Universidad Autónoma de Madrid

excelencia UAM CSIC+

Escuela Politécnica Superior

Task 3
Spark

Introduction

- Nowadays, large amounts of data are gathered and stored for its analysis
  - Data is the new petrol
- Massive data processing is needed for a wide variety of applications
  - Finances, Genetics, Marketing, Artificial Intelligence, etc.
- How can we process these humongous amounts of data in reasonable times?
  - One computer even with GPUs is not enough
  - Need of computer clusters

Introduction

- In addition to the computer cluster, we need a programming model to access and process large datasets:
  - Efficiently
  - In parallel
  - In a distributed way

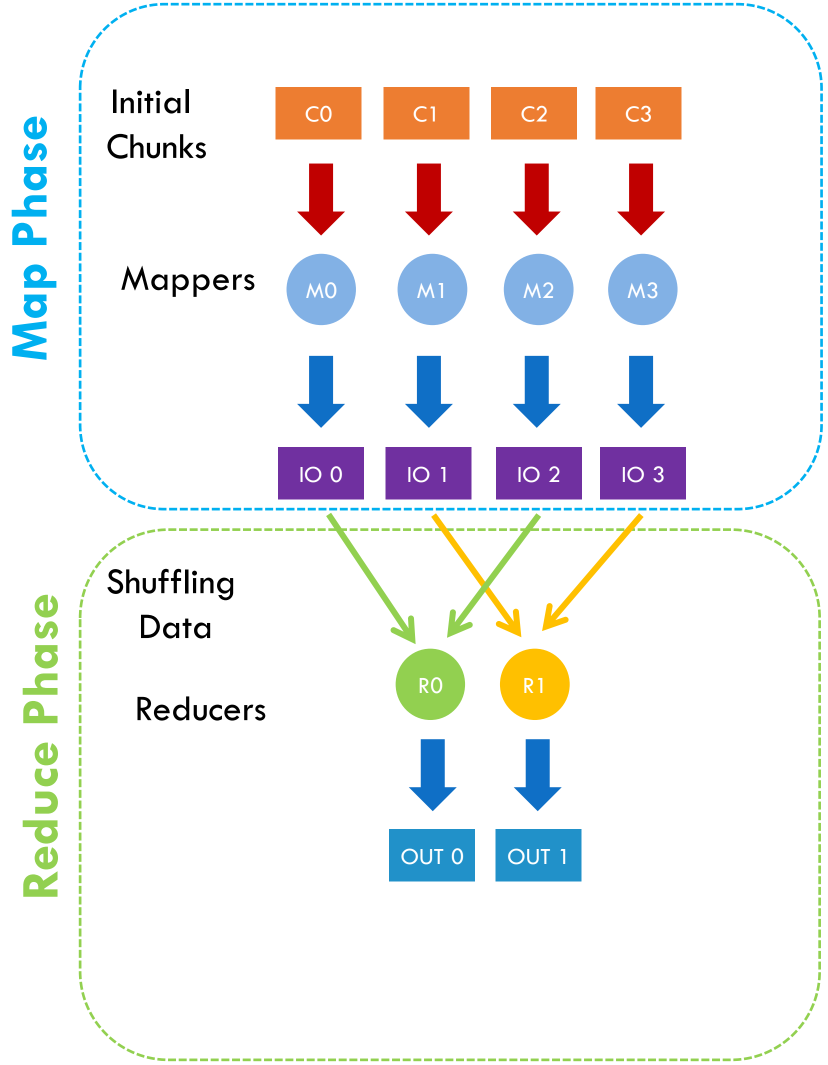
- Solution:
  - Map-Reduce paradigm

Map-Reduce Paradigm

- Map-Reduce is a programming paradigm for data processing.
- Enables massive processing scalability using 100s or 1000s computers (nodes) in parallel.
- Based on two different tasks:
  - Map
  - Reduce
- Map-Reduces divides the workload in independent and isolated tasks that can run in parallel at each node.
- Input data is stored along all the cluster nodes using a distributed file system (HDFS)

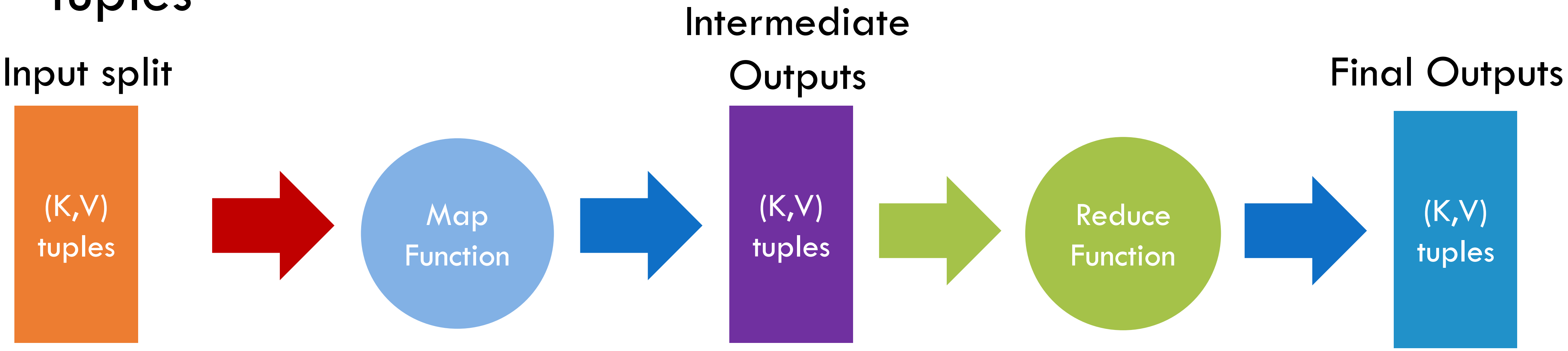
Map-Reduce Paradigm

- Divided in 2 phases:
  - Map:
    - Initial data is stored in chunks
    - Each chunk is processed independently by a Mapper task
    - Mappers produce Intermediate Outputs (IOs) that are assigned to Reducer tasks
  - Reduce:
    - Combining IOs into reducers is called "shuffling process"
    - Reducers process data and produce the final output



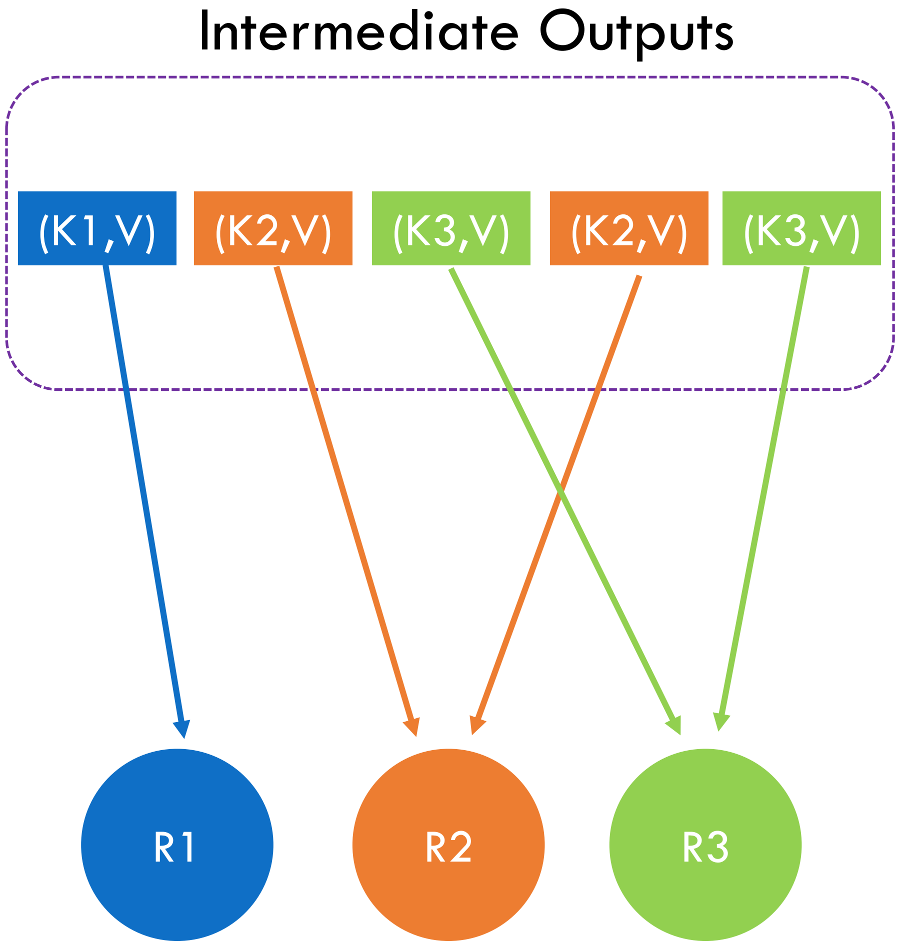
Map-Reduce Paradigm

- The programmer implements the functionality of both Map and Reduce functions
- Data elements are always defined by a key-value tuple (K,V)
- Map and Reduce functions receive and produce key-value tuples



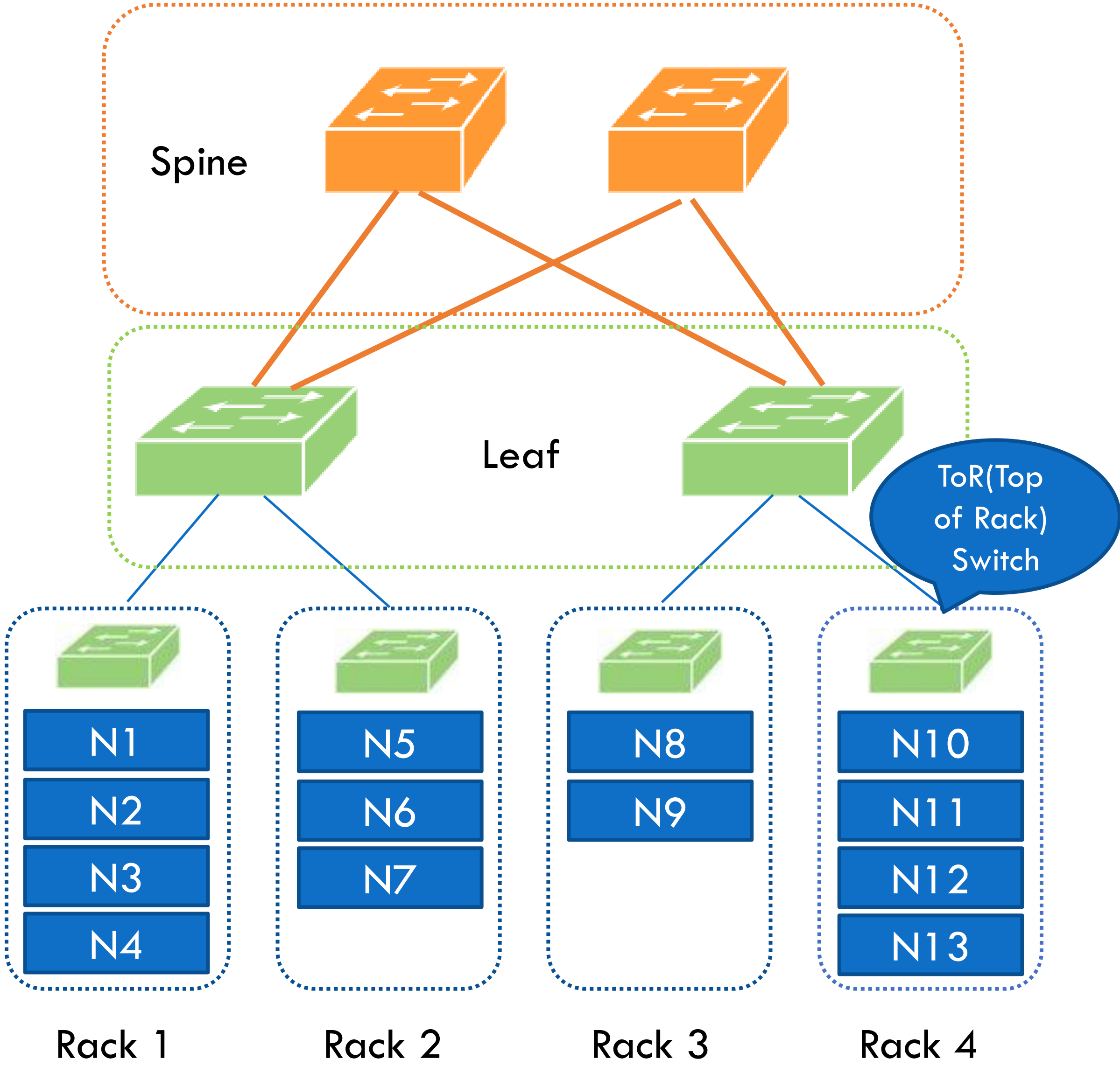
Map-Reduce Paradigm

- Not all intermediate outputs are reduced together
- IOs with the same key are grouped in partitions
- Each partition is assigned to a reducer



Map-Reduce Paradigm

- Task are distributed along nodes of cluster
- Communication and management is performed using communication networks
  - Overall performance depends highly on communications performance
- Typical topology is a tree
  - Data centers are distributed using a leaf-spine architectures
  - BW is higher between nodes on the same rack as opposed to nodes on other racks



Distributed programming frameworks for Big Data.

- **Apache Hadoop:** collection of tools that provide distributed storage and Map-Reduce processing for big data over commodity clusters.
  - Implemented in Java
  - Disk/Storage-oriented → HDFS (Hadoop File System)
- **Apache Spark:** multi-language engine for executing data engineering, data science, and machine learning tasks in a distributed way
  - Evolution of the Map-Reduce Paradigm
  - Memory-oriented Vs disk-oriented

Apache Spark (I)

- MapReduce problems:
  - Many problems aren't easily described as map-reduce
  - Persistence to disk typically slower than in-memory work
- Apache Spark:
  - Use of memory
  - Avoid saving intermediate results to disk
  - **Cache data** for repetitive queries (e.g. for machine learning)
  - Compatible with Hadoop

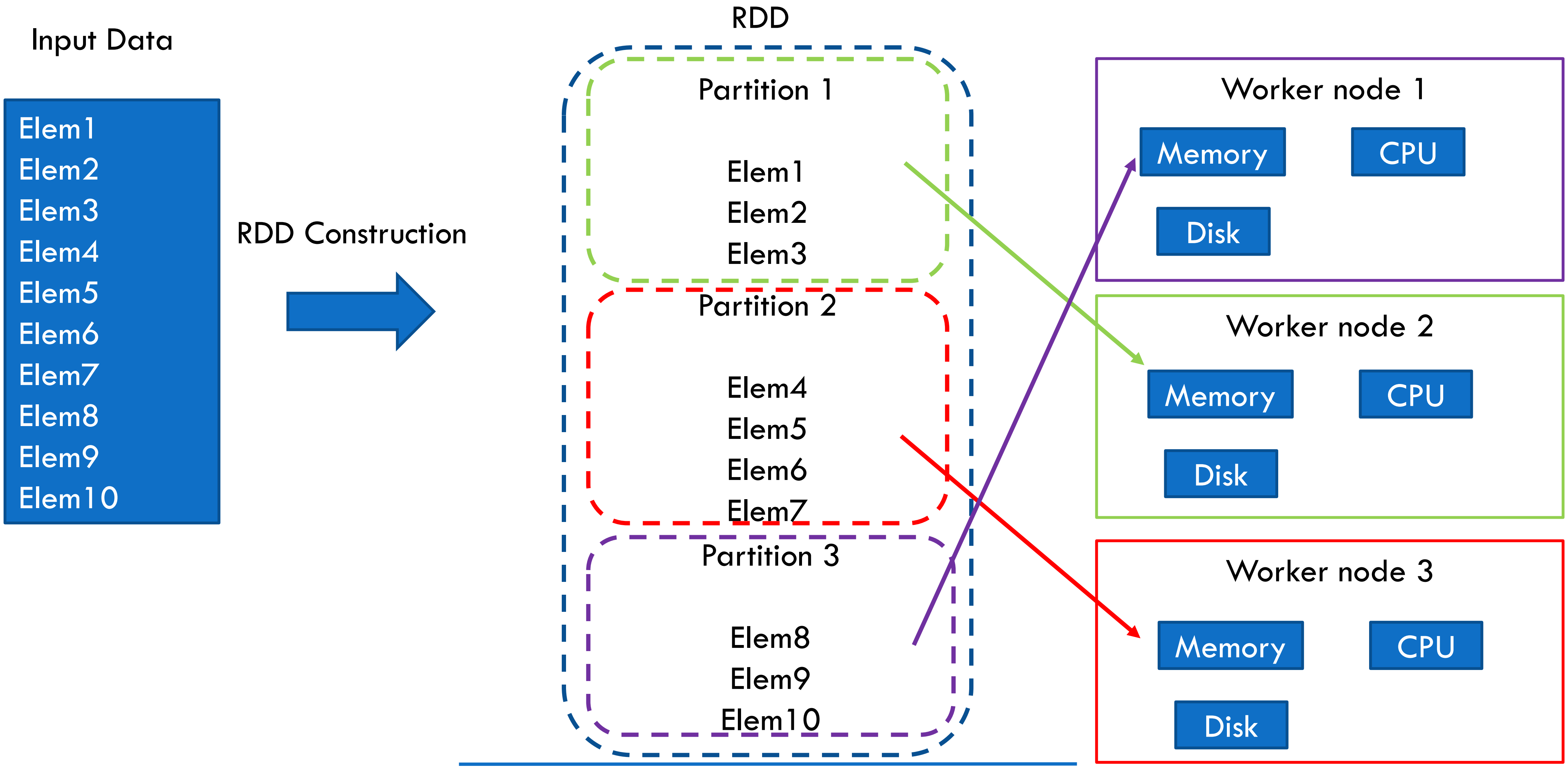
Apache Spark (II)

- Defines a set of operations (transformations and actions) that can be combined at any order
  - For example group by, map, filter, etc.
- Similar to Map-Reduce but based on **RDD (Resilient Distributed Datasets)**
  - Collections of objects distributed across a cluster that **can be manipulated in parallel**
- **Lazy evaluation**: Nothing computed until an action requires it.
- Fault tolerance by means of DAGs (Directed Acyclic Graph)
- Programming APIs in Python, Scala, R, SQL or Java

Resilient Distributed Datasets (RDDs)

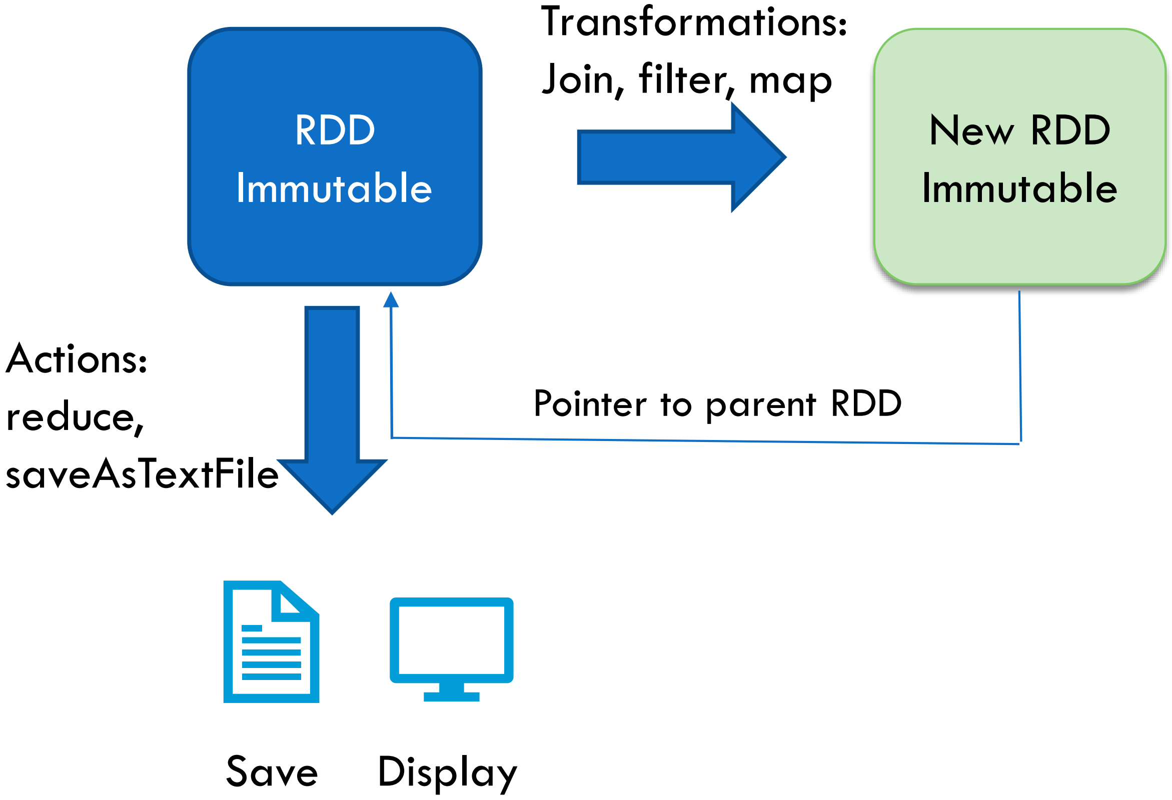
- Collection of partitioned records
  - Immutable → Read-only
  - Operations execute in parallel
- Partitions:
  - By default, equal to the number of CPU cores available in the cluster
  - Each partition reside in a single machine
    - Shuffle operations to move data from one partition to another
  - Stored in memory by default
    - Use of disk if necessary or explicitly defined

Resilient Distributed Datasets (RDDs)



Resilient Distributed Datasets (RDDs)

- ***Transformations:*** build RDDs through deterministic operations on other RDDs
  - Include *map*, *filter*, *join*
  - Lazy operation
- ***Actions:*** return value or export data
  - Include *count*, *collect*, *save*
  - Triggers execution



Working with Spark and Python

- Spark initialization and context creation:

```
from pyspark.sql import SparkSession
APP_NAME = "CAP-lab3"
SPARK_URL = "local[*]"
spark = SparkSession.builder.appName(APP_NAME).master(SPARK_URL).getOrCreate()
sc = spark.sparkContext
```

Loading data

- Creating a Spark RDD
  - Create from list of numbers using **paralellize**

    array = sc.parallelize([1,2,3,4,5,6,7,8,9,10], 2

  - RDDs cannot be printed unless they are reduced (lazy evaluation). The basic reduce operation is **collect**

    print(array.collect())

  - Create from text file using **textFile**

    quijote = sc.textFile("elquijote.txt")

    quijote.take(10)

  - **Take** method gets N first elements of RDD

Transformations

- **map**
  - Applies a function to each element of the RDD and returns a new RDD
  - charsPerLine = quijote.map(lambda s: len(s))
- **flatMap:**
  - Applies a function to each element of the RDD, flattens the results and returns a new RDD
  - allWords = quijote.flatMap(lambda s: s.split())
- **filter**
  - Returns a new RDD containing only the elements that satisfy a predicate.
  - allWordsNoArticles = allWords.filter(lambda a: a.lower() not in ["el", "la"])

Transformations

- **sample:**
  - Samples a fraction of data randomly with or without replacement
  - sampleWords = allWords.sample(withReplacement=True, fraction=0.2, seed=666)
- **union**
- **distinct**

Actions

- **reduce:**
  - Reduces elements of the RDD applying a binary operator
- **collect**
- **take**
- **count**
- **takeOrdered**

Key-Value RDDs

- RDDs indexed by a key. Similar to a Python dict.
- To build K-V RDDs lambda map function must return a tuple:
  - words = allWords.map(lambda e: (e,1))
- Reduce action can be performed over key-grouped values:
  - frequencies = words.reduceByKey(lambda a,b: a+b)

Other operations

- Persistence:
  - Unlike Hadoop, intermediate RDDs are nor preserved and each time we use and action the complete pipeline is executed
  - Use **cache()** method to avoid this
  - By default, only memory-level storage is used. To persist data in disk we must specify it
    - allWords.persist(StorageLevel.MEMORY_AND_DISK)
    - allWords.saveAsTextFile("palabras_parte2")

Spark SQL

- SQL-like Operations over Pandas dataframes
  - show
  - filter
  - select
  - where
  - groupby
- We can execute SQL queries over the dataframe directly:
  - spark.sql("select * from table limit 10").show()
- Pandas dataframes are not totally scalable when using Spark
  - Koalas dataframe library solves this

Assignments

- Execute provided Python notebook on Google Colab
- Answer questions and exercises proposed along the notebook
- Answer the questions in the provided template assignment