

Procesamiento Masivo de datos: Hadoop II

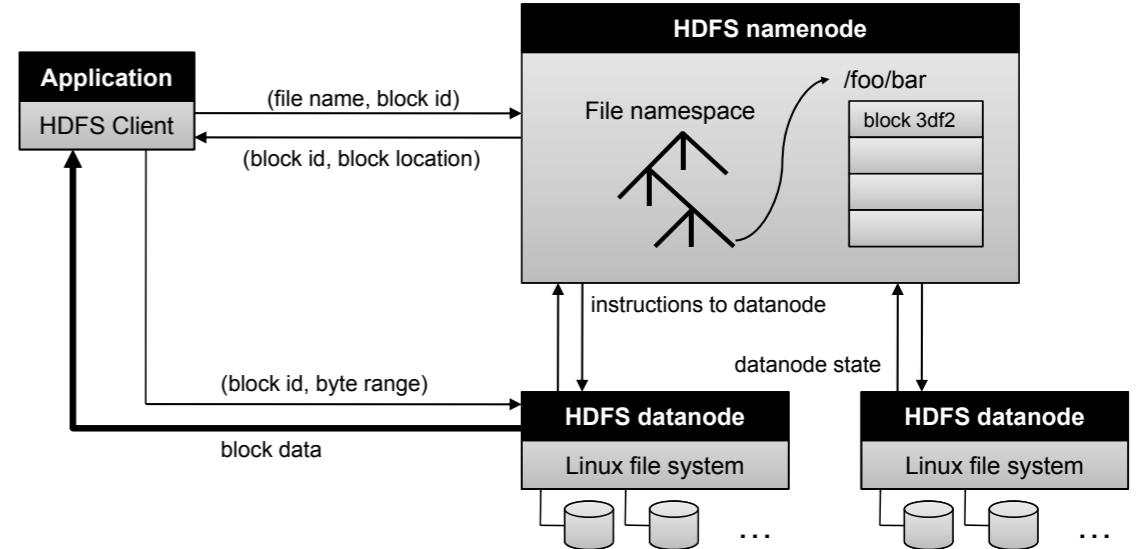
Alex Di Genova

16/10/2023

Hadoop

HDFS

- HDFS proporciona almacenamiento redundante para big data al almacenar los datos en un grupo de computadoras.
- HDFS es una capa de software sobre un sistema de archivos nativo como ext4 o xfs.
- Los archivos HDFS se dividen en bloques, generalmente de 64 MB o 128 MB.
- El tamaño del bloque es la cantidad mínima de datos que se pueden leer o escribir en HDFS.
- Los bloques permiten dividir archivos muy grandes y distribuirlos a muchas máquinas en tiempo de ejecución. Se almacenan diferentes bloques del mismo archivo en diferentes máquinas para proporcionar un procesamiento distribuido más eficiente.
- Los bloques se replican en los DataNodes (3x). Por lo tanto, cada bloque existe en tres máquinas diferentes y tres discos diferentes, y si fallan incluso dos nodos, los datos no se perderán.
 - Almacenamiento en clúster / 3
- El **NameNode** maestro almacena qué bloques componen un archivo y dónde se encuentran esos bloques. El **NameNode** se comunica con los **DataNodes**, los nodos que realmente contienen los bloques en el clúster. Los metadatos asociados con cada archivo se almacenan en la memoria del **NameNode** para realizar búsquedas rápidas.



- hadoop fs -help
- hadoop fs –copyFromLocal kmers.txt kmers.txt
- hadoop fs -mkdir text_db
- hadoop fs –ls .
- hadoop fs –cat kmers.txt | less
- hadoop fs –tail shakespeare.txt | less
- hadoop fs –get kmers.txt ./kmers.from-remote.txt
- hadoop fs –chmod 664 kmers.txt
- hadoop fs -test <ruta>
- hadoop fs -count <ruta>
- hadoop fs -du -h <ruta>

Hadoop

MapReduce: Un modelo de programación funcional

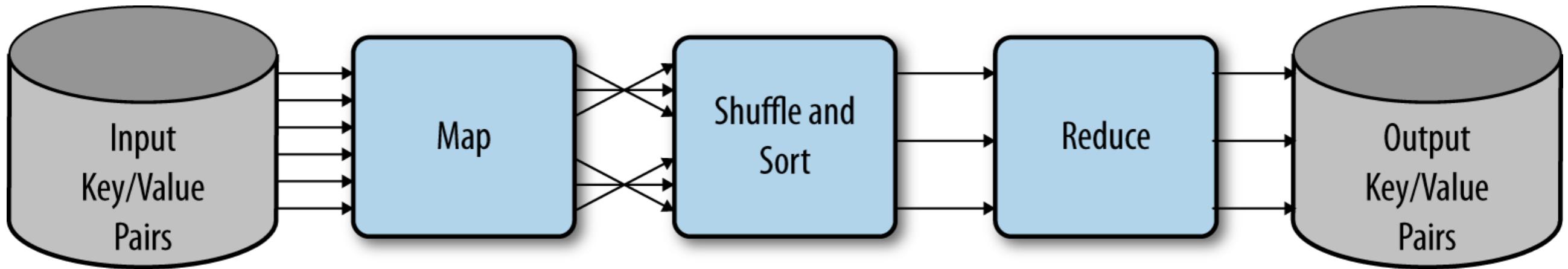
- MapReduce es un marco computacional simple pero muy potente diseñado específicamente **para habilitar el cómputo distribuido (tolerante a fallas) en un grupo de máquinas administradas centralmente.**
 - Idea: múltiples tareas independientes ejecutan una función en fragmentos locales de datos y agregan los resultados después del procesamiento.
 - La programación funcional es un estilo de programación que garantiza las funciones dependen solo de sus entradas, y están cerradas y no comparten estado. Los datos se transfieren entre funciones enviando la salida de una función como entrada a otra función totalmente independiente.
 - Los datos que se operan como entrada y salida en estas funciones no son utilizan pares de "clave/valor" para coordinar el cálculo.

```
def map(key, value):  
    # Perform processing  
    return (intermed_key, intermed_value)
```

```
def reduce(intermed_key, values):  
    # Perform processing  
    return (key, output)
```

Hadoop

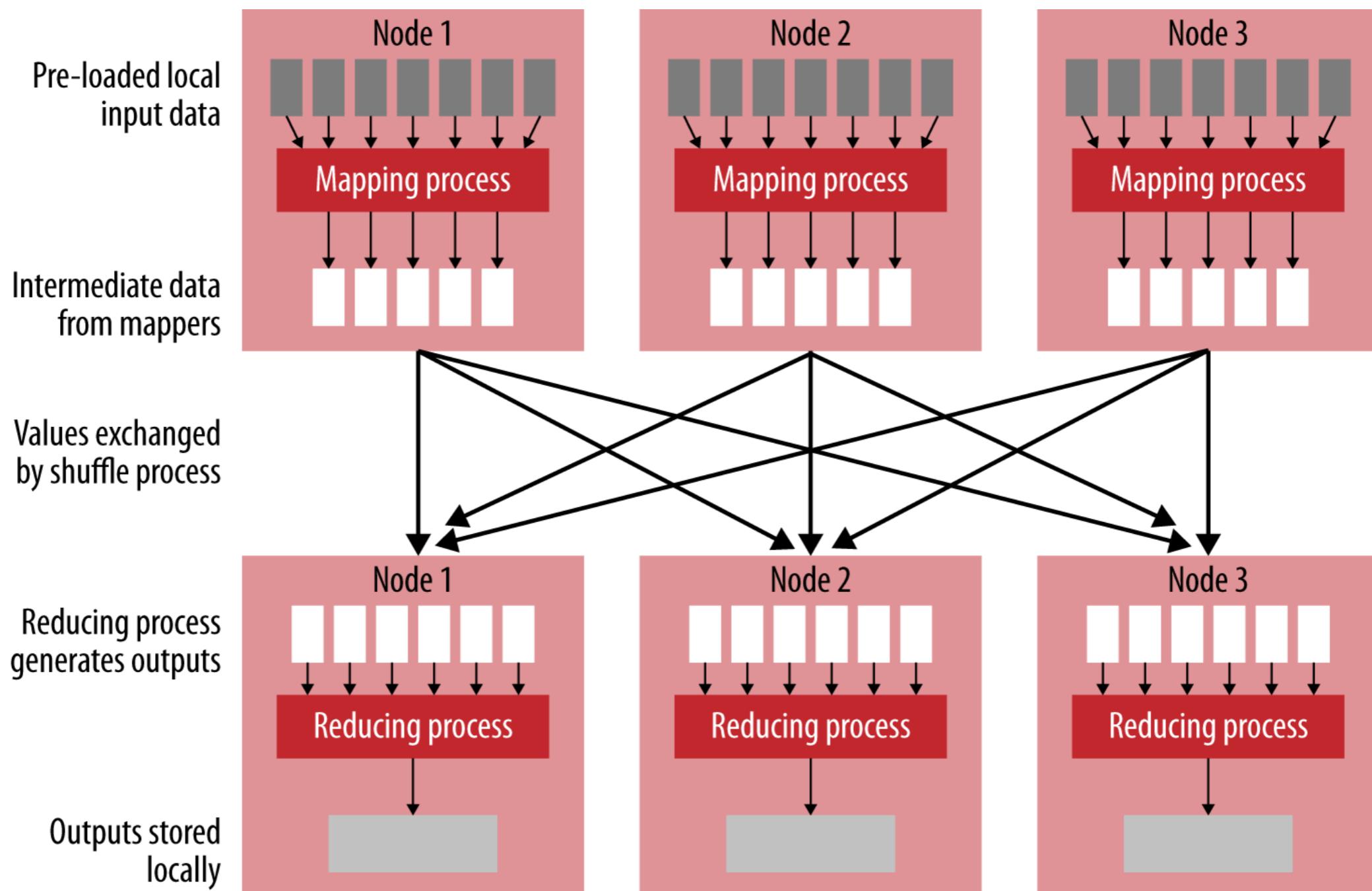
MapReduce: Un modelo de programación funcional



- 1.Los datos locales se cargan en un proceso de mapeo como pares clave/valor de HDFS.
- 2.Los mapeadores generan cero o más pares clave/valor, asignando valores calculados a una clave en particular.
- 3.Luego, estos pares se ordenan/barajan en función de la clave y luego se pasan a un reductor de modo que todos los valores de una clave estén disponibles.
- 4.Los reductores deben generar cero o más pares clave/valor finales, que son la salida.

Hadoop

Map/Reduce



Map/Reduce

- Mapper.py
 - Archivo input.txt
 - 1
 - 2
 - 3
 - 4
 - 5
 - Emitiendo pares clave-valor, donde la clave es el valor y el valor es 1.
- ```
#!/usr/bin/env python
import sys
current_value = None
current_count = 0
for line in sys.stdin:
 line = line.strip()
 value, count = line.split("\t")
 value = int(value)
 count = int(count)
 if current_value == value:
 current_count += count
 else:
 if current_value is not None:
 print(f"{current_value}\t{current_count}")
 current_value = value
 current_count = count
if current_value is not None:
 print(f"{current_value}\t{current_count}")
```
- Reducer suma los valores con la misma clave.
- ```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-* .jar \
-file mapper.py -mapper mapper.py \
-file reducer.py -reducer reducer.py \
-input input.txt -output output
```

Hadoop

Map/Reduce

```
def map(dockey, line):
    for word in value.split():
        emit(word, 1)

def reduce(word, values):
    count = sum(value for value in
values)
    emit(word, count)
```

```
# Input to WordCount mappers
(27183, "The fast cat wears no hat.")
(31416, "The cat in the hat ran fast.")

# Mapper 1 output
("The", 1), ("fast", 1), ("cat", 1), ("wears", 1),
("no", 1), ("hat", 1), (".", 1)

# Mapper 2 output
("The", 1), ("cat", 1), ("in", 1), ("the", 1),
("hat", 1), ("ran", 1), ("fast", 1), (".", 1)

# Input to WordCount reducers
# This data was computed by shuffle and sort

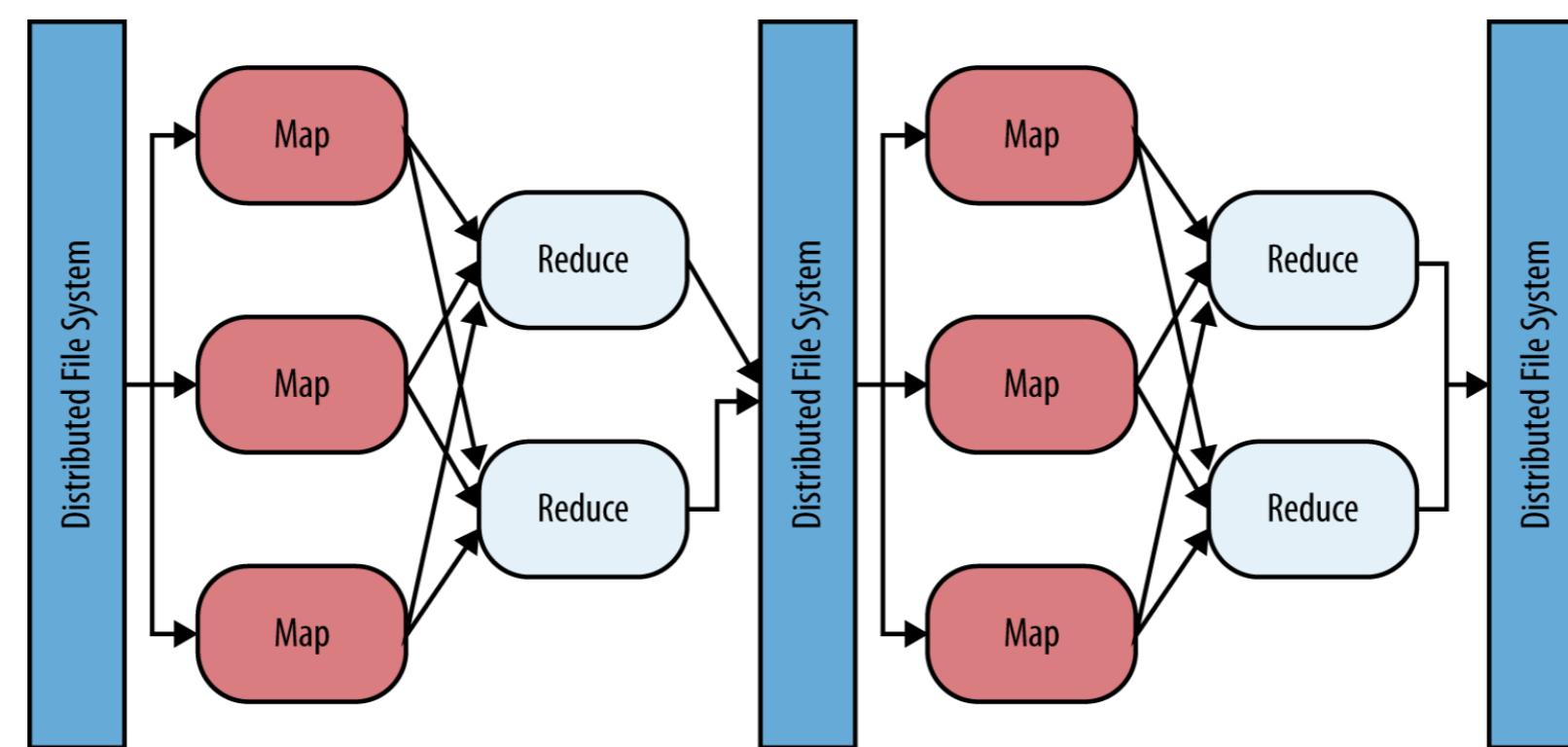
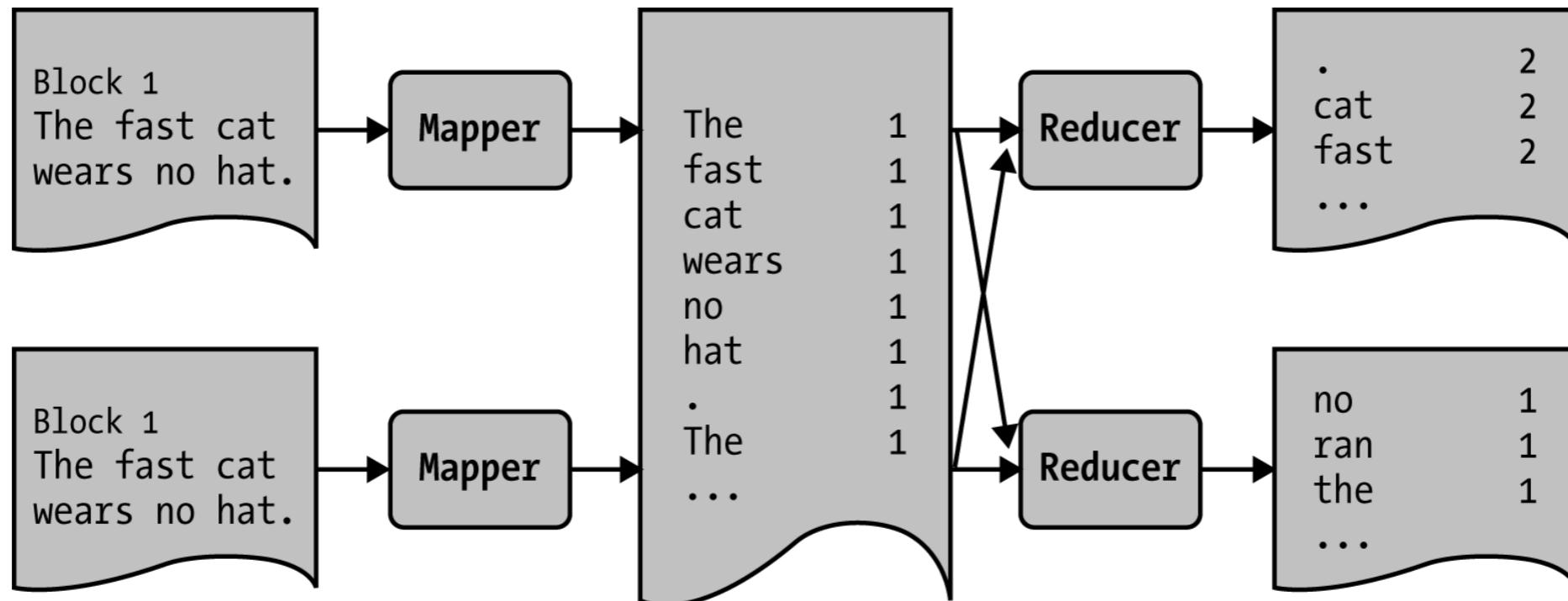
(".", [1, 1])
("cat", [1, 1])
("fast", [1, 1])
("hat", [1, 1])
("in", [1])
("no", [1])
("ran", [1])
("the", [1])
("wears", [1])
("The", [1, 1])

# Output by all WordCount reducers

(".", 2)
("cat", 2)
("fast", 2)
("hat", 2)
("in", 1)
("no", 1)
```

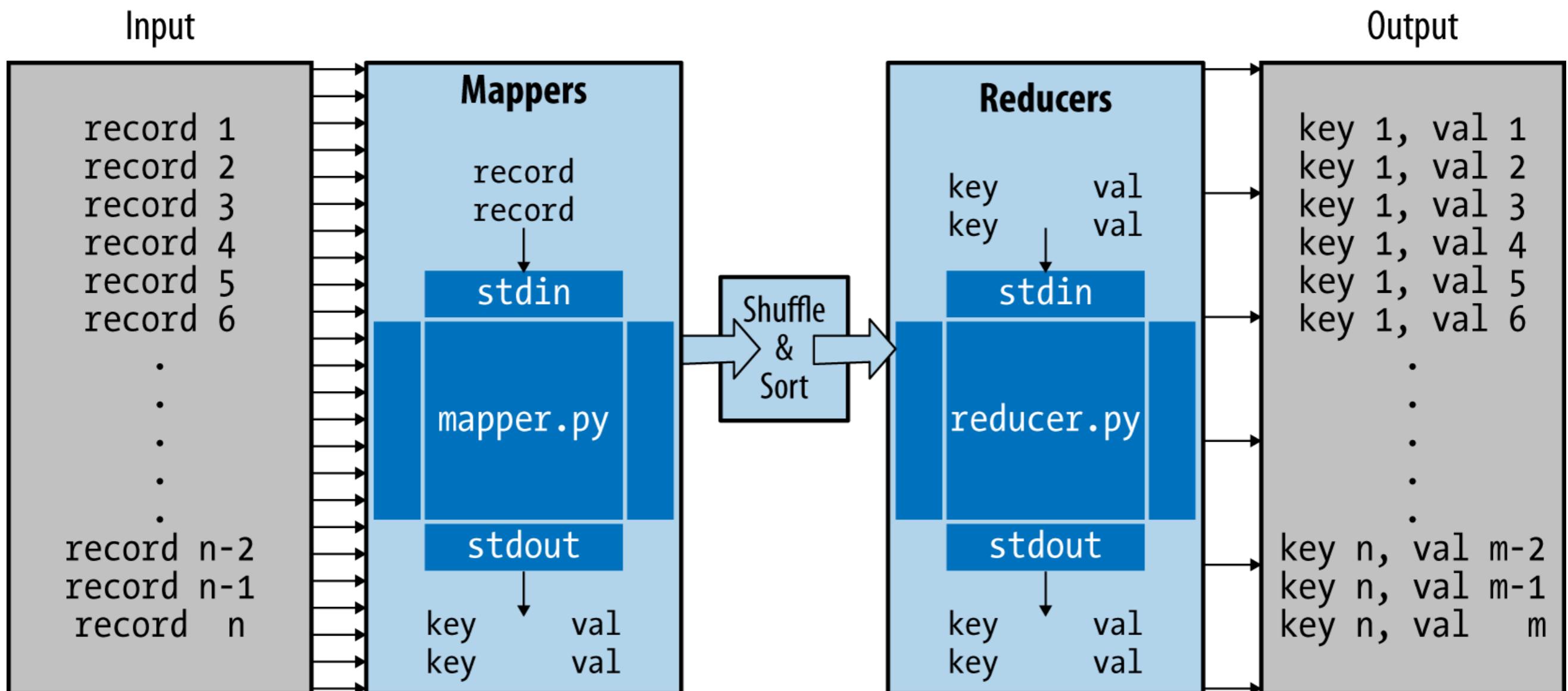
Hadoop

Map/Reduce



Hadoop streaming

Map/Reduce



Google Colab

The screenshot shows a Google Colab notebook titled "Rust-C-basics.ipynb". The notebook contains two sections: "Hola Mundo" and "Arreglos y funciones".

Hola Mundo:

```
[ ] 1 !apt install rustc cargo
[ ] 1 @@writefile helloworld.rs
2 // This is a comment, and is ignored by the compiler
3
4 // This is the main function
5 fn main() {
6     // Statements here are executed when the compiled binary is called
7     // Print text to the console
8     println!("Hola Mundo!");
9 }
10

Writing helloworld.rs

[ ] 1 !rustc /content/helloworld.rs
2 ./helloworld

[ ] Hola Mundo!
```

Arreglos y funciones:

```
[ ] 1 @@writefile arreglos.rs
2
3 use std::mem;
4
5 // This function borrows a slice
6 fn analyze_slice(slice: &[i32]) {
7     println!("primer elemento del arreglo: {}", slice[0]);
8     println!("el ultimo elemento es: {}", slice[slice.len()-1]);
9     println!("el arreglo tiene {} elementos", slice.len());
10 }

11
12 fn main() {
13     // arreglo de largo 5 fijo
14     let xs: [i32; 5] = [1, 2, 3, 4, 5];
15     // arreglo de largo 500 inicializado en 0
16     let ys: [i32; 500] = [0; 500];
17
18     // indices comienzan en 0
19     println!("primer elemento arreglo: {}", xs[0]);
20     println!("tercer elemento del arreglo: {}", xs[2]);
21
22     // `len` retorna el largo del arreglo
23     println!("numero de elementos arreglo xs: {}", xs.len());
24     println!("numero de elementos arreglo ys: {}", ys.len());
25
26     // Arrays are stack allocated
27     println!("tamaño en memoria xs {} bytes", mem::size_of_val(&xs));
28     println!("tamaño en memoria ys {} bytes", mem::size_of_val(&ys));
29     // Arrays can be automatically borrowed as slices
30     println!("pasamos el arreglo por puntero a función");
31     analyze_slice(&xs);
32     analyze_slice(&ys);
33
34     println!("pasamos una slice del array");
35     analyze_slice(&ys[1 .. 4]);
36
37     // Se puede acceder a las matrices de forma segura mediante `.get`,
38     // que devuelve un
39     // `Opción`. Esto se puede combinar como se muestra a continuación, o se puede usar con
40     // `.expect()` si desea que el programa finalice con un buen
41     // mensaje en lugar de continuar.
42     for i in 0..xs.len() + 1 { // iteramos un elemento mas del largo
43         match xs.get(i) {
44             Some(xval) => println!("{}: {}", i, xval),
45             None => println!("Fuera de indice! {} no existe!", i),
46         }
47     }
48 }
```

<https://github.com/adigenova/uohpmd/blob/main/code/HadoopII.ipynb>

Consultas?

Consultas o comentarios?

Muchas gracias