

# Big Data Analytics

## C. Distributed Computing Environments / C.2. Resilient Distributed Datasets: Apache Spark

Lars Schmidt-Thieme

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

# Syllabus

- |            |      |   |
|------------|------|---|
| Tue. 10.4. | (1)  | 0. Introduction                                   |
|            |      | <b>A. Parallel Computing</b>                      |
| Tue. 17.4. | (2)  | A.1 Threads                                       |
| Tue. 24.4. | (3)  | A.2 Message Passing Interface (MPI)               |
| Tue. 1.5.  | —    | — <i>Labour Day</i> —                             |
| Tue. 8.5.  | (4)  | A.3 Graphical Processing Units (GPUs)             |
| Tue. 15.5. | (5)  | (ctd.)  |
| Tue. 22.5. | —    | — <i>Pentecoste Break</i> —                       |
|            |      | <b>B. Distributed Storage</b>                     |
| Tue. 29.5. | (6)  | B.1 Distributed File Systems                      |
| Tue. 5.6.  | (7)  | B.2 Partitioning of Relational Databases          |
| Tue. 12.6. | (8)  | B.3 NoSQL Databases                               |
|            |      | <b>C. Distributed Computing Environments</b>      |
| Tue. 19.6. | (9)  | C.1 Map-Reduce                                    |
| Tue. 26.6. | (10) | C.2 Resilient Distributed Datasets (Spark)        |
| Tue. 3.7.  | (11) | C.3 Computational Graphs (TensorFlow)             |
|            |      | <b>D. Distributed Machine Learning Algorithms</b> |
| Tue. 10.7. | (12) | D.1 Distributed Stochastic Gradient Descent       |

# Outline

1. Introduction
2. Apache Spark
3. Working with Spark
4. MLlib: Machine Learning with Spark

# Outline

1. Introduction

2. Apache Spark

3. Working with Spark

4. MLlib: Machine Learning with Spark

# Core Idea

To implement fault-tolerance for primary/original data:

- ▶ **replication**:
  - ▶ partition large data into parts
  - ▶ store each part several times on different servers
  - ▶ if one server crashes, the data is still available on the others

To implement fault-tolerance for secondary/derived data:

- ▶ **replication**

# Core Idea

To implement fault-tolerance for primary/original data:

- ▶ **replication**:
  - ▶ partition large data into parts
  - ▶ store each part several times on different servers
  - ▶ if one server crashes, the data is still available on the others

To implement fault-tolerance for secondary/derived data:

- ▶ **replication** or
- ▶ **resilience**:
  - ▶ partition large data into parts
  - ▶ for each part, store how it was derived (**lineage**)
    - ▶ from which parts of its input data
    - ▶ by which operations
  - ▶ if a server crashes, recreate its data on the others

# How to store data derivation?

## journal

- ▶ sequence of elementary operations
  - ▶ set an element to a value
  - ▶ remove a value/index from a list
  - ▶ insert a value at an index of a list
  - ▶ ...
- ▶ generic: supports all types of operations
- ▶ but too large
  - ▶ often same size as data itself

# How to store data derivation?

## journal

- ▶ sequence of elementary operations
  - ▶ set an element to a value
  - ▶ remove a value/index from a list
  - ▶ insert a value at an index of a list
  - ▶ ...
- ▶ generic: supports all types of operations
- ▶ but too large
  - ▶ often same size as data itself

## coarse-grained transformations

- ▶ just store
  - ▶ the executable code of the transformations and
  - ▶ the input
    - ▶ either primary data or itself an RDD



# Resilient Distributed Datasets (RDD)

Represented by 5 components:

1. **partition**: a list of parts
2. **dependencies**: a list of parent RDDs
3. **transformation**: a function to compute the dataset from its parents
4. **partitioner**: how elements are assigned to parts
5. **preferred locations**: which hosts store which parts

# Resilient Distributed Datasets (RDD)

Represented by 5 components:

1. **partition**: a list of parts
2. **dependencies**: a list of parent RDDs
3. **transformation**: a function to compute the dataset from its parents
4. **partitioner**: how elements are assigned to parts
5. **preferred locations**: which hosts store which parts

distinction into two types of dependencies:

- ▶ **narrow dependencies**:  
each parent part is used to derive at most one part of the dataset
- ▶ **wide dependencies**:  
some parent part is used to derive several parts of the dataset

# How to cope with expensive operations?

## checkpointing:

- ▶ traditionally,
  - ▶ a long process is broken into several steps A, B, C etc.
  - ▶ after each step, the state of the process is saved to disk
  - ▶ if the process crashes within step B,
    - ▶ it does not have to be run from the very beginning
    - ▶ but can be restarted at the beginning of step B reading its state at the end of step A.

# How to cope with expensive operations?

## checkpointing:

- ▶ traditionally,
  - ▶ a long process is broken into several steps A, B, C etc.
  - ▶ after each step, the state of the process is saved to disk
  - ▶ if the process crashes within step B,
    - ▶ it does not have to be run from the very beginning
    - ▶ but can be restarted at the beginning of step B reading its state at the end of step A.
  
- ▶ in a distributed scenario,
  - ▶ “saving to disk” is not fault-tolerant
  - ▶ replicate the data instead (**distributed checkpointing**)

# Caching

- ▶ RDDs are marketed as technology for in memory cluster computing
- ▶ derived RDDs are not saved to disks, but kept in (distributed) memory
- ▶ derived RDDs are saved to disks on request (checkpointing)
- ▶ allows faster operations

# Limitations

- ▶ RDDs are read-only
  - ▶ as updating would invalidate them as input for possible derived RDDs
- ▶ transformations have to be deterministic
  - ▶ otherwise lost parts cannot be recreated the very same way
  - ▶ for stochastic transformations: store random seed

For more conceptual details see the original paper

- ▶ Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (2012).

# Outline

1. Introduction
2. Apache Spark
3. Working with Spark
4. MLlib: Machine Learning with Spark

# Apache Spark Overview

an open source framework for large scale data processing and analysis.

## Main Ideas:

- ▶ Processing occurs where the data resides
- ▶ Avoid moving data over the network
- ▶ Works with the data in memory



# Apache Spark Overview

an open source framework for large scale data processing and analysis.

## Main Ideas:

- ▶ Processing occurs where the data resides
- ▶ Avoid moving data over the network
- ▶ Works with the data in memory

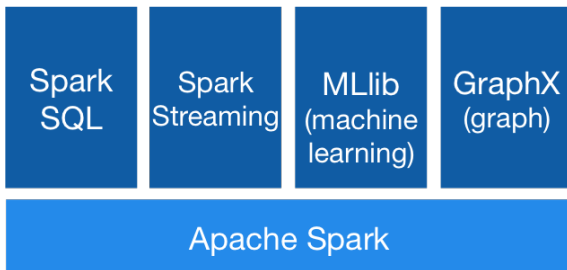
## Technical details:

- ▶ Written in Scala
- ▶ Works seamlessly with Java, Python and R
- ▶ Developed at UC Berkeley

# Apache Spark Stack

- ▶ **Data platform:** Distributed file system /data base
  - ▶ Ex: HDFS, HBase, Cassandra
- ▶ **Execution Environment:** single machine or a cluster
  - ▶ Standalone, EC2, YARN, Mesos
- ▶ **Spark Core:** Spark API
- ▶ **Spark Ecosystem:** libraries of common algorithms
  - ▶ MLlib, GraphX, Streaming

# Apache Spark Ecosystem



# How to use Spark

Spark can be used through:

- ▶ The **Spark Shell**
  - ▶ Available in Python and Scala
  - ▶ Useful for learning the framework
- ▶ **Spark Applications**
  - ▶ Available in Python, Java and Scala
  - ▶ For “serious” large scale processing

# Outline

1. Introduction
2. Apache Spark
3. Working with Spark
4. MLlib: Machine Learning with Spark

# Working with Spark

- ▶ Working with Spark requires accessing a **Spark Context**:
  - ▶ Main entry point to the Spark API
  - ▶ Already preconfigured in the shell
- ▶ Most of the work in Spark is a set of operations on **Resilient Distributed Datasets (RDDs)**:
  - ▶ Main data abstraction
  - ▶ The data used and generated by the application is stored as RDDs

# Spark Java Application

```
1 import org.apache.spark.api.java.*;
2 import org.apache.spark.SparkConf;
3 import org.apache.spark.api.java.function.Function;
4
5 public class HelloWorld {
6     public static void main(String[] args) {
7         String logFile = "/home/lst/system/spark/README.md";
8         SparkConf conf = new SparkConf().setAppName("Simple Application");
9         JavaSparkContext sc = new JavaSparkContext(conf);
10        JavaRDD<String> logData = sc.textFile(logFile).cache();
11
12        long numAs = logData.filter(new Function<String, Boolean>() {
13            public Boolean call(String s) { return s.contains("a"); }
14        }).count();
15
16        long numBs = logData.filter(new Function<String, Boolean>() {
17            public Boolean call(String s) { return s.contains("b"); }
18        }).count();
19
20        System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
21    }
22 }
```

# Compile and Run

0. install spark (here in `~/system/spark`)

1. compile:

```
1 javac -cp ~/system/spark/lib/spark-assembly-1.6.1-hadoop2.6.0.jar HelloWorld.java
```

2. create jar archive:

```
1 jar cf HelloWorld.jar HelloWorld*.class
```

3. run:

```
1 ~/system/spark/bin/spark-submit --master local --class HelloWorld HelloWorld.jar
```



# Spark Interactive Shell (Python)

```
1 > ./bin/pyspark
2 Welcome to
3
4   /_/_/_/_/
5  /_/_/_/_/
6  /_/_/_/_/
7  /_/_/_/_/
8
9 Using Python version 2.7.6 (default, Jun 22 2015 17:58:13)
10 SparkContext available as sc, SQLContext available as sqlContext.
11 >>>
```

# Spark Context

The Spark Context is the main entry point for the Spark functionality.

- ▶ It represents the connection to a Spark cluster
- ▶ Allows to create RDDs
- ▶ Allows to broadcast variables on the cluster
- ▶ Allows to create Accumulators

# Resilient Distributed Datasets (RDDs)

- ▶ A Spark application stores data as RDDs
  - ▶ **Resilient**: if data in memory is lost it can be recreated (fault tolerance)
  - ▶ **Distributed**: stored *in memory* across different machines
  - ▶ **Dataset**: data coming from a file or generated by the application
- ▶ A Spark program is about operations on RDDs
  - ▶ RDDs are **immutable**: operations on RDDs may create new RDDs but never change them

# Resilient Distributed Datasets (RDDs)

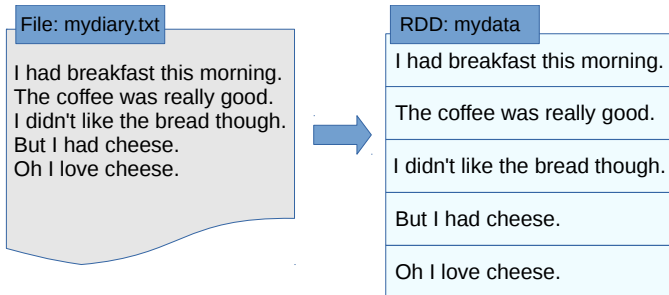


- ▶ RDD elements can be stored in different machines
  - ▶ transparent to the developer
- ▶ data can have various data types

# RDD Data types

- ▶ An element of an RDD can be of any type as long as it is **serializable**
- ▶ Examples:
  - ▶ Primitive data types: integers, characters, strings, floats, ...
  - ▶ Sequences: lists, arrays, tuples ...
  - ▶ Pair RDDs: key-value pairs
  - ▶ Serializable Scala/Java objects
- ▶ A single RDD may contain elements of different types
- ▶ Some specific element types have additional functionality

# Example: Text file to RDD



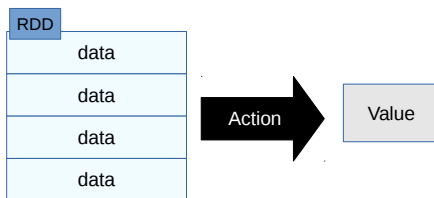
# RDD Operations / Actions

There are two types of RDD operations:

1. **Actions**: return a value based on the RDD.

► Examples:

- `count`: returns the number of elements in the RDD
- `first()`: returns the first element in the RDD
- `take(n)`: returns an array with the first  $n$  elements in the RDD

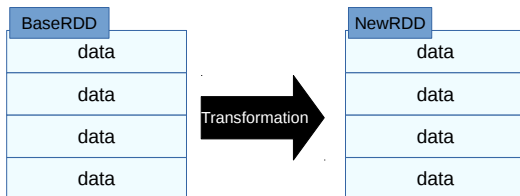


# RDD Operations / Transformations

2. **Transformations**: create a new RDD based on the current one.

► Examples:

- **filter**: returns the elements of an RDD which match a given criterion
- **map**: applies a particular function to each RDD element
- **reduce**: aggregates the elements of a specific RDD

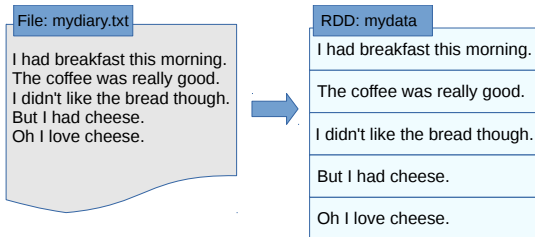


► RDD transformations are **lazy**

- they are only executed once an action requires a result to be computed



# Action Examples

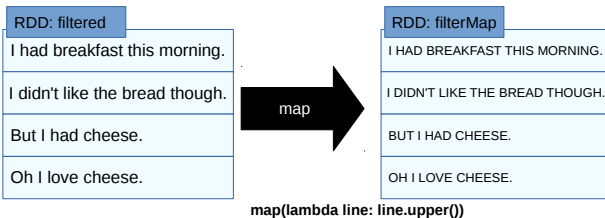
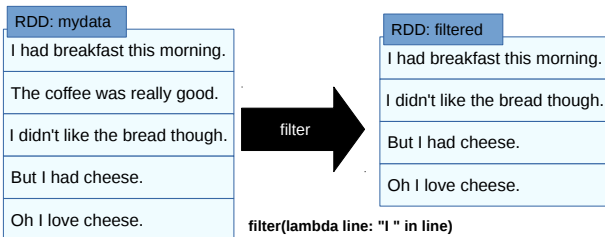


```

1 >>> mydata = sc.textFile("mydiary.txt")
2 >>> mydata.count()
3 5
4 >>> mydata.first()
5 u'I had breakfast this morning.'
6 >>> mydata.take(2)
7 [u'I had breakfast this morning.', u'The coffee was really good.']

```

# Transformation Examples



# Transformation Examples

```
1 >>> filtered = mydata.filter(lambda line: "I" in line)
2 >>> filtered.count()
3 4
4 >>> filtered.take(4)
5 [u'I had breakfast this morning.',
6  u'I didn't like the bread though.",
7  u'But I had cheese.',
8  u'Oh I love cheese. ']
9 >>> filterMap = filtered.map(lambda line: line.upper())
10 >>> filterMap.count()
11 4
12 >>> filterMap.take(4)
13 [u'I HAD BREAKFAST THIS MORNING.',
14  u" I DIDN'T LIKE THE BREAD THOUGH.",
15  u' BUT I HAD CHEESE.',
16  u' OH I LOVE CHEESE. ']
```

# Operations on Numbers

Numeric RDDs have special operations:

- ▶ `mean()`

- ▶ `min()`

- ▶ `max()`

- ▶ ...

```
1 >>> linelens = mydata.map(lambda line: len(line))
2 >>> linelens.collect()
3 [29, 27, 31, 17, 17]
4 >>> linelens.mean()
5 24.2
6 >>> linelens.min()
7 17
8 >>> linelens.max()
9 31
10 >>> linelens.stddev()
11 6.0133185513491636
```

# Operations on Key-Value Pairs

- ▶ Pair RDDs contain pairs (tuples of two elements):  $(K, V)$ 
  - ▶ elements called keys and values
- ▶ Keys and values can be of any type
- ▶ Useful for implementing MapReduce algorithms
- ▶ Examples of operations:
  - ▶ `groupByKey`
  - ▶ `reduceByKey`
  - ▶ `aggregateByKey`
  - ▶ `sortByKey`
  - ▶ ...

## Word Count Example

Map:

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

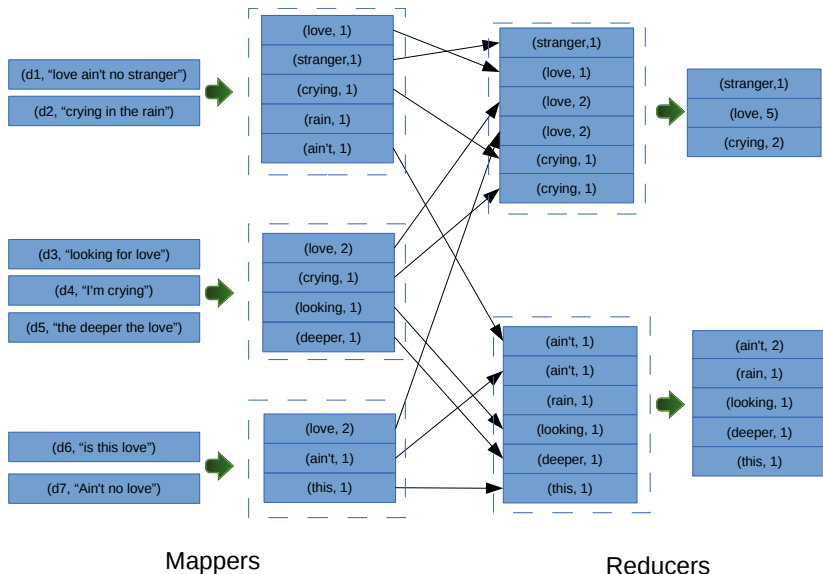
$$(d_n, (w_1, \dots, w_M)) \mapsto ((w, c_w)_{w \in W: c_w > 0})$$

Reduce:

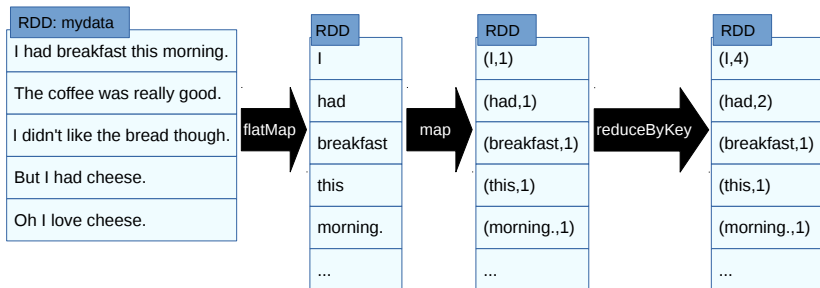
- ▶ Input: word-(count list) pairs
- ▶ Output: word-count pairs

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

# Word Count Example



# Word Count on Spark



```

1 >>> counts = mydata.flatMap(lambda line: line.split("_")) \
2     .map(lambda word: (word, 1)) \
3     .reduceByKey(lambda x, y: x + y)
  
```



# ReduceByKey

```
1 .reduceByKey(lambda x, y: x + y)
```

- ▶ ReduceByKey works somewhat different from the MapReduce reduce function:

- ▶ It combines two values associated with the same key.
- ▶ Must be **commutative**:

$$\text{reduceByKey}(x, y) = \text{reduceByKey}(y, x)$$

- ▶ Must be **associative**:

$$\begin{aligned} \text{reduceByKey}(\text{reduceByKey}(x, y), z) = \\ \text{reduceByKey}(x, \text{reduceByKey}(y, z)) \end{aligned}$$

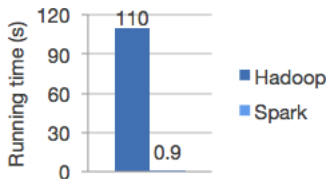
- ▶ Spark does not guarantee in which order the reduceByKey functions are executed.

# Shuffle Operations

- ▶ operations are broken automatically into tasks
- ▶ each task is executed separately, each on some node
  - ▶ e.g., for **ReduceByKey**: one task for each key
- ▶ each task needs all its input data locally on the node it is executed on
  - ▶ e.g., for **ReduceByKey**: all key/value pairs for a specific key
- ▶ if data is not already partitioned by that key, it needs to be repartitioned (**shuffled**)
  - ▶ expensive operation (network, IO)

# MapReduce in Spark and Hadoop

- ▶ Spark provides a much more efficient MapReduce implementation than Hadoop:
  - ▶ Higher level API
  - ▶ In memory storage (less I/O overhead)
  - ▶ Chaining MapReduce operations is simplified
    - ▶ sequence of MapReduce passes can be done in one job
- ▶ Spark vs. Hadoop on training a logistic regression model:



Source: Apache Spark. <https://spark.apache.org/>

# Outline

1. Introduction
2. Apache Spark
3. Working with Spark
4. MLlib: Machine Learning with Spark

# Overview

MLlib is a Spark Machine Learning library containing implementations for:

- ▶ Computing Basic Statistics from Datasets
- ▶ Classification and Regression
- ▶ Collaborative Filtering
- ▶ Clustering
- ▶ Feature Extraction and Dimensionality Reduction
- ▶ Frequent Pattern Mining
- ▶ Optimization Algorithms

# Logistic Regression with MLLib

## 1. Import required packages:

```
1 from pyspark.mllib.regression import LabeledPoint
2 from pyspark.mllib.util import MLUtils
3 from pyspark.mllib.classification import LogisticRegressionWithSGD
```

## 2. Read the data (LibSVM format):

```
1 dataset = MLUtils.loadLibSVMFile(sc, 'data/mllib/sample_libsvm_data.txt')
```

## 3. Train the model:

```
1 model = LogisticRegressionWithSGD.train(dataset)
```

## 4. Evaluate the model:

```
1 labelsAndPreds = dataset.map(lambda p: (p.label, model.predict(p.features)))
2 trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count()
3 / float(dataset.count())
4 print('Training Error={}'.format(trainErr))
```

# Summary

- ▶ **Resilient Distributed Datasets (RDDs)** make large-scale distributed computations **fault-tolerant**
  - ▶ RDDs keep track of **dependencies**:
    - ▶ which part of a output data depends on which part of the input data
    - ▶ which coarse-grained **transformation** is used
  - ▶ if a server storing a part of a dataset fails, this part will be **recomputed**.
- ▶ RDDs are **immutable**, transformations have to be deterministic.
  - ▶ so that RDDs can be recomputed always exactly the same
- ▶ RDDs provide **in memory cluster computing**
  - ▶ derived RDDs can be saved to disk on request (**checkpointing**)

# Further Readings

- ▶ Original Spark paper:
  - ▶ Zaharia et al. [2010]
- ▶ A broad introduction to Spark:
  - ▶ Karau et al. [2015]



# References

- Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark, Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015.
- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, pages 10–10, Boston, MA, 2010. USENIX Association.