Introduction to Spark

Louis Jachiet
Spark Motivation
IBM and Apache Spark
Apache Spark is a fast and general engine for large-scale data processing.

- **Speed**: Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- **Ease of Use**: Write applications quickly in Java, Scala, Python, R.
- **Generality**: Combine SQL, streaming, and complex analytics.
- **Runs Everywhere**: Spark runs on Hadoop, Mesos, standalone, or in the cloud.

https://spark.apache.org/
Resilient Distributed Datasets (RDDs)

• An RDD is a fault-tolerant collection of “objects” that can be operated on in parallel.

• RDDs are created:
  • parallelizing an existing collection in your driver program, or
  • referencing a dataset in an external storage system or
  • building on other RDDs

• RDDs can be seen as the “old” interface but also the backbone of Spark
Data Frames (DFs)

- A Data Frame is a fault-tolerant “table”
- DFs support:
  - schemas and automatic schema inference
  - complex queries easily written
  - lots of advanced algorithms
  - automatic optimization
- DFs is the “new” interface of Spark
Spark Ecosystem

Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)

Apache Spark
Spark Resilient Distributed Datasets
Key ideas for Spark

1. Keep a trace of how data was constructed
Resilient Distributed Datasets (RDD)

Key ideas for Spark

1. Keep a trace of how data was constructed
2. Computation failure is rare
Key ideas for Spark

1. Keep a trace of how data was constructed
2. Computation failure is rare
3. Provide high(er)-level constructs and interactivity
Simple operations on RDD

\[ D_1 \quad D_1.filter(f) \]
Simple operations on RDD

$D_1 \xrightarrow{f(0,A)} D_1 . map(f)$

$\xrightarrow{f(1,B)}$

$\xrightarrow{f(0,C)}$

$\xrightarrow{f(2,D)}$
Simple operations on RDD

\[ D_1 \] \rightarrow \quad D_1.mapValues(f) 

\begin{align*} 
2 & \quad D & \quad 2 & \quad f(D) \\
0 & \quad C & \quad 0 & \quad f(C) \\
1 & \quad B & \quad 1 & \quad f(B) \\
0 & \quad A & \quad 0 & \quad f(A) 
\end{align*}
Simple operations on RDD

\[ D_1 \xrightarrow{\text{flatMap}(f)} D_1 \]

- \( f(0,A) \)
- \( f(1,B) \)
- \( f(1,B) \)
- \( f(2,D) \)
- \( f(2,D) \)
### Simple operations on RDD

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>{\eta}</td>
</tr>
<tr>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>{\beta}</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>{\alpha, \gamma}</td>
</tr>
</tbody>
</table>
```

\[ D_1 \quad D_1.groupByKey() \]
Simple operations on RDD

\[
\begin{array}{c|c}
2 & D \\
0 & C \\
1 & B \\
0 & A \\
\end{array} \quad \begin{array}{c|c}
3 & \eta \\
0 & \gamma \\
1 & \beta \\
0 & \alpha \\
\end{array}
\]

\[D_1 \quad D_2\]
Simple operations on RDD

\[
\begin{align*}
D_1 & = & \begin{array}{c|c}
2 & D \\
0 & C \\
1 & B \\
0 & A \\
\end{array} \\
D_2 & = & \begin{array}{c|c}
3 & \eta \\
0 & \gamma \\
1 & \beta \\
0 & \alpha \\
\end{array}
\end{align*}
\]

\[D_1.\text{union}(D_2)\]
Simple operations on RDD

\[
\begin{array}{c|c}
2 & D \\
0 & C \\
1 & B \\
0 & A \\
\end{array}
\quad
\begin{array}{c|c}
3 & \eta \\
0 & \gamma \\
1 & \beta \\
0 & \alpha \\
\end{array}
\quad
\begin{array}{c|c}
1 & B \\
0 & C \\
1 & A \\
0 & C \\
\end{array}
\]

\[D_1 \quad D_2\]

\[D_1.join(D_2)\]
### Simple operations on RDD

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>D</td>
<td>3</td>
<td>η</td>
</tr>
<tr>
<td>0</td>
<td>C</td>
<td>0</td>
<td>γ</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>1</td>
<td>β</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>0</td>
<td>α</td>
</tr>
</tbody>
</table>

- $D_1$  
- $D_2$  

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{D}, \emptyset</td>
<td>3</td>
<td>\emptyset, {η}</td>
</tr>
<tr>
<td>1</td>
<td>{B}, {β}</td>
<td>0</td>
<td>{A, C}, {α, γ}</td>
</tr>
</tbody>
</table>

$D_1$.coGroup2($D_2$)
Practical Spark
### Spark RDD API (extract)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>limits the number of records</td>
</tr>
<tr>
<td>map</td>
<td>transform records</td>
</tr>
<tr>
<td>flatMap</td>
<td>maps each record to 0, 1 or more elements</td>
</tr>
<tr>
<td>distinct</td>
<td>eliminates duplicates</td>
</tr>
</tbody>
</table>

**Manipulating RDDs**
### Spark RDD API (extract)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupByKey</td>
<td>corresponds to a shuffle</td>
</tr>
<tr>
<td>reduceByKey(f)</td>
<td>reduce for each key</td>
</tr>
<tr>
<td>foldByKey(f)</td>
<td>fold for each key</td>
</tr>
<tr>
<td>keyBy(f)</td>
<td>create pair-RDD from RDD</td>
</tr>
<tr>
<td>mapValues</td>
<td>transforms only the value</td>
</tr>
</tbody>
</table>

**Manipulating pair-RDDs**
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cartesian</td>
<td>build the cartesian product</td>
</tr>
<tr>
<td>join</td>
<td>behaves like SQL join (supposes two pair-RDDs)</td>
</tr>
<tr>
<td>union</td>
<td>takes the union of two RDDs</td>
</tr>
<tr>
<td>intersection</td>
<td>takes the intersection of two RDDs</td>
</tr>
<tr>
<td>subtract</td>
<td>takes the difference of two RDDs</td>
</tr>
<tr>
<td>coGroup</td>
<td>generalized groupByKey (supposes two pair-RDDs)</td>
</tr>
</tbody>
</table>

**Combining several RDDs**
Spark RDD API (extract)

textFile(path) creates an RDD with an item per line
saveAsTextFile(path) saves an RDD with an item per line
sortWith(f) sort according to comparison f

Tools for RDDs

take(n) retrieves the n first elements
collect() retrieves whole RDD
count() counts the number of items
reduce combines all elements
fold combines all elements with an initial value
aggregate fold+reduce
foreach apply a function on each element

Actions on RDDs
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)

Spark’s Python API
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)

Spark’s Python API
Exercise revisited (easy)

**Input**

You are given a list of pairs \((k_i, v_i)\) where \(k_i\) is a string and \(v_i\) an integer.

**Problem**

Compute the average value for each key.

**Example**

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 42</td>
<td>A (\frac{42 + 12}{2} = 27)</td>
</tr>
<tr>
<td>B 17</td>
<td>B (\frac{17 + 99}{2} = 58)</td>
</tr>
<tr>
<td>A 12</td>
<td></td>
</tr>
<tr>
<td>B 99</td>
<td></td>
</tr>
</tbody>
</table>
Input
You are given two lists of items.

Problem
Compute the list of item appearing in the first one but not in the second.

Example

<table>
<thead>
<tr>
<th>INPUT 1</th>
<th>INPUT2</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

OUTPUT
B
Exercise revisited (hard)

Input
You are given the Twitter following list: each record is a pair \((A_i, B_i)\) indicating that account \(A_i\) follows \(B_i\).

Problem
Compute the accounts that have more followers than followees.

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>
Exercise revisited (hardest)

Input

You are given the Twitter following list: each record is a pair \((A_i, L_i)\) indicating that account \(A_i\) follows the accounts in the list \(L_i\).

Problem

Compute for each account \(A\) the list of accounts that are followed by an account followed by \(A\).

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B,D</td>
</tr>
<tr>
<td>B</td>
<td>C,D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td>C,D</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
</tbody>
</table>
What does this program do?

A = sc.textFile("hdfs:///user/ljachiet/a.txt")
   .flatMap(lambda x: [(v,1) for v in x.split()])
   .reduceByKey(lambda x,y: x+y)
   .filter(lambda x: len(x[0])>2)

B = sc.textFile("hdfs:///user/ljachiet/b.txt")
   .flatMap(lambda x: [(v,1) for v in x.split()])
   .reduceByKey(lambda x,y: x+y)
   .filter(lambda x: len(x[0])>2)

A.join(B).collect()
Spark internals
Hadoop internals for Map-Reduce

Mapper task:
- Record Reader
- Map
- Key-value sort

Reducer task:
- Fetch
- Sort
- Reduce

HDFS
Let us consider:

\[
A = \text{textFile("hdfs://A.txt")}.\map(\lambda x: [(v, 1) \text{ for } v \text{ in } x.\text{split()}]).\filter(\lambda x: \text{len}(x[1]) > 2)
\]

\[
B = \text{textFile("hdfs://B.txt")}.\flatMap(\lambda x: x.\text{split()}).\keyBy(\lambda x: x.\text{split(":")}[0])
\]

A.join(B)
Spark internals

HDFS

Executor 1 → Executor 2 → Executor 3 → Executor 4 → Executor 5

Filter

Executor 1 → Executor 2 → Executor 3

Map

Executor 1 → Executor 2 → Executor 3

flatMap

Executor 4 → Executor 5

KeyBy

Executor 4 → Executor 5

join

Executor 6 → Executor 7 → Executor 8 → Executor 9
Spark internals

HDFS

Filter
Executor 1  Executor 2  Executor 3

Execute 6

Join
Executor 7  Executor 8  Executor 9

Map
Executor 1  Executor 2  Executor 3

KeyBy
Executor 4  Executor 5

flatMap
Executor 4  Executor 5

HDFS
Spark internals

Filter
- Executor 1
- Executor 2
- Executor 3

Map
- Executor 1
- Executor 2
- Executor 3

flatMap
- Executor 4
- Executor 5

KeyBy
- Executor 4
- Executor 5

join
- Executor 6
- Executor 7
- Executor 8
- Executor 9

HDFS
Spark internals

HDFS

join

Executor 6
Executor 7
Executor 8
Executor 9

Filter
Executor 1
Executor 2
Executor 3

Map
Executor 1
Executor 2
Executor 3

KeyBy
Executor 4
Executor 5

flatMap
Executor 4
Executor 5

HDFS
Spark internals

HDFS

Map
Executor 1  Executor 2  Executor 3

Filter
Executor 1  Executor 2  Executor 3

join
Executor 6  Executor 7  Executor 8  Executor 9

KeyBy
Executor 4  Executor 5

flatMap
Executor 4  Executor 5

HDFS
Iterations
Spark is especially competitive for jobs requiring long chain of individual jobs.
Spark is especially competitive for jobs requiring long chain of individual jobs.

Such jobs are often required by data mining algorithm (e.g. the gradient descent for logistic regression).
# Compute logistic regression gradient for a matrix of data points

def gradient(matrix, w):
    Y = matrix[:, 0]  # point labels (first column of input file)
    X = matrix[:, 1:]  # point coordinates
    # For each point (x, y), compute gradient function, then sum these up
    return ((1.0 / (1.0 + np.exp(-Y * X.dot(w))) - 1.0) * Y * X.T).sum(1)

def add(x, y):
    x += y
    return x

for i in range(iterations):
    print("On iteration %i" % (i + 1))
    w -= points.map(lambda m: gradient(m, w)).reduce(add)

From Spark: examples/src/main/python/logistic_regression.py
Caching and persistence
RDD materialization are only triggered by an *action*. 
RDD materialization are only triggered by an *action*.

Spark caches by default the materialization, but the user can specify the caching.
## Caching

<table>
<thead>
<tr>
<th>Type</th>
<th>Space</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY*</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>high</td>
<td>med</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>
# Compute logistic regression gradient for a matrix of data points

def gradient(matrix, w):
    Y = matrix[:, 0]  # point labels (first column of input file)
    X = matrix[:, 1:]  # point coordinates
    # For each point \((x, y)\), compute gradient function, then sum these up
    return ((1.0 / (1.0 + np.exp(-Y * X.dot(w))) - 1.0) * Y * X.T).sum(1)

def add(x, y):
    x += y
    return x

for i in range(iterations):
    print("On iteration %i" % (i + 1))
    w -= points.map(lambda m: gradient(m, w)).reduce(add)

From Spark: examples/src/main/python/logistic_regression.py
Spark API: Shared Variables

```python
>>> broadcastVar = sc.broadcast([1, 2, 3])

>>> broadcastVar.value
[1, 2, 3]
```

Spark’s Python API
Spark API: Shared Variables

```scala
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

Spark’s Scala API
Broadcast<int[]> broadcastVar =
   sc.broadcast(new int[] {1, 2, 3});

broadcastVar.value();
// returns [1, 2, 3]

Spark’s Java API
Spark ecosystem
Spark Streaming is an extension of Spark that allows processing data stream using micro-batches of data.
Discretized Streams (DStreams)

- Discretized Stream or DStream represents a continuous stream of data,
  - either the input data stream received from source, or
  - the processed data stream generated by transforming the input stream.

- Internally, a DStream is represented by a continuous series of RDDs
Discretized Streams (DStreams)

- Any operation applied on a DStream translates to operations on the underlying RDDs.

```
lines DStream
lines from time 0 to 1
flatMap operation
lines from time 1 to 2
lines from time 2 to 3
lines from time 3 to 4
words DStream
words from time 0 to 1
words from time 1 to 2
words from time 2 to 3
words from time 3 to 4
```
Spark Streaming provides windowed computations, which allow transformations over a sliding window of data.
val conf = new SparkConf().setMaster("local[2]").setAppName("WCount")
val ssc = new StreamingContext(conf, Seconds(1))

// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split(" "))

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()

ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
Spark SQL and DataFrames

- Spark SQL is a Spark module for structured data processing.
- It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine.
- A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database.
Spark Machine Learning Libraries

- **MLLib** contains the original API built on top of RDDs.
- **spark.ml** provides higher-level API built on top of DataFrames for constructing ML pipelines.

![Spark ML Pipeline Diagram]

- `Pipeline (Estimator)`
- `Tokenizer` → `HashingTF` → `Logistic Regression`
- `Pipeline.fit()`
  - Raw text → Words → Feature vectors → Logistic Regression Model
• **MLLib** contains the original API built on top of RDDs.
• **spark.ml** provides higher-level API built on top of DataFrames for constructing ML pipelines.
• GraphX optimizes the representation of vertex and edge types when they are primitive data types

• The property graph is a directed multigraph with user defined objects attached to each vertex and edge.

Property Graph

Vertex Table

<table>
<thead>
<tr>
<th>Id</th>
<th>Property (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(rxin, student)</td>
</tr>
<tr>
<td>7</td>
<td>(jgonzal, postdoc)</td>
</tr>
<tr>
<td>5</td>
<td>(franklin, professor)</td>
</tr>
<tr>
<td>2</td>
<td>(istoica, professor)</td>
</tr>
</tbody>
</table>

Edge Table

<table>
<thead>
<tr>
<th>SrcId</th>
<th>DstId</th>
<th>Property (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>Collaborator</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Advisor</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Colleague</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>PI</td>
</tr>
</tbody>
</table>
Spark GraphX

// Assume the SparkContext has already been constructed
val sc: SparkContext

// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
  (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),
  Edge(5L, 3L, "advisor"),
  Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
Spark Dataframes
SQL model

SQL tables

Tuple

Attribute

Relation
Dataframes are basically RDD with an explicit schema but:

1. untyped data
2. can be optimized
val movies = ... // of type RDD[(int,string,string)]
...
val dfMovies = movies.toDF("movieId","title","genre")
class Movie(movieId: Int, title: String, genre: String)
...
val movies = ... // of type RDD[Movie]
...
val dfMovies = movies.toDF
val myRDD = something //
...
val mySchema = StructType(List(
    StructField("number", IntegerType, true),
    StructField("word", StringType, true)))
...
val myDF = spark.createDataFrame(myRDD,mySchema)
Creating Dataframes From Files

Spark can read from:

1. CSV
2. JSON
3. Parquet
4. etc.

val myDF = spark.read.format("csv").
  option("header","true").
  load("/datasets/movie_small/ratings.csv")
Spark can read from:

1. CSV
2. JSON
3. Parquet
4. etc.

```scala
import org.apache.spark.sql.types.{StructType, StructField, FloatType, IntegerType, LongType};
val mySchema = StructType(List(StructField("userId", IntegerType, true),
                                StructField("movieId", IntegerType, true),
                                StructField("rating", FloatType, true),
                                StructField("timestamp", LongType, true)))
val myDF = spark.read.format("csv").option("header","true").
schema(mySchema).
load("/datasets/movie_small/ratings.csv")
```
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select</code></td>
<td>projection on some columns</td>
</tr>
<tr>
<td><code>agg</code></td>
<td>aggregation</td>
</tr>
<tr>
<td><code>groupBy</code></td>
<td>use in conjunction with <code>agg</code></td>
</tr>
<tr>
<td><code>join</code></td>
<td>inner join</td>
</tr>
<tr>
<td><code>filter</code></td>
<td>filter some columns</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>equivalent to <code>take(n)</code></td>
</tr>
<tr>
<td><code>orderBy</code></td>
<td>sort by a given column</td>
</tr>
<tr>
<td><code>where</code></td>
<td>condition on join</td>
</tr>
<tr>
<td><code>union</code></td>
<td>union</td>
</tr>
<tr>
<td><code>show</code></td>
<td>print 20 first entries</td>
</tr>
<tr>
<td><code>printSchema</code></td>
<td>print schema</td>
</tr>
<tr>
<td><code>as</code></td>
<td>name table</td>
</tr>
<tr>
<td><code>drop</code></td>
<td>remove records with NULL</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>replace NULL with value</td>
</tr>
</tbody>
</table>
```scala
val schema2 = StructType(List(
  StructField("movieId", IntegerType),
  StructField("title", StringType)))
val movieDF = spark.read.format("csv").
  option("header","true").
  schema(schema2).
  load("/datasets/movie_small/movies.csv")
myDF2.join(movieDF,
  myDF2("movieId")===movieDF("movieId")).
show()
```
Exercise revisited (easy)

Input

You are given a list of pairs \((k_i, v_i)\) where \(k_i\) is a string and \(v_i\) an integer.

Problem

Compute the average value for each key.

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
</table>
| A     | \(
\frac{42 + 12}{2}
\) = 27 |
| B     | \(
\frac{17 + 99}{2}
\) = 58 |
Exercise revisited (medium)

**Input**

You are given two lists of items.

**Problem**

Compute the list of item appearing in the first one but not in the second.

**Example**

<table>
<thead>
<tr>
<th>INPUT 1</th>
<th>INPUT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
</tbody>
</table>

OUTPUT

| B       |
Exercise revisited (hard)

Input
You are given the Twitter following list: each record is a pair \((A_i, B_i)\) indicating that account \(A_i\) follows \(B_i\).

Problem
Compute the accounts that have more followers than followees.

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
</tbody>
</table>
Exercise revisited (hardest)

**Input**

You are given the Twitter following list: each record is a pair \((A_i, L_i)\) indicating that account \(A_i\) follows the accounts in the list \(L_i\).

**Problem**

Compute for each account \(A\) the list of accounts that are followed by an account followed by \(A\).

**Example**

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B,D</td>
</tr>
<tr>
<td>B</td>
<td>C,D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td>C,D</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
</tbody>
</table>
Datasets
Datasets

Dataframes can be generalized into *Datasets*, giving the best of both worlds.
Spark SQL
myDF.createOrReplaceTempView("tablename")

sqlDF = spark.sql(`SELECT * FROM tablename'')
sqlDF.show()
Exercise revisited (easy)

Input
You are given a list of pairs \((k_i, v_i)\) where \(k_i\) is a string and \(v_i\) an integer.

Problem
Compute the average value for each key.

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(\frac{42 + 12}{2} = 27)</td>
</tr>
<tr>
<td>B</td>
<td>(\frac{17 + 99}{2} = 58)</td>
</tr>
<tr>
<td>A</td>
<td>42</td>
</tr>
<tr>
<td>B</td>
<td>17</td>
</tr>
<tr>
<td>A</td>
<td>12</td>
</tr>
<tr>
<td>B</td>
<td>99</td>
</tr>
</tbody>
</table>
Exercise revisited (medium)

Input
You are given two lists of items.

Problem
Compute the list of item appearing in the first one but not in the second.

Example

<table>
<thead>
<tr>
<th>INPUT 1</th>
<th>INPUT2</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>B</td>
</tr>
</tbody>
</table>
Exercise revisited (hard)

Input
You are given the Twitter following list: each record is a pair \((A_i, B_i)\) indicating that account \(A_i\) follows \(B_i\).

Problem
Compute the accounts that have more followers than followees.

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th></th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

OUTPUT:
- E
- D
- C
Exercise revisited (hardest)

Input
You are given the Twitter following list: each record is a pair \((A_i, L_i)\) indicating that account \(A_i\) follows the accounts in the list \(L_i\).

Problem
Compute for each account \(A\) the list of accounts that are followed by an account followed by \(A\).

Example

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B,D</td>
</tr>
<tr>
<td>B</td>
<td>C,D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td>C,D</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
</tbody>
</table>
Playing With The Movie Lens Dataset