

# Chapter 12

## Big Data with Map / Reduce

Dr. Steffen Herbold

[herbold@cs.uni-goettingen.de](mailto:herbold@cs.uni-goettingen.de)

# Outline

- Overview
- MapReduce
- Apache Hadoop
- Apache Spark
- Summary

# Repetition: Definition of Big Data

## Definition of Big Data

- Following Gartner's IT Glossary:

- Big data is high-**volume**, high-**velocity** and/or high-**variety** information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation.

- The three Vs

- Volume
- Velocity
- Variety



Some people actually use 10 Vs to define big data!

- Variability
- Veracity
- Validity
- Vulnerability
- Volatility
- Visualization
- Value



GEORG-AUGUST-UNIVERSITÄT  
GÖTTINGEN

Data Science and Big Data Analytics

What are the innovative forms of information processing?

# Parallelism is Mandatory

In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox.

We shouldn't be trying for bigger computers, but for more systems of computers

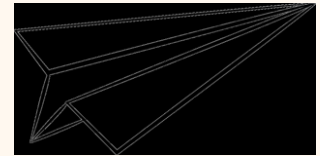
– Grace Hopper



# Parallel Programming Models

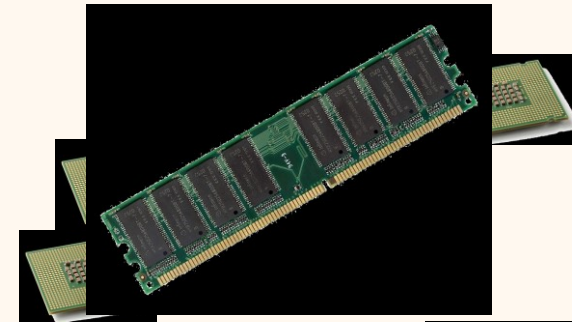
- Message Passing

- Independent tasks on local data
- Tasks interact by exchanging messages



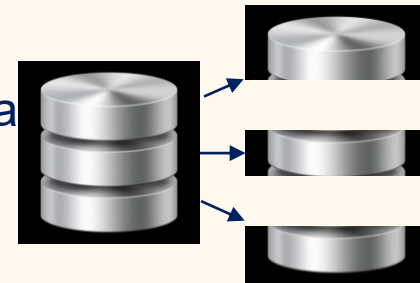
- Shared memory

- Tasks share common address space
- Tasks interact by reading/writing in this space



- Data parallelization

- Tasks execute independent operations on partitions of data
- Well suited for problems that are “embarrassingly parallel”



# Traditional Infrastructures

Data storage



Database or Storage  
Area Network (SAN)

Compute cluster



...



Compute Nodes

Result / Insight



# Message Passing / Shared Memory

Data storage



Compute cluster



...



Result / Insight



Each node may load complete data!  
→ Does not scale

# Data Parallelization

Data storage



Compute cluster



...



Result / Insight



Each node only loads partition  
→ Scales better  
→ Still requires transferring all data over the network



# Data Locality as Solution



If moving data is the problem,  
stop moving the data!

- Not supported by traditional clusters / infrastructures
  - Clusters for sharing CPUs, not storage
  - Storage made for IO, not computations

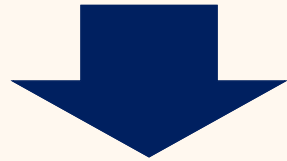


# Core concept of Big Data Technologies

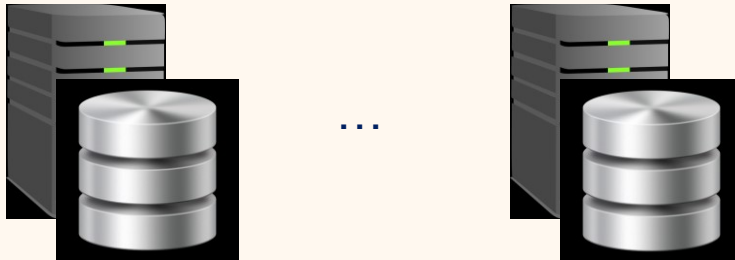
Parallelization



Data Locality



Compute Cluster with Distributed Storage



Our examples:



# Outline

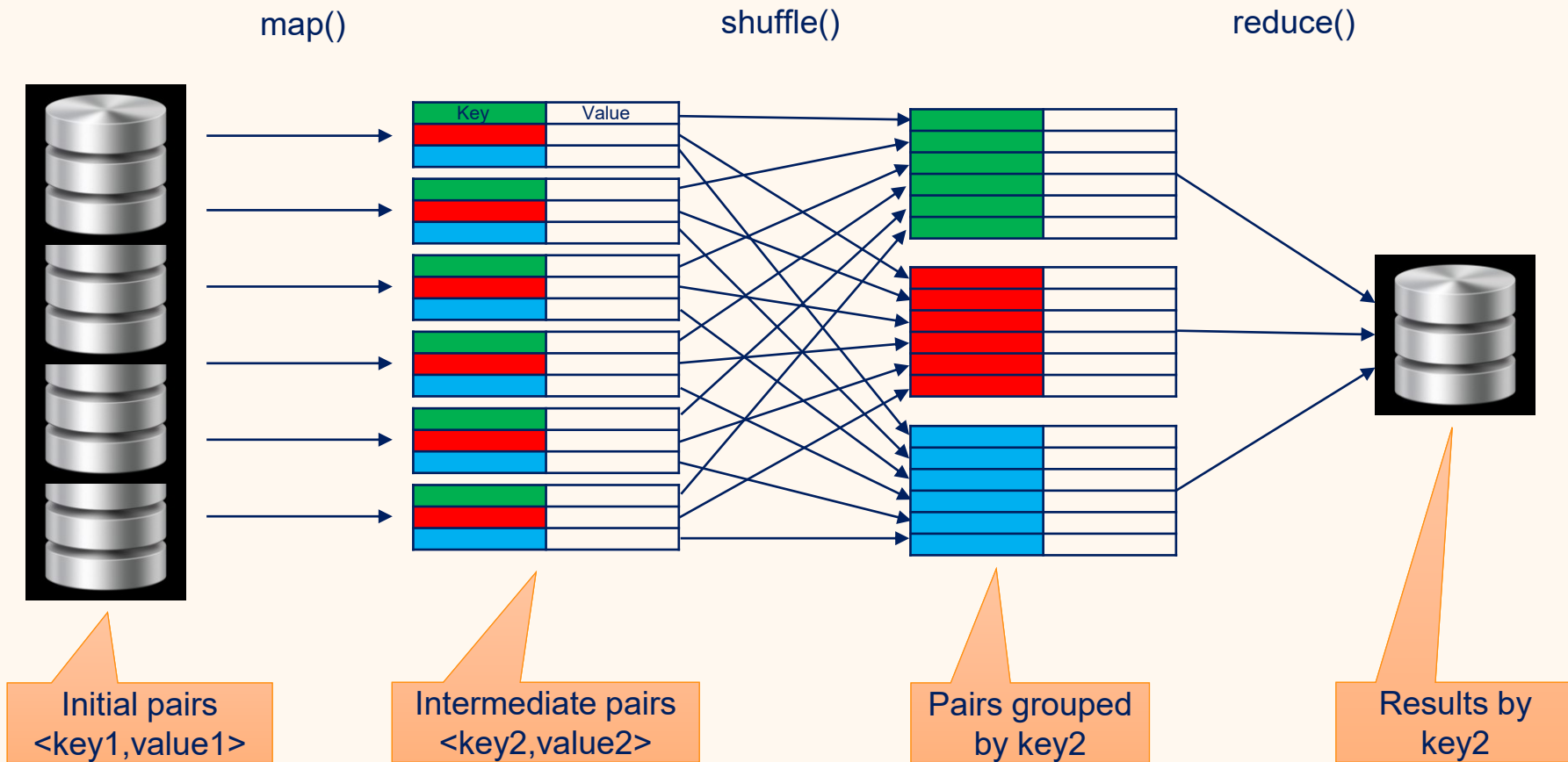
- Overview
- **MapReduce**
- Apache Hadoop
- Apache Spark
- Summary

# MapReduce

- Programming model for data parallelization
  - Published in 2004 by Google
- map() and reduce() functions for data processing
  - Based on transformations of key-value pairs
- shuffle() function for arranging intermediate results
- Distribution via master/worker paradigm
  - Supports high availability / recoverability
  - Discussed together with hadoop

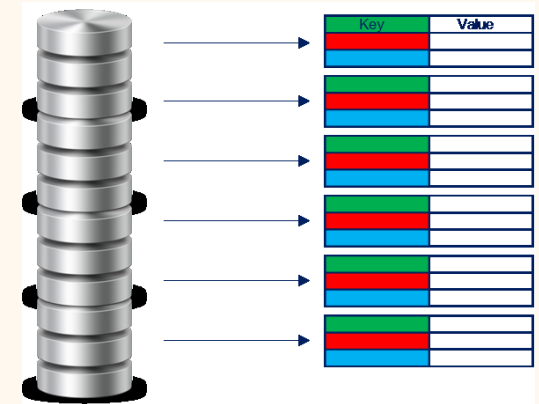
Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM*, 51.1 (2008): 107-113.

# Overview of MapReduce



# The map() Function

- Concept from functional programming



- Applies a function to every item in the input **seperately**

- `map(fun, <key1,value1>) → list(<key2, value2>)`
- Functions are usually user-defined

Data parallelization  
trivial

- Input keys and output keys can be different
  - Also different types

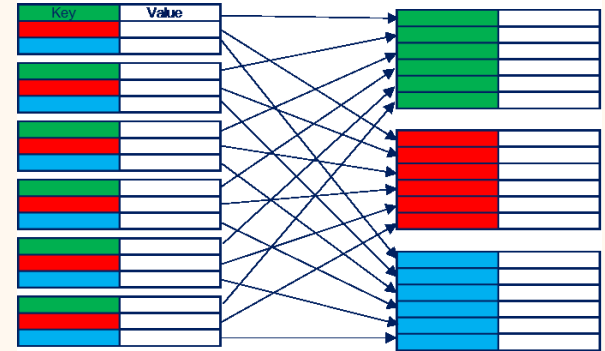
- Output is a list, i.e., one mapping can have multiple outputs

- All list elements must have same types



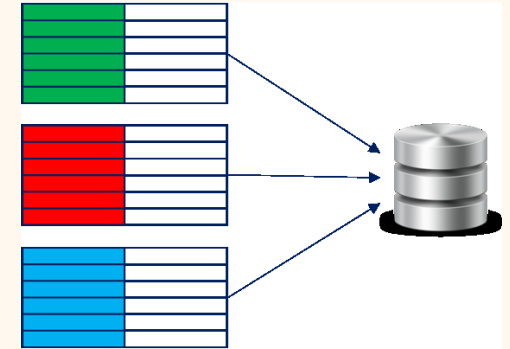
In the initial MapReduce implementation, all keys and values were strings, users where expected to convert the types if required as part of the map/reduce functions

# The shuffle() Function



- Organize data by key
  - `shuffle(list(<key2,value2>)) → list(<key2, list(value2)>)`
- Often includes sorting by key for efficiency
- Shuffle does not wait for map() to finish
  - Once a <key2,value2> is available it can be shuffled
  - Reduces waiting times
- Provided by the MapReduce framework
  - Can be overridden by users to optimize for use case

# The reduce() Function



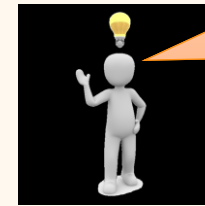
- Related to fold from functional programming
- Aggregates  $\langle \text{key2}, \text{value2} \rangle$  pairs with the same key
  - Single value per key
- Results in a list of values, one for each key
  - $\text{reduce}(\text{fun}, \text{list}(\langle \text{key2}, \text{list}(\text{value2}) \rangle)) \rightarrow \text{list}(\text{value2})$
  - Functions are usually user defined



# Parallelization with MapReduce


- Input can be read in chunks
  - Parallelism for creation of initial key-value pairs
- `map()` can be computed each key-value pair independently
  - Parallelism potential only limited by amount of data
- `shuffle()` can start working as soon as first key-value pair is processed
  - Limits waiting times
- `reduce()` can run in parallel for different keys
  - Need not wait for map to complete
  - Can already start when all results for a key are available

# Word Counts with MapReduce



This is the “Hello World” for MapReduce

- Our Data: 


- Initial <key1,value1> pairs:
  - <sentence1, “what is your name”>
  - <sentence2, “the name is bond james bond”>
- map() function: emit <word,1> for each word in a sentence
  - <sentence1, “what is your name”>
    - <“what”,1>, <“is”,1>, <“your”,1>, <“name”,1>
  - <sentence2, “the name is bond james bond”>
    - <“the”, 1>, <“name”,1>, <“is”,1>, <“bond”,1>, <“james”,1>, <“bond”,1>
- reduce() function: key concatenated with sum of values
  - <“what”, list(1)> → “what: 1”
  - <“bond”, list(1,1)> → “bond: 2”
  - ...

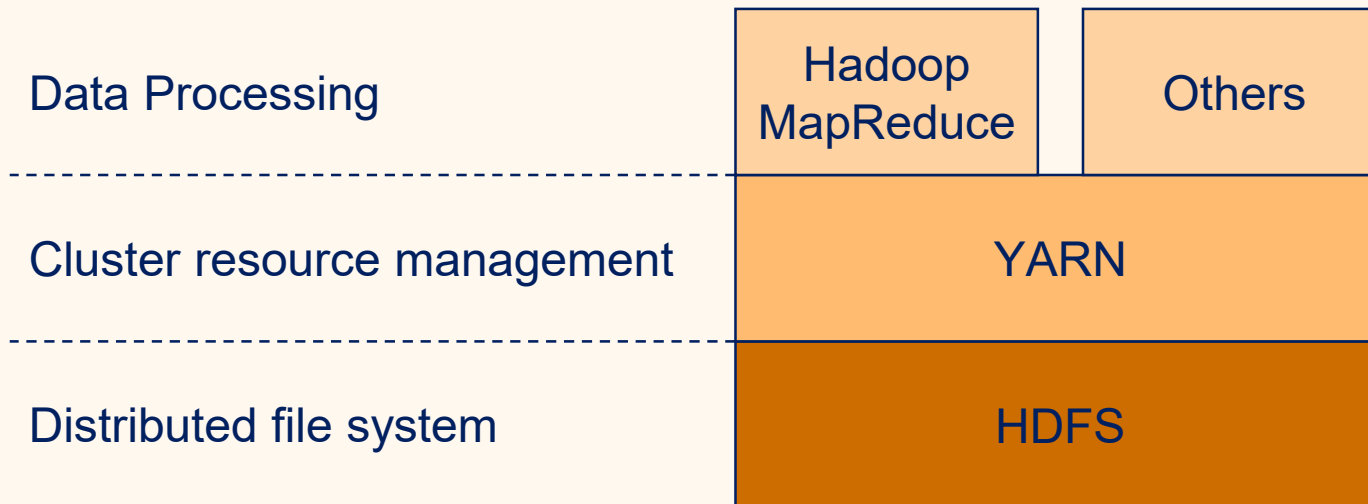
# Outline

- Overview
- MapReduce
- **Apache Hadoop**
- Apache Spark
- Summary

# Overview of Hadoop

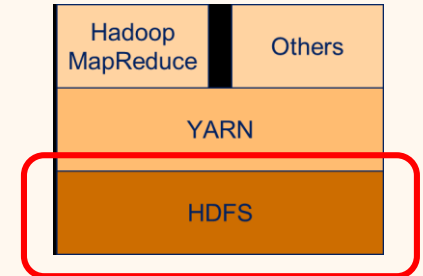


- Open-source implementation of MapReduce
  - Supported by all major cloud providers
  - Used by many large companies, e.g., Twitter, Facebook, Amazon, ...



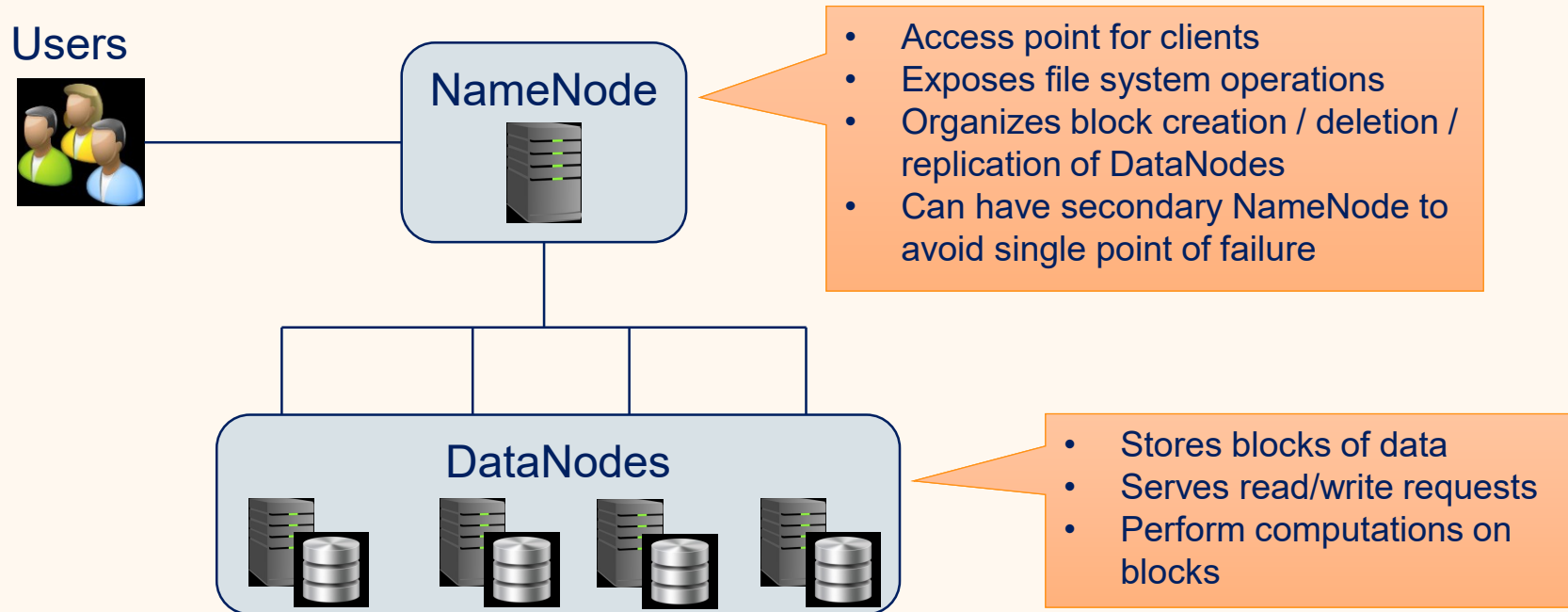
# Hadoop Distributed File System (HDFS)

- Core component of Hadoop

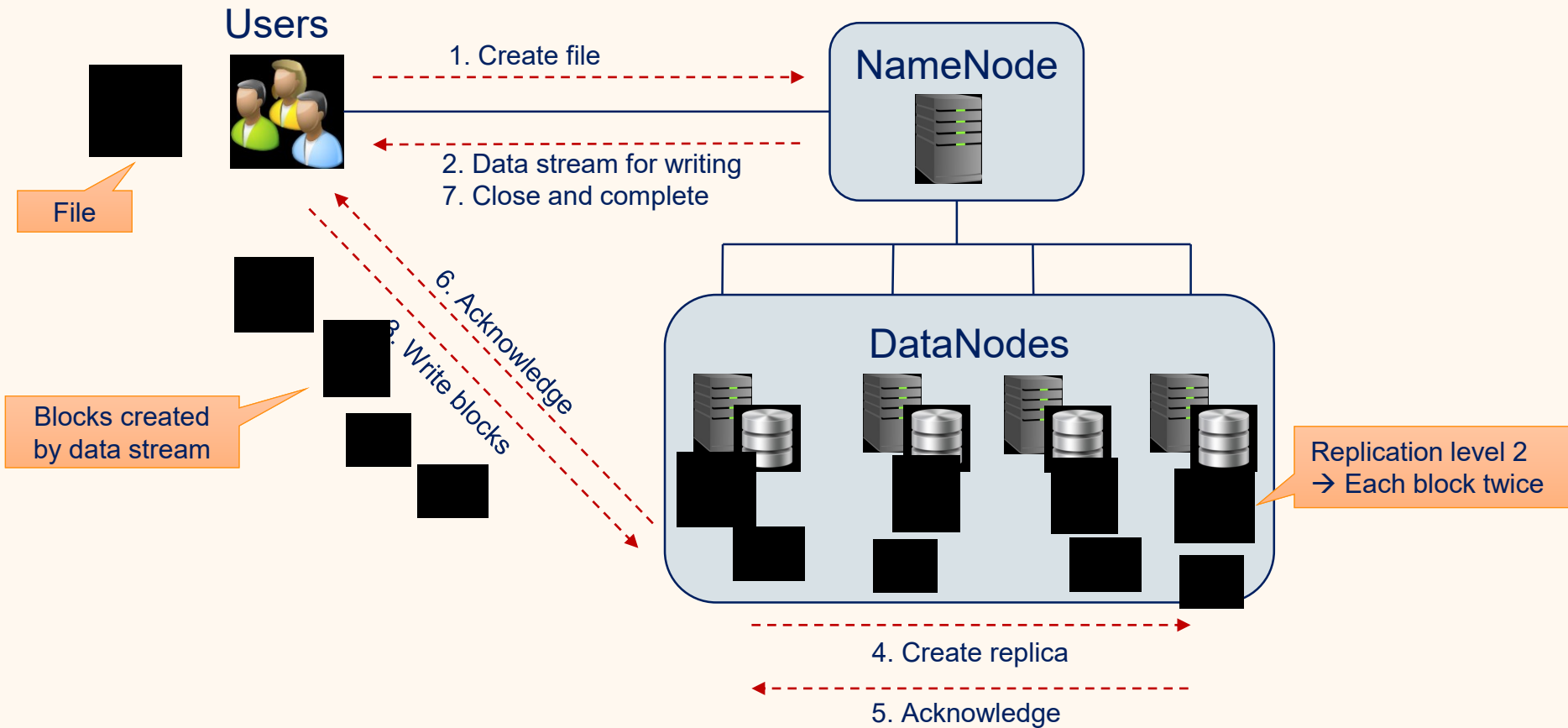


- Goals of HDFS
  - High throughput instead of low latency
  - Support for large files and data sets
  - Moving computation instead of moving data (→ Data locality)
  - Resiliency against hardware failures
- Uses a master/slave architecture

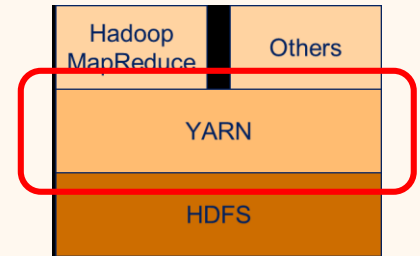
# Overview of HDFS



# Example: Write File



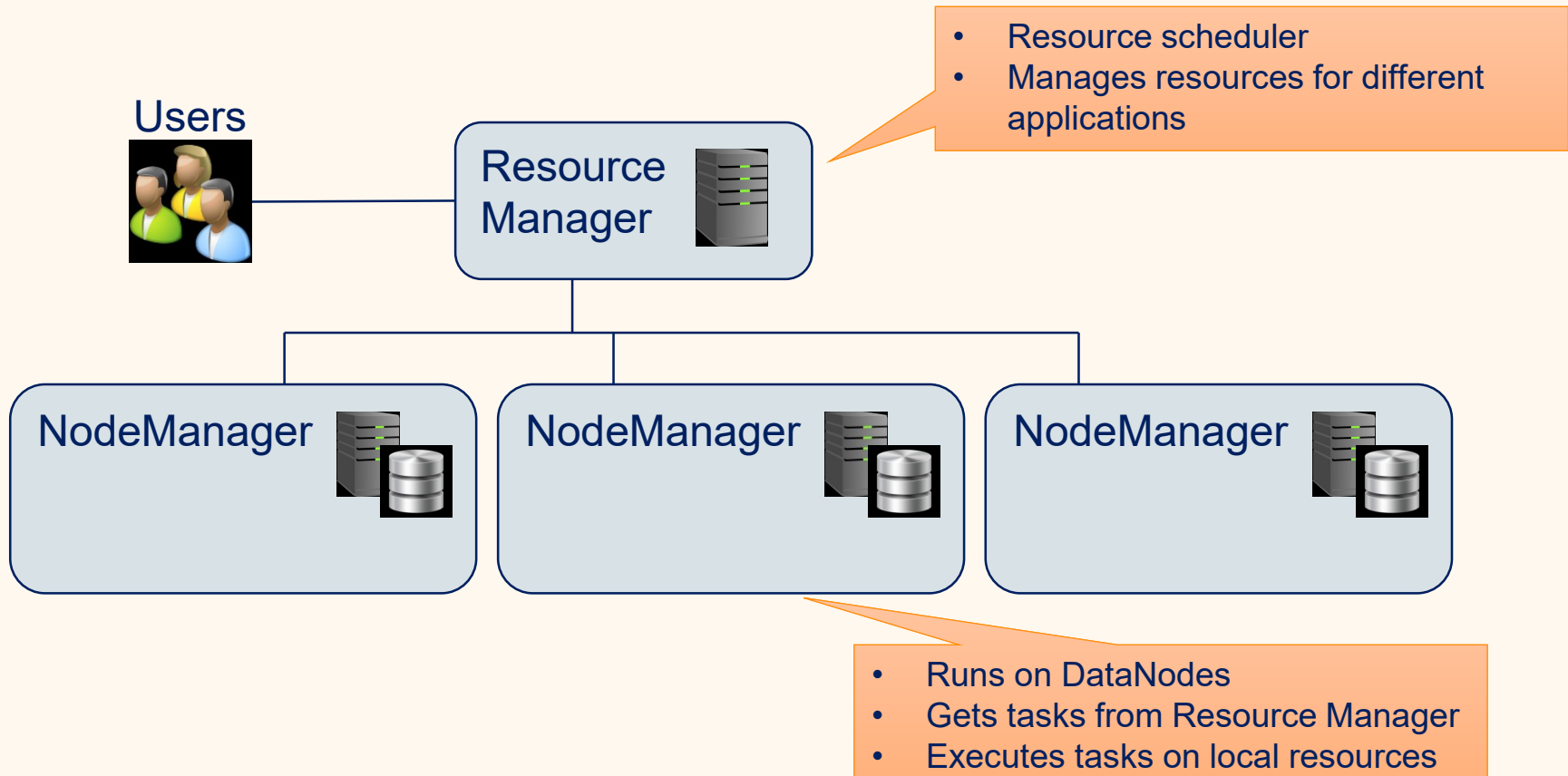
# Computing with Hadoop



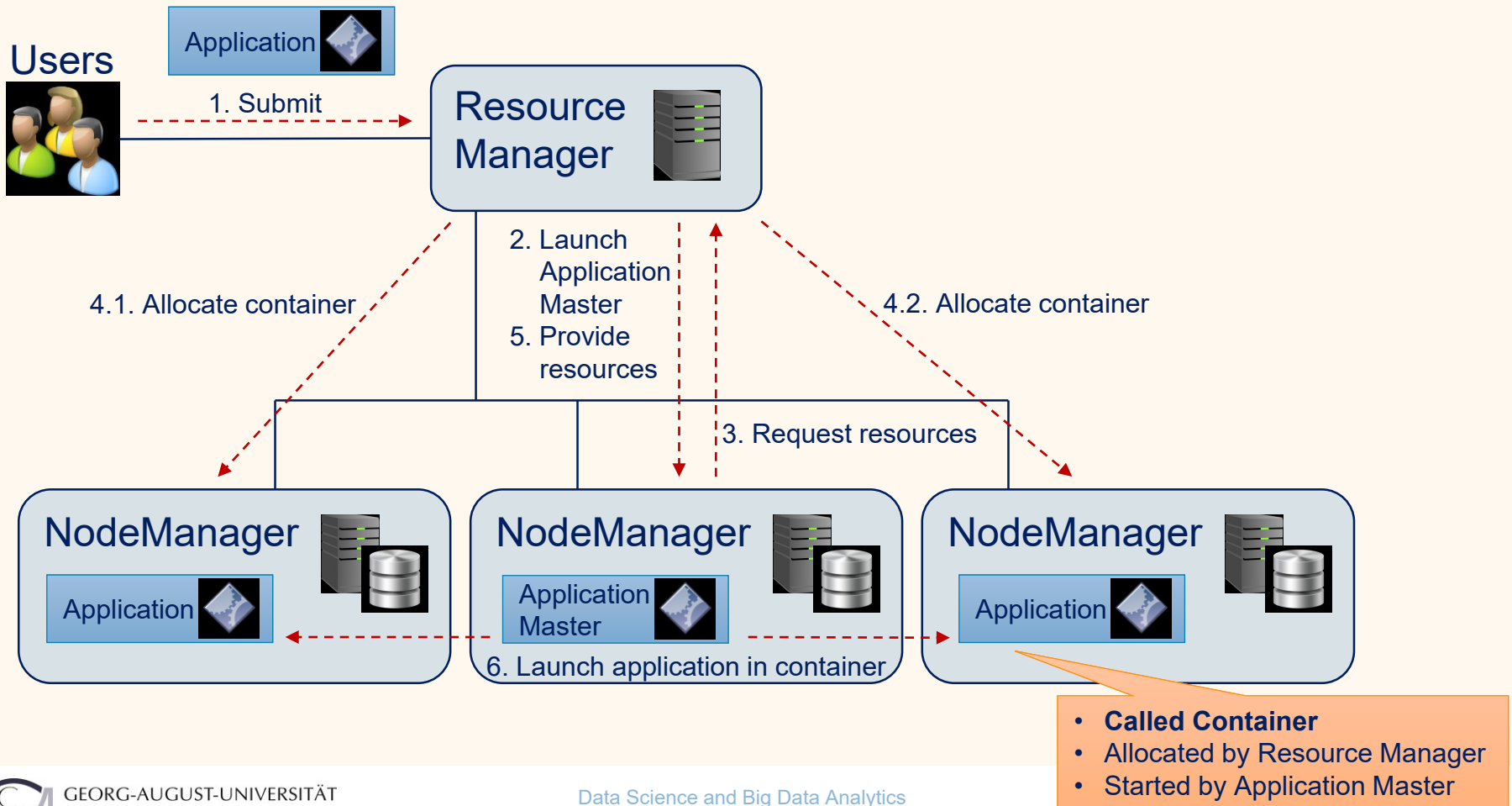
- HDFS „only“ distributed block storage
- Each DataNode should also serve as compute node
  - Map / Reduce / Shuffle tasks
  - Ideally also general compute tasks
- Each resource should only be used by one task
  - CPU cores
  - Memory
  - No overutilization
- Resources should be used as efficiently as possible
  - No underutilization



# YARN for Resource Management



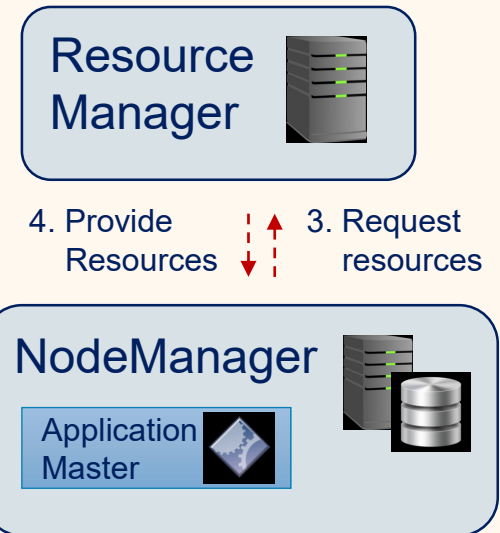
# Running Applications with YARN



# Resource Requests

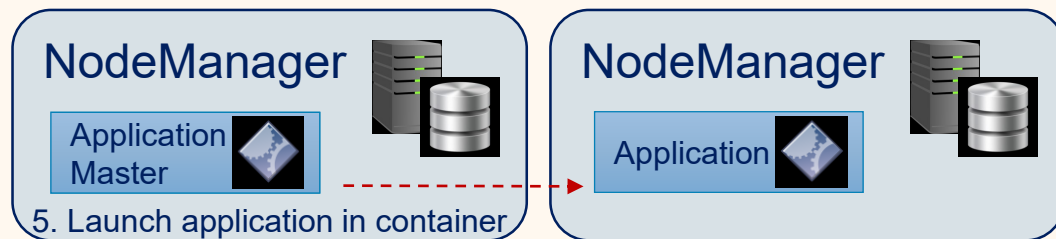
- Send from Application Master to Resource Manager

- Name of the resource
  - Can be used to select specific hosts or racks
- Priority
  - Only within the application
  - Allows application to have an internal scheduler
- Resource requirements
  - Memory
  - CPU cores
- Number of containers

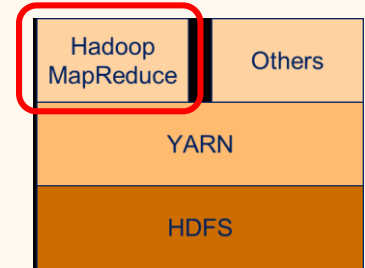


# Launching Containers

- Executes parts of the application
  - For example, map, reduce, or shuffle tasks
- Requires commands to start application
- Environment configuration
  - E.g., environment variables of application call
- Can access local resources
  - Binaries, HDFS files/blocks



# MapReduce with Hadoop




- Implementation of MapReduce on top of YARN
- Users define applications as sequences of map/reduce tasks
- Hadoop specifies as MRAppMaster YARN container
- MRAppMaster manages execution of tasks
- Two execution modes
  - Java applications
  - Streaming mode

# Java Applications

- MapReduce applications defined by Jobs programmatically
- Job class used to
  - Specify input
  - Register mapper
  - Register reducer
  - Specify output
- Mapper and reducer tasks defined by extending classes
- Compiled Jar is submitted to resource manager for execution

# Example for a Mapper

- Hadoop Mapper for Word Count example



The diagram shows four orange callout boxes pointing to the generic types in the Mapper signature:   
- "Type of input key" points to `Object`   
- "Type of input value" points to `Text`   
- "Type of output key" points to `Text`   
- "Type of output value" points to `IntWritable`

```
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 {  
        // text into tokens  
        StringTokenizer itr = new StringTokenizer(value.toString());  
        while (itr.hasMoreTokens()) {  
            // add an output pair <word,1> for each token  
            word.set(itr.nextToken());  
            context.write(word, one);  
        }  
    }  
}
```

# Example for a Reducer

- Hadoop Reducer for the Word Count example

Type of input value      Type of output key  
Type of input key      Type of output value

```
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 {  
        // calculate sum of word counts  
        int sum = 0;  
        for (IntWritable val : values) {  
            sum += val.get();  
        }  
        result.set(sum);  
        // write result  
        context.write(key, result);  
    }  
}
```



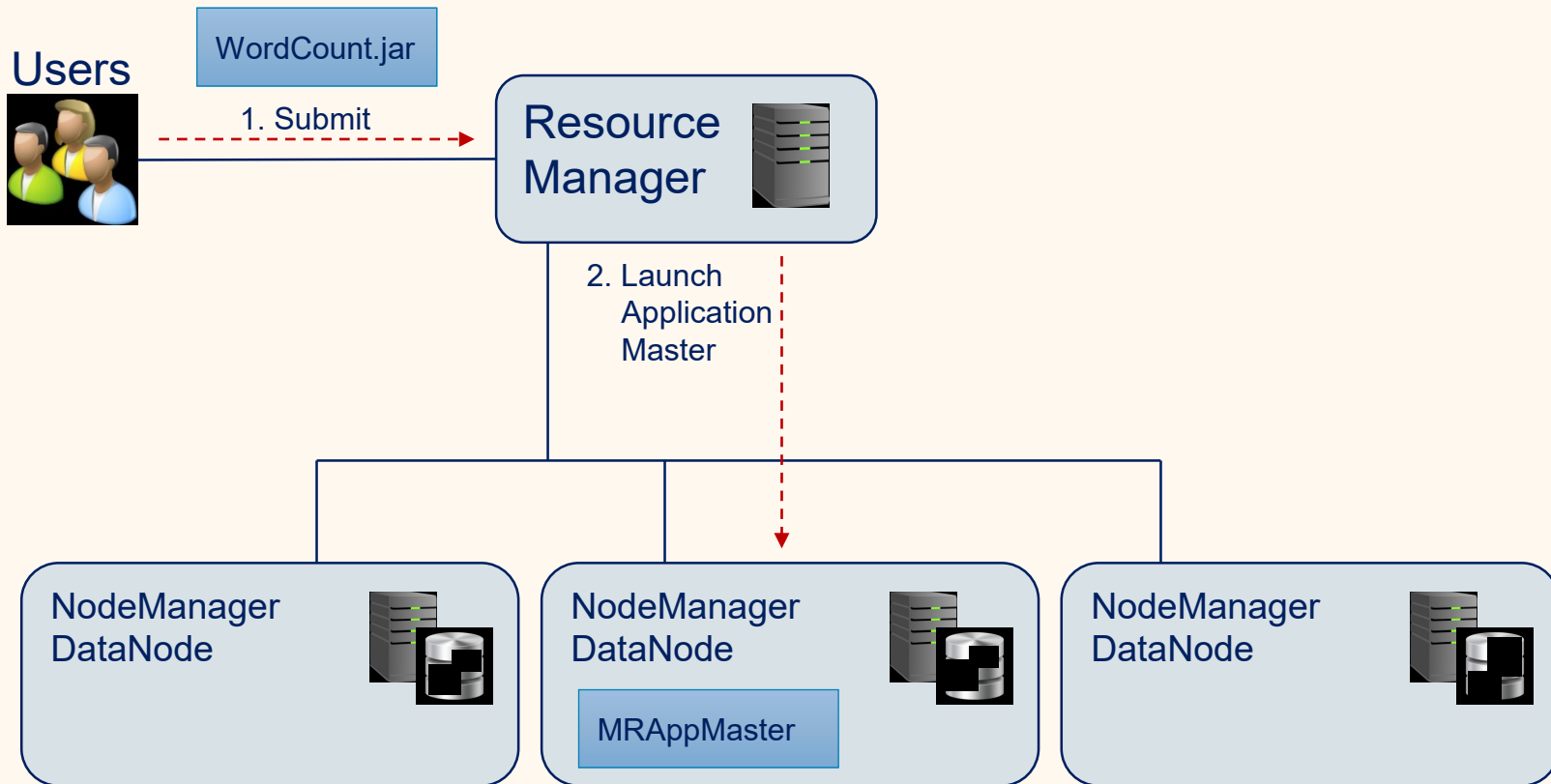
# Example for a Job definition

- Hadoop Job for the Word Count example

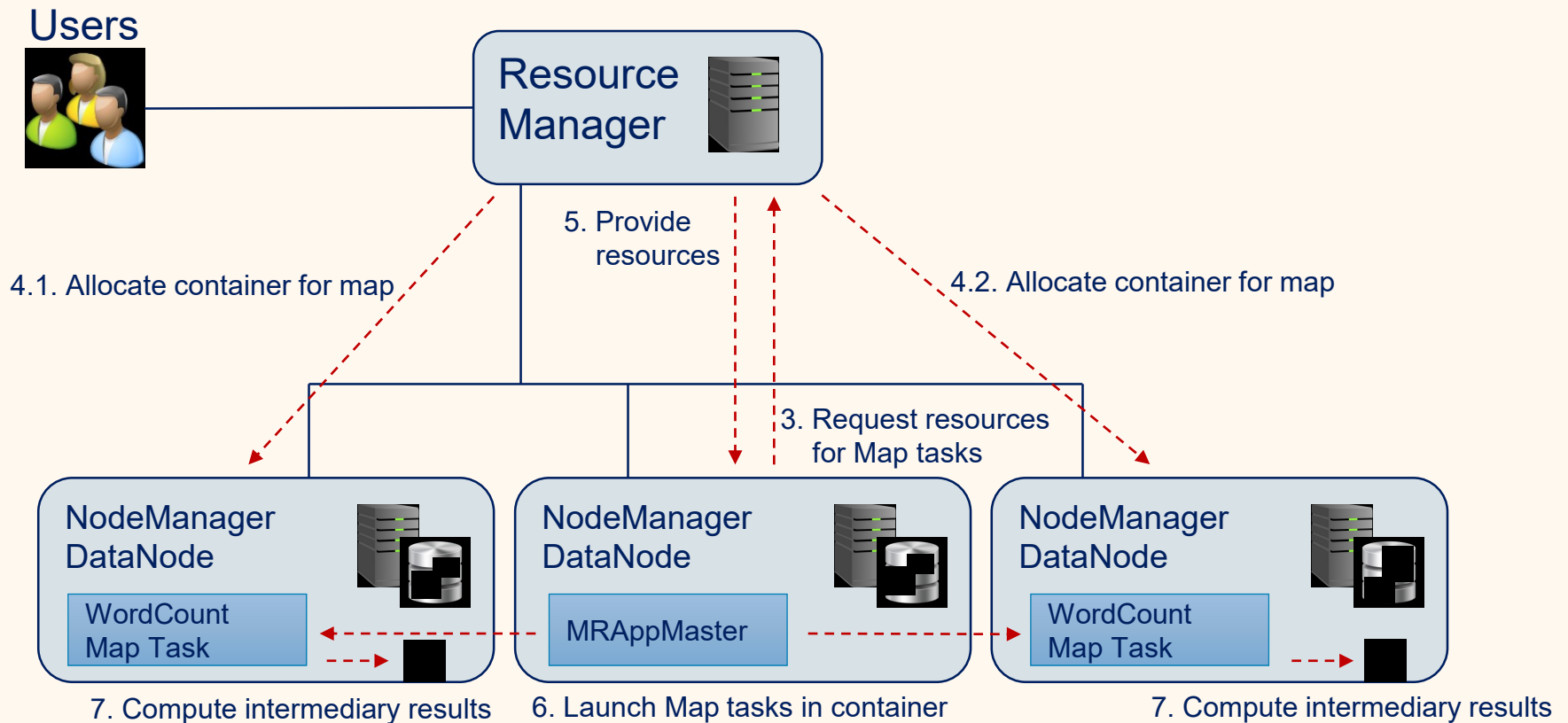
```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        // Hadoop configuration  
        Configuration conf = new Configuration();  
  
        // create a Job with the name "word count"  
        Job job = Job.getInstance(conf, "word count");  
        job.setJarByClass(WordCount.class);  
  
        // set mapper, reducer, and output types  
        job.setMapperClass(TokenizerMapper.class);  
        job.setReducerClass(IntSumReducer.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        // specify input and output files  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        // run job and wait for completion  
        job.waitForCompletion(true);  
    }  
}
```

Reads file line by line

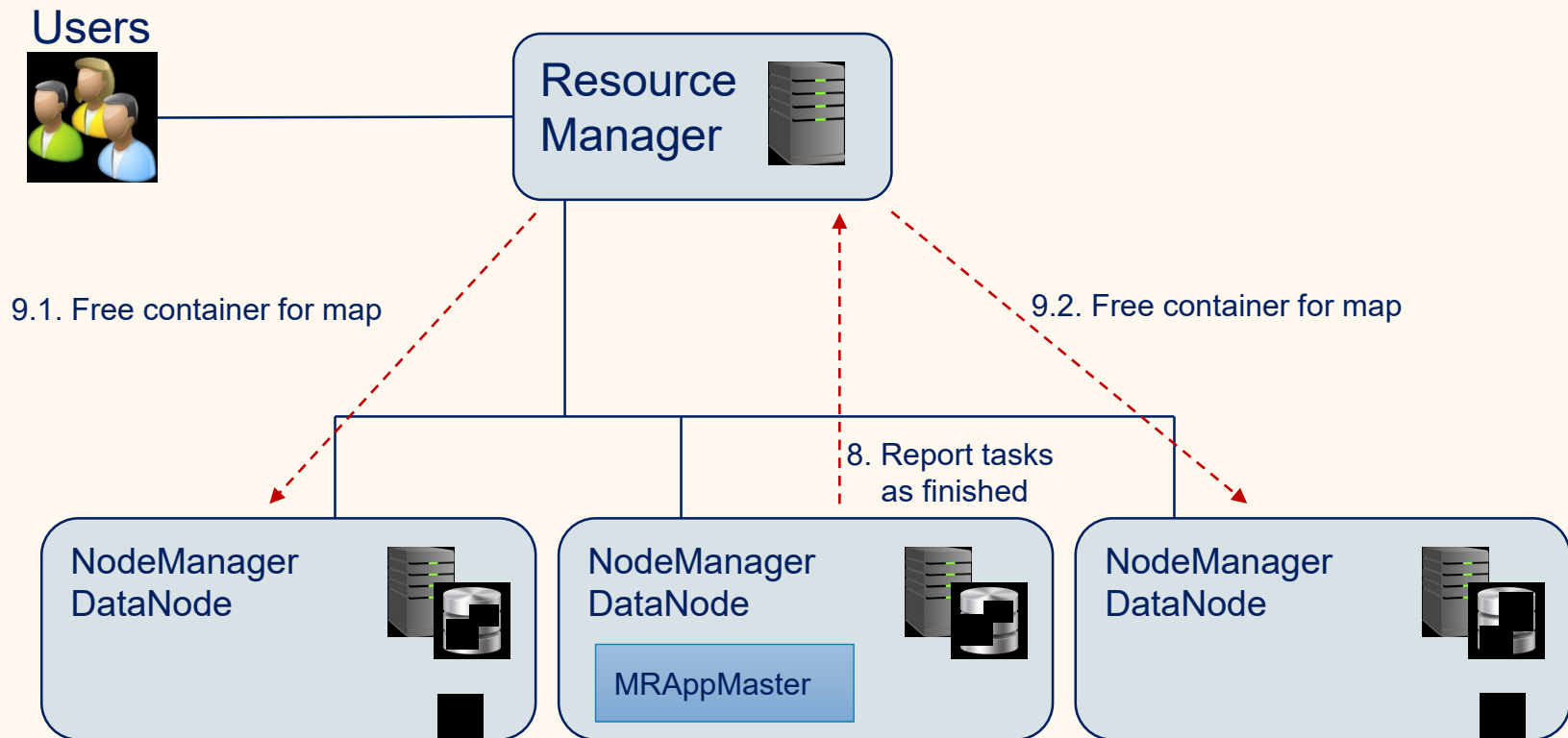
# Execution of Word Count with Hadoop



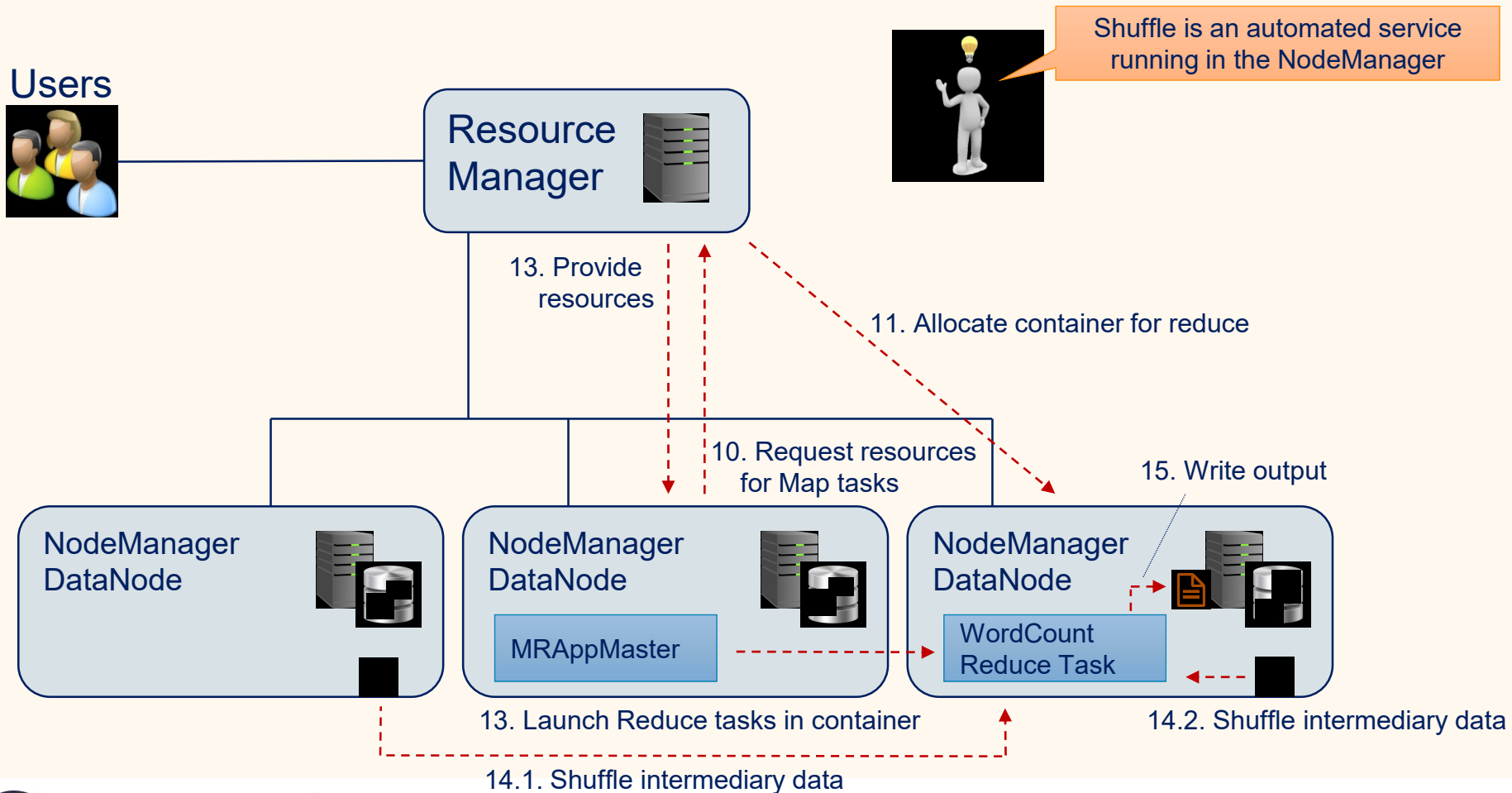
# Execution of Word Count with Hadoop



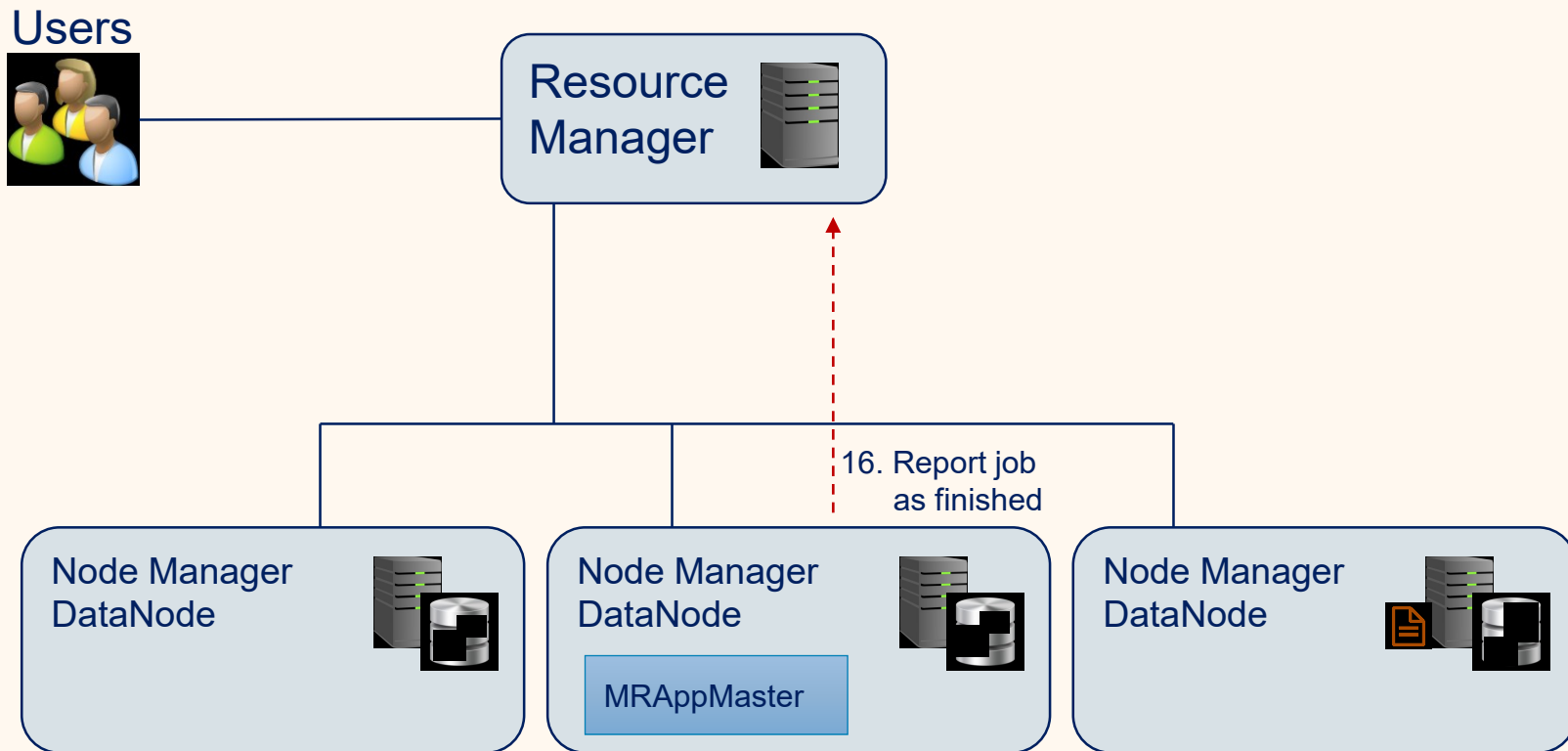
# Execution of Word Count with Hadoop



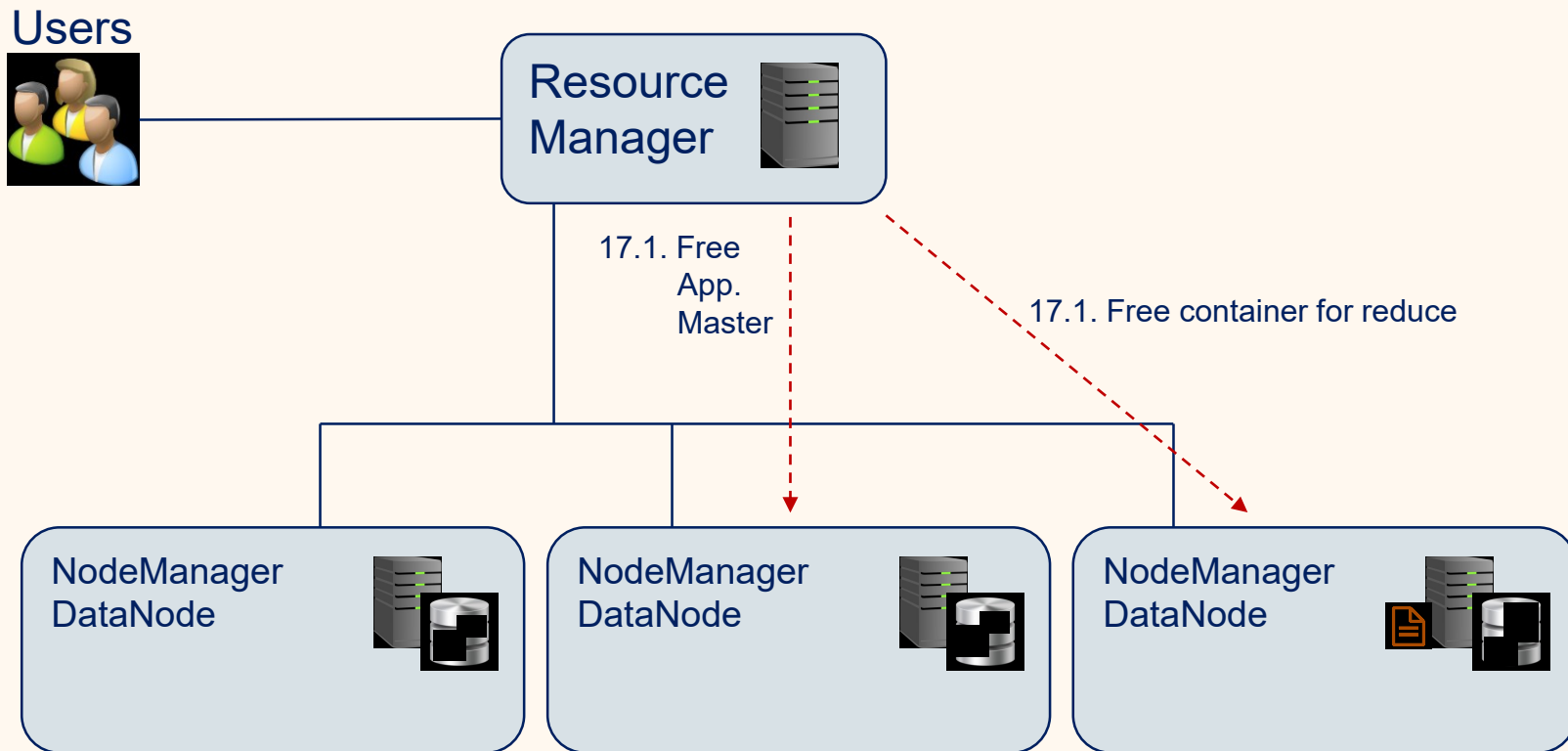
# Execution of Word Count with Hadoop



# Execution of Word Count with Hadoop



# Execution of Word Count with Hadoop



# Streaming Mode of Hadoop

- Provided Java Application that implements Hadoop MapReduce
- Input and output by line-wise file processing
  - Same as the handler we showed in the word count example
  - Formats can be defined using command line parameters
- Mapper/Reducer can be any executable application

```
hadoop jar hadoop-streaming-2.9.1.jar \  
-input myInputDirs \  
-output myOutputDir \  
-mapper mapper.py \  
-reducer reducer.py \  
-file mapper.py \  
-file reducer.py
```

Java Application that implements  
Hadoop MapReduce

Input and output locations

Executables used as  
mapper/reducer

Copies local files to compute  
nodes



# Word Count with Python

```
#!/usr/bin/env python
"""mapper.py"""

import sys

# read from standard input
for line in sys.stdin:
    # split the line into words
    words = line.strip().split()
    # increase counters
    for word in words:
        # print output pairs to standard output
        # key and value separated by tabulator
        print '%s\t%s' % (word, 1)
```

```
#!/usr/bin/env python
"""reducer.py"""

from operator import itemgetter
import sys

# init current word and counter as not existing
current_word = None
current_count = 0
word = None

# read from standard input
for line in sys.stdin:
    # read output from mapper.py
    word, count = line.strip().split('\t', 1)
    count = int(count)

    # Hadoop shuffle sorts by key
    # -> all values with same key are next to each other
    if current_word == word:
        # same word -> increase count
        current_count += count
    else:
        if current_word:
            # write result to standard output
            print('%s\t%s' % (current_word, current_count))
        # reset counter and update current word
        current_count = count
        current_word = word

# output for last word
if current_word == word:
    print('%s\t%s' % (current_word, current_count))
```

# Additional Important Parts of Hadoop

- **Combiner**
  - Reducer function that runs locally before shuffling the data to another node
  - Often same as reducer, but not always
    - Requires functions to be chainable / idempotent
  - Can reduce network traffic
  - Example:
    - Word count reducer can also be used as combiner
    - The shuffled pairs would not all have a count of 1 anymore, but the word counts of the data of that node
- **MapReduce Job History Server**
  - Collects information about the history of MapReduce jobs
    - Log files, start/end times, job state
  - Can be used by users to see status of their jobs

# Limitations of Hadoop

- Multiple Map and/or Reduce steps require multiple jobs
  - Can still be defined in a single Java file
  - Output of one job can be input of another job
  - Jobs can have dependencies, e.g., one waiting for the completion of other jobs
  - Dependencies must be modeled by the programmer
- All communication between jobs via the file system
  - Bad for multiple computations on the same data, e.g., chained map functions

# Outline

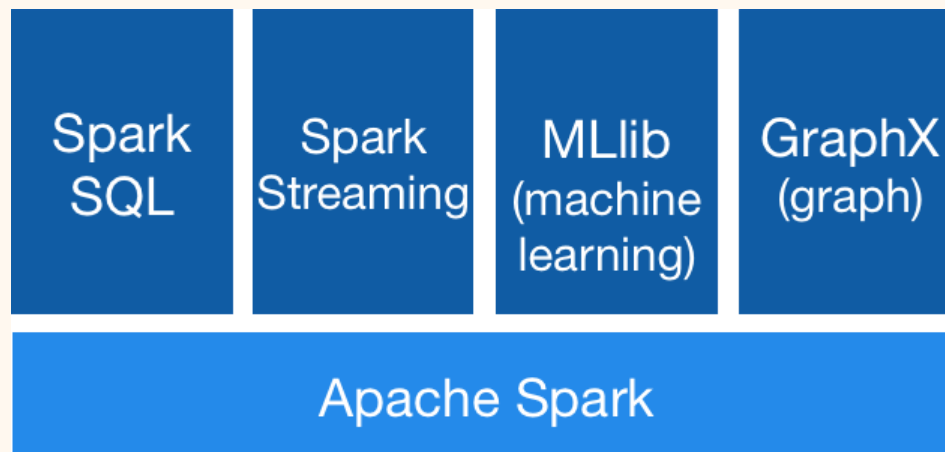
- Overview
- MapReduce
- Apache Hadoop
- **Apache Spark**
- Summary

# Apache Spark



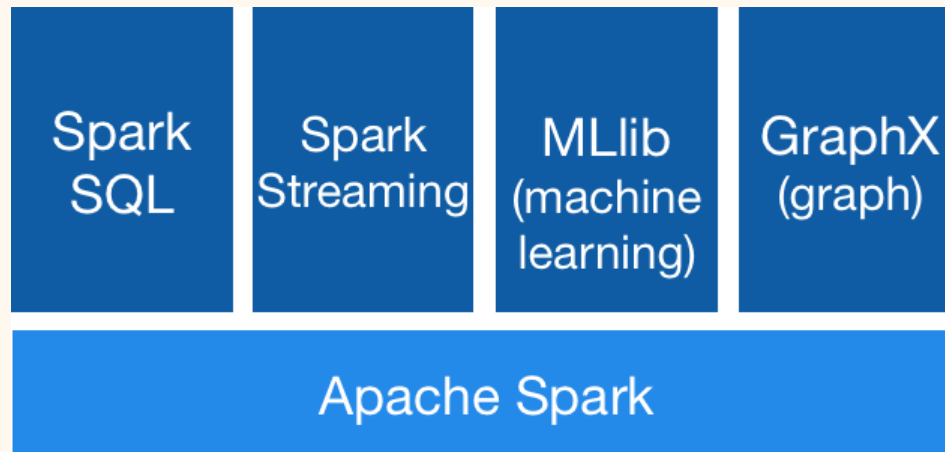
- Engine for large-scale data processing
- Designed to resolve limitations of Hadoop for data analysis
- Supports in-memory analysis
  - Good support for iterative algorithms
- Supports arbitrary combinations of Map and Reduce tasks
  - Users do not need to care about specific jobs and their dependencies

# Spark Stack (I)



- **SparkSQL** for SQL-like queries and data frame generation
- **Spark Streaming** for live processing of streaming data

# Spark Stack (II)



- **MLlib** for machine learning algorithms
  - > 20 algorithms for clustering, regression, and classification
- **GraphX** for graph algorithms

# Data Structures used by Spark

- Not a file system like Hadoop, instead two important in-memory data structures
- Resilient Distributed Dataset (RDD)
  - Abstraction layer for data operations
  - Immutable partitions of elements
  - All elements in a RDD can be processed in parallel
  - Support map, reduce, filtering, user-defined functions, and persistence
- Data frame
  - Higher abstraction built on RDDs
  - Similar to R / pandas data frames
  - Usually generated using the SparkSQL API



# Infrastructures to Execute Apache Spark

- Spark allows setup of clusters for computing
  - Not the standard way to use spark
- Compatible with many existing infrastructures instead
- Computing
  - Hadoop/YARN and Apache Mesos
- Storage
  - Hadoop/HDFS, Cassandra, HBase, MongoDB, ...

# Programming with Apache Spark

- Natively implemented in Scala
  - JVM language with type inference and functional programming concepts
- Also provides APIs for
  - Java
  - Python (PySpark)
  - R (SparkR)

# Word Count with PySpark

```
text_file = sc.textFile("hdfs://data.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://wc.txt")
```

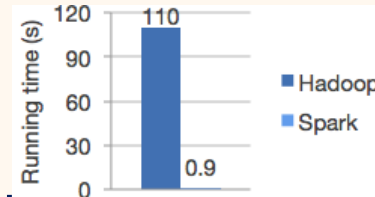
flatMap can map input to multiple outputs

reduceByKey reduces the data set by merging (i.e. reducing) key-value pairs with the same key

Python lambda functions: anonymous function with parameters a,b that returns the result of the computation after the colon, i.e., a+b

# Two major differences to Hadoop

- In-memory
  - RDDs and data frames are handled in-memory if possible
  - Vast speed-up
    - Example: Logistic Regression



- Does not provide own distributed storage back-end

# Spark Ecosystem

- Different execution modes
- Large open source community
- Many technologies on top of Spark
  - <https://spark-packages.org/>
- Still rapidly developing
  - Core concepts stable
  - 2.4.0 release this fall
  - 3.0.0 release planned for next year

# Outline

- Overview
- MapReduce
- Apache Hadoop
- Apache Spark
- **Summary**

# Summary

- Big data processing requires dedicated infrastructures
  - Moving data not feasible
  - Move computation to data
- MapReduce as programming model for big data processing
- Apache Hadoop as big data framework
  - HDFS distributed and reliable file system
  - YARN for resource management
  - Provides a MapReduce framework
- Apache Spark for in-memory computations with big data
  - Compatible with different infrastructures, including Hadoop
  - Provides API for machine learning