UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Sistemas de almacenamiento y procesado distribuido

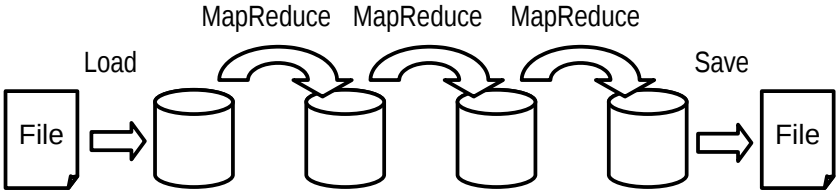
Laboratorio 4

Javier Esparza Peidro – jesparza@dsic.upv.es

Contenido

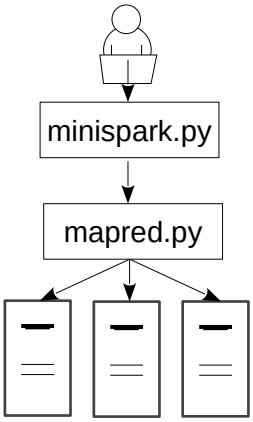
1. Introducción

En este laboratorio se extiende el resultado obtenido en el laboratorio 3, con el objetivo de implementar una versión limitada de Spark, que denominaremos MiniSpark, y que permitirá la ejecución encadenada de múltiples tareas MapReduce.



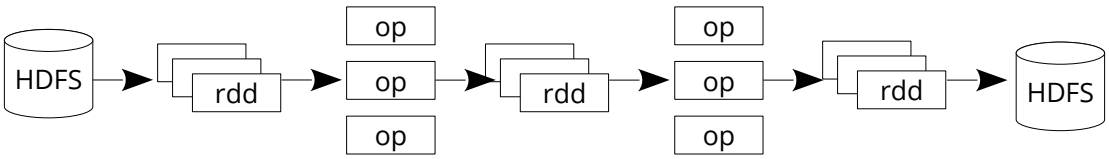
El funcionamiento de MiniSpark será similar al de Spark, primero se cargará un DataSet a partir de uno o varios ficheros, después se aplicarán transformaciones encadenadas, cada una generando un nuevo DataSet. Finalmente, se guardará el último DataSet en uno o más ficheros.

Para ello, MiniSpark se basará en el algoritmo MapReduce que se implementó en el laboratorio 3, dando lugar a la siguiente arquitectura genérica.



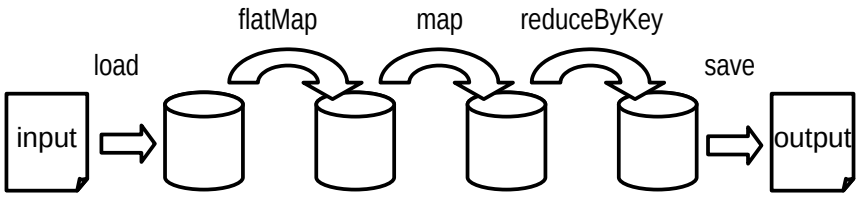
2. MiniSpark

Spark es una plataforma de computación distribuida que permite el procesamiento de grandes volúmenes de datos en paralelo. El núcleo de Spark lo representan los RDDs o los Resilient Distributed Datasets. Se trata de colecciones de datos distribuidas que pueden ser procesadas en paralelo, utilizando para ello diversas transformaciones.



De manera similar, en MiniSpark vamos a implementar el concepto de **DataSet**. Un DataSet es una colección de datos, que inicialmente residirá en la máquina local, y que podrá ser procesada en paralelo, utilizando **transformaciones**. Para ejecutar las transformaciones utilizaremos el algoritmo **MapReduce** implementado en el laboratorio 3.

Para fijar conceptos, veamos un ejemplo. El siguiente diagrama describe cómo podríamos contar las palabras de un texto en Spark:



A continuación se presenta un posible código que ilustra cómo se podría implementar este proceso haciendo uso de MiniSpark desde la perspectiva del usuario final:

```
import minispark
import utils
ds = minispark.load("input")
ds2 = ds.flatMap(utils.split)
ds3 = ds2.map(utils.word)
ds4 = ds3.reduceByKey(utils.sum)
ds4.save("output")
```

Como se puede observar, tras importar el módulo *minispark*, el primer paso consiste en cargar un DataSet a partir de los ficheros incluidos en el directorio “input”. Para ello se utiliza la función minispark.load(path). Una vez se ha creado un primer DataSet, el resto de transformaciones se aplican en cadena, generando nuevos DataSets. El último DataSet se almacena en el directorio “output”.

Una limitación de nuestra implementación será que las funciones de usuario que se utilicen deben estar almacenadas en un fichero externo, es decir, no pueden ser funciones definidas en el propio intéprete interactivo de Python. Por ello, necesitamos del fichero utils.py, que se presenta a continuación:

```
def split(x): return x.split()
def word(x): return (x,1)
def sum(accum, x): return x if accum is None else accum + x
```

3. DataSet

En MiniSpark un DataSet representa una colección de datos de sólo lectura sobre los que se pueden efectuar diversas transformaciones en paralelo. Cada transformación genera un nuevo DataSet.

Internamente, un DataSet contiene una colección de pares (clave, valor) almacenados en uno o varios ficheros. Para simplificar, el/los ficheros que consitutye/n un DataSet se almacenarán bajo un mismo directorio en la máquina local, siguiendo la convención “minispark/datasets/<uuid>”, donde <uuid> representa un identificador único que se genera en el instante de creación de un DataSet.

En los ficheros que constituyen un DataSet, cada dato se almacenará en una línea de texto con el siguiente formato:

<key>\t<value>

Donde <key> será un String y <value> será cualquier valor representado por un String en formato JSON.

Para obtener un primer DataSet es necesario partir de un fichero o de un directorio, y utilizar la función *minispark.load(path)*. Esta función creará un directorio “minispark/datasets/<uuid>”, y copiará en su interior los ficheros de datos originales.

Además, la función *minispark.load(path)* se asegurará de que estos ficheros de datos cumplen el formato <key>\t<value> especificado anteriormente. Si no es así, deberán ser transformados del siguiente modo: se asumirá que cada línea del fichero origen es un valor, y se generará una nueva clave a partir de dicho valor, utilizando el algoritmo de resumen SHA. De este modo dos valores iguales poseerán la misma clave.

Podemos usar la utilidad *toKey()* para ello:

```
import hashlib
def toKey(val):
    hex = hashlib.md5(val.encode()).hexdigest()
    return int(hex, 16)
```

El esqueleto de la función *minispark.load(path)* podría quedar así:

```
import uuid
import os
import json
def load(path):
  # 1. Generate UUID for DataSet
  uid = uuid.uuid4()
  os.makedirs(f"minispark/datasets/{uid}")

  # 2. Copy/Transform files from <path> to minispark/datasets/{uid}
  if os.path.isdir(path):
    files = [os.path.join(path, subpath) for subpath in os.listdir(path)]
  else:
    files = [path]

  # create dataset format
  for file in files:
    fin = open(file, "rt")
    fout = open(f"minispark/datasets/{uid}/{os.path.basename(file)}", "wt")
    for line in fin:
      if len(line.split("\t")) == 2:
        fout.write(line)
      else:
        fout.write(f"{toKey(line)}\t{json.dumps(line)}\n")
    fin.close()
    fout.close()

  # 3. Create DataSet pointing to the correct path
  return DataSet(f"minispark/datasets/{uid}")
```

La clase DataSet contiene todas las operaciones que se pueden efectuar sobre la colección de datos, y pueden ser de dos tipos:

- **Transformaciones**: transforman el DataSet origen y siempre generan un nuevo DataSet destino. Algunos ejemplos son: map(mapper), flatMap(mapper), reduceByKey(reducer), sortByKey(), filter(pred), union(), intersection(), etc.
- **Acciones**: se utilizan para finalizar una cadena de transformaciones, bien para obtener el resultado en memoria, bien para almacenarlo permanentemente en un fichero o directorio. Algunos ejemplos son: collect(n), save(path), reduce(reducer), etc.

Para obtener un listado más completo de todas estas operaciones, así como sus parámetros y funcionamiento se recomienda consultar la API de PySpark.

Incluiremos estas operaciones en la clase DataSet. A continuación se presenta un esqueleto con algunas de ellas:

```
class DataSet:
  def __init__(self, path):
    self.path = path

  # Transformations
  def map(self, mapper): pass
  def flatMap(self, mapper): pass
  def reduceByKey(self, reducer): pass
  ...
  # Actions
  def collect(self, n=None): pass
  def save(self, path): pass
  def reduce(self, reducer): pass
  ...
```

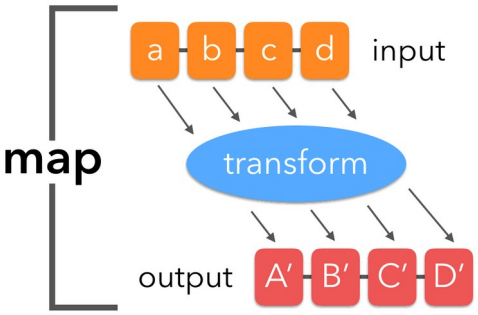
Todas las transformaciones harán uso del algoritmo MapReduce implementado en el laboratorio 3, y que posee el siguiente prototipo:

```
def run(input, output, mapper, reducer, config={})
```

Por lo tanto, para cada transformación será necesario idear un “plan”, que conste de dos fases, una fase Map y una fase Reduce, y que genere los resultados esperados. En las siguientes secciones proponemos un par de ejemplos ilustrativos.

4. La transformación map()

Si consultamos la API de PySpark comprenderemos que la transformación *DataSet.map(mapper)* recibe una función *mapper* como parámetro y genera un nuevo DataSet con todos los datos transformados. Para ello, la función *mapper(data)* es invocada para cada dato del DataSet origen, devolviendo el dato transformado.



Esta funcionalidad encaja bastante bien con la fase Map de nuestro algoritmo MapReduce pero podemos identificar algunas diferencias fundamentales:

1. En el algoritmo MapReduce, el mapper es un script que recibe datos por la entrada estándar y genera resultados por la salida estándar. En MiniSpark el mapper es una función.

2. En el algoritmo MapReduce, los datos generados por el mapper son pares (clave, valor), en MiniSpark los datos de entrada pueden ser de cualquier tipo, y los datos de salida pueden ser de cualquier tipo.

3 En el algoritmo MapReduce, el mapper puede generar múltiples pares (clave, valor) por cada dato de entrada. En MiniSpark la función *mapper(data)* genera un dato de salida (de cualquier tipo) por cada dato de entrada (de cualquier tipo).

Teniendo en cuenta estas diferencias, intentaremos implementar esta transformación usando el algoritmo MapReduce.

MapReduce siempre necesitará dos scripts: el script mapper, que ejecutará en la fase Map, y el script reducer, que ejecutará en la fase Reduce. Según la transformación de MiniSpark que implementemos, estos scripts tendrán un contenido u otro. Proponemos crear un directorio “operations/” y depositar en su interior estos scripts usando la siguiente convención:

- *<T>_map.py*: para el script que implementa el mapper en una transformación T
- *<T>_reduce.py*: para el script que implementa el reducer en una transformación T

Siguiendo estas convenciones tendríamos una estructura de proyecto similar a ésta:

```
lab4/
\_ mapred.py
\_ mapred.json
\_ minispark.py
\_ partition.py
\_ operations/
    \_ map_map.py
    \_ map_reduce.py
    \_ flatMap_map.py
    \_ flatMap_reduce.py
    \_ reduceByKey_map.py
    \_ reduceByKey_reduce.py
    ...
```

Ahora nos centraremos en diseñar el contenido de los ficheros *map_map.py* y *map_reduce.py*, necesarios para implementar la transformación *DataSet.map(mapper)*. Empezamos por el fichero *map_map.py*, que se utilizará en la fase Map del algoritmo MapReduce. Este script recibirá cada dato en una línea de la entrada estándar y deberá imprimir el dato transformado por la función *mapper* definida por el usuario en la salida estándar. Por tanto, deberá conocer la función *mapper* definida por el usuario, pero es dinámica y diferente en cada invocación a *DataSet.map(mapper)*. Debemos por tanto generar distintos scripts *map_map.py* para cada invocación a *DataSet.map(mapper)*, que contengan la definición del *mapper* definido por el usuario y su invocación.

Esto en general pasará en todas las transformaciones, y para las dos fases Map y Reduce. Proponemos por tanto generar estos “scripts personalizados” en un directorio “minispark/operations/<uid>” donde <uid> representa un identificador único que asignaremos a

cada transformación invocada sobre un DataSet. Los “scripts personalizados” se generarán a partir de las “plantillas” que se ubican en el directorio “operations/”. Estas plantillas incluirán algunas “marcas genéricas” que deberán ser sustituídas por las funciones especificas definidas por el usuario.

Por ejemplo, una primera implementación para la plantilla “*operations/map_map.py”* podría ser la siguiente:

```
#!/bin/python3
import sys
import hashlib
import json

def toKey(val):
  hex = hashlib.md5(val.encode()).hexdigest()
  return int(hex, 16)

$MAPPER_DEF

for line in sys.stdin:
  key, value = line.split("\t")
  value = json.loads(value)

  data = $MAPPER(value)

  if type(data) == tuple:
    key = data[0]
    value = json.dumps(data[1])
  else:
    value = json.dumps(data)
    key = toKey(value)

  print(f"{key}\t{value}")
```

En este script es posible observar dos identificadores especiales *$MAPPER* y *$MAPPER_DEF*, que representan huecos que deberán ser rellenados por el nombre del mapper del usuario y su definición respectivamente. Como se puede observar, siempre asumiremos que los datos de entrada al DataSet poseen la sintaxis <key>\t<value> especificada anteriormente, y se generan datos con exactamente la misma sintaxis.

Una posible implementación para el script “operations/map_reduce.py” podría ser como sigue:

```
#!/bin/python3
import sys

for line in sys.stdin:
    print(line, end="")
```

En este caso, el trabajo a hacer se limita a volcar por la salida estándar lo mismo que se recibe por la entrada, ya que la transformación *DataSet.map(mapper)* no acumula ni agrega ningún tipo de resultado.

Por último, faltaría implementar el método *DataSet.map(mapper)* que dá sentido a todo el proceso. A continuación se presenta una posible implementación:

```
def map(self, mapper):
  # 1. Generate UUID for operation
  opUid = uuid.uuid4()
  os.makedirs(f"minispark/operations/{opUid}")

  # 2. Generate specific mapper from template
  fin = open("operations/map_map.py", "rt")
  data = fin.read()
  fin.close()
  data = data.replace("$MAPPER_DEF", inspect.getsource(mapper))
  data = data.replace("$MAPPER", mapper.__name__)
  fout = open(f"minispark/operations/{opUid}/mapper.py", "wt")
  fout.write(data)
  fout.close()

  # 3. Generate specific reducer from template
  fin = open("operations/map_reduce.py", "rt")
  data = fin.read()
  fin.close()
  fout = open(f"minispark/operations/{opUid}/reducer.py", "wt")
  fout.write(data)
  fout.close()

  # 4. Run MapReduce
  # - Generate UUID for resulting dataset
  dsUid = uuid.uuid4()

  mapred.run(
    self.path,
    f"minispark/datasets/{dsUid}",
    f"minispark/operations/{opUid}/mapper.py",
    f"minispark/operations/{opUid}/reducer.py"
  )
  # 5. Return new DataSet
  return DataSet(f"minispark/datasets/{dsUid}")
```

Como puede observarse en el código, los identificadores especiales *$MAPPER* y *$MAPPER_DEF* son reemplazados por el nombre del mapper y su definición respectivamente. Para la obtención de la definición de la función se utiliza la función *inspect.getsource()*. Hay que tener en cuenta que esta estrategia **sólo funcionará si la definición de la función se incluye en un fichero .py**, es decir, no funcionaría con una función lambda o una función definida directamente en el intérprete interactivo de Python.

5. La transformación reduceByKey()

En el caso de la transformación *DataSet.reduceByKey(reducer)* cobra mayor relevancia la fase de Reduce, en lugar de la fase Map.

Proponemos el siguiente ejemplo de código para la plantilla “operations/reduceByKey_map.py”:

```
#!/bin/python3
import sys
for line in sys.stdin:
    print(line, end="")
```

El siguiente código podría funcionar para la plantilla “operations/reduceByKey_reduce.py”:

```
#!/bin/python3
import sys
import hashlib
import json

$REDUCER_DEF

curr_key = None
curr_accum = None

for line in sys.stdin:
    key, value = line.split("\t")
    value = json.loads(value)
    if key == curr_key:
        curr_accum = $REDUCER(curr_accum, value)
    else:
        if curr_key:
            print(f"{curr_key}\t{json.dumps(curr_accum)}")
        curr_key = key
        curr_accum = $REDUCER(None, value)

if curr_key:
    print(f"{curr_key}\t{json.dumps(curr_accum)}")
```

Finalmente, a continuación se propone una posible implementación para el método *DataSet.reduceByKey(reducer)*:

```
def reduceByKey(self, reducer):
  # Generate UUID for operation
  opUid = uuid.uuid4()
  os.makedirs(f"minispark/operations/{opUid}")

  # 1. Generate mapper
  fin = open("operations/reduceByKey_map.py", "rt")
  data = fin.read()
  fin.close()
  fout = open(f"minispark/operations/{opUid}/mapper.py", "wt")
  fout.write(data)
  fout.close()

  # 2. Generate reducer
  fin = open("operations/reduceByKey_reduce.py", "rt")
  data = fin.read()
  fin.close()
  data = data.replace("$REDUCER_DEF", inspect.getsource(reducer))
  data = data.replace("$REDUCER", reducer.__name__)
  fout = open(f"minispark/operations/{opUid}/reducer.py", "wt")
  fout.write(data)
  fout.close()

  # 3. Run MapReduce
  # - Generate UUID for resulting dataset
  dsUid = uuid.uuid4()
```

```
    mapred.run(
        self.path,
        f"minispark/datasets/{dsUid}",
        f"minispark/operations/{opUid}/mapper.py",
        f"minispark/operations/{opUid}/reducer.py"
    )

    return DataSet(f"minispark/datasets/{dsUid}")
```

6. Trabajo a realizar

El resto del laboratorio consiste en implementar las operaciones de la clase DataSet. Se puede utilizar como guía la API de la clase RDD en PySpark. Como ya hemos visto, en nuestras implementaciones es necesario idear un plan para que las transformaciones se implementen utilizando el algoritmo MapReduce.

Podría suceder que al aplicar ciertas transformaciones encadenadas el resultado no fuera el esperado. Será necesario identificar estas situaciones. En general esto se puede resolver modificando ligeramente el algoritmo MapReduce implementado en el laboratorio 3. Recordar que la función run() posee un diccionario con opciones, que podríamos utilizar para variar ligeramente su ejecución.