# Programowanie w chmurze na platformie Java EE

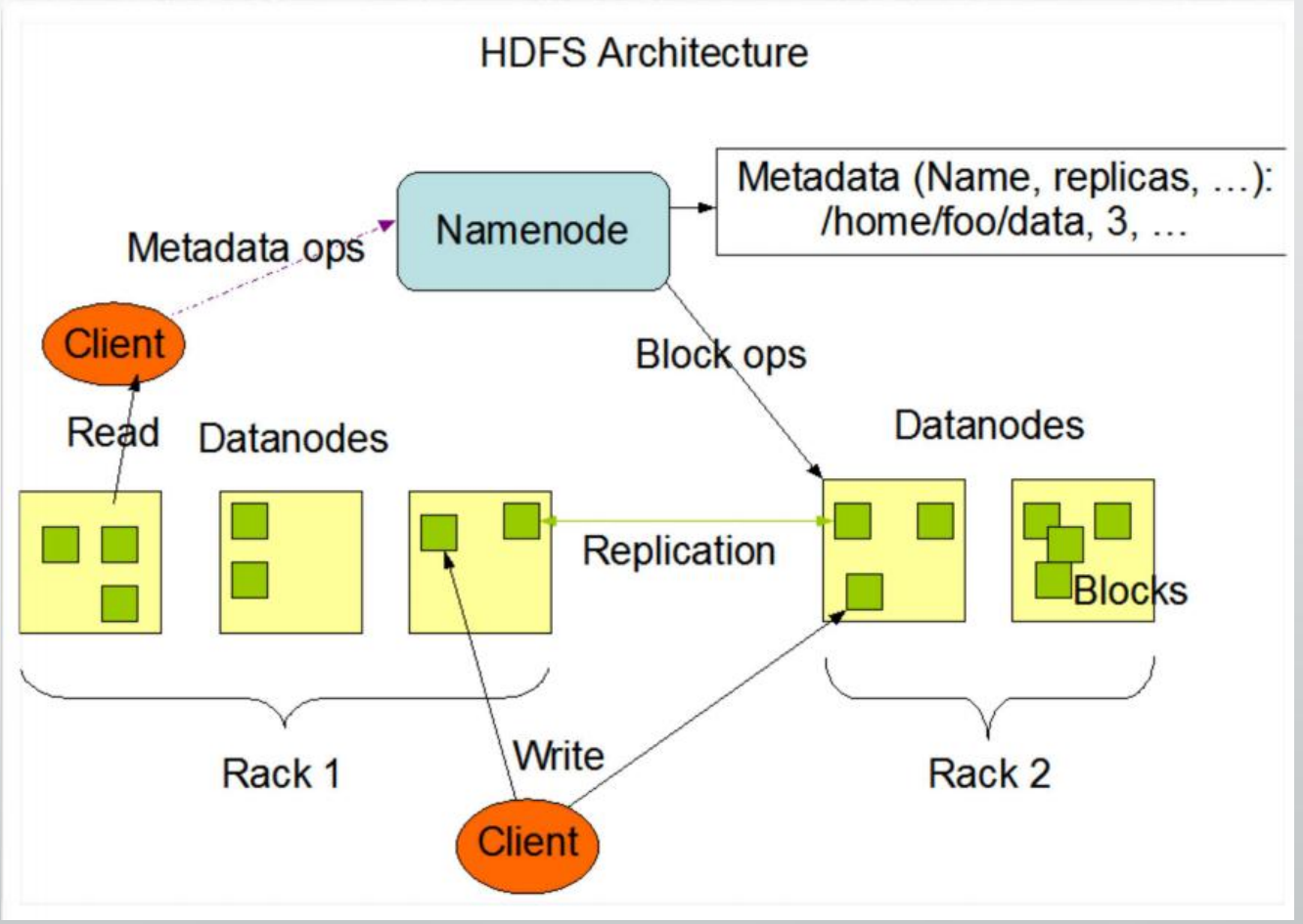
Wykład 3 - dr inż. Piotr Zając

# Hadoop

- Apache Hadoop is a framework for running applications on large cluster built of commodity hardware. The Hadoop framework transparently provides applications both reliability and data motion
- https://hadoop.apache.org/
- Hadoop implements a computational paradigm named Map/Reduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster.
- In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster.

# Hadoop HDFS

- Master/slave architecture
- Single NameNode, a master server that manages the file system namespace and regulates access to files by clients
- In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on
- HDFS exposes a file system namespace and allows user data to be stored in files



# HDFS Architecture

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant.

- HDFS is highly fault-tolerant
- Designed to be deployed on low-cost hardware
- Provides high throughput access to application data
- Suitable for applications that have large data sets

HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is part of the Apache Hadoop Core project.

# HDFS Data Sets

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size.  Thus, HDFS is tuned to support large files. It should:

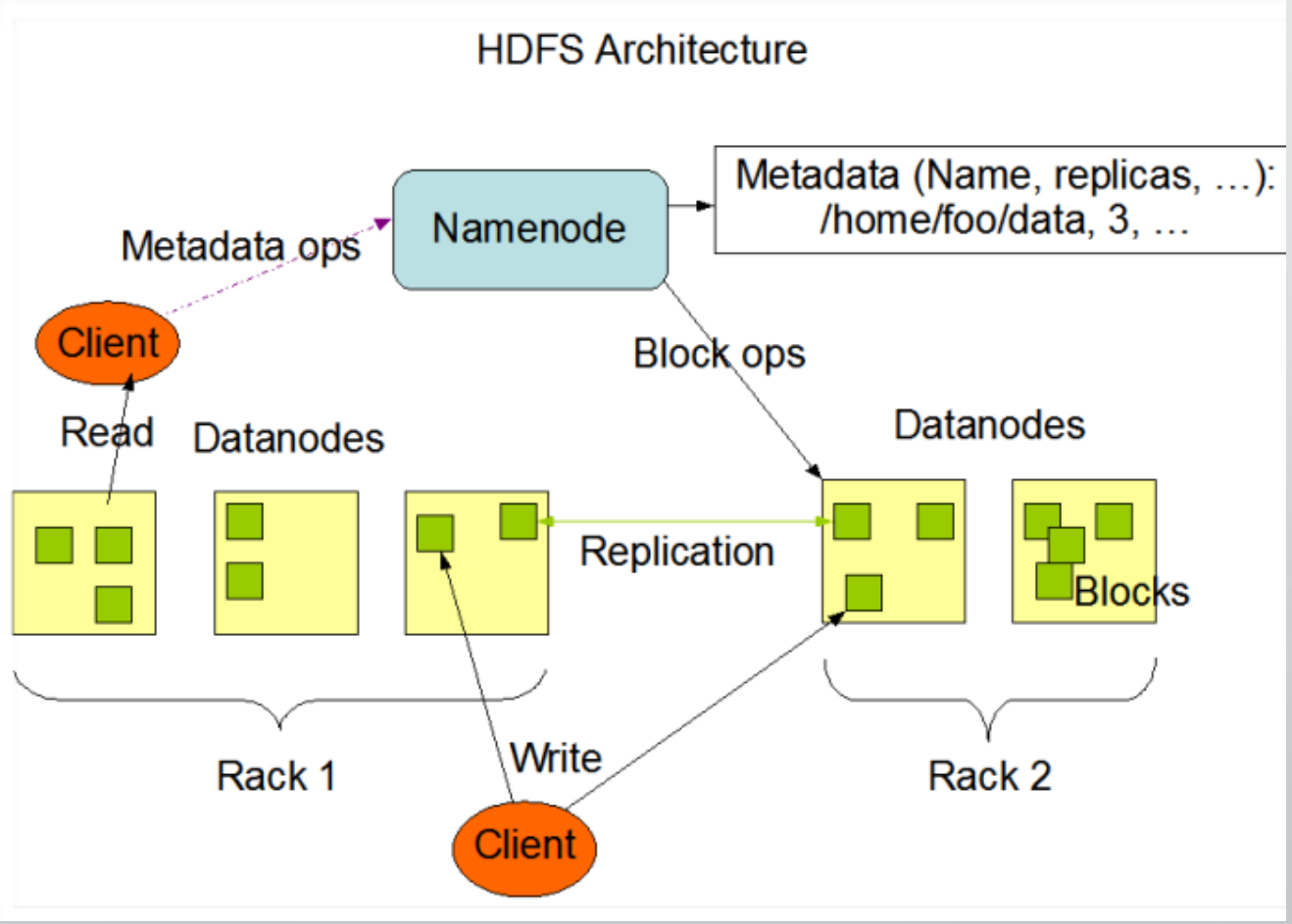
- Provide high aggregate data bandwidth
- Scale to hundreds of nodes in a single cluster
- Support tens of millions of files in a single instance

# „Moving Computation is Cheaper than Moving Data"

- A computation requested by an application is much more efficient if it is executed near the data it operates on.
- This is especially true when the size of the data set is huge. This minimises network congestion and increases the overall throughput of the system.
- It is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located

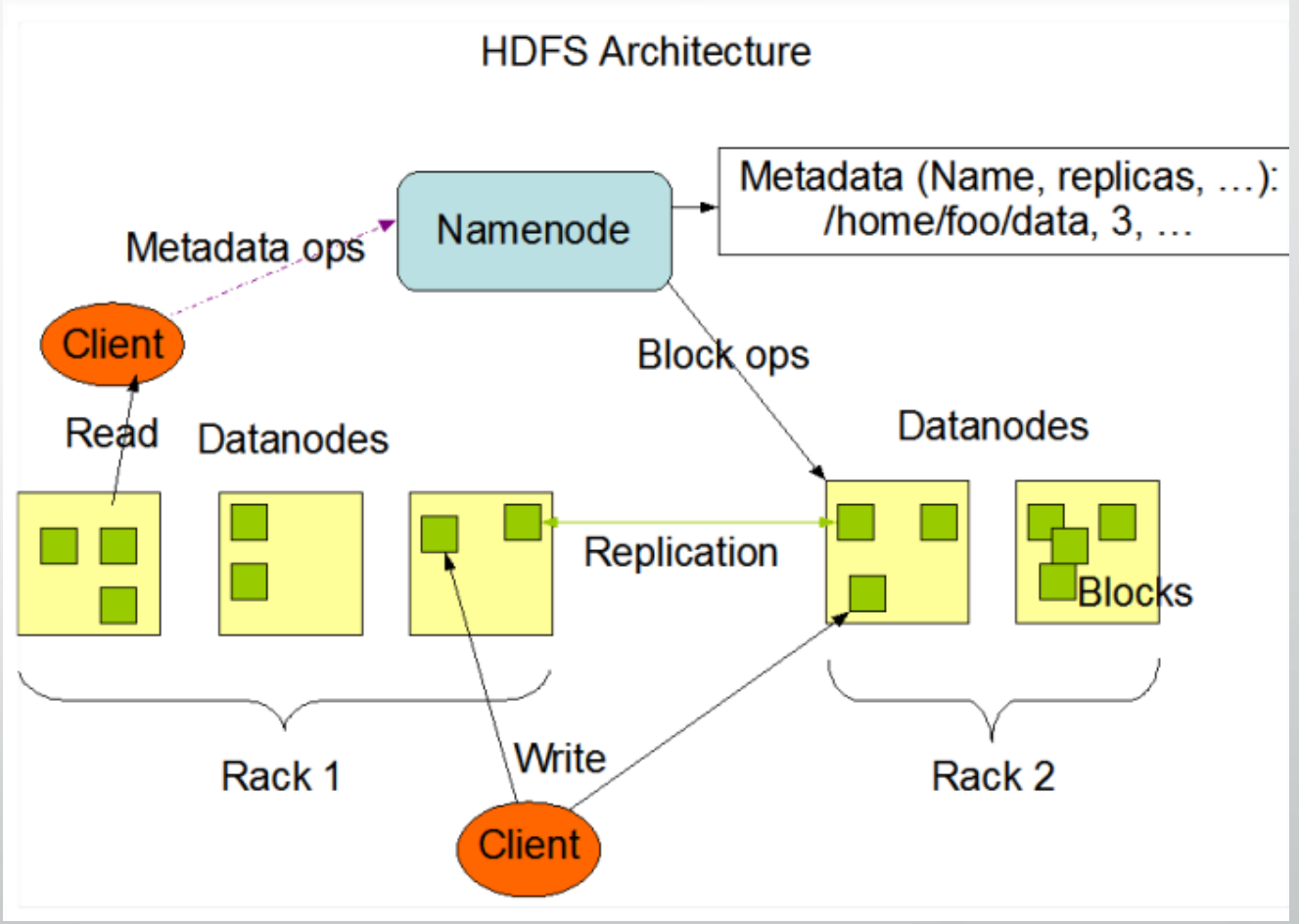
# Namenode

The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.



# Datanode

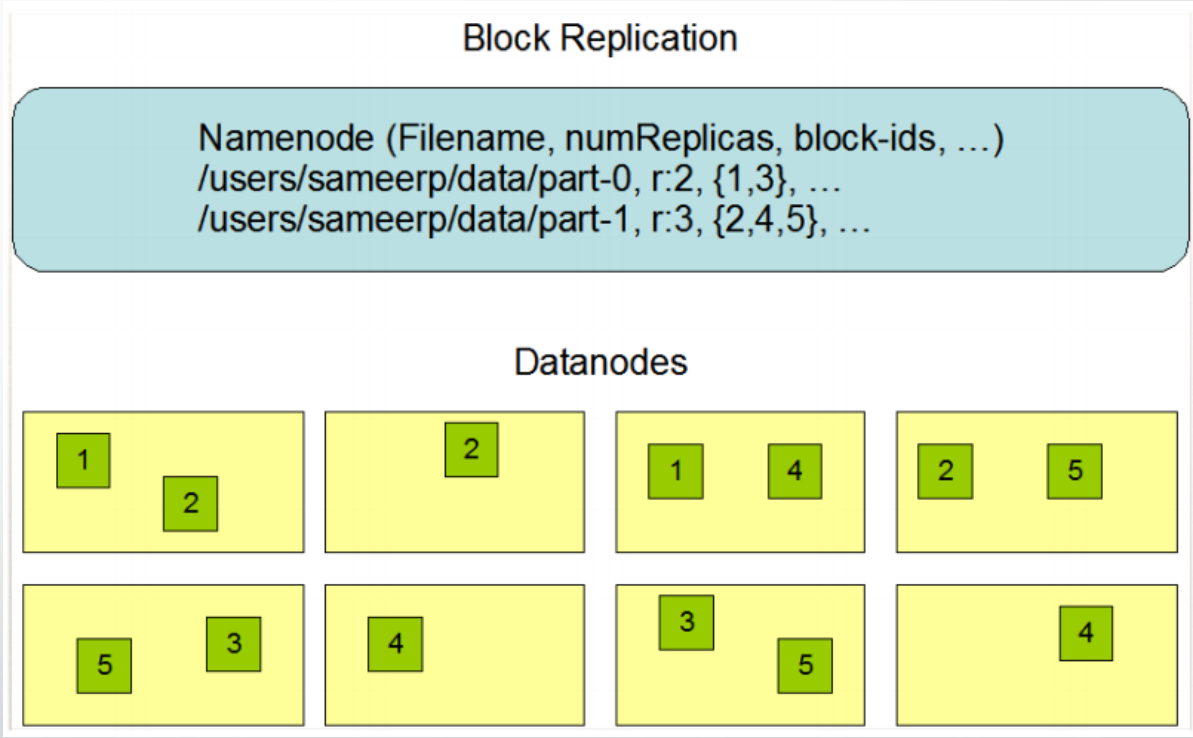
- Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes
- The DataNodes are responsible for serving read and write requests from the file system's clients
- The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.



# Replication

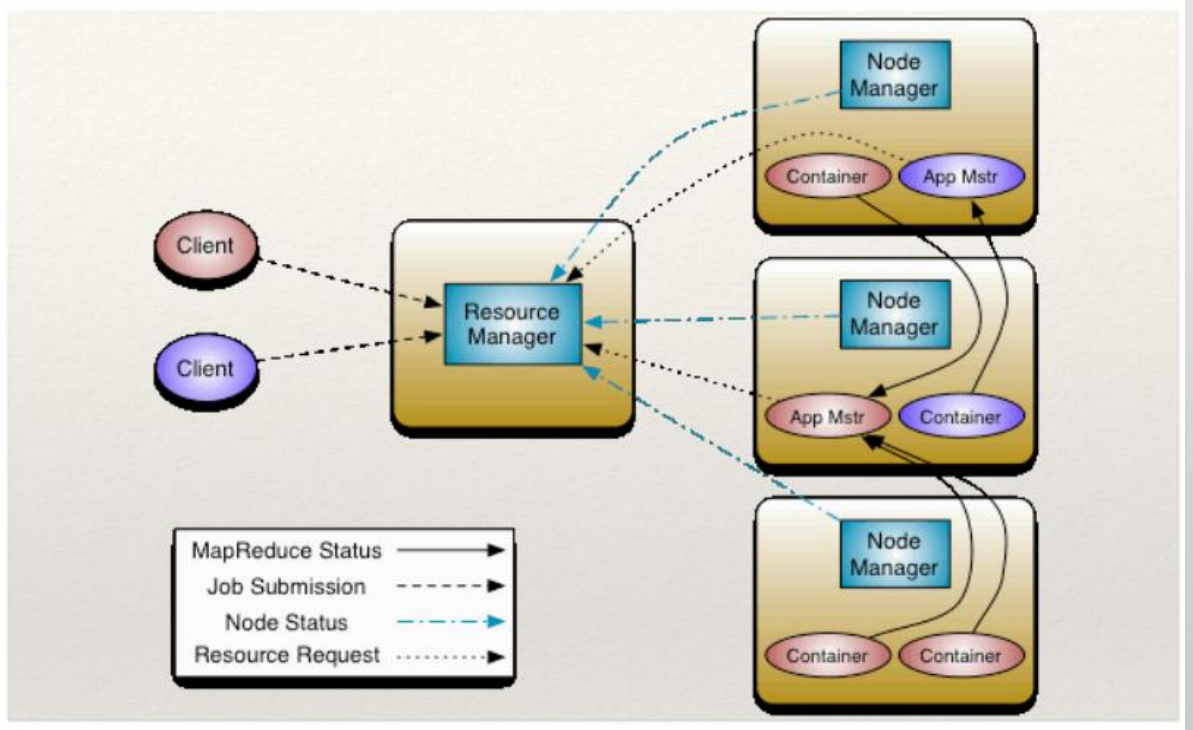
- HDFS is designed to reliably store very large files across machines in a large cluster.
- It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size.
- The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file.
- Application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later.

Files in HDFS are write-once and have strictly one writer at any time.



# YARN

- YARN is the component responsible for allocating containers to run tasks, coordinating the execution of said tasks, restart them in case of failure, among other housekeeping.
- It also has 2 main components: a ResourceManager which keeps track of the cluster resources and NodeManagers in each of the nodes which communicate with the ResourceManager and sets up containers for execution of tasks.

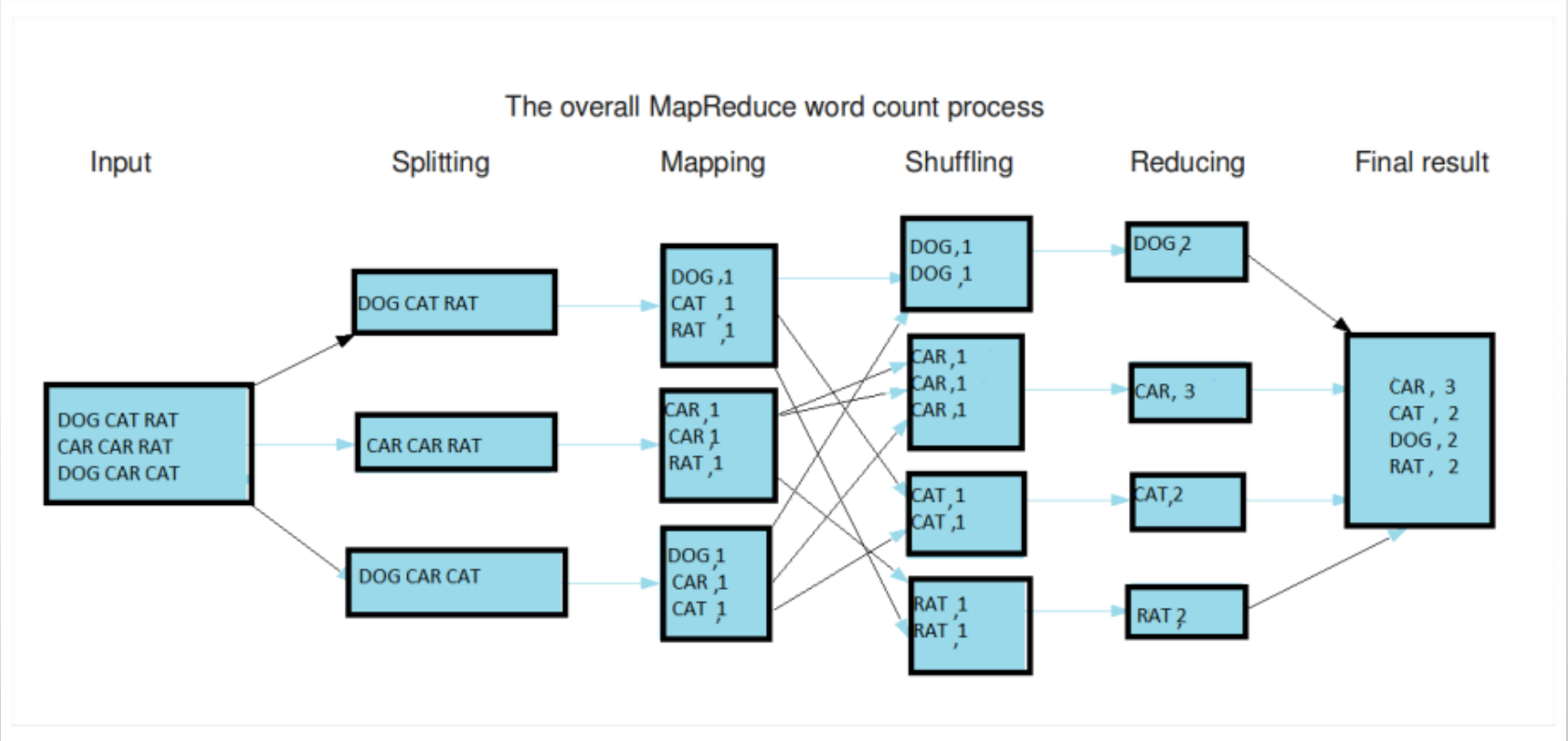


# Hadoop MapReduce

- input
  - set of files
  - directory
- output
  - directory
  - set of results files
- most important methods are:
  - map()
  - reduce()

# MapReduce Stages

- Map
- Shuffle
- Reduce



# MapReduce

- Typical implementation uses Mapper and Reducer interfaces
- Most important implemented methods are:
  - map()
  - reduce()
- Class implementing Mapper
- Class implementing Shuffling
- Class implementing Reducer

# MapReduce API

```
public class WordCount {
     public static class TokenizerMapper
          extends Mapper<Object, Text, Text, IntWritable>{
     public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
}
     public static class IntSumReducer
          extends Reducer<Text,IntWritable,Text,IntWritable> {
     public void reduce(Text key, Iterable<IntWritable> values,
     Context context
     ) throws IOException, InterruptedException {
     }
}
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);
}
}
```

# Map Stage

- Parsing function maps input key/value pairs to set of intermediate key/value pairs
- One map for each InputSplit generated by InputFormat
- http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Mapper.html

```
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);
     }
}
```

For the given sample input the first map emits:
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
The second map emits:
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>

# Shuffle Stage

Combiner:

- Mapper output is combined according to defined rules
- Merges duplicates
- Executes local aggregation

The output of the first map:
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
The output of the second map:
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>

```
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);
     }
}
```

Combined output of the first map:
< Bye, 1>
< Hello, 1>
< World, 2>
The combined output of the second map:
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>

# Reduce stage

- Uses Mapper output as input if there is no class defined to proces data after Mapper
- Uses data processed by implemented classes, that do the processing after Mapper stage
- Computes final result

```
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);
     }
}
```

The reducer output:
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>

# Job

- Primary interface for user-job interaction with ResourceManager
- MapReduce job configuration: setters for Mapper, Combiner, Reducer
- Job.submit() - submit job to cluster and return immediately
- Job.waitForCompletion(boolean) - submit job and wait for it to finish

```
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);
```

# Job Input

InputFormat

- validate the input for the job
- split-up input files into InputSplit instances assigned to separate Mapper
- blocksize of FileSystem is the upper bound for splits
- blocksize can be defined by user
- it is recommended to implement RecordReader for InputSplit

# Job Output

OutputFormat

- validate the output of the job - for example by checking that the output directory doesn't already exist
- provide RecordWriter to write job results files

# Exercise 1a: Pre-Hadoop

- Create Gitlab repo:
  - progchmur_2018_2019_d_imie_nazwisko
  - Add **piotr.zajac** as developer
- Download cluster setup files from: fiona.dmcs.pl/~pzajac
- Setup cluster with 1 master and 2 slaves
  - Complete the hadoop configuration files (all files in files/hadoop/.. )
  - Run the machines with vagrant
  - ssh to hadoop-master
  - Run the demo.sh script
- Push to repo

# Exercise 1b: Hadoop

- Environment
  - maven
  - ide ( e.g. IntelliJ, netbeans)
  - working hadoop environment
- Write Map Reduce algorithm. The program should be started inside hadoop-multinode environment. Run it as a batch executable with hadoop jar command
- input: file containing JSON lines (download from fiona.dmcs.pl/~pzajac)
- output: list of standard deviation calculated for each finger in series side from input file

The JSON lines contain data with metadata:


```
{"timestamp":1521403485000,"features2D":{"fifth":62.957324981689,"fourth":
79.980072021484,"first":58.463665008545,"third":85.255867004395,"second":
76.443199157715},"side":"R","features3D":{"palm-section":{"factor": {"value":"6.952650410179064e-310"}},"defects-
distance-proportion":"","finger-length":"","nofinger-shape-factor":{"factor":{"value":"6.9531293302065641e-
310"}},"finger-volume": {"little":"0","index":"0","thumb":"0","middle":"0","ring":"0"}},"series":3,"sample":8}
```


the data fields are features2D[first], features2D[second], features2D[third], features2D[fourth], features2D[fifth]
the metadata are: side, series, sample, finger
side - hand left/right
sample - samples of single hand
series - set of hand sequence
finger - number of a hand finger, contains finger length

# Exercise 1b : Hadoop

Example of MapReduce output :

.....

L-5-first stdDev(1,20)

....

L-25-second stdDev(1,20)

....

Where stdDev(1,20) is the value of the standard deviation calculated for each finger of the samples in series side set.

Calculated standard deviation for each finger of all samples of a set defined with series, side, finger number, there are 20 samples for each series side combination each scan contain 5 fingers length.

Directory input should be created in users directory and the input data file placed inside.

Results should be saved in output directory in users directory.