Big Data Analytics

7. Resilient Distributed Datasets: Apache Spark

Lars Schmidt-Thieme

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

original slides by Lucas Rego Drumond, ISMLL

Outline

Outline

Core Idea

To implement fault-tolerance for primary/original data:

- **replication**:
  - partition large data into parts
  - store each part on 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 on 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 a 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 depenencies**:
  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

Spark Overview

Apache Spark is 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

Spark Overview

Apache Spark is 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
- Work 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

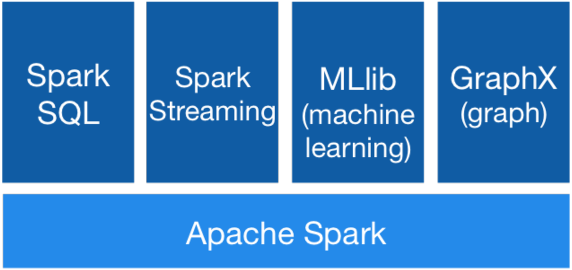
- Standalone, EC2, YARN, Mesos

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

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

```
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class HelloWorld {
  public static void main(String[] args) {
    String logFile = "/home/lst/system/spark/README.md";
    SparkConf conf = new SparkConf().setAppName("Simple␣Application");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("a"); }
    }).count();

    long numBs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("b"); }
    }).count();

    System.out.println("Lines␣with␣a:␣" + numAs + ",␣lines␣with␣b:␣" + numBs);
  }
}
```

Compile and Run

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

1. compile:

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

2. create jar archive:

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

3. run:

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

Spark Interactive Shell (Python)

```
$ ./bin/pyspark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.3.1
      /_/

Using Python version 2.7.6 (default, Jun 22 2015 17:58:13)
SparkContext available as sc, SQLContext available as sqlContext.
>>>
   
```

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**

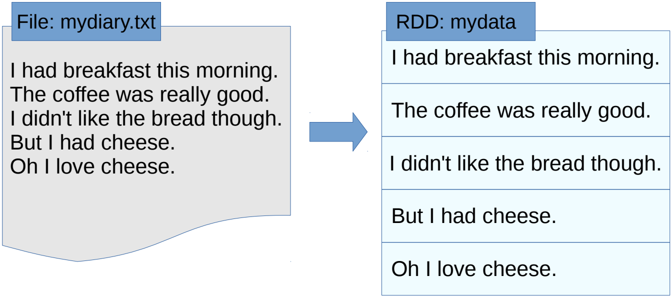
Example:

- Primitive data types: integers, characters, strings, floating point numbers, ...
- Sequences: lists, arrays, tuples ...
- Pair RDDs: key-value pairs
- Serializable Scala/Java objects

A single RDD may have elements of different types

Some specific element types have additional functionality

Example: Text file to RDD



RDD operations

There are two types of RDD operations:

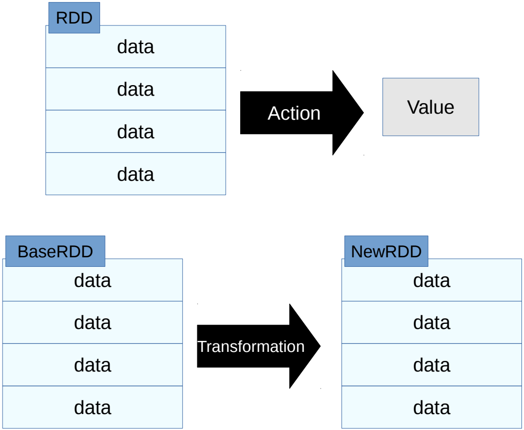
- **Actions**: return a value based on the RDD
- Example:
  - `count`: returns the number of elements in the RDD
  - `first()`: returns the first element in the RDD
  - `take(n)`: returns an array with the first *n* elements in the RDD

RDD operations

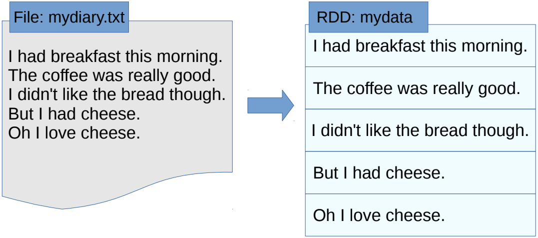
There are two types of RDD operations:

- **Actions**: return a value based on the RDD
- Example:
  - `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
- **Transformations**: creates a new RDD based on the current one
- Example:
  - `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

Actions vs. Transformations

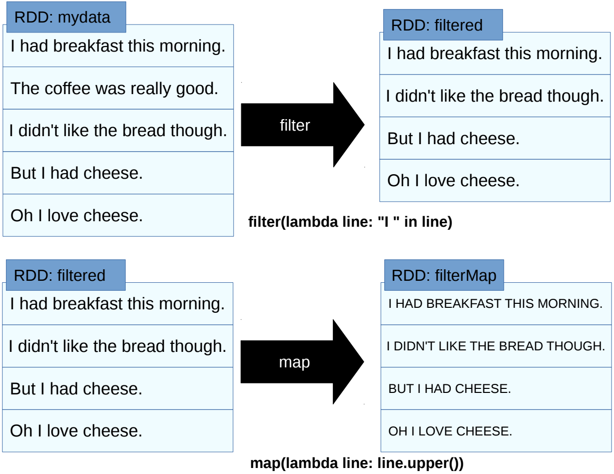


Actions examples




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

Transformation examples



Transformations examples

```python
>>> filtered = mydata.filter(lambda line: "I␣" in line)
>>> filtered.count()
4
>>> filtered.take(4)
[u'I␣had␣breakfast␣this␣morning.',
 u"I␣didn't␣like␣the␣bread␣though.",
 u'But␣I␣had␣cheese.',
 u'Oh␣I␣love␣cheese.']
>>> filterMap = filtered.map(lambda line: line.upper())
>>> filterMap.count()
4
>>> filterMap.take(4)
[u'I␣HAD␣BREAKFAST␣THIS␣MORNING.',
 u"I␣DIDN'T␣LIKE␣THE␣BREAD␣THOUGH.",
 u'BUT␣I␣HAD␣CHEESE.',
 u'OH␣I␣LOVE␣CHEESE.']
```

Operations on specific types

Numeric RDDs have special operations:

- mean()
- min()
- max()
- ...

```
>>> linelens = mydata.map(lambda line: len(line))
>>> linelens.collect()
[29, 27, 31, 17, 17]
>>> linelens.mean()
24.2
>>> linelens.min()
17
>>> linelens.max()
31
>>> linelens.stdev()
6.0133185513491636
```

Operations on Key-Value Pairs

Pair RDDs contain a two element tuple: (K, V)

Keys and values can be of any type

Extremely 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

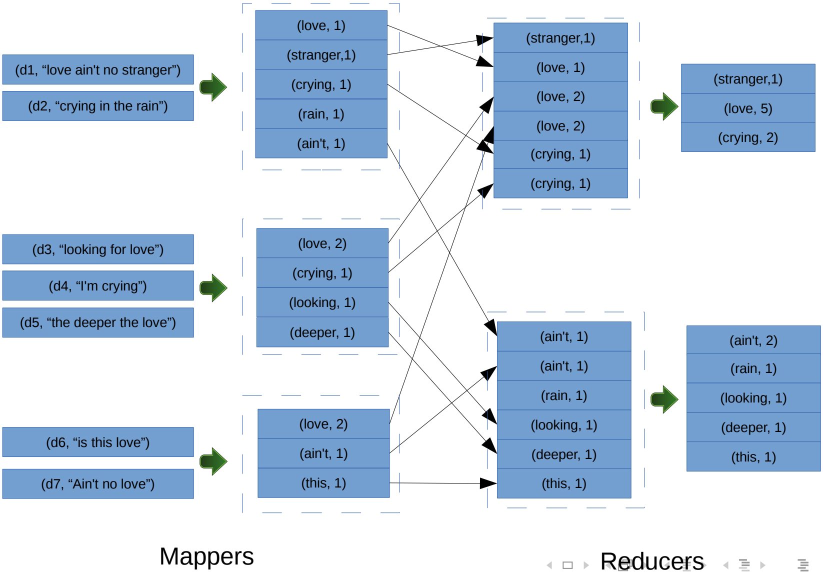
$$(d_k, \text{``}w_1, \ldots, w_m'') \mapsto [(w_i, c_i)]$$

Reduce:

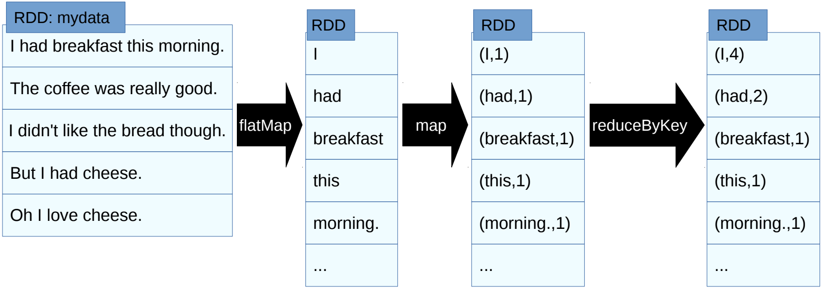
- Input: word-(count list) pairs
- Output: word-count pairs

$$(w_i, [c_i]) \mapsto (w_i, \sum_{c \in [c_i]} c)$$

Word Count Example



Word Count on Spark




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

ReduceByKey

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

`ReduceByKey` works a little different from the MapReduce `reduce` function:

- It takes two arguments: combines two values at a time associated with the same key
- Must be **commutative**: `reduceByKey(x,y) = reduceByKey(y,x)`
- Must be **associative**: `reduceByKey(reduceByKey(x,y), z) = reduceByKey(x,reduceByKey(y,z))`

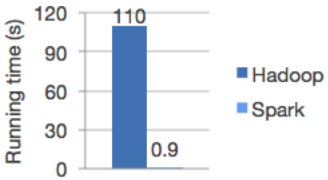
Spark does not guarantee on which order the `reduceByKey` functions are executed!

Considerations

Spark provides a much more efficient MapReduce implementation then 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

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

Import necessary packages:

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

Read the data (LibSVM format):

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

Logistic Regression with MLLib

Train the Model:

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

Evaluate:

```
labelsAndPreds = dataset
    .map(lambda p: (p.label, model.predict(p.features)))
trainErr = labelsAndPreds
    .filter(lambda (v, p): v != p)
    .count() / float(dataset.count())
print("Training Error = " + str(trainErr))
```