Spark

Amos Azaria and Merav Chkroun

Spark

- Spark allows us to compute functions that are massively parallelizable, on large amounts of data.
- Spark usually relies on the map-reduce paradigm (and the Hadoop file-system).
- Spark has a Scala, Java and Python API (with Scala being the most popular).
- We will be using PySpark which is a Python interpreter for Spark
- Installation on Linux (use VirtualBox if needed):
  - Make sure you have Java installed (and JAVA_HOME set):
    - sudo apt-get install default-jre
  - Make sure you have python installed (is a part of Ubuntu)
  - Download Spark from: http://spark.apache.org/downloads.html
  - Unzip
  - (export PYSPARK_PYTHON=python3)
  - ./bin/pyspark
- (To install on windows see: http://stackoverflow.com/questions/25481325/how-to-set-up-spark-on-windows)

Resilient Distributed Dataset (RDD)

- Spark’s primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD).
- We can create an RDD from a file (which can be written using the Hadoop file system).
  - text_file = sc.textFile("myDir/story.txt")
- The text file is read line by line (i.e. is split using "\n").
- We can perform transforms on an RDD and get another RDD.
  - lines_with_hello = text_file.filter(lambda line: "hello" in line)
- We can use collect() to convert an RDD to a list.
- RDD resembles Streams in Java streams.

Word Count

- The file story.txt was
  https://s3.amazonaw
  datasets/nietzsche.tx

```
>>> text_file = sc.textFile("myDi
>>> word_counts = text_file.flat
                .map(lambda word
                .reduceByKey(lamb
                .collect()
>>> for word,count in word_cou
        print("the word: \"%s\" 
```

Result:

```
…
the word: "retrograde" appears 1 time(s)
the word: "grounds" appears 4 time(s)
the word: "VOUS" appears 2 time(s)
the word: ""Flatterers" appears 1 time(s)
the word: "injustice;" appears 1 time(s)
the word: "reciprocity," appears 1 time(s)
the word: "inflicted" appears 2 time(s)
the word: "limbs." appears 1 time(s)
the word: "christened" appears 2 time(s)
the word: "majority--where" appears 1 time(s)
the word: "three-fourths" appears 1 time(s)
the word: "dish," appears 1 time(s)
the word: "73." appears 1 time(s)
the word: "ENVIRONMENT," appears 1 time(s)
the word: ""honesty";" appears 1 time(s)
…
```

Sort by Key

- To sort by the key we y (before we call collec

```
>>> word_counts = text_fil
            .map(lambda word
            .reduceByKey(lam
            .sortByKey() \
            .collect()
>>> for word,count in word
           print("the word: \"                    )
```

Result:

```
the word: "" appears 2032 time(s)
the word: ""=Man" appears 1 time(s)
the word: ""A" appears 2 time(s)
the word: ""AWAY" appears 1 time(s)
the word: ""Ah," appears 1 time(s)
the word: ""All" appears 1 time(s)
the word: ""And" appears 2 time(s)
the word: ""Another" appears 1 time(s)
the word: ""Are" appears 2 time(s)
the word: ""BIG" appears 1 time(s)
the word: ""BY" appears 1 time(s)
the word: ""Bad!" appears 1 time(s)
the word: ""Be" appears 1 time(s)
the word: ""Better" appears 1 time(s)
the word: ""Beyond" appears 1 time(s)
```

Sort by Value

- To sort by value we d
  then sort by the key e
  sort in descending o

```
>>> word_counts = (text_f
                   .map(lambda
                   .reduceByKe
                   .map(lambda
                   #False is for
                   .sortByKey(Fa
                   .map(lambda
                   .collect())

>>> for word,count in wor
           print("the word: \                                    )
```

```
Result:

the word: "the" appears 5839 time(s)
the word: "of" appears 4560 time(s)
the word: "and" appears 3562 time(s)
the word: "to" appears 2716 time(s)
the word: "" appears 2032 time(s)
the word: "in" appears 1995 time(s)
the word: "a" appears 1896 time(s)
the word: "is" appears 1857 time(s)
the word: "that" appears 1242 time(s)
the word: "as" appears 1172 time(s)
the word: "it" appears 908 time(s)
the word: "for" appears 808 time(s)
the word: "which" appears 783 time(s)
the word: "be" appears 740 time(s)
the word: "with" appears 665 time(s)
```

The cache action

- When reading large files, we can use the cache() action to load the file into memory so the subsequent operations will be executed faster.

  ```
  >>>text_file.cache()
  ```

- Note that once cache() is called, there is no need to call it again (this is an action, not a transformation).

Bi-Grams

- In natural language processing, we many are interested in the appearances of pairs of words (bi-grams).
- For example, for the following sentence:

  "I did it you did it you did it"

  We get the following bi-grams count:

  [((I, did), 1), ((did, it), 3), ((it, you), 2), ((you, did), 2)]
- This is useful for part of speech tagging, feature extraction, story generation, etc.
- Tri-grams are the same with triples, and in general, we call this method n-grams.

Bi-grams (zip function)

- zip is a built-in function in Python, which receives two lists (or more) and returns a list of tuples from both lists. E.g:

```
>>>zip([1, 2, 3, 6], [10, 16, 23, 57]) =
      [(1,10), (2, 16), (3, 23), (6, 57)]
```

- *Equivalent* to the following:

```
def zip(lst1, lst2):
      out = []
      for i in range(min(len(lst1), len(lst2))):
            out.append((lst1[i], lst2[i]))
      return out
```

Bi-grams (cont.)

Result:

```
[(('of', 'the'), 910), (('in', 'the'), 498), (('to', 'the'),
327), (('it', 'is'), 240), (('to', 'be'), 186), (('of', 'a'),
171), (('and', 'the'), 157), (('for', 'the'), 149),
(('that', 'the'), 138), (('is', 'the'), 131)]
```

```
>>>def bigram(line):
    words = line.split()
    return zip(words, words[1:])

>>>pairs = text_file.flatMap(bigram)
>>>count = pairs.map(lambda x: (x, 1)).reduceByKey(lambda a, b: a + b)

>>>print(count.collect()[0:10])
>>>print(count.map(lambda xy: (xy[1],xy[0]))
                    .sortByKey(False)
                    .map(lambda xy: (xy[1],xy[0]))
                    .collect()[0:10])
```

HDFS

- Hadoop File-System is a highly distributed file system.
- We will not be using HDFS, but for any *real* use of Spark, it is recommended to install Hadoop in order to support HDFS.

Running a Python script using Spark

- In order to import Spark into Python we need first to define the environment variable SPARK_HOME:
  - export SPARK_HOME=.
- Then we need to install findspark
  - pip install findspark
- Then, in the script we write:
  - import findspark
    - findspark.init()
  - from pyspark import SparkContext
  - sc = SparkContext("local[*]", "Simple App")
  - …

* Means the maximum number of cores.
You can replace it with any other number.

Pi Estimation

```
import random
samples = 10**8
def inside(p):
        x, y = random.random(), random.random()
        return x*x + y*y < 1
hits = sc.parallelize(range(0, samples)) \
       .filter(inside).count()
print("Pi is approx. %f" % (4.0 * hits / samples))
```

```
Result:

Pi is approx. 3.141091
```

Java Streams Equivalent

```
static boolean inside(Object p)
{
    Random rand = new Random();
    double x = rand.nextDouble();
    double y = rand.nextDouble();
    return x * x + y * y < 1;
}


public static void main(String[] args)
{
    int samples = (int)1E+8;
    long start_time = System.nanoTime();
    long hits = IntStream.rangeClosed(0, samples).filter(Main::inside).count();
    long end_time = System.nanoTime();
    System.out.println("Pi is approx. " + (4.0 * hits / samples));
    System.out.println("Execution took: " + ((end_time-start_time)*1E-9) + " seconds");
}
```

Adding .parallel() did not help

Result:

```
Pi is approx. 3.14139924
Execution took: 8.663336708000001 seconds
```

When to Use Spark?

- On my laptop, using the Pi estimation example, Java Streams actually outperformed Spark, however when using a workstation (with 24 cores) Spark was twice as fast.
- Spark is intended for use with systems that:
  - Run on multiple machines / many cores
  - Have the data spread-out on multiple machines (HDFS = big data!)
  - Highly scalable

Data-frames

- Data-frames are the equivalent of Spark to tables in relational databases.
- We can create a data-frame from:
  - A relational database (and similar DBMS such as Cassandra)
  - An RDD
  - A CSV file
  - XML / JSON and other sources
- We can use SQL(!) to query Data-frames.

Data-frames example

```
from pyspark.sql import Row
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
student_list = [(111, 'Chaya', 'Glass', 21), (222, 'Tal','Negev',28),
        (333, 'Gadi', 'Golan', 24), (444, 'Moti', 'Cohen', 23)]
student_rdd = sc.parallelize(student_list)
students_rows = student_rdd.map(lambda x:
        Row(id=int(x[0]),age=int(x[3]), firstName=x[1], lastName=x[2]))
df_students = sqlContext.createDataFrame(students_rows)
df_students.show()
```

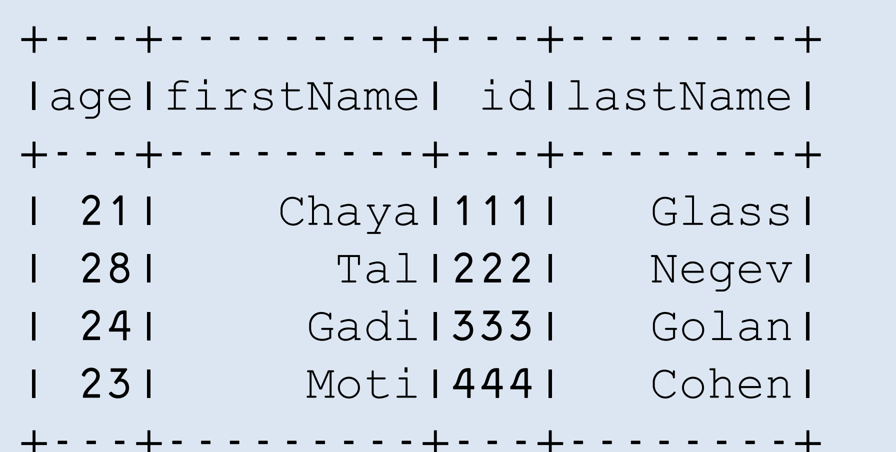
```
+---+---------+---+--------+
|age|firstName| id|lastName|
+---+---------+---+--------+
| 21|    Chaya|111|   Glass|
| 28|      Tal|222|   Negev|
| 24|     Gadi|333|   Golan|
| 23|     Moti|444|   Cohen|
+---+---------+---+--------+
```

Create Data-frame from json

```
students_json = '[{"id":"111","firstName":"Chaya","lastName":"Glass","age":23},{"id":"222","firstName":"Tal","lastName":"Negev","age":28},{"id":"333","firstName":"Gadi","lastName":"Golan","age":24},{"id":"444","firstName":"Moti","lastName":"Cohen","age":23}]'

df = sqlContext.read.json(sc.parallelize([students_json]))

df.show()
```



```
+---+---------+---+--------+
|age|firstName| id|lastName|
+---+---------+---+--------+
| 21|    Chaya|111|   Glass|
| 28|      Tal|222|   Negev|
| 24|     Gadi|333|   Golan|
| 23|     Moti|444|   Cohen|
+---+---------+---+--------+
```

Create DF from json file

df = sqlContext.read.json(“myDir\students.json”, multiLine=True)

df.show()

```
+---+---------+---+--------+
|age|firstName| id|lastName|
+---+---------+---+--------+
| 21|    Chaya|111|   Glass|
| 28|      Tal|222|   Negev|
| 24|     Gadi|333|   Golan|
| 23|     Moti|444|   Cohen|
+---+---------+---+--------+
```

.withColumn()

- .withColumn() allows us to add an additional column (which is usually dependant on other columns).

df_students2 = df_students.withColumn('age_plus_id', df_students.age + df_students.id)

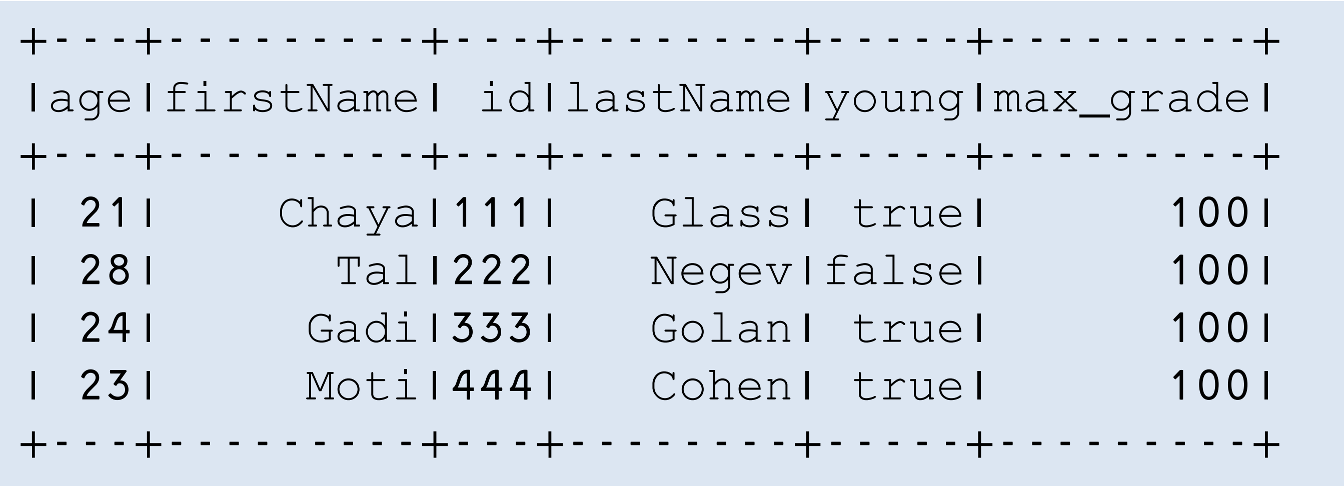
df_students2.show()

```
+---+---------+---+--------+-----------+
|age|firstName| id|lastName|age_plus_id|
+---+---------+---+--------+-----------+
| 21|    Chaya|111|   Glass|        132|
| 28|      Tal|222|   Negev|        250|
| 24|     Gadi|333|   Golan|        357|
| 23|     Moti|444|   Cohen|        467|
+---+---------+---+--------+-----------+
```

.withColumn() (cont.)

- The second argument to withColumn must be a column, so if we want to enter our own data, it must be converted to a column.

```
from pyspark.sql import functions
from pyspark.sql.functions import lit  #lit for literal
df_students2 = df_students.withColumn('young',
        functions.when(df_students.age < 25,
        True).otherwise(False)).withColumn('max_grade', lit(100))
df_students2.show()
```



```
+---+---------+---+--------+-----+---------+
|age|firstName| id|lastName|young|max_grade|
+---+---------+---+--------+-----+---------+
| 21|    Chaya|111|   Glass| true|      100|
| 28|      Tal|222|   Negev|false|      100|
| 24|     Gadi|333|   Golan| true|      100|
| 23|     Moti|444|   Cohen| true|      100|
+---+---------+---+--------+-----+---------+
```

SQL Query in Spark

```
df_students.registerTempTable("student_table")
ending_with_n = sqlContext.sql("SELECT firstName, lastName, id FROM student_table WHERE lastName LIKE '%n'")
ending_with_n.show()
```

```
+---------+--------+---+
|firstName|lastName| id|
+---------+--------+---+
|     Gadi|   Golan|333|
|     Moti|   Cohen|444|
+---------+--------+---+
```

Map-Reduce Example

- Suppose we have documents with the following structure:

```
{
    _id: ObjectId("50a8240b927d5d8b5891743c"),
    cust_id: "abc123",
    ord_date: new Date("Oct 04, 2012"),
    status: 'A',
    amount: 25,
    items: [ { sku: "mmm", qty: 5, price: 2.5 },
             { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

SKU = Stock Keeping Unit is an item identifier.

Can we do this in a relational DB, With SQL?

We will need to assume a different structure a relational data-base.

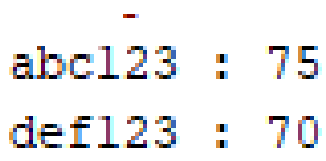
- We would like to get the total **amount** paid by each cust_id with status 'A'.

Solving in spark with RDD

- We can solve it using map-reduce:

```
df = sqlContext.read.json(“MyDir\orders.json",
multiLine=True)
rdd = df.rdd
calc_amount = rdd.filter(lambda a: a.status == 'A') \
        .map(lambda row: (row.cust_id, row.amount)) \
        .reduceByKey(lambda a,b: a+b) \
        .collect()
for cust_id, sum_amount in calc_amount:
    print('%s : %s' %(cust_id, sum_amount))
```

279,840 orders RDD:
time: 5.2894 seconds

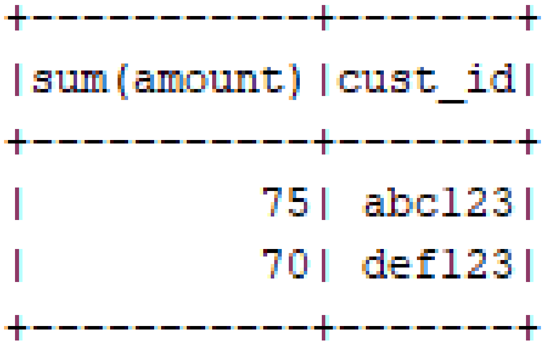


```
abc123 : 75
def123 : 70
```

Solving in Spark with Dataframe-1

- Assume we would have all the data in a json file. can we obtain the query using spark?

```
df = sqlContext.read.json(“myDir\orders.json",
multiLine=True)
df.registerTempTable("orders_table")
amount = sqlContext.sql("SELECT sum(amount),
cust_id FROM orders_table WHERE status == 'A'
group by cust_id")
amount.show()
```

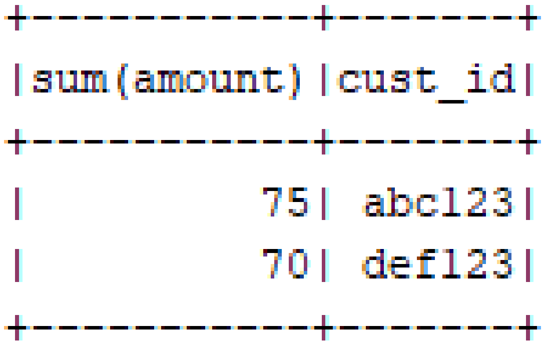


```
+-----------+-------+
|sum(amount)|cust_id|
+-----------+-------+
|         75| abc123|
|         70| def123|
+-----------+-------+
```

279,840 orders DF (select):
time: 0.2224 seconds

Solving in Spark with Dataframe-2

```
from pyspark.sql import functions
df.filter(df.status == 'A')\
    .groupBy(df.cust_id)\
    .agg(functions
        .sum(functions.col("amount")))\
    .show()
```



```
+-----------+-------+
|sum(amount)|cust_id|
+-----------+-------+
|         75| abc123|
|         70| def123|
+-----------+-------+
```

279,840 orders DF:
time: 1.30952 seconds

RDD v. Data Frames

RDD	Data Frames
“How” to do	“What” to do
Unstructured Data	Structured and semi-structured data
Less optimal and efficient	optimization and performance benefits available with DataFrames