INTRODUCTION TO APACHE SPARK

2. APACHE SPARK AND RDDS

REMINDER: HOMEWORKS

- Send me the first one by Slack or email by Friday.
- Next one expected before the following class (24/02)
- Send the exported notebook (.ipynb extension)
- Please include both student name and no space (TPName_FirstName_LastName.ipynb) in the file name.

LEARNING RESOURCES

- Spark Documentation Website
- API docs: Scala, Python
- Databricks learning notebooks
- Book: Learning Spark 2nd Ed.
- Online Book: Apache Spark Internals
- StackOverflow: apache-spark and pyspark
- Practice!

PHILOSOPHY

Spark computing framework hides complexity of fault tolerance and slow servers ("*stragglers*").

"Here's an operation, run it on all the data."

- I do not care where it runs
- Feel free to run it twice on different nodes

API

Spark is implemented in Scala, runs on the JVM (Java Virtual Machine)

Multiple Application Programming Interfaces (APIs):

- Scala (JVM)
- Java (JVM)
- Python
- R

This course is based on Python API, as it is easier to learn than Scala and Java APIs for non-programmers.

R API: *younger*, outlying from the three others due to R syntax.

ARCHITECTURE

When you interact with Spark through the API, you are sending instructions to the *driver*.

Driver: central coordinator which communicates with the distributed workers called *executors*.

ARCHITECTURE

The driver

- Creates a logical directed acyclic graph (DAG) of operations
- Merges operations that can be merged
- Splits the operations in *tasks* (smallest unit of work in Spark)
- Schedules the tasks and send them to the executors
- Tracks data and tasks

RDD ENTRY POINT: THE SPARKCONTEXT OBJECT

You interact with the driver through the `SparkContext` object, created as follows:

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf = conf)
```

When using notebooks or any Spark interactive shell, `SparkContext` is automatically created within the `SparkSession` (more details in next class).

RDDS AND RUNNING MODEL

Spark programs are written in terms of operations on **Resilient Distributed Datasets** (RDDs)

RDD: immutable distributed collections of objects spread across the cluster disks or memory

RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Parallel transformations or actions can be applied to RDDs.

RDDs are automatically rebuilt on machine failure.

CREATING A RDD

From an iterable object:

```
lines = sc.parallelize(iterable_object)
```

From a text file:

```
lines = sc.textFile("/path/to/file.txt")
```

where `lines` is the resulting RDD, and `sc` the spark context.

OPERATIONS ON RDD

Two families of operations can be performed on RDDs:

- Transformations: operations on RDDs which return a new RDD. Lazy evaluation.
- Actions: operations on RDDs that return some other data type. Trigger computation.

Lazy evaluation: When a transformation is called on an RDD, the operation is not immediately performed. Spark internally records that this operation has been requested. The computation is triggered only if an action requires the result of this transformation at some point.

TRANSFORMATIONS

transformation	description
`map(f)`	apply `f` to each element of the RDD

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.map(lambda x: range(1, x)).collect())
[ [1], [1, 2], [1, 2, 3] ]
```

TRANSFORMATIONS

transformation	description
`flatMap(f)`	apply `f` to each element of the RDD, then flattens the results

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
```

TRANSFORMATIONS

transformation	description
`filter(f)`	Return an RDD consisting of only elements that pass the condition `f` passed to `filter()`

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.filter(lambda x: x % 2 == 0).collect())
[2, 4]
```

TRANSFORMATIONS

transformation	description
`distinct()`	Removes duplicates
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement

TRANSFORMATIONS - PSEUDO SET OPERATIONS

transformation	description
`union(otherRdd)`	Returns union with `otherRdd`
`intersection(otherRdd)`	Returns intersection with `otherRdd`
`subtract(otherRdd)`	Return each value in `self` that is not contained in `otherRdd`.

TRANSFORMATIONS - SET OPERATIONS

transformation	description
`cartesian(otherRdd)`	Return the Cartesian product of this RDD and another one

```
>>> rdd = sc.parallelize([1, 2])
>>> rdd2 = sc.parallelize(["a", "b"])
>>> sorted(rdd.cartesian(rdd2).collect())
[(1, "a"), (1, "b"), (2, "a"), (2, "b")]
```

ACTIONS

action	description
`collect()`	Return all elements from the RDD

```
>>> rdd = sc.parallelize([1, 2, 3, 3])
>>> rdd.collect()
[1, 2, 3, 3]
```

ACTIONS

action	description
`count()`	Return the number of elements in the RDD
`countByValue()`	Return the count of each unique value in the RDD as a dictionary of `{value: count}` pairs.

ACTIONS

action	description
`take(n)`	Return `n` elements from the RDD (deterministic)
`top(n)`	Return first `n` elements from the RDD (decending order)
`takeOrdered(num, key=None)`	Get the N elements from a RDD ordered in ascending order or as specified by the optional key function.

ACTIONS

action	description
`reduce(op)`	Combines elements of an RDD binary operator 'op'.
`fold(zeroValue, op)`	Combines elements of an RDD using

Bonus question: what is the algebraic difference between reduce and fold? The function `op(x, y)` is allowed to modify x and return it as its result value to avoid object allocation; however, it should not modify y.

ACTIONS

action	description
`aggregate(zero, seqOp, combOp)`	Similar to reduce() but used to return a different type.

```
>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
>>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)
(10, 4)
>>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)
(0, 0)
```

The functions `op(x, y)` is allowed to modify x and return it as its result value to avoid object allocation; however, it should not modify y.

`seqOp` can return a different result type than the type of this RDD.

ACTIONS

action	description
`foreach(f)`	Apply a function `f` to each element of a RDD

Performs an action on all of the elements in the RDD without returning any result to the driver.

Ex. : insert records into a database with using `f`

The `foreach()` action lets us perform computations on each element in the RDD without bringing it back locally.

LAZY EVALUATION AND PERSISTENCE

Spark RDDs are lazily evaluated

Each time an action is called on a RDD, this RDD and all its dependencies are recomputed.

If you plan to reuse a RDD multiple times, you sould use *persistence*.

method	description
`cache()`	Persist the RDD in memory
`persist(storageLevel)`	Persist the RDD according to `storageLevel`

These methods must be called before the action, and do not trigger the computation.

```
pyspark.StorageLevel(useDisk, useMemory, useOffHeap,
    deserialized, replication=1)
```

argument	description
`useDisk`	Allow caching to use disk if `True`
`useMemory`	Allow caching to use memory if `True`
`useOffHeap`	Store data outside JVM heap. Useful if using Tachyon
`deserialized`	Cache data without serialization
`replication`	Number of replications of the cached data

You can pass these constants to `StorageLevel` if you do not want to remember the parameters

```
DISK_ONLY = StorageLevel(True, False, False, False, 1)
DISK_ONLY_2 = StorageLevel(True, False, False, False, 2)
MEMORY_AND_DISK = StorageLevel(True, True, False, True, 1)
MEMORY_AND_DISK_2 = StorageLevel(True, True, False, True, 2)
MEMORY_AND_DISK_SER = StorageLevel(True, True, False, False, 1
MEMORY_AND_DISK_SER_2 = StorageLevel(True, True, False, False,
MEMORY_ONLY = StorageLevel(False, True, False, True, 1)
MEMORY_ONLY_2 = StorageLevel(False, True, False, True, 2)
MEMORY_ONLY_SER = StorageLevel(False, True, False, False, 1)
MEMORY_ONLY_SER_2 = StorageLevel(False, True, False, False, 2)
OFF_HEAP = StorageLevel(False, False, True, False, 1)
```

What if you attempt to cache too much data to fit in memory ?

Spark will automatically evict old partitions using a Least Recently Used (LRU) cache policy:

- For the memory-only storage levels, it will recompute these partitions the next time they are accessed
- For the memory-and-disk ones, it will write them out to disk.

You can also `unpersist()` RDDs to manually remove them from the cache.

PASSING FUNCTIONS

Warning: when passing functions, you can inadvertently serialize the object containing the function. If you pass a function that:

- is the member of an object
- contains references to fields in an object

then Spark sends the entire object to worker nodes, which can be much larger than the bit of information you need.

Sometimes this can also cause your program to fail, if your class contains objects that Python can’t figure out how to pickle.

Example: Passing a function with field references (don’t do this!)

```
class SearchFunctions(object):
    def __init__(self, query):
        self.query = query

    def isMatch(self, s):
        return self.query in s

    def getMatchesFunctionReference(self, rdd):
        # Problem: references all of "self" in "self.isMatch"
        return rdd.filter(self.isMatch)

    def getMatchesMemberReference(self, rdd):
        # Problem: references all of "self" in "self.query"
        return rdd.filter(lambda x: self.query in x)
```

Instead, just extract the fields you need from your object into a local variable and pass that in.

Example: Python function passing without field references

```
class WordFunctions(object):

    ...

    def getMatchesNoReference(self, rdd):
        # Safe: extract only the field we need into a local va
        query = self.query
        return rdd.filter(lambda x: query in x)
```

PAIR RDD: KEY-VALUE PAIRS

For numerous tasks, such as aggregations tasks, storing information as `(key, value)` pairs into RDD is very convenient.

Such RDDs are called `PairRDD`.

Forming a `PairRDD` often requires some ETL (Extract, Transform, Load) work to get the data into `(key, value)` format.

Pair RDDs expose new operations such as grouping together data with the same key, and grouping together two different RDDs.

EXAMPLE: CREATING A PAIR RDD USING THE FIRST ELEMENT OF A LIST AS A KEY

```
>>> rdd = sc.parallelize([[1, "a", "fe"], [2, "b", "de"],
 [2, "c", "fe"], ...])
>>> pairs = rdd.map(lambda x: (x[0], x[1:]))
```

Calling `sc.parallelize()` on an in-memory Python pair collection creates a `PairRDD`

TRANSFORMATIONS A SINGLE PAIR RDD

transformation	description
`keys()`	Return an RDD containing the keys.
`values()`	Return an RDD containing the values.
`sortByKey()`	Return an RDD sorted by the key.
`mapValues(f)`	Apply a function `f` to each value of a pair RDD without changing the key.

TRANSFORMATIONS A SINGLE PAIR RDD

transformation	description
`flatMapValues(f)`	Pass each value in the key-value pair RDD through a flatMap function `f` without changing the keys. Very useful for tokenization.

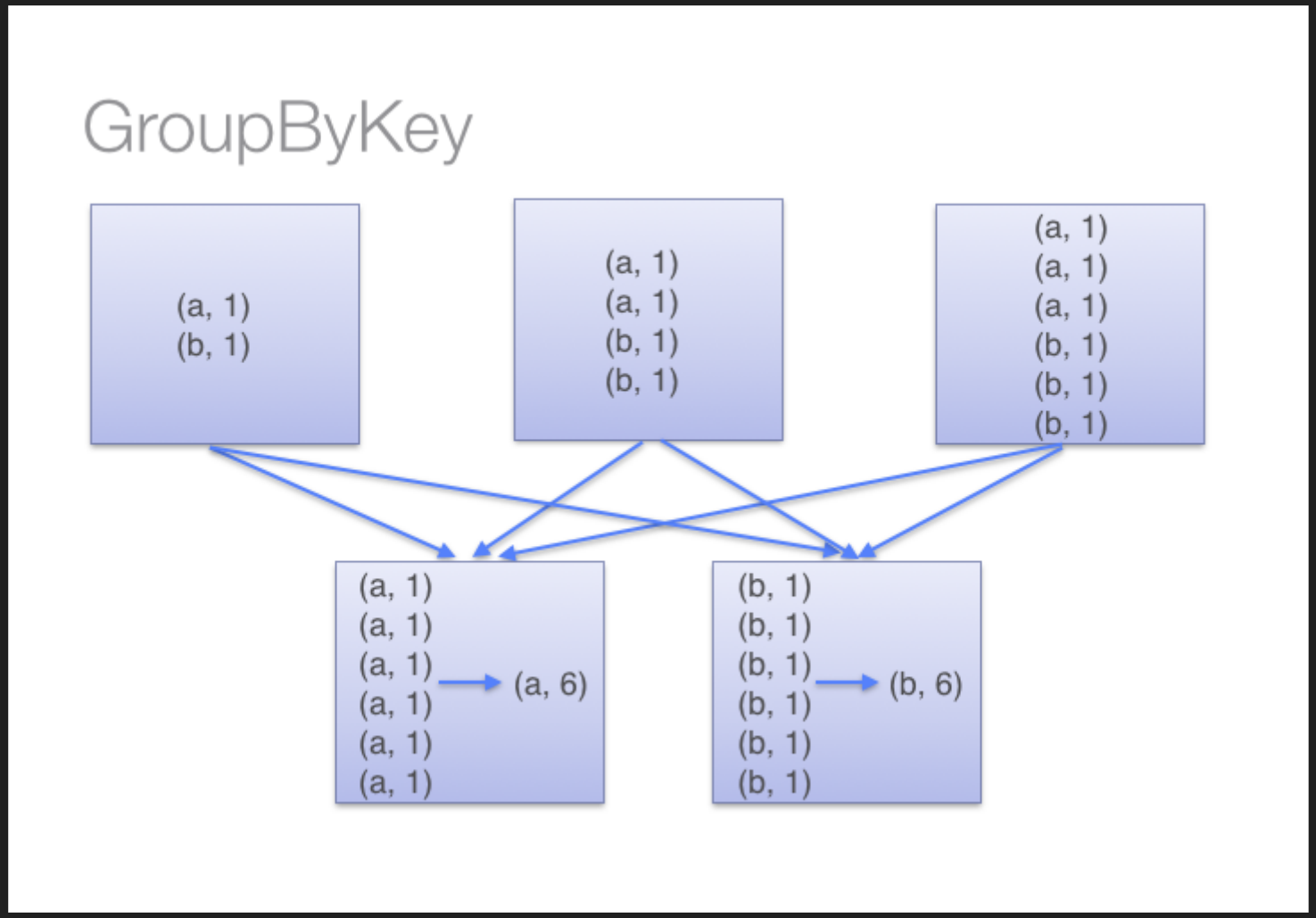
```
>>> texts = sc.parallelize([("a", "x y z"), ("b", "p r")])
>>> tokenize = lambda x: x.split(" ")
>>> texts.flatMapValues(tokenize).collect()
[('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
```

TRANSFORMATIONS A SINGLE PAIR RDD

transformation	description
`groupByKey()`	Group values with the same key

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.groupByKey().mapValues(list).collect())
[('a', [1, 1]), ('b', [1])]
```

GROUPBYKEY

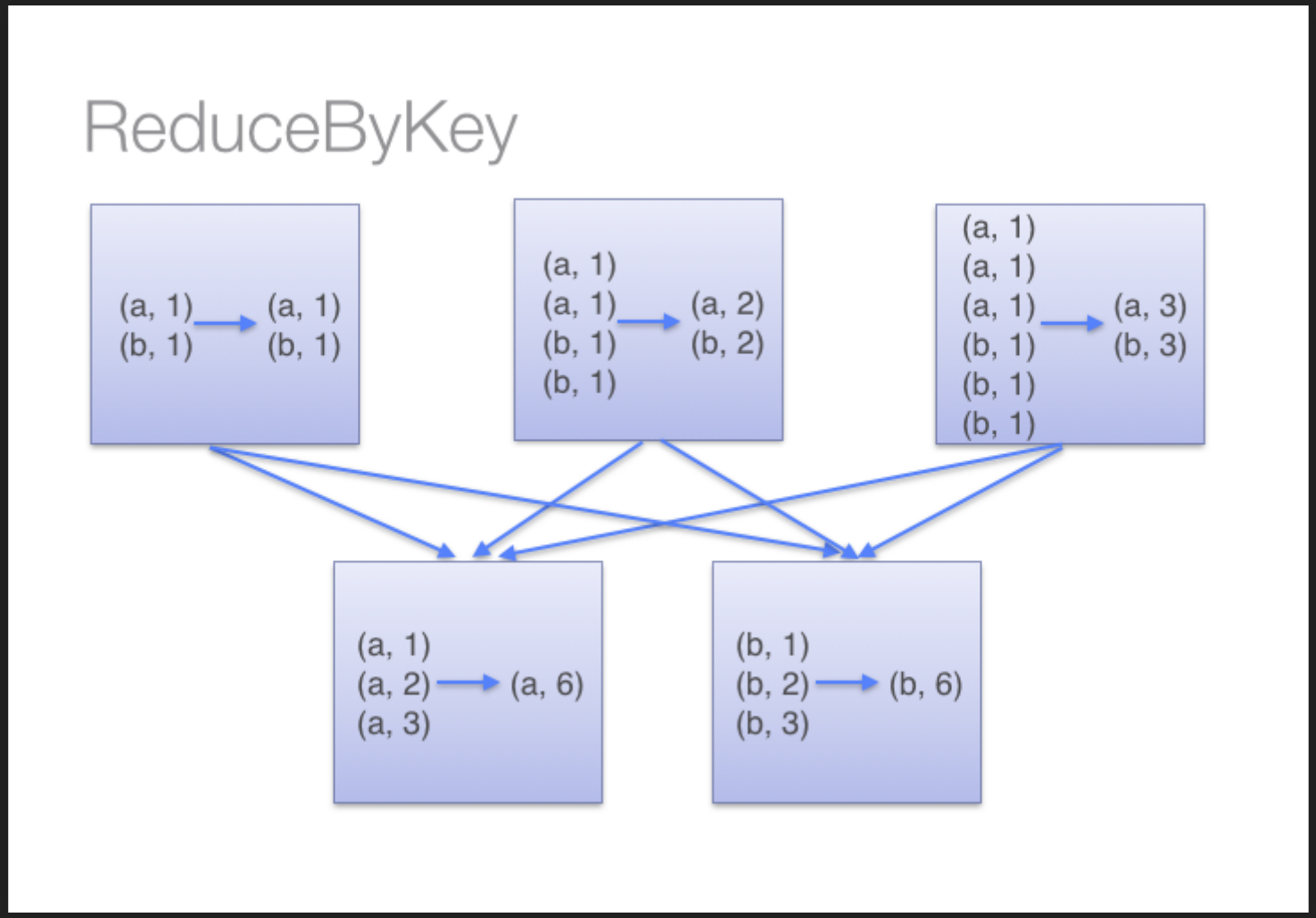

GroupByKey
(a, 1)
(b, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 6)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 6)

TRANSFORMATIONS A SINGLE PAIR RDD

transformation	description
`reduceByKey(f)`	Merge the values for each key using an associative and commutative reduce function `f`.
`foldByKey(f)`	Merge the values for each key using an associative reduce function `f`.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.reduceByKey(add).collect())
[('a', 2), ('b', 1)]
```

REDUCEBYKEY



TRANSFORMATIONS A SINGLE PAIR RDD

transformation	description
`combineByKey( createCombiner, mergeValue, mergeCombiners, [partitioner])`	Generic function to combine the elements for each key using a custom set of aggregation functions.

Turns an `RDD[(K, V)]` into a result of type `RDD[(K, C)]`, for a “combined type” `C` which can be different of `V`.

TRANSFORMATIONS A SINGLE PAIR RDD

The user must pass the following functions:

- createCombiner, which turns a V into a C (e.g., creates a one-element list)
- mergeValue, to merge a V into a C (e.g., adds it to the end of a list)
- mergeCombiners, to combine two C's into a single one.

```
>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> def add(a, b): return a + str(b)
>>> sorted(x.combineByKey(str, add, add).collect())
[('a', '11'), ('b', '1')]
```

TRANSFORMATIONS TWO PAIR RDDS

transformation	description
`subtractByKey(other)`	Remove elements with a key present in the `other` RDD.
`join(other)`	Inner join with `other` RDD.
`rightOuterJoin(other)`	Right join with `other` RDD.
`leftOuterJoin(other)`	Left join with `other` RDD.

TRANSFORMATIONS TWO PAIR RDDS

transformation	description
`cogroup(other)`	Group data from both RDDs sharing the same key.

```
>>> rdd = sc.parallelize([(1, 2), (3, 4), (3, 6)])
>>> other = sc.parallelize([(3, 9)])
>>> sorted(rdd.cogroup(other).collect())
[(1,([2],[])), (3, ([4, 6],[9]))]
```

ACTIONS TWO PAIR RDDS

action	description
`countByKey()`	Count the number of elements for each key.
`lookup(key)`	Return all the values associated with the provided `key`.
`collectAsMap()`	Return the key-value pairs in this RDD to the master as a Python dictionary.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()
>>> m[1]
2
>>> m[3]
4
```

DATA PARTITIONNING

Some operations on PairRDDs, shuch as `join`, require to scan the data more than once.

Partitionning the RDDs in advance can reduce network communications.

When a key-oriented dataset is reused multiple times, partitionning can lead to performance increase.

In Spark: you can choose which keys will appear on the same node, but no explicit control of which worker node each key goes to.

In practice, you can specify the number of partitions with

```
rdd.partitionBy(100)
```

You can also use a custom partition function `h` s.t. `h(key)` returning a hash.

```
import urlparse
def hash_domain(url):
return hash(urlparse.urlparse(url).netloc)
rdd.partitionBy(20, hash_domain) # Create 20 partitions
```

To have finer control on partitionning, you must use the Scala API.

Warning:

The hash function you pass will be compared by identity to that of other RDDs.

If you want to partition multiple RDDs with the same partitioner, pass the same **function object** (e.g., a global function) instead of creating a new lambda for each one!

Warning:

The hash function you pass will be compared by identity to that of other RDDs.

If you want to partition multiple RDDs with the same partitioner, pass the same **function object** (e.g., a global function) instead of creating a new lambda for each one!

SECOND LAB:
PYSPARK RDDS