Resilient Distributed Datasets (RDDs)

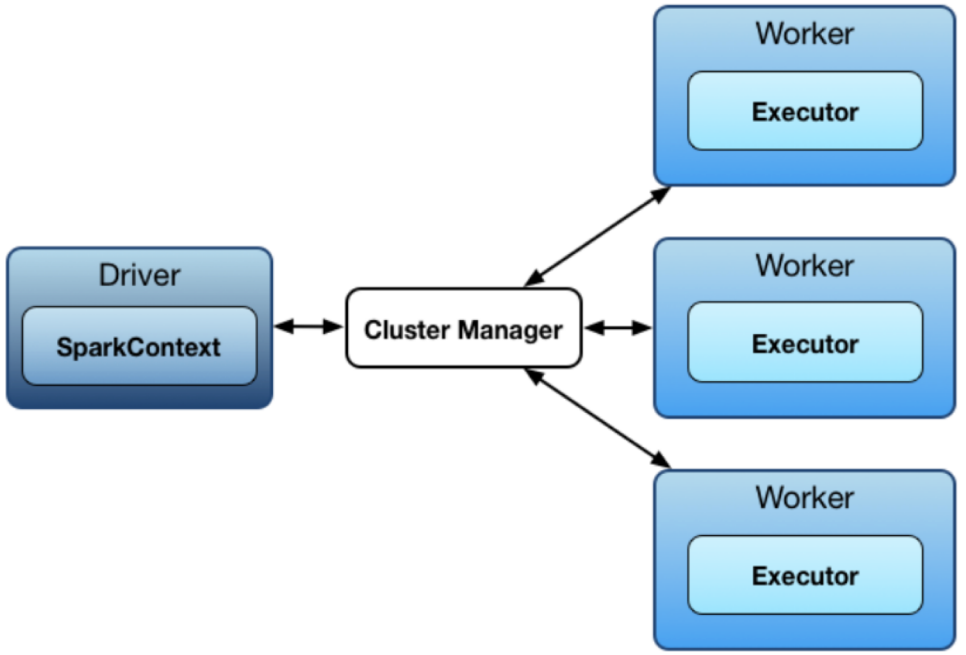
A brief Introduction

Resilient Distributed Datasets (RDDs)

- Primary abstraction that allow Spark to distribute data
- Fault tolerant – if they are destroyed they can be recreated by the driver and sent to a new worker
- Immutable – once created you cannot change them. Instead you perform transformations on them and create new RDDs.
- Unstructured and semi-structured data
- Remain in memory
- Many input sources : HDFS, S3, csv, json

RDD operations

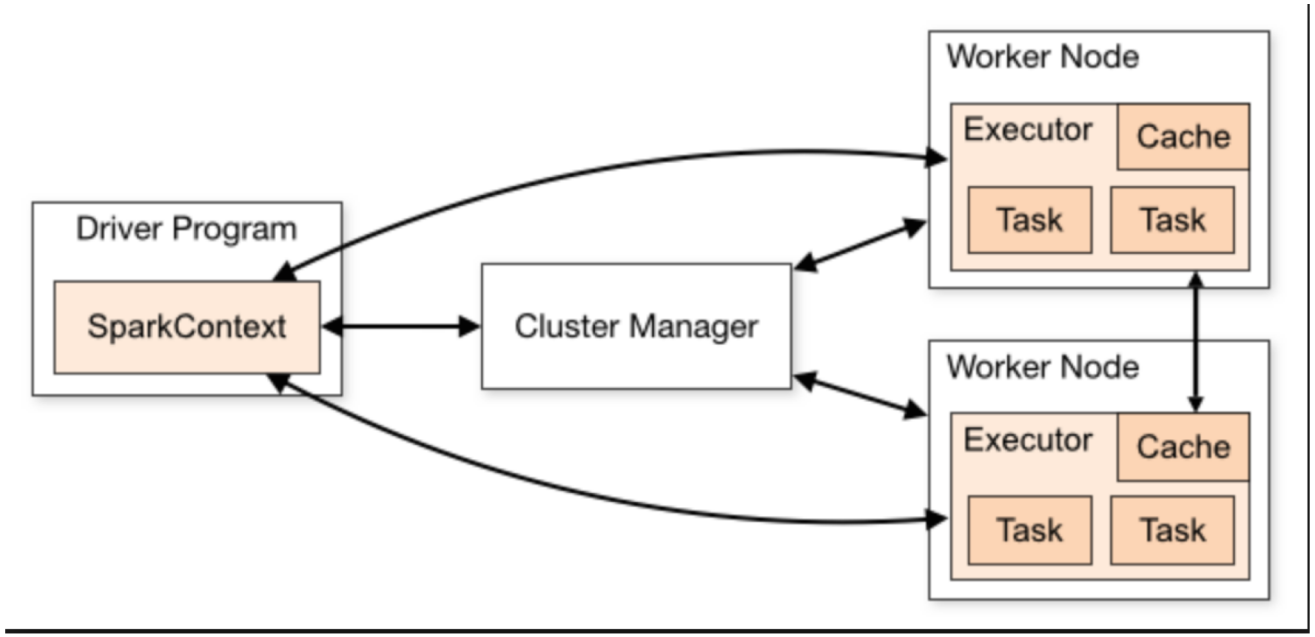
- Transformations – map, flatmap, join
  - Lazy operations
  - Take place on the worker nodes
- Actions – count, distinct, reduce
  - Eager operations
  - Results returned to the driver and summarized or written to storage



Reference

Your entry point into Spark is the Spark Context

```
import pyspark
sc = pyspark.SparkContext('local[*]')
```



Creating RDDs

You can create an RDD in one of three ways:

- ***Parallelizing*** an existing collection in your driver program
- ***Referencing a dataset*** in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat
- By ***transformations*** on another RDD

Resilient Distributed Datasets (RDDs)

Parallelizing an existing collection in your driver program

```
>>>  data = [1, 2, 3, 4, 5]                    //Python List
>>>  distData = sc.parallelize(data)           // RDD
```

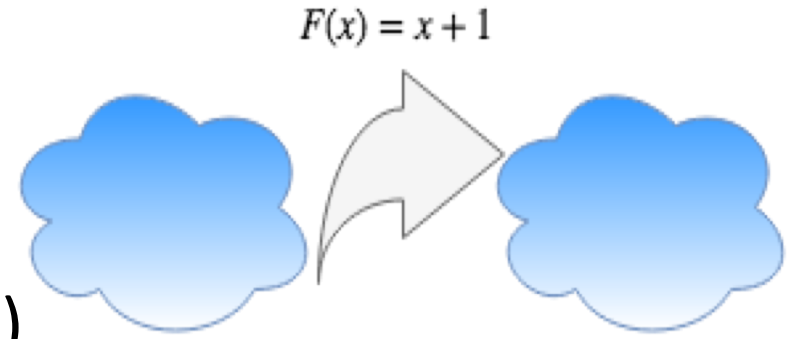
Referencing a dataset

```
>>> distFile = sc.textFile("data.txt")       //RDD
```

Resilient Distributed Datasets (RDDs)

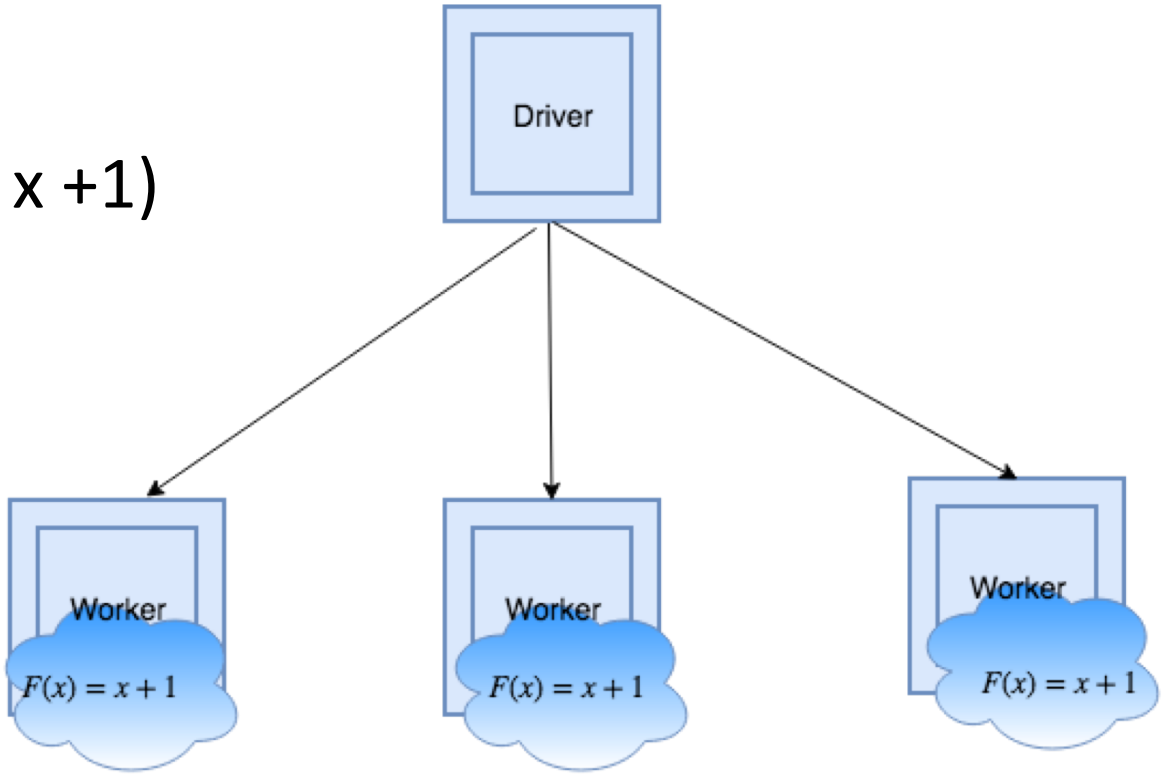
Given an RDD you can create a new RDD using ***transformations***.

```
>>>  data = [1, 2, 3, 4, 5]
>>>  rdd1 = sc.parallelize(data)
>>>  rdd2 = rdd1.map(lambda x : x + 1)
```



Resilient Distributed Datasets (RDDs)

rdd2 = rdd1.***map***(lambda x : x +1)



Transformations on RDDs

Transform one RDD to another

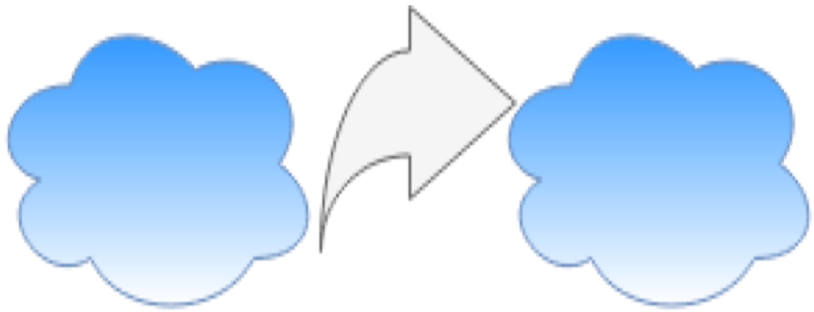
map

flatmap

filter

fold

aggregate



Reference

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)   # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1,
0, 1, 2}
```

Transformations are lazy

To get the results of ***a transformation*** back to the driver, you must issue an ***action***

```
>>>  data = [1, 2, 3, 4, 5]
>>>  rdd1 = sc.parallelize(data)
>>>  rdd2 = rdd1.map(lambda x : x + 1)
        ###  Nothing happens until »collect» is executed
>>>  rdd2.collect()
```

Resilient Distributed Datasets (RDDs)

- Actions
  - collect
  - count
  - reduce
  - take(n)

Reference

Basic Actions - Eager

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile(“hdfs://file.txt”)
```

Resilient Distributed Datasets (RDDs)

In [1]:
```
import pyspark
sc = pyspark.SparkContext('local[*]')
```

In [2]:
```
data = [1, 2, 3, 4, 5]
rdd1 = sc.parallelize(data)
rdd2 = rdd1.map(lambda x : x + 1)
print(type(rdd2))
```

```
<class 'pyspark.rdd.PipelinedRDD'>
```

In [3]:
```
mylist = rdd2.collect()
print(type(mylist))
```

```
<class 'list'>
```

In [5]:
```
print(mylist)
```

```
[2, 3, 4, 5, 6]
```

Lazy Evaluation - An execution plan, a DAG (directed acyclic graph) of tasks is sent to the workers. It is not executed until an action is run