Processing Big Data

Introduction

- How to efficiently process large amount of data?
  - Use many machines
  - Use many cores
- How to efficiently use many machine/cores
  - Use a suitable programming model
- What is a programming model
  - A way to write some program,
  - with access to a limited set of functions,
    - provided by libraries or frameworks
  - and an environment to execute it.

MapReduce

The MapReduce programming model

- Popularized by a paper from Google : **MapReduce: simplified data processing on large clusters (2008)**
- Simple model with 2 basic operations
  - Map
  - Reduce
- Assume data are structured as (key,value)
- Apply successive map and reduce operations
  - Not necessarily limited to 1 map and 1 reduce in theory

Map and Reduce

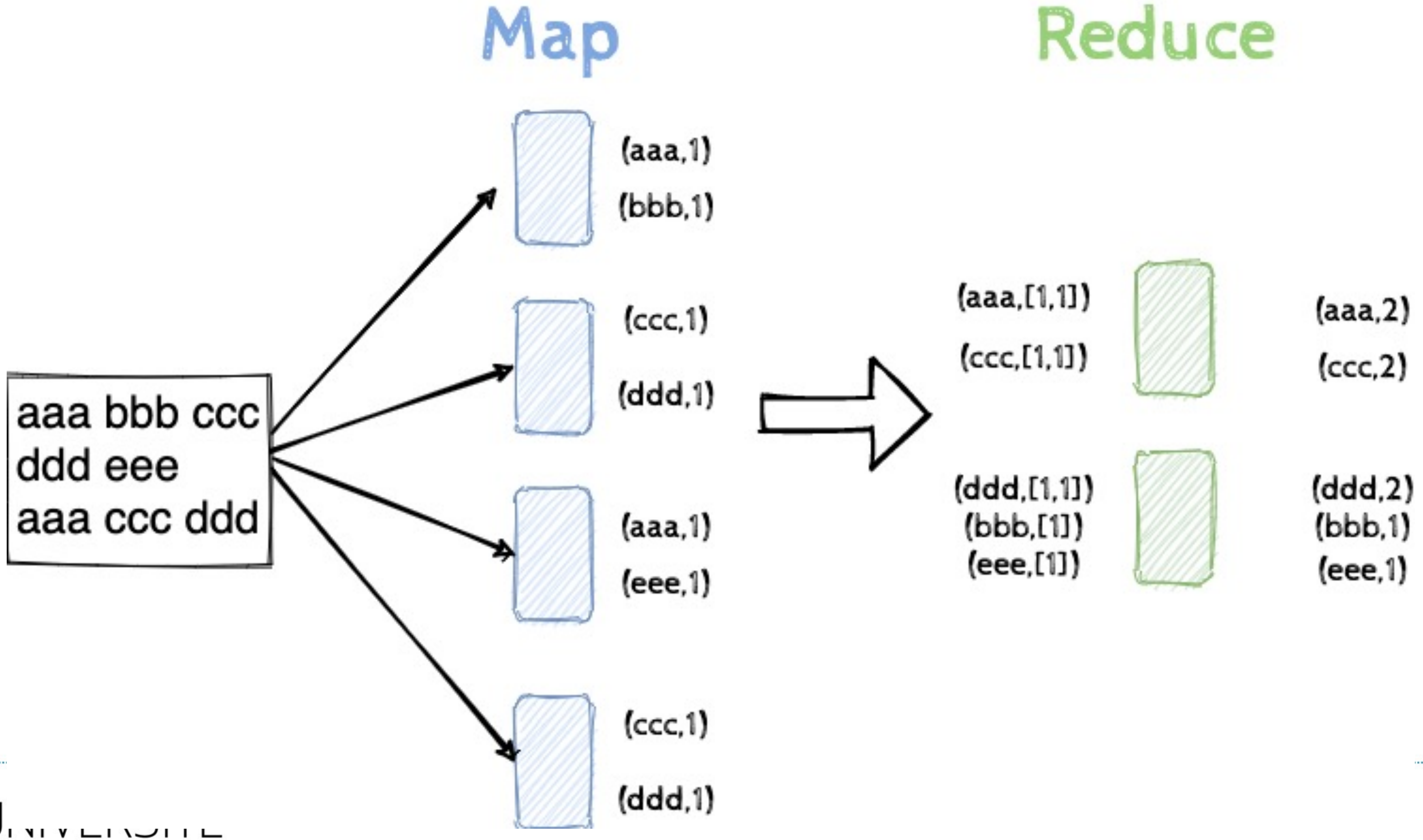
- Map and reduce are functions with defined input-output
- Map :
  - Uses a single (key, value) to produce multiple pairs
  - Input : (key, value) or (key,_)
  - Output : One or many (key, value) pairs

can be different

- Reduce :
  - Gets all values associated with a given key and produce new pairs
  - Input : (key, [value1, value2, …, valueN])
  - Output : One or many (key, value) pairs

Example

- Word count
  - Take a text, produce a list of (word, Nb occurences)
- Map function :
  - Assume each word appears only once
  - Use word as key and add value 1
  - Input : (word, _)
  - Output : (word, 1)
- Reduce function :
  - Sum all ‘1’ for a given key (word)
  - Input : (word, [1,1,1,1,1,1])
  - Output : (word, 6)

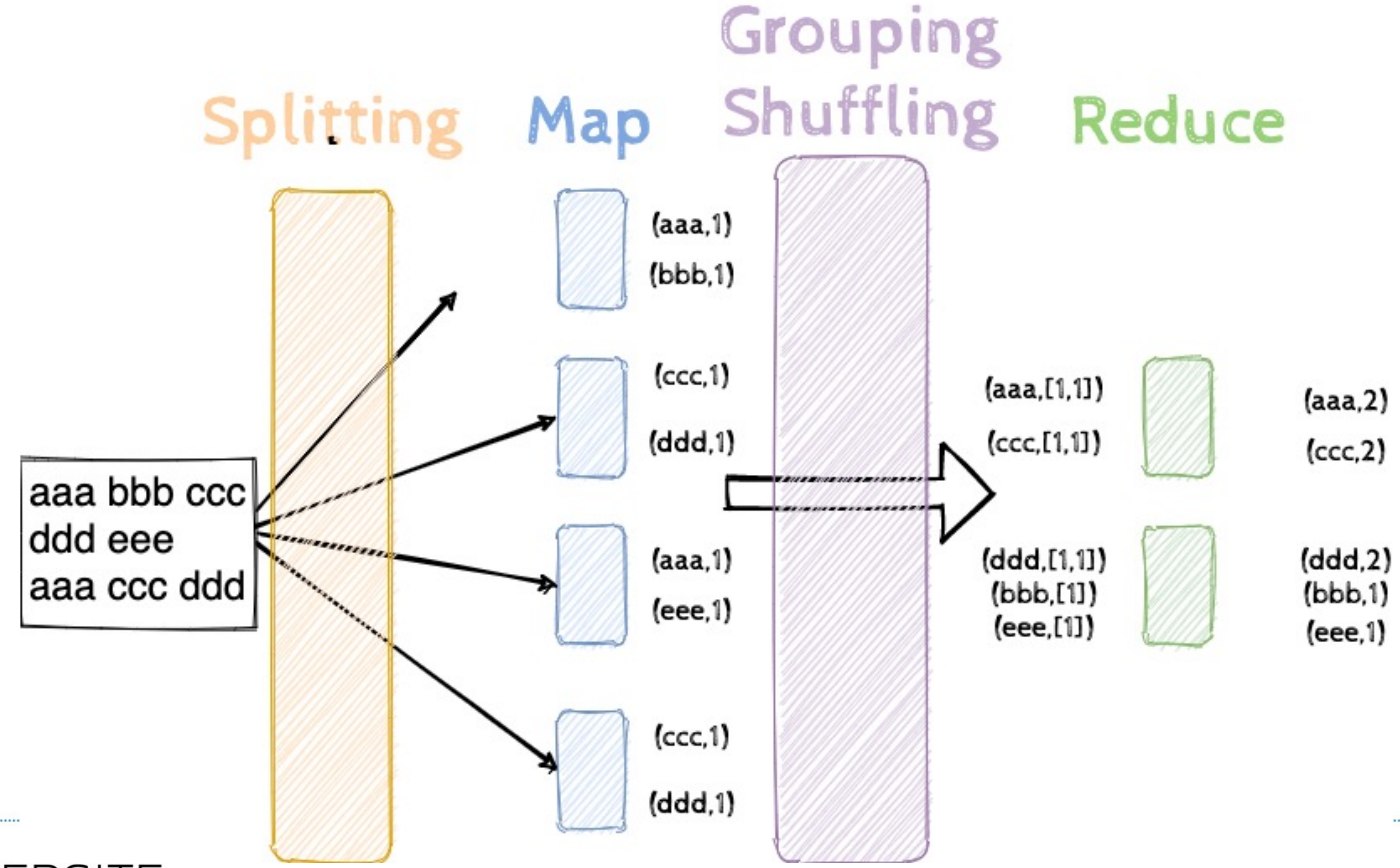

Map
Reduce
aaa bbb ccc
ddd eee
aaa ccc ddd
(aaa,1)
(bbb,1)
(ccc,1)
(ddd,1)
(aaa,1)
(eee,1)
(ccc,1)
(ddd,1)
(aaa,[1,1])
(ccc,[1,1])
(ddd,[1,1])
(bbb,[1])
(eee,[1])
(aaa,2)
(ccc,2)
(ddd,2)
(bbb,1)
(eee,1)

Benefits of MapReduce

- A lot of real world problems can be expressed as MapReduce
- Maps and reduce functions can be executed in parallel
  - Map works on independent data
  - Reduce works on a single key
- Reduce can have inner parallelism
  - If complex, can reduce with multiple threads

Implementations questions

- How is the input split into individual pairs for mappers?
- How are output of map grouped by key and sent to the correct reduce ?
- How is final result written ?
- All these are answered by a framework
  - All follow the same model but implementation may vary
- Basically, a MapReduce application has 4 phases
  - Splitting, Mapping, Grouping/Shuffling, Reducing

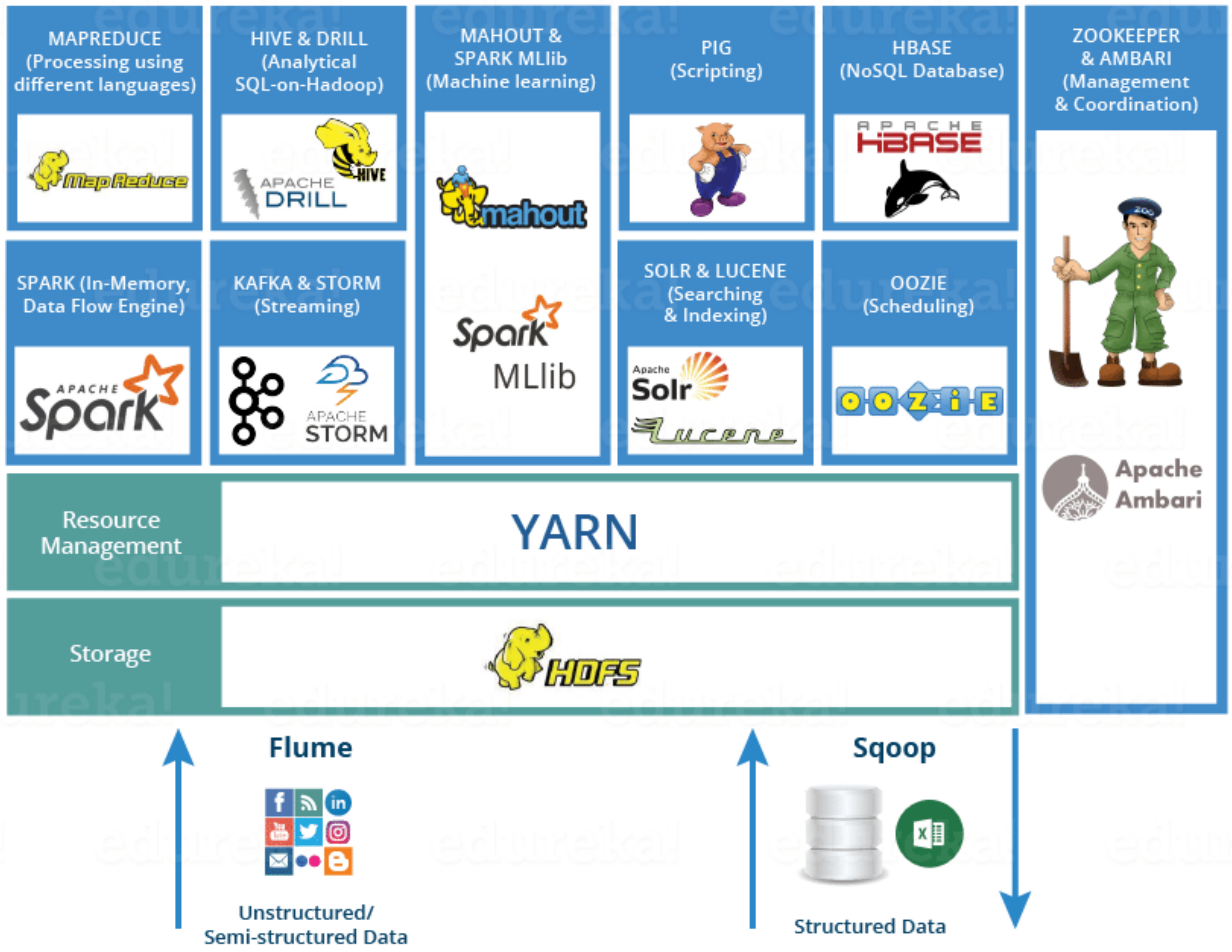

Grouping
Shuffling
Splitting
Map
Reduce
aaa bbb ccc
ddd eee
aaa ccc ddd
(aaa,1)
(bbb,1)
(ccc,1)
(ddd,1)
(aaa,1)
(eee,1)
(ccc,1)
(ddd,1)
(aaa,[1,1])
(ccc,[1,1])
(ddd,[1,1])
(bbb,[1])
(eee,[1])
(aaa,2)
(ccc,2)
(ddd,2)
(bbb,1)
(eee,1)

MapReduce

The Hadoop Framework

Introduction

- Hadoop is an open source implementation of Google MapReduce
- Provides full stack of services/frameworks
  - Not only MapReduce since version 2
- Most important components
  - HDFS
  - YARN
  - MapReduce



https://www.edureka.co/blog/hadoop-ecosystem

YARN

- Yet Another Resource Negotiator
- Manages resources (nodes)
  - Knows the nodes available
  - Can provide nodes to an application
- YARN doesn’t know or care about applications
  - So it can be used by any kind of application
- Main benefits
  - Nodes can be shared between user/applications
  - New applications/frameworks can use it and avoid managing resources
- Yarn is used outside of Hadoop

Writing Hadoop MapReduce Programs

- A MapReduce program is called a Job
- Main language is Java
  - But can use any executable or script
- Map function implemented in a Mapper class
- Reduce function implemented in a Reducer class
- Default implementation
  - Splitting
    - If input is text file, key is offset, value is whole line
  - Shuffling
    - Based on hash value of key
- Package *org.hadoop.mapereduce*

(5280, "abc")

Datatypes

- Hadoop relies on its own serialization
  - More efficient than standard Java
  - Relies on the *Writable* interface
- Any key or value implements *Writable*
- Common ones
  - IntWritable : 32-bit Integer
  - LongWritable : 64-bit Integer
  - DoubleWritable : 64-bit Double
  - Text : String

Writing a mapper

- A mapper takes *(key,value)* and produces *(key,value)*
- Steps for implementing your own mapper
  1. Decide on the data types of input and output
  2. Extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
     - Set the types of all generics
  3. Implements *public void map(KEYIN, VALUEIN, Context)*
- Sending data to reducer is done through Context

Writing a reducer

- A reducer takes *(key,[values])* and produces *(key,value)*
- Steps for implementing your own reducer
    1. Decide on the data types of input and output
        - Input comes from the mapper, output can be anything
    2. Extends Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
        - Set the types of all generics
    3. Implements *public void reduce(KEYIN, Iterable<VALUEIN>, Context)*
- Output is done through Context

Mapper example

to reducer

```java
public static class Map extends 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, Context context) throws… {
        String line = value.toString();
        //do something smart here
        context.write(word, one);
    }
}
```

From Mapper

Reducer Example

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

 public void reduce(Text key, Iterable<IntWritable> values, Context context)throws… {
    int sum = 0;
     //do also something smart here
    context.write(key, new IntWritable(sum));
   }
}
```

Writing the rest and compiling

- In the *main(…)* method
  - Create a Job instance
  - Configure InputFiles, Mapper, Reducer, Output key and value…
- Everything has to be packaged as a jar
  - Use maven for dependencies and packaging

```java
public static void main(String [] args) throws Exception
{
  Configuration c=new Configuration();
  String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
  Path input=new Path(files[0]);
  Path output=new Path(files[1]);
  Job j=new Job(c,"wordcount");
  j.setJarByClass(WordCount.class);
  j.setMapperClass(MapForWordCount.class);
  j.setReducerClass(ReduceForWordCount.class);
  j.setOutputKeyClass(Text.class);
  j.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(j, input);
  FileOutputFormat.setOutputPath(j, output);
  System.exit(j.waitForCompletion(true)?0:1);
}
```

Execution

- A Job is submitted to a Hadoop Cluster as jar
  - hadoop command
- All files have to be in HDFS
  - All paths relative to HDFS
- Number of mapper instances
  - Automatically decided based on the size (blocks) of input
- Number of reducers
  - Computed, can be set manually
  - Ideal value depends on the output of mappers

Execution - 2

- Mappers/Reducers are executed close to data if possible
  - Use replication factor
- Result of a Job is written to HDFS
  - In a directory
- Output directory contains 1 file per reducer instance
  - Named *part-000xxx*
- Jobs cannot overwrite existing files
  - Remember to remove previous results before new execution

```
Error Launching job : Output directory hdfs://localhost:9000/result[redacted] already exists
```

Hadoop Streaming

- A simple tool to use almost anything as map and reduce functions
  - Part of the standard Hadoop distribution
- Mappers and Reducers can be any exec or script
  - Works with Python
- Mappers
  - Read from files on HDFS
  - Write to standard output as text with tab as (key value) separator
- Reducers
  - Read from standard input, assume tab as separator
  - Get (key value) pairs (!)
  - Write to standard output, automatically saved to file

Hadoop Streaming

$(key, [val_1, val_2, val_3])$

(key, val_1)
(key, val_2)
(key, val_3)

- Limitations
  - No real grouping/shuffling
  - Reduce receives multiple (key,value) pairs
  - Relies on files and STDIN/STDOUT for data transfer
- Example :
  - hadoop jar hadoop-streaming-3.1.4.jar -input /sample-text-file.txt -output /results -mapper mapper.py -reducer reducer.py