An Insufficient Introduction to Spark

Part 2: RDDs and operations on them

Riccardo Murri <riccardo.murri@gmail.com>

2019-02-11

Spark

What is Spark?

Apache Spark is a general-purpose distributed computation framework.

- computation model based on directed acyclic graphs (DAG)
  - Spark does the mapping onto Map/Reduce stages
  - can do its own scheduling if no M/R engine available
- supports interactive use:
  - good for data exploration
- can keep data in-memory:
  - good for loop-intensive algorithms
- has a rich feature-set to make programming easier (in Scala, Java, Python, R)
  - including ML and graph-processing libraries

Flexibility of Spark runtime

The spark runtime can be deployed on:

- a single machine (local)
- a set of pre-defined machines (stand-alone)
- a dedicated cluster (YARN/Mesos)

The development workflow is that you start small (local) and scale up to one of the other solutions, depending on your needs and resources.

Often, you don't need to change *any* code to go between these methods of deployment!

Recall what was hard about distributed computing:

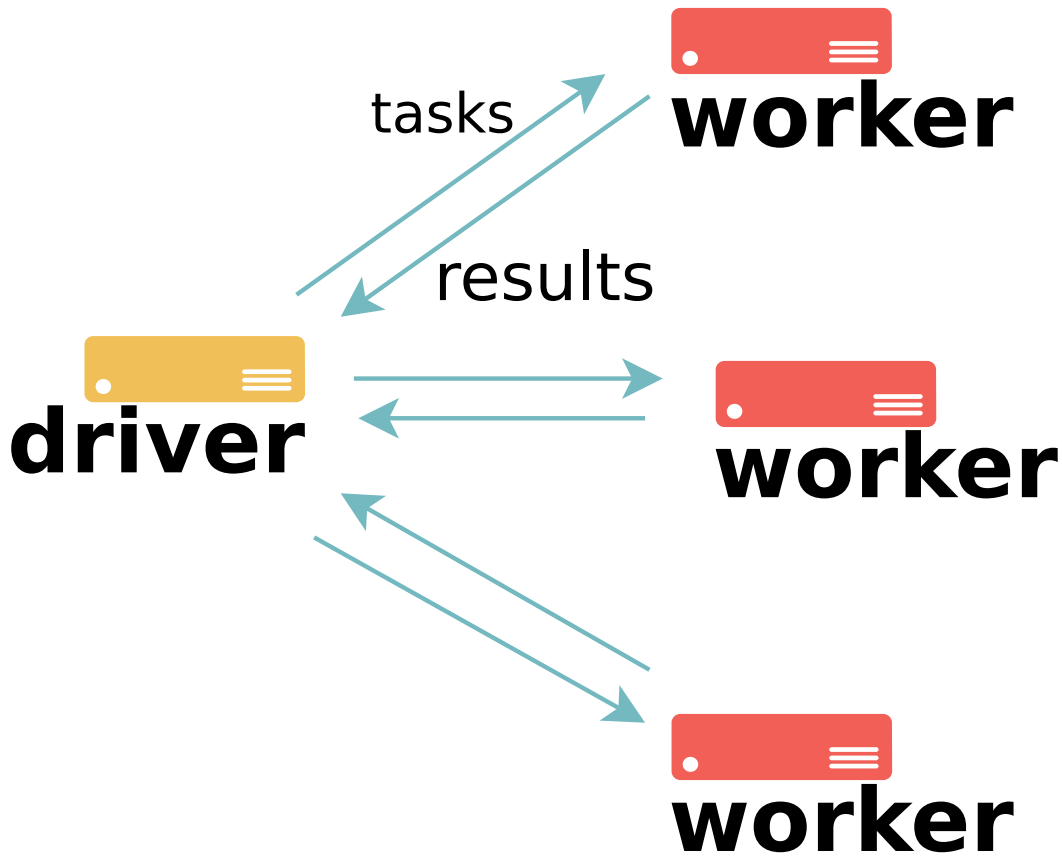
1. distributing work to the available resources
2. orchestrating task execution
3. collecting results

This is what a “framework” like Spark does for us.

At its most basic, it consists of a **driver** and **workers**.

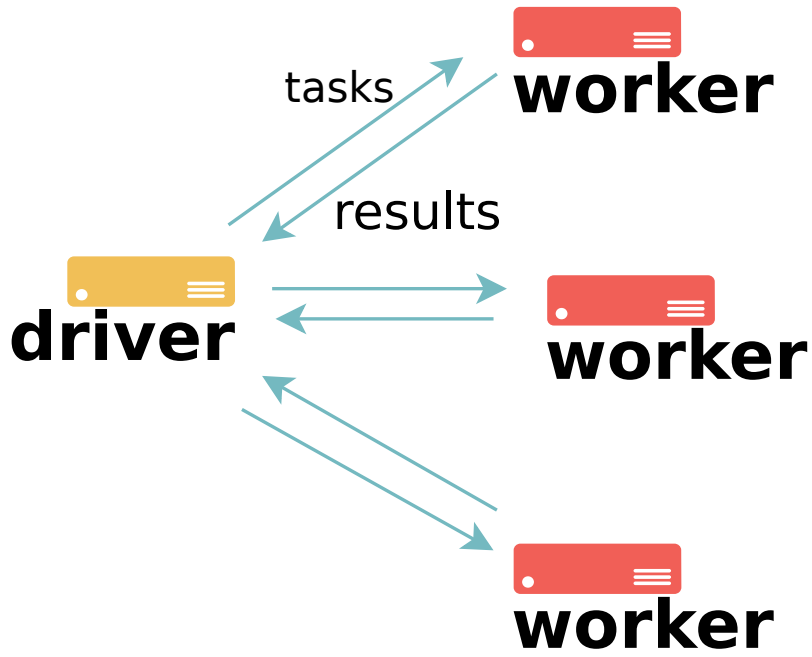
Spark Architecture Overview

A running Spark system consists of a single **driver** and a set of **workers**.



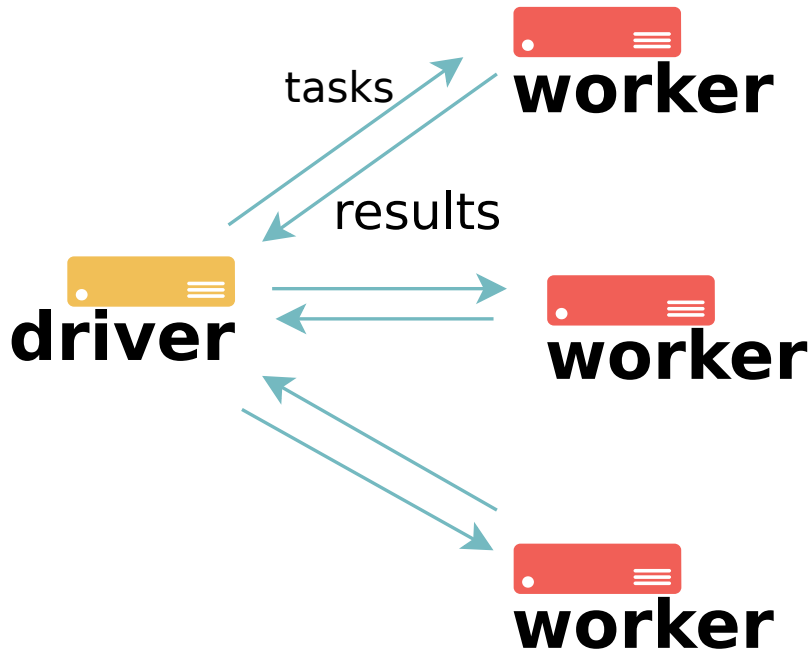
Driver

- coordinates the work to be done
- keeps track of tasks
- communicates with the workers (and the user)



Workers

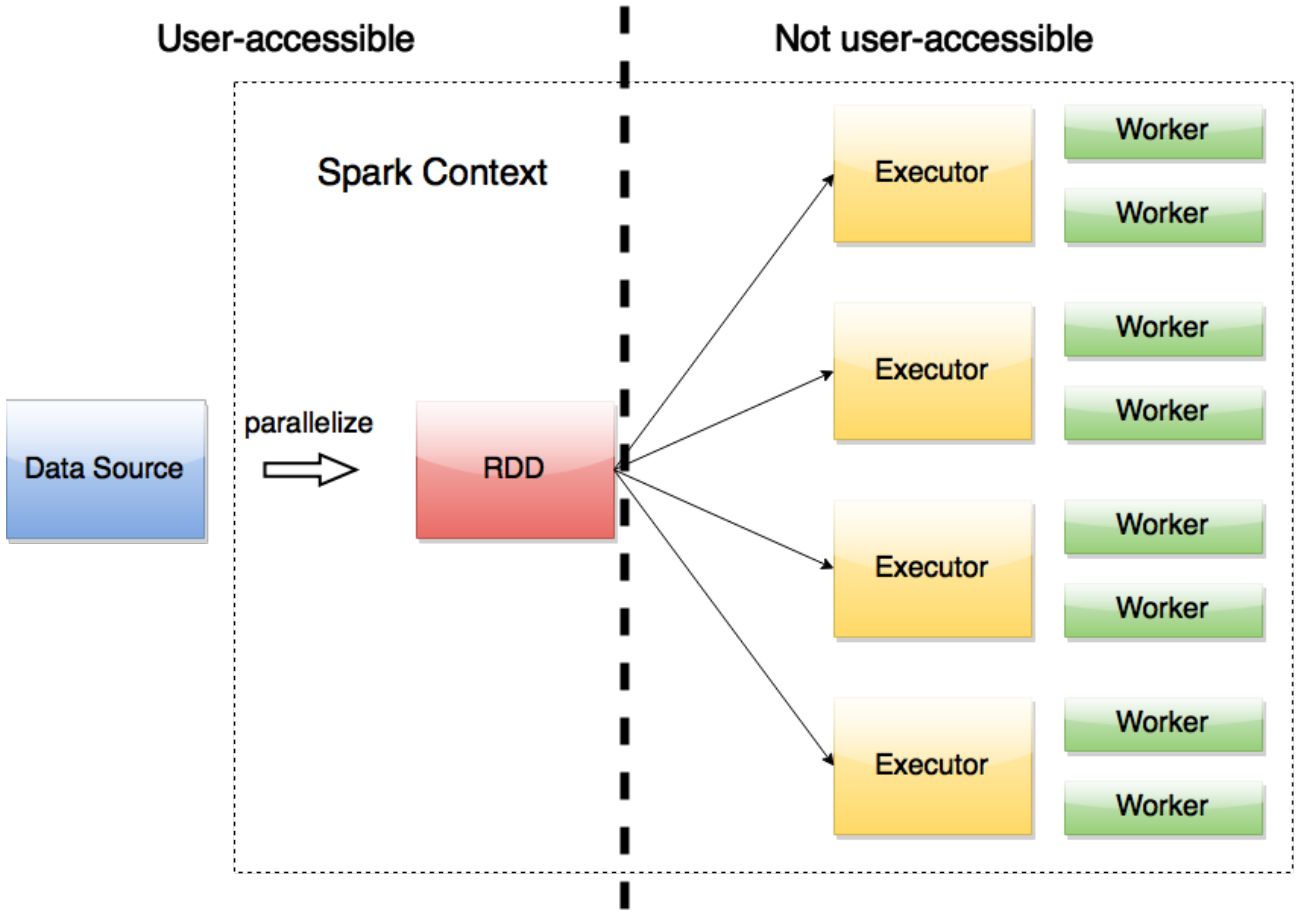
- receive tasks to be done from the driver
- store data in-memory or on disk
- perform calculations
- return results to the driver



RDDs: basic data manipulation

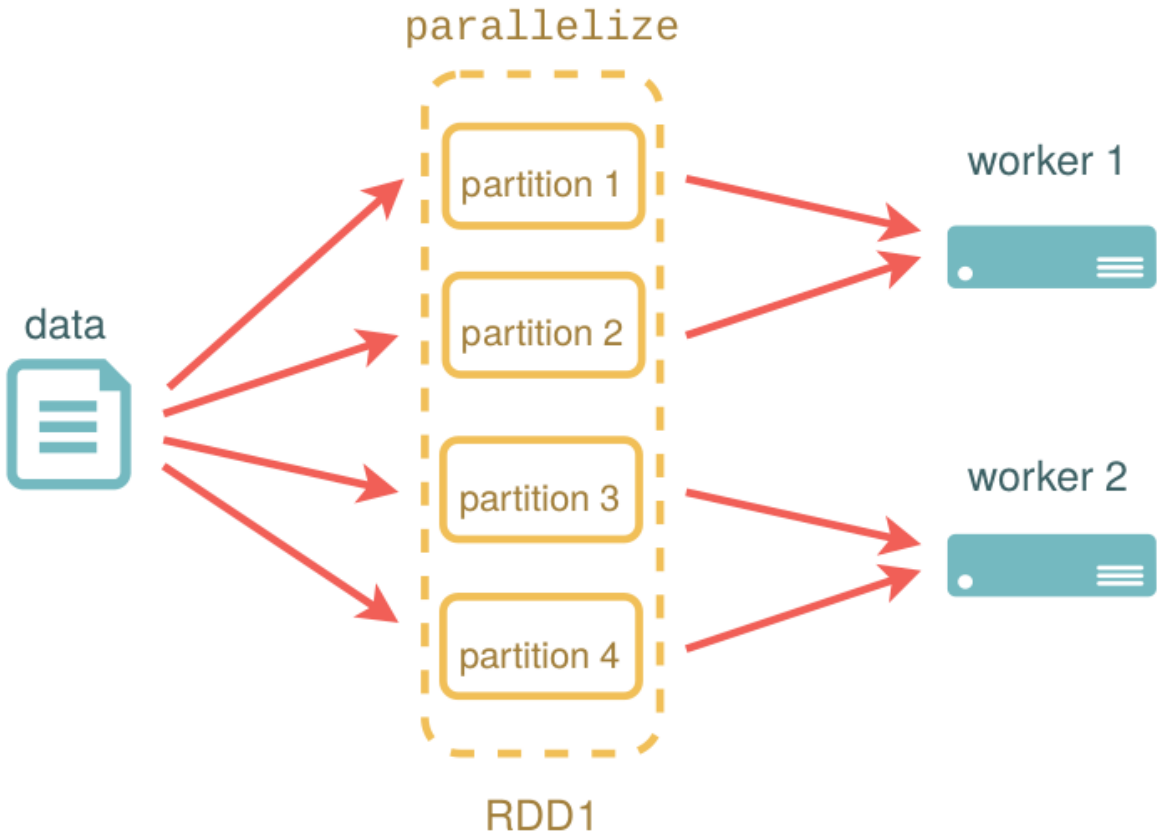
Spark Context

The user's entry point to Spark is the **Spark Context** which provides an interface to generate RDDs (i.e., inject data into the system).



Spark Context

The user's entry point to Spark is the **Spark Context** which provides an interface to generate RDDs (i.e., inject data into the system).




```
rdd1 = sc.parallelize(data, 4)
```

Creating RDDs

```
rdd1 = sc.parallelize(data, 4)
```

This is the "Spark Context" Python object.

It is automatically available in our Jupyter notebooks and in the `pyspark` shell.

Creating RDDs

```
rdd1 = sc.parallelize(data, 4)
```

The `.parallelize()` method is used to *copy* data from Python into the Spark system.

Creating RDDs

```
rdd1 = sc.parallelize( data , 4)
```

Any Python sequence can be used as data.

Creating RDDs

```
rdd1 = sc.parallelize(data, 4)
```

This is the number of "partitions" to divide the data into. Each partition can be processed *independently* by a worker.

Variable `sc.defaultParallelism` holds the default number of executors; a good starting point for the number of partitions is `sc.defaultParallelism*4` — adjust up or down depending on size of the data.

Loading data

```
rdd2 = sc.textFile('hdfs:///shakespeare.txt.gz')
```

For *actual, real* data processing you would rather read data from a file or another data source!

Loading data

```
rdd2 = sc.textFile('hdfs:///shakespeare.txt.gz')
```

There are multiple functions for loading data in different ways:

- `sc.textFile()`: load and chunk a text file (possibly compressed)
- `binaryFiles()`: `wholeTextFiles()`: Read a directory of files. Each file is read as a single record and returned in a key-value pair, where the key is the path of each file, the value is the content of each file.
- `binaryRecords()`: read a binary file and chop it in records of the specified length.

Loading data

```
rdd2 = sc.textFile('hdfs://shakespeare.txt.gz')
```

All paths can be a local filesystem path (prefix `file://`), a HDFS location (prefix `hdfs://`), or any otherfilesystem visible from the *entire* Hadoop cluster.

Creating RDDs

```
rdd1 = sc.parallelize(data, 4)
```

```
rdd2 = sc.textFile('hdfs:///shakespeare.txt.gz')
```

An RDD is the result of entering unstructured data into Spark.

RDD: Resilient Distributed Dataset

An RDD is the primary interface of every Spark application:

- **ordered immutable collection** of arbitrary data
- provides an interface to the user to access and operate on the data
- keeps track of lineage
- tracks distribution of data across the workers

Spark applications feed data into RDDs and subsequently operate on them to compute the desired result.

Transformations and Actions

Once an RDD is created, it is **immutable**.

There are two classes of operations that can be applied to a given RDD:

- **transformations:** create a new (output) RDD by applying some operation on the data of the input RDD;
- **actions:** take data out of an RDD and into another data structure (e.g., list or dictionary)

Actions

Actions take data out of an RDD and into a host language data structure (e.g., list or dictionary)

- converts to host language native data structures
- output data is collected on the *driver* process: risk of a memory overflow!

Actions on all RDDs

Actions available on *all* RDDs include:

- `collect` – return a list of all elements of the RDD to the driver (often a bad idea!!)
- `count` – return the number of elements of an RDD
- `countApproxDistinct` – return estimation of number of unique elements of an RDD
- `take`, `takeOrdered`, `takeSample` – yield a desired number of items to the driver
- `first` – returns the first element of the RDD to the driver

Reducing an RDD to one element

There are multiple actions reducing a whole RDD to a single value.

```
val = rdd.reduce(f)
```

Go through partitions, applying *binary* function `f(x,y)` to the first two values, then to the result and the 3rd value, and so on – then apply the procedure again to combine results from partitions into one. Function `f` *must be* commutative and associative.

```
val = rdd.fold(start, f)
```

Like `reduce`, but combines first value of the RDD with provided `start`.

```
val = rdd.aggregate(start, f1, f2)
```

Like `fold`, but combines elements within a partition using `f2`, and combines results from different partitions using `f1`.

Actions on numeric RDDs

These actions are defined for all RDDs, but will only work for an all-numeric one.

- `max`, `mean`, `min`, `stdev` – basic statistics operations on numeric RDDs
- `sum` – add all the elements in an RDD

Exercise 2.A: How many lines are there in text file `hdfs:///shakespeare.txt.gz`?

Exercise 2.B: Compute the product of the sequence of numbers `[1, 2, 3, 4, 5]` using PySpark.

Exercise 2.C *(advanced)*: A very simple technique for checksumming a stream of bytes is computing the "bit parity" of each bit position (i.e., number of "1" bits).

Compute the bit parity checksum of the above ASCII text file using PySpark.

Hint: you can compute parity of two bytes *a* and `b` by combining them with Python's `a^b` operator (XOR); it is a commutative and associative operation.

Actions on Key/Value RDDs

A special place is taken by RDDs whose elements are key/value pairs.

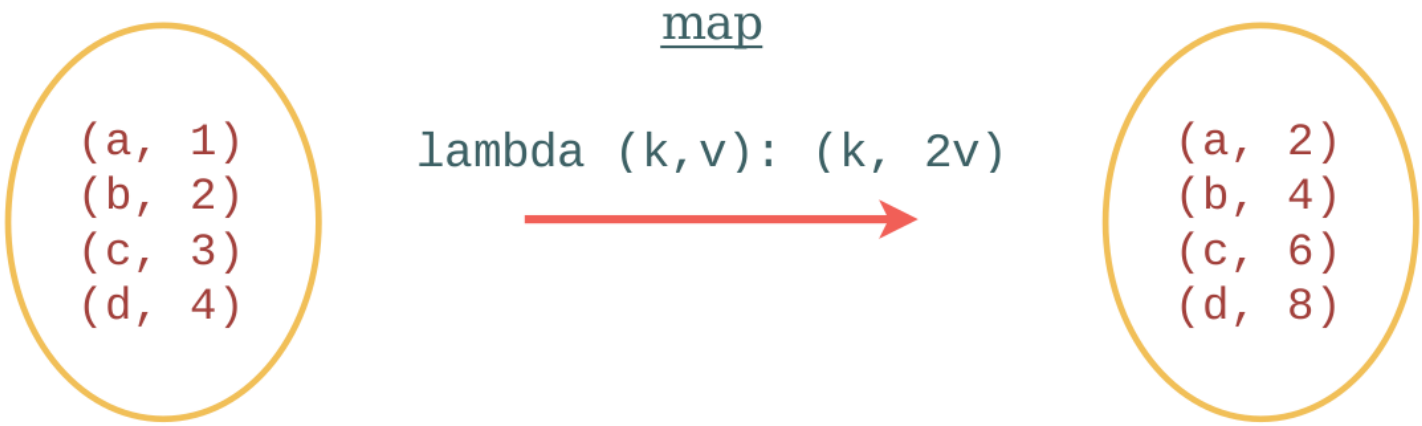
- `aggregateByKey` – Like `aggregate` but aggregate separately the values of each key.
- `collectAsMap` – like `collect` but returns a dictionary to the driver which makes it easy to lookup the keys
- `countByKey` – return the number of elements for each key
- `countByValue` – return the count of each *unique* value
- `lookup` – return the list of values for a key
- `reduceByKey`, `foldByKey` – Merge the values of each key using a given operator

Transformations

Transformations create a new (output) RDD by applying some operation on the data of the input RDD.

`map` transformation

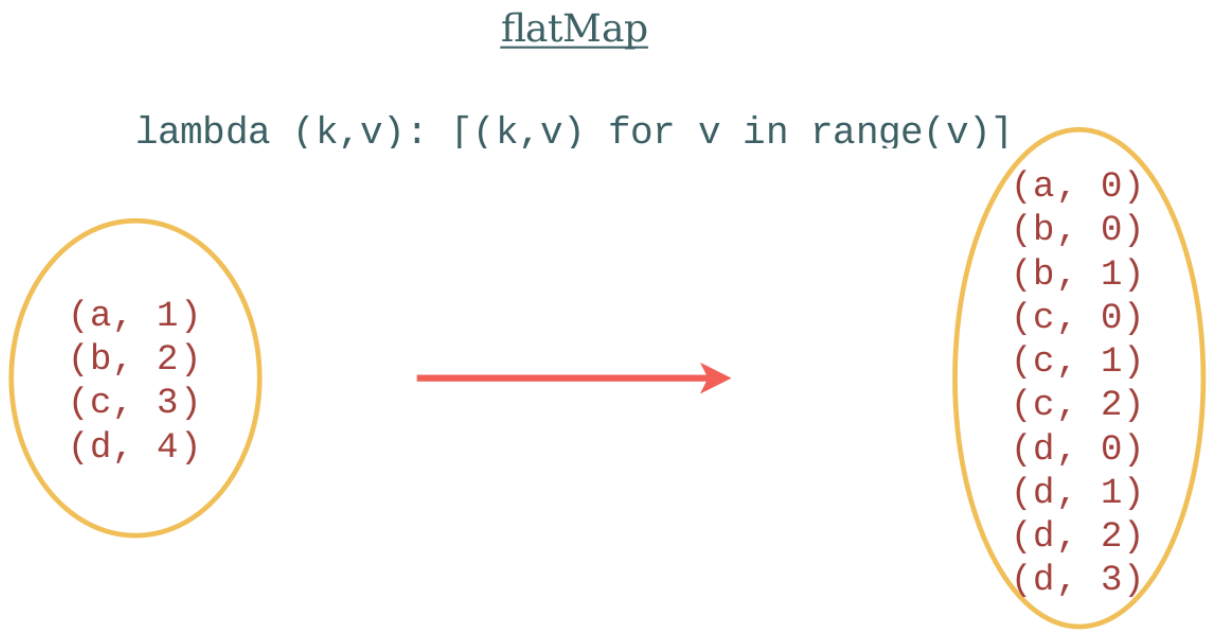
Create a new RDD by applying a $1-1$ function F to each element of a given RDD.




```
rdd2 = rdd1.map(F)
```

`flatMap` transformation

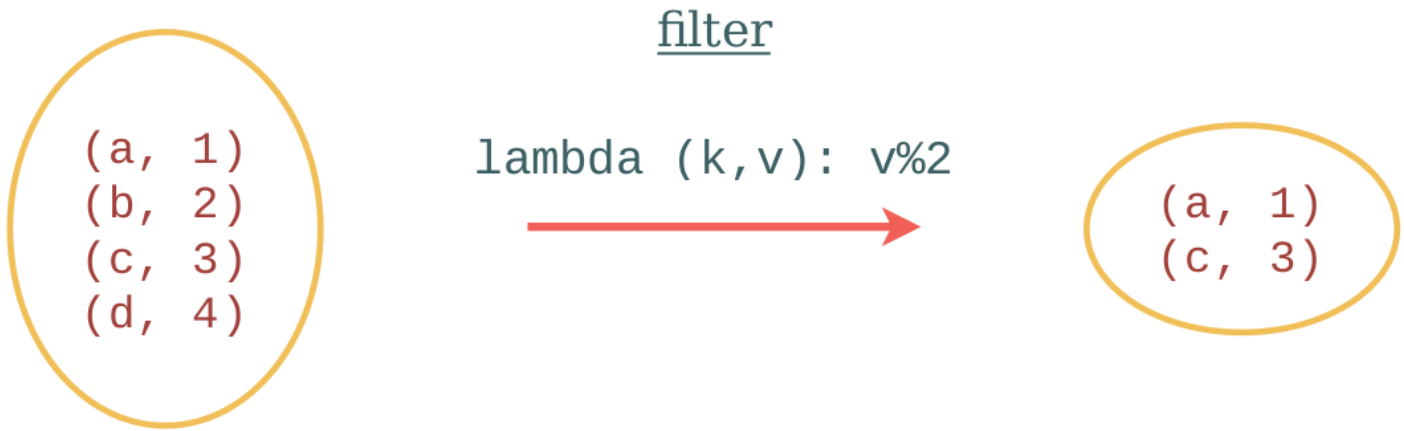
Apply a (possibly one-to-many) function *F* to each element of a given RDD. Expect that *F* produces a list for each element of the input RDD, and make a new (output) RDD from the concatenation of such lists.




```
rdd2 = rdd1.flatMap(F)
```

`filter` transformation

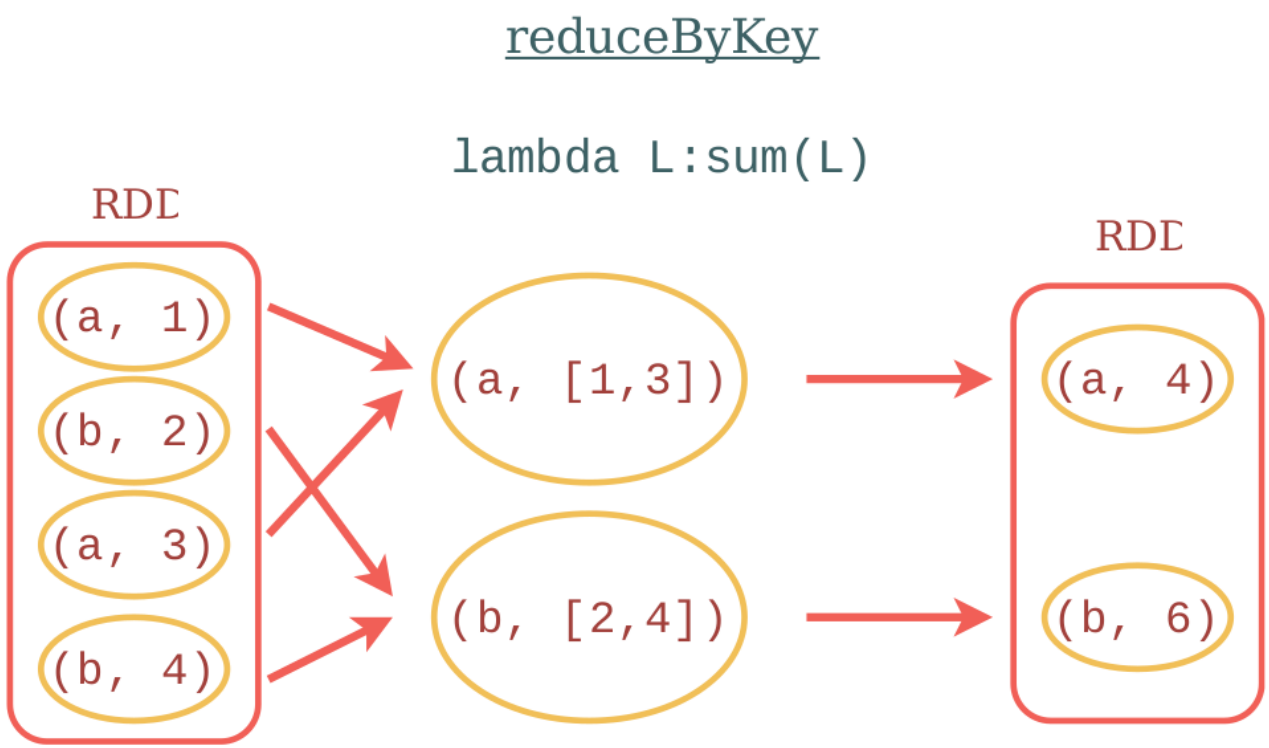
Create a new RDD by selecting elements from the input RDD on which function *F* takes a "true" value.




```
rdd2 = rdd1.filter(F)
```

`reduceByKey` transformation

Group elements by key and reduce the resulting sequence of values to form a new RDD.



Other transformations on Key/Value RDDs

Other transformations include:

- `rdd2 = rdd1.groupBy(f)`
  Create a key/value RDD `rdd2` using function `f` to compute the key corresponding to each value in `rdd1`.
- `rdd2 = rdd1.keys()`
  `rdd2 = rdd1.values()`
  `rdd2` is the set of keys or values of `rdd1`
- `rdd2 = rdd1.mapValues(f)`
  Pass each value in `rdd1` through function `f` without changing the keys.
- `rdd1 = rdd2.zip(rdd3)`
  `rdd1` is the RDD of pairs whose first element comes from `rdd2` and second element from `rdd3`

Set-theoretic transformations

Other transformations include:

- `rdd3 = rdd1.cartesian(rdd2)`
  `rdd3` is the set of pairs `(a, b)` where `a` is an element of `rdd1` an `b` is an element of `rdd2`
- `rdd3 = rdd1.intersection(rdd2)`
  `rdd3 = rdd1.subtract(rdd2)`
  `rdd3 = rdd1.union(rdd2)`
  `rdd3` is the set-theoretic intersection/difference/union of `rdd1` and `rdd2`

Other transformations

Other transformations include:

- `rdd2 = rdd1.distinct()`
  only retain the unique elements of the entire RDD
- `rdd2, rdd3 = rdd1.randomSplit([w1, w2])`
  Split `rdd1` into `rdd2`, `rdd3` by randomly assigning elements with probabilities `w1`, `w2`.
- `rdd2 = rdd1.sortBy(f)`
  `rdd2` has the same elements as `rdd1`, sorted so that `f` takes ascending values.

Lineage

Transformations are evaluated "lazily": only executed once an *action* is performed.

- When an RDD is transformed, this **transformation** is not automatically carried out.
- Instead, the system remembers how to get from one RDD to another and only executes whatever is needed for the **action** that is being done.
- This allows one to build up a complex "pipeline" and easily tweak/rerun it in its entirety.

Exercise 2.D: Implement a the "word count" algorithm using PySpark, and use it to count the words in file `hdfs:///shakespeare.txt.gz`

What are the 5 most frequent words?

Hint: The "word count" algorithm is comprised of the following steps:

- Read lines of a text file into an RDD;
- Transform the RDD by splitting each line at blank spaces;
- Create a key/value RDD by pairing each word with the value `1`;
- Sum the values associated with each word.

Exercise 2.E: Write a solution to Exercise 1.E using PySpark.

Recap of RDD usage

1. Create a `SparkSession` object.[1]
2. Inject data into Spark:

   ```
   rdd = sc.parallelize(data)
   ```

3. Build a computation DAG by chaining *transformations*:

   ```
   step1 = rdd.flatMap(lambda line:
                  [(w,1) for w in line.split()])
   step2 = step1.reduceByKey(lambda a,b: a+b)
   ```

4. Extract final data with *actions*:

   ```
   wc = step2.collectAsMap()
   ```

[1] In Spark 1.x this was called `SparkContext`.