MapReduce in Spark

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Software Development Technologies
Master studies, second semester
Academic year 2017/18 (winter course)

Review of the previous lectures

- Mining of massive datasets.
- Classification and regression.
- Evolution of database systems:
  - Operational (OLTP) vs. analytical (OLAP) systems.
  - Relational model vs. multidimensional model.
  - Design of data warehouses.
  - NoSQL.
  - Processing of massive datasets.

Outline

Outline

Motivation

- Traditional DBMS vs. NoSQL

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- Computational burden $\rightarrow$ distributed systems

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- Computational burden $\rightarrow$ distributed systems
  - Scaling-out instead of scaling-up

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- Computational burden $\rightarrow$ distributed systems
  - Scaling-out instead of scaling-up
  - Move-code-to-data

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust – are intended to run on commodity hardware; desinged with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust – are intended to run on commodity hardware; desinged with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.
- Scalable – scales linearly to handle larger data by adding more nodes to the cluster.

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust – are intended to run on commodity hardware; desinged with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.
- Scalable – scales linearly to handle larger data by adding more nodes to the cluster.
- Simple – allow users to quickly write efficient parallel code.

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?
- How much can we gain by such implementation?

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?
- How much can we gain by such implementation?
- Let us focus on the word count problem ...

Word count

- Count the number of times each word occurs in a set of documents:

Do as I say, not as I do.

Word	Count
as	2
do	2
i	2
not	1
say	1

Word count

- Let us write the procedure in pseudo-code for a single machine:

Word count

- Let us write the procedure in pseudo-code for a single machine:

```
define wordCount as Multiset;

for each document in documentSet {
   T = tokenize(document);

   for each token in T {
      wordCount[token]++;
   }

}

display(wordCount);
```

Word count

- Let us write the procedure in pseudo-code for many machines:

Word count

- Let us write the procedure in pseudo-code for many machines:
  - First step:

```
define wordCount as Multiset;

for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}

sendToSecondPhase(wordCount);
```

Word count

- Let us write the procedure in pseudo-code for many machines:
  - First step:

```
define wordCount as Multiset;

for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}

sendToSecondPhase(wordCount);
```

  - Second step:

```
define totalWordCount as Multiset;

for each wordCount received from firstPhase {
   multisetAdd (totalWordCount, wordCount);
}
```

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition the intermediate data (that is, wordCount) from phase one.

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition the intermediate data (that is, wordCount) from phase one.
  - Shuffle the partitions to the appropriate machines in phase two.

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition the intermediate data (that is, wordCount) from phase one.
  - Shuffle the partitions to the appropriate machines in phase two.
  - Ensure fault tolerance.

Outline

MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:

MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:
  - Map: the map function is written to convert input elements to key-value pairs.

MapReduce

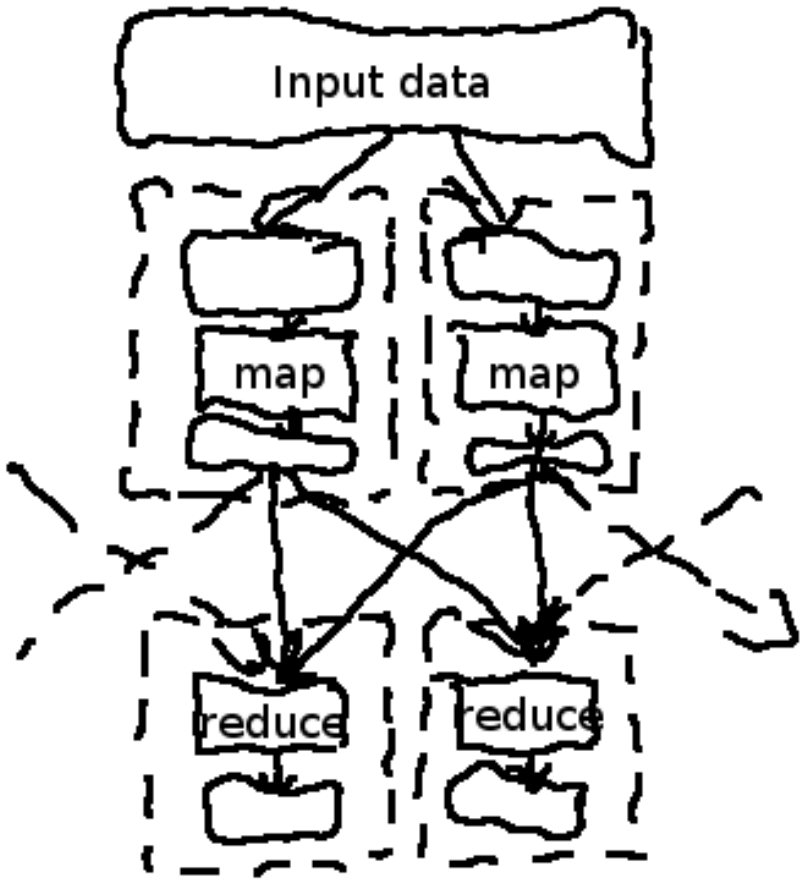
- MapReduce programs are executed in two main phases, called mapping and reducing:
  - Map: the map function is written to convert input elements to key-value pairs.
  - Reduce: the reduce function is written to take pairs consisting of a key and its list of associated values and combine those values in some way.

MapReduce

- The complete data flow:

	Input	Output
map	`(<k1, v1>)`	`list(<k2, v2>)`
reduce	`(<k2, list(<v2>)`	`list(<k3, v3>)`

Figure: The complete data flow



MapReduce

- The complete data flow:

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - The key-value pairs are processed in arbitrary order.

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - The key-value pairs are processed in arbitrary order.
  - The output of all the mappers are (conceptually) aggregated into one giant list of `<k2,v2>` pairs. All pairs sharing the same `k2` are grouped together into a new aggregated key-value pair: `<k2,list(v2)>`.

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - The key-value pairs are processed in arbitrary order.
  - The output of all the mappers are (conceptually) aggregated into one giant list of `<k2,v2>` pairs. All pairs sharing the same `k2` are grouped together into a new aggregated key-value pair: `<k2,list(v2)>`.
  - The framework asks the reducer to process each one of these aggregated key-value pairs individually.

Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.

Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.
- **Combiner** – perform local aggregation (the reduce step) on the map node.

Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.
- **Combiner** – perform local aggregation (the reduce step) on the map node.
- **Partitioner** – divide the key space of the map output and assign the key-value pairs to reducers.

WordCount in MapReduce

- Map:
  - For a pair `<k1,document>` produce a sequence of pairs `<token,1>`, where `token` is a token/word found in the document.

```
map(String filename, String document) {
    List<String> T = tokenize(document);

    for each token in T {
      emit ((String)token, (Integer) 1);
    }

}
```

WordCount in MapReduce

- Reduce
  - For a pair <word, list(1, 1, ..., 1)> sum up all ones appearing in the list and return <word, sum>, where sum is the sum of ones.

```
reduce(String token, List<Integer> values) {
    Integer sum = 0;

    for each value in values {
        sum = sum + value;
    }

    emit ((String)token, (Integer) sum);
}
```

Matrix-vector Multiplication

- Let $\boldsymbol{A}$ to be large $n \times m$ matrix, and $\boldsymbol{x}$ a long vector of size m.
- The matrix-vector multiplication is defined as:

Matrix-vector Multiplication

- Let $\boldsymbol{A}$ to be large $n \times m$ matrix, and $\boldsymbol{x}$ a long vector of size m.
- The matrix-vector multiplication is defined as:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{v},$$

where $\boldsymbol{v} = (v_1, \ldots, v_n)$ and

$$v_i = \sum_{j=1}^{m} a_{ij} x_j.$$

Matrix-vector multiplication

- Let us first assume that m is large, but not so large that vector $\boldsymbol{x}$ cannot fit in main memory, and be part of the input to every Map task.
- The matrix $\boldsymbol{A}$ is stored with explicit coordinates, as a triple (i, j, a_{ij}).
- We also assume the position of element x_j in the vector $\boldsymbol{x}$ will be stored in the analogous way.

Matrix-vector multiplication

- **Map**:

Matrix-vector multiplication

- **Map**: each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.

Matrix-vector multiplication

- **Map**: each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.
- **Reduce**:

Matrix-vector multiplication

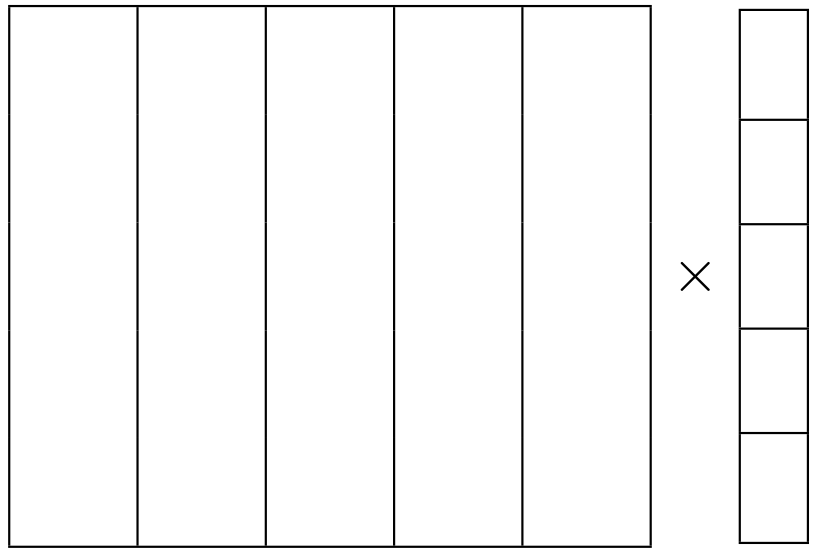
- **Map**: each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.
- **Reduce**: a reduce task has simply to sum all the values associated with a given key i. The result will be a pair (i, v_i) where:

$$v_i = \sum_{j=1}^{m} a_{ij}x_j.$$

Matrix-Vector Multiplication with Large Vector v

Matrix-Vector Multiplication with Large Vector v

- Divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height.



- The ith stripe of the matrix multiplies only components from the ith stripe of the vector.
- Thus, we can divide the matrix into one file for each stripe, and do the same for the vector.

Matrix-Vector Multiplication with Large Vector v

- Each Map task is assigned a chunk from one the stripes of the matrix and gets the entire corresponding stripe of the vector.
- The Map and Reduce tasks can then act exactly as in the case where Map tasks get the entire vector.

Outline

Spark

- Spark is a fast and general-purpose cluster computing system.

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,
  - and Spark Streaming.

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,
  - and Spark Streaming.
- For more check `https://spark.apache.org/`

Spark

- Spark uses Hadoop which is a popular open-source implementation of MapReduce.

Spark

- Spark uses Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.

Spark

- Spark uses Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.
- Hadoop Distributed File System (HDFS) is responsible for distributed storage.

Installation of Spark

- Download Spark from

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`
- To play with Spark there is no need to install HDFS ...

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`
- To play with Spark there is no need to install HDFS ...
- But, you can try to play around with HDFS.

HDFS

- Create new directories:

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

- Copy the input files into the distributed filesystem:

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

```
hdfs dfs -ls /user/myname/
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

```
hdfs dfs -ls /user/myname/
hdfs dfs -cat /user/myname/data.txt
```

WordCount in Hadoop

```
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;

public class WordCount {

 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 in Hadoop

```
  (...)
  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);
    }
  }

  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 in Spark

- The same code is much simpler in Spark
- To run the Spark shell type: `./bin/spark-shell`
- The code

```
val textFile = sc.textFile("~/data/all-bible.txt")
val counts = (textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _))
counts.saveAsTextFile("~/data/all-bible-counts.txt")
```

Alternatively:

```
val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).
     count()
```

Matrix-vector multiplication in Spark

- The Spark code is quite simple:

```
val x = sc.textFile("~/data/x.txt").map(line => {val t = line.split(","); (t(0).
    trim.toInt, t(1).trim.toDouble)})
val vectorX = x.map{case (i,v) => v}.collect
val broadcastedX = sc.broadcast(vectorX)
val matrix = sc.textFile("~/data/M.txt").map(line => {val t = line.split(","); (t
    (0).trim.toInt, t(1).trim.toInt, t(2).trim.toDouble)})
val v = matrix.map { case (i,j,a) => (i, a * broadcastedX.value(j-1))}.reduceByKey(
    _ + _)
v.toDF.orderBy("_1").show
```

Outline

Summary

- Motivation: New data-intensive challenges like search engines.
- MapReduce: The overall idea and simple algorithms.
- Spark: MapReduce in practice.

Bibliography

- A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*.
  Cambridge University Press, 2011
  `http://infolab.stanford.edu/~ullman/mmds.html`

- J.Lin and Ch. Dyer. *Data-Intensive Text Processing with MapReduce*.
  Morgan and Claypool Publishers, 2010
  `http://lintool.github.com/MapReduceAlgorithms/`

- Ch. Lam. *Hadoop in Action*.
  Manning Publications Co., 2011