Data Analytics

with Spark

Peter Vanroose

GSE NL Nat.Conf.

16 November 2017

Almere - Van Der Valk

“Digital Transformation”

Data Analytics with Spark

Outline :

- Data analytics - history
- Spark and its predecessors
- Spark and Scala - some examples
- Spark libraries: SQL, streaming, MLlib, GraphX

Data analytics
1

Wikipedia:

- process of **inspecting**, cleansing, transforming, **modeling** data;
  => **discover** info, suggest conclusions, support **decision-making**
- Related terms: business intelligence (BI); data mining; statistics

Business Intelligence (BI):

- relies heavily on **aggregation**; focus on business information

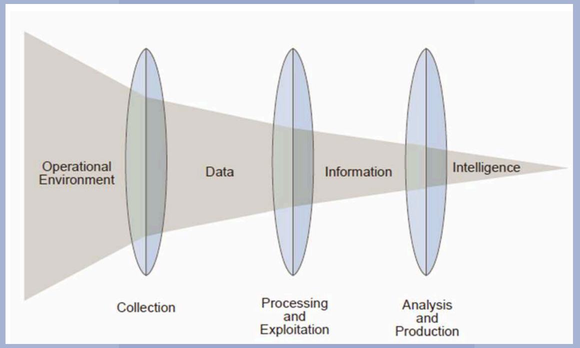
Data mining:

- modeling & knowledge discovery for **predictive** purposes

Statistical data analysis:

- **descriptive** statistics: **visualisation** (scatter plot, histogram, ...)
- **exploratory** data analysis (EDA): discover "features" in data
- **confirmatory** data analysis (CDA): confirm/falsify hypotheses
- **predictive** analytics: build statistical **models** => classification
  e.g. linear regression; text analytics

Spark Analytics



RDBMS (e.g. Db2) 2.1

- On-Line Analytical Processing (OLAP)
  - **aggregation** (SUM, COUNT, AVG) + grouping sets
- can easily answer **BI** questions, like:
  - turnover, revenue => overview per year, month, region, product
  - TOP-10 analysis (10 best customers, 10 most promising new markets, 10 least profitable products, ...)

    => requires “total sorting” (= n log n) + showing just first part

    could use pre-sorted data (indexes) => not always possible !
- typical setup: data **warehouses**
  - make data **available** to BI tools (ETL)
  - heavy pre-sorting & pre-summarizing
  - dimensional modeling (several granularities)

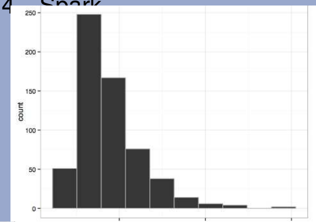
Data & data analytics - the “classic” tools (2)

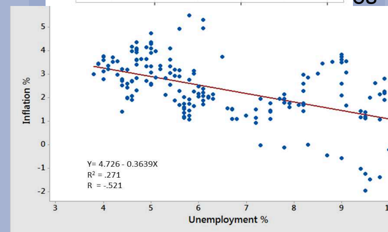
Statistical software (e.g. SPSS, R) 2.2

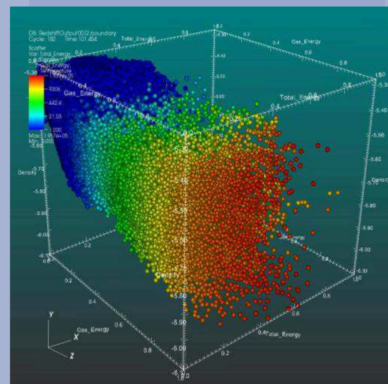
- graphical possibilities (better than Excel)
  - scatter plot (correlation), heat map, ...
  - histogram (frequency distrib.), bar chart (ranking), pie chart, ...
  - time series (line chart)
  - geographic, geospatial
- statistical functionality
  - hypothesis testing (e.g. t-test); confidence intervals
  - normality tests
  - **modeling** (linear regression, correlation); with reliability
  - **clustering**; pattern recognition; (un)supervised learning
  - trend analysis
  - support for taking business policy decisions

    => answer questions like “what if price increased”
- **Machine Learning** = new term for old functionality ...

Spark Analytics







Data & data analytics - the “classic” tools (3)

Typical “machine learning” applications 2.3

- **Examples:**
  - **spam filters**
  - **virus scanners**
  - **break-in detection**
  - **OCR**
  - **search engines with “educated guesses” cf Google search**
- **clustering**
- **dimension reduction, e.g. PCA (principal component analysis)**
- **pattern recognition**
  - trained from labeled “training” data (“golden” data)
  - originated from computer vision
- **classification, e.g. decision trees**

Enormous amounts of data 3.1

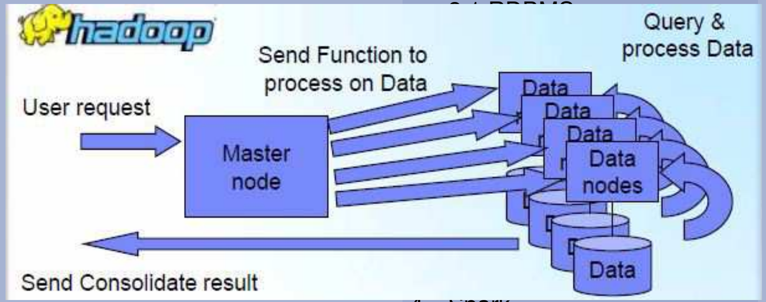
- **the 3 Vs** => need for a new framework?
  - volume (TB / PB / ZB / YB)
  - velocity (real-time analysis)
  - variety (unstructured & semi-structured data)
- “Big Data” => **Hadoop**
  - assumes a cluster of commodity hardware (**sharding** - scale out)
  - fail-safe because of **redundance**
- but ... less data **consistency** guarantees
  - because of the CAP theorem (Brewer, 2000)

    can only have 2 out of 3: **c**onsistency, **a**vailablility, **p**artitioned
  - BASE instead of ACID
- Hadoop’s analytical frame work: **MapReduce**

  => “access path” responsibility : the programmer

Spark Analytics

Hadoop



- **Apache project (`http://hadoop.apache.org/`)**
- **implemented in Java => runs in JVM**
- **“Function to Data”:**
  - **partitioned data resides on different cluster nodes**
  - **parallelized algorithm runs on all data nodes (=processing nodes)**
  - **data flow is important!**

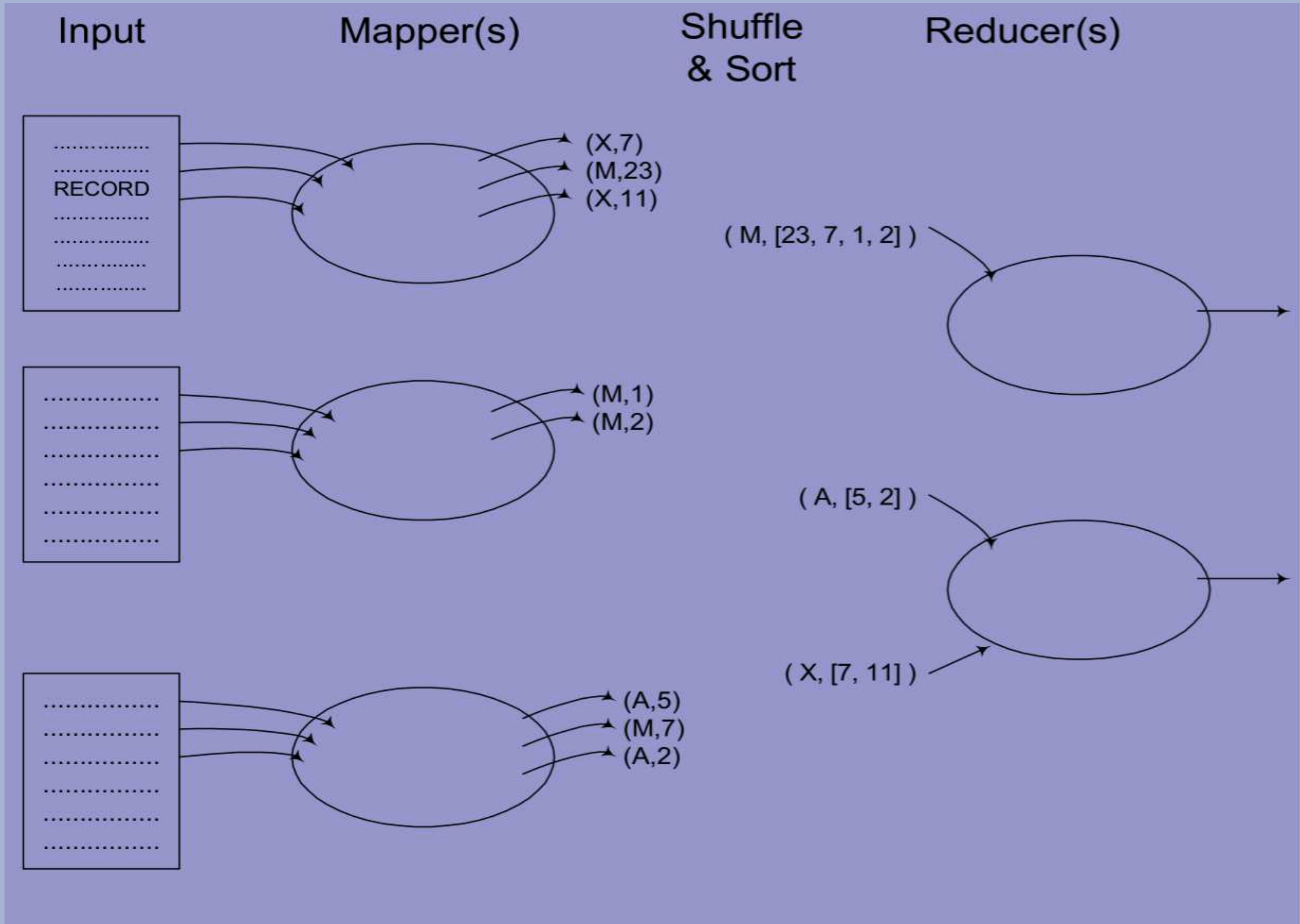
    *** minimize data flow between nodes**

    *** optimize data flow between consecutive steps of algorithm**

    => Directed Acyclic Graph (DAG) between MapReduce steps

    => “clever” combination of different Map & Reduce steps
    for optimal performance

Hadoop: MapReduce



Spark Analytics

Hadoop: HDFS & Yarn

HDFS 3.4

- **Hadoop Distributed File System**
- **storage abstraction layer**
  - **single HDFS “file” is actually a set of fragments / partitions**
  - **residing on different cluster nodes**
  - **with duplicates (replication factor; default: 3)**
- **end user sees a “normal” hierarchical file system**

  hdfs:/user/peter/myfile.txt
  - **command-line interface (Linux style) & API**
    - put & get files between client & cluster
    - move/rename, remove, append to
    - head & tail
    - no update !

Yarn 3.5

- **Yet Another Resource Negotiator => job scheduler for MR steps**

Spark Analytics

Hadoop: Pig and Hive

- **Pig** (an Apache project – http://pig.apache.org/)
  - High-level language interface, compiles into Hadoop MapReduce
  - Easily readable formulation for standard design patterns
  - Data is represented as “objects”, “variables”
  - Example:

```
logs = LOAD 'mytext.txt' USING PigStorage(' ');              /* space-delimited input */
data = FOREACH logs GENERATE $0 AS ip, $6 AS webpage;    /* fields 0 and 6 */
valid = FILTER data BY  ip MATCHES '^10(\\.\\d+){3}$';      /* a valid IP address */
STORE valid INTO 'weblog.out';
```

- **Hive** (an Apache project – http://hive.apache.org/)
  - SQL-like interface
  - like Pig, translates “standard” questions into optimal MapReduce implementation
  - Example:

```
CREATE TABLE weblog (ip STRING, ..., webpage STRING)
            ROW FORMAT DELIMITED  FIELDS TERMINATED BY ' '  ;
SELECT     webpage,COUNT(*)
FROM        weblog      WHERE ip LIKE '10.%'
GROUP BY webpage;
```

Spark Analytics

Data & data analytics - the “Big Data” tools (5)

Spark

3.7

- **has learned from Big Data history (esp. Hadoop, Hive) & from R, Python, Jupyter Notebook, Zeppelin, Mahout, Storm, Avro, ...**
- **tries to combine the best elements of all its predecessors**
- **top-down approach instead of bottom-up:**
  - **good, simple user interface, prevent making “stupid mistakes”:**
    - **fast prototyping**: command interface (interactively)
    - provide for same programming language for the final algorithm (e.g. to run multiple times, or in a continuous setup)
    - provide for a data flow pipeline via **immutable** objects & their methods ==> *functional programming*
  - **provides for simple integration with existing frameworks:**
    - data sources & sinks: HDFS, local filesystem, URLs, data streams
    - Hadoop framework (which runs on Java and hence on JVM)
    - Yarn or similar resource negotiator / workload balancer
    - Simple RDBMS interface; connections to Cassandra, MongoDB, ...
- **better than its predecessors:** e.g. **in-memory** where possible

- **Spark from scratch:**
  - **no need for a cluster**
    - develop & test on stand-alone system (local or cloud)
    - your Spark prototype programs will easily deploy on cluster
  - **download & install software on a Linux system**
    - or download a preconfigured virtual image (VMware / VirtualBox)
      e.g. CDH from https://www.cloudera.com/downloads/
      or HDP from https://hortonworks.com/downloads/
  - **a typical Spark installation also contains**
    - Hadoop (with HDFS, MapReduce, Yarn) or Mesos
    - Java 8 compiler (JDK 1.8)
    - Scala compiler
- **preconfigured cloud solutions available**
  - **AWS (Amazon Web Services) EMR (Elastic MapReduce), EC2**
  - **Google Cloud Platform(`https://cloud.google.com/hadoop/`)**
  - **IBM Cloud: `https://www.ibm.com/cloud/spark` (Watson, BigInsights)**

Spark - command-line interface

```
[Linux]$ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc (master = local[*], app id = local-1510673299900).
SQL context available as sqlContext.

scala>
```

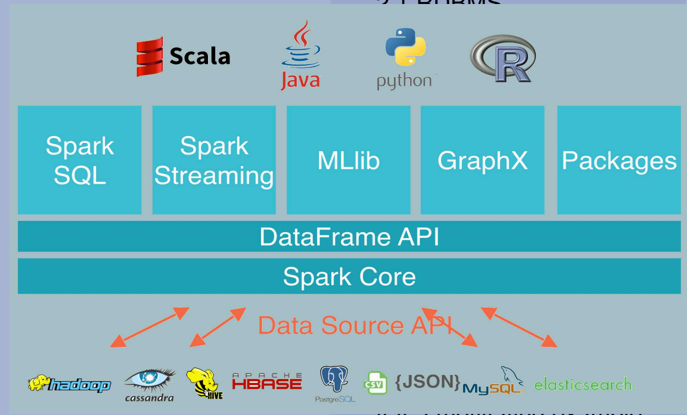
- 1st principal **interface** is interactive command-line: **Spark shell**
- shell syntax: uses **Scala**
  - a (new) functional programming language
  - Spark is itself largely implemented in Scala
  - Scala *can* run on JVM (hence bytecode compatible with Java)
- similar interface provided for Python, R (, Java)

Spark - design

4.2

- **A unified computing engine + set of libraries (& APIs)**
- **For parallel data processing on a cluster**
- **Hides the “ugly details” of MapReduce**
  - **user steps in at a level similar to Pig or Hive**
  - **Spark translates your data flow processing requests to distributed algorithms**

    (not necessarily MapReduce)
- **Spark “core” provides**
  - **limited set of “transformations” & “actions” (see further)**
  - **on distributed data objects (so-called RDDs)**
- **RDD: Resilient Distributed Dataset: a data abstraction**
  - **similar to an RDBMS table, an R data frame, or a JSON file**
  - **but distributed (cluster)**
  - **accessible though a “handle”: a (Scala) object (= variable)**
  - **lazy evaluation where possible:** program flow = DAG (dependencies)



Spark Analytics

Spark - history

- 2009-2012: Berkeley research project (AMPLab)
- 2010: open sourced (BSD license)
- 2013: original authors started Databricks
- 2013: Apache project (Apache license)
- Febr. 2014: top-level Apache project
- > 1000 contributors
- versions:
  - May 2014: version 1.0 (Nov. 2016: v. 1.6.3)
  - July 2016: version 2.0 (July 2017: v. 2.2.0)
- Spark Summits:
  - December 2013: San Francisco
  - since then: every year; 3-day event;
    (Europe: October 2016: Brussels; October 2017: Dublin)
- **IBM**: "strategic product" (partnership Databricks; integr. BlueMix)

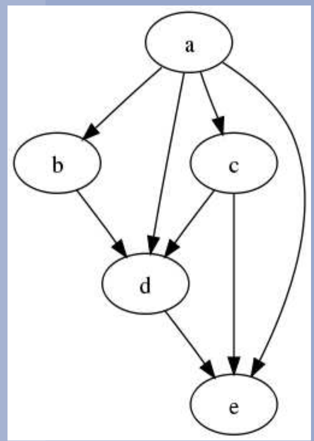
A motivating example

Suppose we have an HDFS file mytext.txt, containing some text.
Count the word frequencies in the file, and write the answer to HDFS file count.out :

```
[Linux]$ wget -O mytext.txt https://nl.lipsum.com/feed/html?amount=150
[Linux]$ hadoop fs -put mytext.txt
[Linux]$ spark-shell
scala> val textFile = sc.textFile("hdfs:/user/peter/mytext.txt")
textFile: org.apache.spark.rdd.RDD[String] = hdfs:/user/peter/mytext.txt MapPartitionsRDD[1]
scala> val words = textFile.flatMap( line => line.split(" ") )
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2]
scala> val words_as_key_val = words.map( word => (word, 1) )   // or just:   map( (_, 1) )
words_as_key_val: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3]
scala> val words_with_counts = words_as_key_val.reduceByKey( (v1,v2) => v1 + v2 )
words_with_count: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4]
scala> words_with_counts.saveAsTextFile("hdfs:/user/peter/count.out")
[Linux]$ hadoop fs -ls count.out
-rw-r--r--   1 peter users          0 2017-11-16 15:23 count.out/_SUCCESS
-rw-r--r--   1 peter users       6395 2017-11-16 15:23 count.out/part-00000
-rw-r--r--   1 peter users       6262 2017-11-16 15:23 count.out/part-00001
[Linux]$ hadoop fs -cat count.out/*
(interdum,42)
(mi.,22)
(erat,60)
(fames,13)
(urna,48)
(nunc,,16)
<etc...>
[Linux]$ spark-shell       #   do the same, using a single spark (scala) instruction:
scala> sc.textFile("hdfs:/user/peter/mytext.txt").flatMap(_.split(" ")).map( (_, 1) ).
     |    reduceByKey(_ + _).saveAsTextFile("hdfs:/user/peter/count2")
```

Transformations & actions

- **Both can be applied on RDDs**
  (or actually: RDDs have "methods" of both types)
- **Transformations convert an RDD into a new RDD**
  - **since RDDs are immutable, they never change once created**
  - **the new RDD is not "instantiated":**
    a transformation is just a dependency between two RDDs
  - **multiple transformations can be applied to one RDD => DAG**
- **Only when action is applied, the full dependency chain is activated**
  (including all intermediate transformations)
  - **examples: write to physical data stream; show on screen**
  - **result of action is not an RDD (but local variable)**
- **On activation:**
  - **transformations can be combined into a single MapReduce step**
  - ***notorious example*: sorting followed by top-n filtering**



Spark core

- **Provides basic support for RDDs & basic transformations & actions**
  - the "Spark **context**" (sc) is the user "handle" to the cluster
  - RDD: immutable key-value list; stored on the cluster
    (on HDFS, or in a NoSQL database, or cached in memory, ...)
  - examples of **transformations**:

read from file: `a = sc.textFile("source name or URL")`

create from local:

```
p = sc.parallelize(Array(2,3,5,7,11))
l = sc.range(1,1001)
```

shorten (filter) the RDD list, e.g. based on a text search criterion:

```
b = a.filter( x => x.contains("search-term") )
```

(note the "=>" notation (**lambda expression**): filter arg is *function*)

"vertical" transformation:
e.g. split in words, take 5th element, take largest of two, ...:

```
c = b . map( x => x.split(" ") ) // treat rows separately
d = c . map( x => if (x(0) > x(1)) x(0) else x(1) )
e = b . flatMap( x => x.split(" ") )  // "flat list"

g = e . map(x => (x,1)) . reduceByKey( (v1,v2) => v1+v2 )
```

Spark Analytics

Spark core (2)

- **examples of actions:**

  summaries, like `c.count()`

  "generic" summaries, like finding row with max 10th field:

```
maxc = c . reduce( (a,b) => if (a(9)>b(9)) a else b )
```

  explicitly converting RDD to local (non-RDD) variable:

```
l = c . collect()  // full RDD as Array
v = c . first()    // first element of that array
w = c . take(5)    // first 5 elements
t = c . top(5)     // last 5 elements
```

- **caching** data for faster access, i.e., load in memory on cluster:

```
cc = c.cache()
```

Spark SQL and DataFrames

- **DataFrame:**
  - name and concept comes from **R**
  - is sort of RDD (distributed data value):
    - * is like an RDBMS **table**: with rows & columns
    - * columns have names; default names: _1, _2, etc.
    - * in contrast to RDD, storage is **columnwise**
  - RDD can be converted to DataFrame with method `toDF()`
  - more prominent since Spark version 2.x
- **Spark SQL**
  - add-on library of Spark
  - similar to Hive
  - manipulates DataFrames using (standard) SQL
  - can read in DataFrames from Avro, Parquet, ORC, JSON, JDBC
  - allows to e.g. join tables from different sources

Spark SQL and DataFrames (2)

Example:

```
[Linux]$ spark-shell
scala> val courses = sc.parallelize(Array(
             (1067,"Db2 for z/OS fundamentals",3,475.00),
             (  87,"SQL workshop",2,450.00),
             (1686,"Big data in practice using Spark",2,500.00),
             (  25,"SAS programming fundamentals",3,450.00) ) )
courses: org.apache.spark.rdd.RDD[(Int,String,Int,Double)] =  ParallelCollectionRDD[1]
scala> val coursetable = courses.toDF("cid","ctitle","cdur","cdprice")
coursetable: org.apache.spark.sql.DataFrame =[cid:int,ctitle:string,cdur:int,cdprice:double]
scala> coursetable.show()
+----+--------------------+----+-------+
| cid|              ctitle|cdur|cdprice|
+----+--------------------+----+-------+
|1067|Db2 for z/OS fund...|   3|  475.0|
|  87|        SQL workshop|   2|  450.0|
|1686|Big data in pract...|   2|  500.0|
|  25|SAS programming f...|   3|  450.0|
+----+--------------------+----+-------+
scala> val cheap = coursetable .where("cdprice < 500") .filter(col("ctitle").like("%Db2%"))

// Only from here on, we start using the Spark SQL library:
scala> coursetable.registerTempTable("courses")
scala> val tot = sqlContext.sql("SELECT sum(cdur*cdprice) AS total
                                  FROM courses WHERE cdprice < 500")
tot: org.apache.spark.sql.DataFrame = [total: double]
scala> tot.collect()
res2: Array[org.apache.spark.sql.Row] = Array([3675.0])
```

- **Production applications should run in (e.g.) the JVM**
  and access the cluster through e.g. Yarn, Mesos, or stand-alone
- **Production version (compiled Scala program)**
  should not differ too much from the *Fast prototyping* version
  (created in iteractive spark-shell)
- **Example:**

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object MyProg {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("MyProg").setMaster("local[4]")
    val context = new SparkContext(conf)
    val textFile = context.textFile(args(0))
    val words = textFile.flatMap( line => line.split(" ") )
    val words_as_key_val = words.map( word => (word, 1) )
    val words_with_counts = words_as_key_val.reduceByKey( (v1,v2) => v1 + v2 )
    words_with_counts.saveAsTextFile(args(1))
    context.stop()
  }
}

#  compile the above into a jar file, then:
[Linux] spark-submit --class MyProg    MyJar.jar   hdfs:/user/peter/mytext.txt \
                                                    hdfs:/user/peter/count.out
```

the Spark APIs (2)

Similar programming interfaces exist for:

- **Java 8**
  - program will look very similar to Scala version ...
  - running in the JVM is 100% identical to running a Scala program
- **Python**
  - is an interpreted language => no compiling necessary
  - interactive or non-interactive Python script:

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("MyProg").setMaster("local[4]")
sc = SparkContext(conf=conf)
textFile = sc.textFile("hdfs:/user/peter/mytext.txt")
<etc...>
```

- **R**
  - is an interpreted language => no compiling necessary
  - interactive or non-interactive R script:

```
install.packages("sparkR", dep=TRUE)   # needed only once
library(sparkR)                        # optionally "import" it
sc <- sparkR.init()
sqlContext <- sparkRSQL.init(sc)
<etc...>
```

Spark Streaming

- For data "in motion": live data streams
  (not stored in e.g. HDFS)
  - examples: Twitter feeds, audio/video streams
    typically through sockets or TCP/IP ports
  - supported sources include Kafka, Flume, Twitter, Kinesis, ...
  - data not stored any longer than needed for processing
  - data will be "cut up" into **batches** of given size & given overlap
- **DStream** (discretized stream) object: is an RDD *sequence*
- Example:

```
import org.apache.spark.streaming._
val ssc = new StreamingContext(sc, Seconds(1))        // batch interval: 1 second
val lines = ssc.socketTextStream("localhost", 50000)  // port 50000 on localhost
val words = lines.flatMap(_.split(" "))               // a DStream object
val words_with_counts = words.map((_, 1)).reduceByKey(_ + _)
words_with_counts.print()
// the above will run once a second :
ssc.start() ; ssc.awaitTermination()
```

Spark Analytics

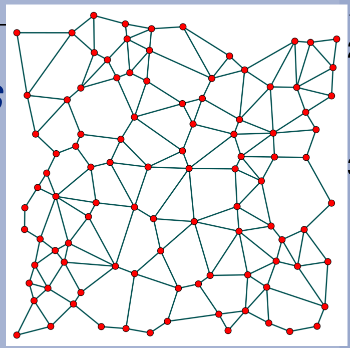
MLlib

- **collection of Machine Learning algorithms:**
  - basic statistics
  - classification & regression (model fitting)
  - unsupervised learning
  - clustering
  - pattern mining
  - and much more!

Example:

```
// start from a DataFrame with columns "label" and "features" (required names)
val mydata = sqlContext.read.format("libsvm").load("mydata.csv")
res1: org.apache.spark.sql.DataFrame = [label: double, features: vector]
import org.apache.spark.ml.classification._   // make LogisticRegression available
val logregr = new LogisticRegression() .
                                  setMaxIter(10) .          // sets param(s) for the algorithm
                                     setElasticNetParam(0.8)
val model = logregr.fit(mydata)     // fits the model to the data
println(s"Coefficients: ${model.coefficients} Intercept: ${model.intercept}")
val predictions = model.transform(someTestData)    //  apply the model to some data
```

GraphX



- **Spark library; contains functions for processing *graphs***
  - **examples:**
    - **web pages** & their hyperlinks (href)
    - social graphs
  - **Graph needs 2 RDDs for its representation: Vertices & Edges**
  - **both Vertices & Edges have "attributes" (data type e.g. *String*)**

```
val my_vertices : RDD[(VertexId, String)] = sc.textFile(...).map(...)
val my_edges: RDD[(VertexId, VertexId, String)] = sc.textFile(...).map(...)
val my_graph = Graph(my_vertices, my_edges)
// apply the famous Google PageRank iterative algorithm:
val ranks = my_graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("users.txt").map { line => val fields = line.split(",") ;
                                                   (fields(0).toLong, fields(1)); }
val ranks = users.join(ranks).map { case (id, (name, rank)) => (name, rank); }
// Print the result
println(ranks.collect().mkString("\n"))
```

Questions, remarks, feedback, ... ?

TRAINING & CONSULTING

Thank you!

Peter Vanroose

ABIS Training & Consulting

pvanroose@abis.be

Spark Analytics