib/to
```

Compiling and Running Map Reduce Classes

1. Set paths

```
export PATH=PATH : /home/lst/system/hadoop/binexportHADOOP_CLASSPATH =(JAVA_HOME)/lib/tc
```

2. Compile sources:

```
hadoop com.sun.tools.javac.Main −sourcepath . MRJobStarter1.java
```

Compiling and Running Map Reduce Classes

1. Set paths

```
export PATH=$PATH:/home/lst/system/hadoop/binexportHADOOP_CLASSPATH=(JAVA_HOME)/lib/tc
```

2. Compile sources:

```
hadoop com.sun.tools.javac.Main -sourcepath . MRJobStarter1.java
```

3. Package all class files of the job into a jar:

```
jar cf job.jar MRJobStarter1.class TokenizerMapper.class
```

Compiling and Running Map Reduce Classes

1. Set paths

```
export PATH=PATH : /home/lst/system/hadoop/binexportHADOOP_CLASSPATH =(JAVA_HOME)/lib/tc
```

2. Compile sources:

```
hadoop com.sun.tools.javac.Main -sourcepath . MRJobStarter1.java
```

3. Package all class files of the job into a jar:

```
jar cf job.jar MRJobStarter1.class TokenizerMapper.class
```

4. Run the jar:

```
hadoop jar job.jar MRJobStarter1 /ex1/input /ex1/output
```

   - the output directory must not yet exist.

Example 1 / Inputs and Output

- input 1:

```
Hello World Bye World
```

- input 2:

```
Hello Hadoop Goodbye Hadoop
```

- output:

```
lst@lst-uni:~> hdfs dfs -ls /ex1/output.ex2
Found 2 items
-rw-r-r-   2 lst supergroup          0 2016-05-24 18:59 /ex1/output.ex2/_SUCCESS
-rw-r-r-   2 lst supergroup         66 2016-05-24 18:59 /ex1/output.ex2/part-r-00000
lst@lst-uni:~> hdfs dfs -cat /ex1/output.ex2/part-r-00000
Bye     1
Goodbye 1
Hadoop  1
Hadoop  1
Hello   1
Hello   1
World   1
World   1
```

Reducers

- extend baseclass **Reducer<KI,VI,KO,VO>** (org.apache.hadoop.mapreduce)
  - types: KI = input key, VI = input value,
    KO = output key, VO = output value.

Reducers

- extend baseclass **Reducer<KI,VI,KO,VO>** (org.apache.hadoop.mapreduce)
  - types: KI = input key, VI = input value,
    KO = output key, VO = output value.

- overwrite **reduce(KI key, Iterable<VI> value, Context ctxt)**
  - **Context** is an inner class of **Reducer<KI,VI,KO,VO>**
    - **write(KO,VO)**: write next output pair

Reducers

- extend baseclass **Reducer<KI,VI,KO,VO>** (org.apache.hadoop.mapreduce)
  - types: KI = input key, VI = input value,
    KO = output key, VO = output value.

- overwrite **reduce(KI key, Iterable<VI> value, Context ctxt)**
  - **Context** is an inner class of **Reducer<KI,VI,KO,VO>**
    - **write(KO,VO)**: write next output pair

- optionally,
  setup(Context ctxt): set up reducer
  - called once before first call to **reduce**

- **cleanup(Context ctxt)**: clean up reducer
  - called once after last call to **reduce**

Example 2 / Reducer

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public 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);
    }
}
```

Example 2 / Job Runner

```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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MRJobStarter2 {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setMapperClass(TokenizerMapper.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);
  }
}
```

Example 2 / Output

```
lst@lst-uni:~> hdfs dfs -cat /ex1/output.2/part*
Bye     1
Goodbye 1
Hadoop  2
Hello   2
World   2
```

Example 3 / Job Runner: Only Mapper and Combiner

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MRJobStarter3 {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(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);
  }
}
```

Example 3 / Output

```
lst@lst-uni:~> hdfs dfs -cat /ex1/output.3/part*
Bye     1
Goodbye 1
Hadoop  2
Hello   1
Hello   1
World   2
```

Example 4 / Job Runner: Combiner and Reducer

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MRJobStarter4 {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    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);
  }
}
```

The output is the same as only with a reducer,
but less intermediate data is moved.

Predefined Mappers

- **Mapper**: (org.apache.hadoop.mapreduce):

$$m(k, v) := ((k, v))$$

- **InverseMapper** (org.apache.hadoop.mapreduce.lib.map):

$$m(k, v) := ((v, k))$$

- **ChainMapper** (org.apache.hadoop.mapreduce.lib.chain)

$$\text{chain}_{m,\ell}(k, v) := (\ell(k', m') \mid (k', v') \in m(k, v))$$

- **TokenCounterMapper** (org.apache.hadoop.mapreduce.lib.map):

$$m(k, v) := ((k', 1) \mid k' \in \text{tokenize}(v))$$

Predefined Mappers (2/2)

- **RegexMapper** (org.apache.hadoop.mapreduce.lib.map):

$$m(k, v) := ((k', 1) \mid k' \in \text{find-regex}(v))$$

Regex pattern to search for and
groups to report
can be set via configuration options

PATTERN	mapreduce.mapper.regex
GROUP	mapreduce.mapper.regexmapper..group

Predefined Reducers

- **Reducer**: (org.apache.hadoop.mapreduce):

$$r(k, V) := (k, V)$$

- **IntSumReducer**, **LongSumReducer** (org.apache.hadoop.mapreduce.lib.reduce):

$$m(k, V) := (v, \sum_{v \in V} v))$$

- **ChainReducer** (org.apache.hadoop.mapreduce.lib.chain)

$$\text{chain}_{r,s}(k, V) := s(r(k, V))$$

Job Default Values

mapper class:	class org.apache.hadoop.mapreduce.Mapper
combiner class:	null
reducer class:	class org.apache.hadoop.mapreduce.Reducer
output key class:	class org.apache.hadoop.io.LongWritable
output value class:	class org.apache.hadoop.io.Text

Further Topics

- Controlling map-reduce jobs
  - YARN
  - chaining map-reduce jobs
  - iterative algorithms
- Controlling the number of mappers and reducers
- Managing resources required by all mappers or reducers
- Input and output using relational databases
- Streaming
- Examples, examples, examples

References

Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly and Associates, Beijing, 2 edition, May 2013. ISBN 978-1-4493-4468-9.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04 Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, volume 6, 2004.

Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107, January 2008. ISSN 00010782. doi: 10.1145/1327452.1327492.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53 (1):72–77, 2010.

Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 4 edition, 2015.