MapReduce Primitivo, Procesamiento Distribuido y Paralelo

Competencias

- Comprenderá la necesidad de procesar datasets de forma paralela.
- Comprenderá las limitantes del procesamiento paralelo.
- Comprenderá las diferencias del procesamiento local versus distribuido.

Motivación

Hasta el momento, tenemos conocimiento sobre cómo operar con scripts de Python desde la nube mediante los servicios AWS ElasticMapReduce y AWS S3. Si bien podemos trabajar con ellos de manera directa sin necesidad de implementar más pasos, deseamos sacar el máximo provecho posible a nuestras instancias de trabajo. Para ello, en esta lectura aprenderemos sobre el procesamiento paralelo en el contexto de Big Data. Mediante la paralelización y parcelización de nuestro trabajo en elementos, lograremos hacer un uso más eficaz de nuestros recursos computacionales.

Una vez que tengamos definidos los elementos básicos de cómo funciona el paralelismo, estaremos en pie de interiorizarnos en el ecosistema Hadoop y cómo éste logra parcelar archivos. El procesamiento paralelo es considerado uno de los métodos más costo-eficiente para la implementación de solución con grandes cantidades de datos.

La idea seminal del procesamiento en paralelo y distribuido

La idea seminal detrás del procesamiento paralelo y el distribuído es el principio de divide y vencerás. Supongamos que estamos a cargo de una central de abastecimiento de frutas que provee de frutas a todo el país. Nuestro objetivo es contar el stock existente entre todas las frutas que acaban de llegar en el día. Un segundo objetivo es que la tarea debe ser terminada el mismo día, para evitar que se acumulen frutas a contar. En sí, la tarea es trivial: **necesitamos tener una persona que se dedique a contar todas las frutas que han llegado**.

Un elemento a considerar es que la eficiencia y eficacia en la ejecución será condicional al tamaño de la tarea a procesar. Así, si estamos ante la tarea de contar 100.000 frutas y clasificarlas por su tipo, probablemente no alcanzaremos a completar la tarea en el día. Esto nos obligará a tener una sobrecarga en el trabajador, así como aumentar las chances de saturación y error en el proceso. La solución alternativa, que es dividir el proceso por roles, se ejemplifica en la siguiente imágen:

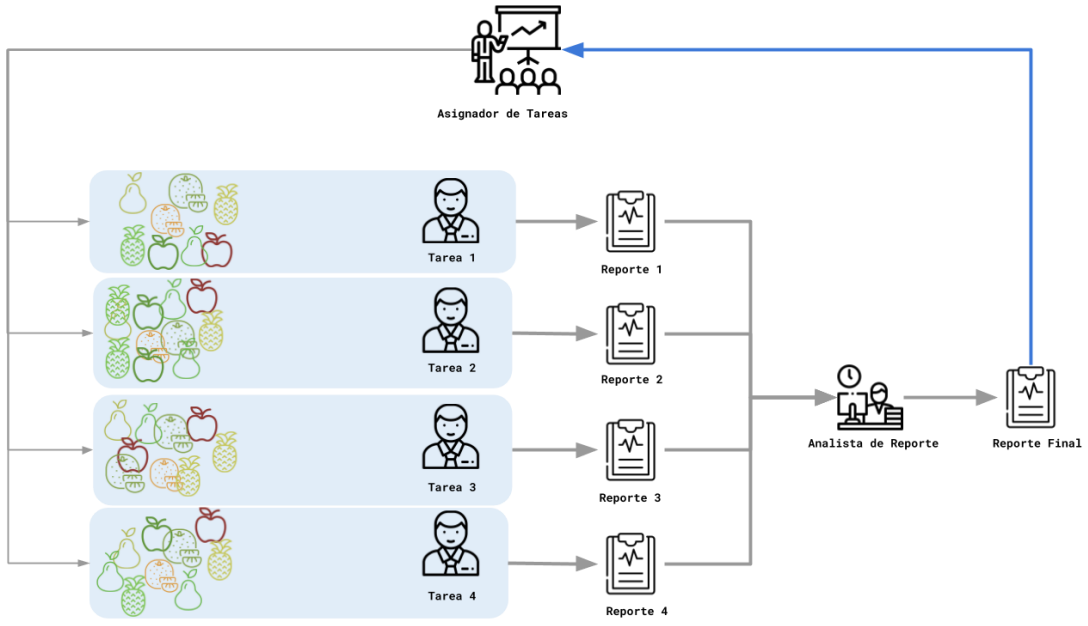

Imagen 1. Distribución por roles.

El proceso de **paralelizar** la tarea parte por la definición de qué es lo que necesitamos, tanto como resultado final así como en cantidad de recursos asignados. Este paso se encapsula en el "asignador de tareas", quien es el responsable de definir el objetivo y cuántas personas estarán a cargo. El segundo paso es la parcelación de todos los datos de entrada. En este caso, vamos a dividir toda la fruta ingresada a la central de abastecimiento en partes iguales, y asignarle a ésta un trabajador que estará encargado de ejecutar la tarea asignada. Este proceso se va a repetir para cada parcela remanente de fruta.

Un aspecto que está implícito en este proceso es el hecho que **todos los trabajadores saben cuál es la tarea y qué hay que devolver**. Este punto es importante de destacar, dado que si uno de los trabajadores no está claro en la tarea, el resultado no será concordante con el de sus compañeros.

Cada trabajador entregará un informe intermedio de toda la fruta que contó en su parcela de frutas. Este informe se pasa al analista de reporte, quien concatenará el resultado de todos los informes intermedios y devolverá el resultado final al asignador de tareas.

Definición del procesamiento paralelo

Por procesamiento paralelo hacemos referencia a la capacidad de un computador de ejecutar tareas dentro de sus procesadores de manera simultánea. Un aspecto importante es el hecho que los sistemas que prestan apoyo en procesamiento paralelo es cómo se implementa la memoria RAM. La opción más común es dentro de un computador con una cantidad finita de procesadores, estos compartirán el acceso a la memoria. Esta opción se conoce como **shared memory systems**. Otra opción es que dado una cantidad finita de procesadores, la memoria RAM se particiona en cada uno de estos. Este proceso se conoce como **distributed memory system**.

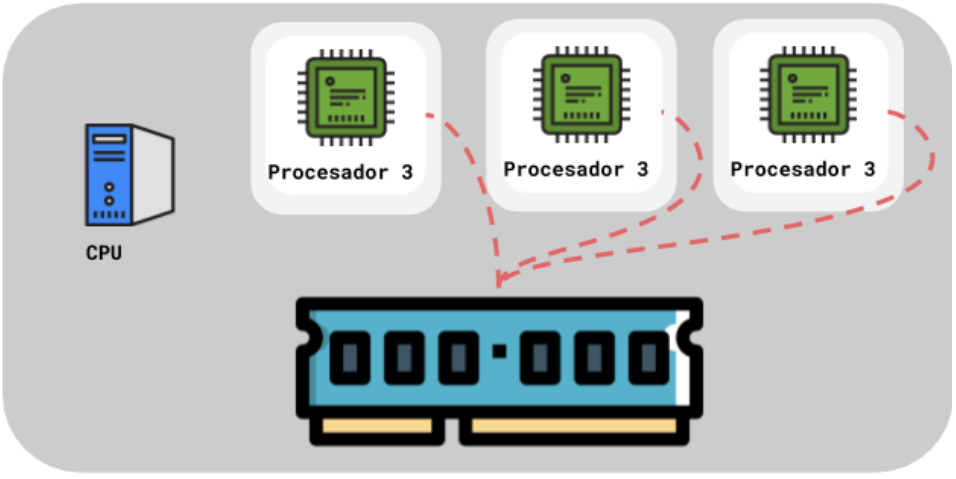

Imagen 2. Procesamiento paralelo.

Definición del procesamiento distribuído

La otra variante de nuestra estrategia de dividir y vencer, es el procesamiento distribuído. La principal virtud y diferencia del procesamiento distribuído es el hecho que tendremos múltiples computadores a nuestra disposición, con los cuales podremos dividir una misma tarea en distintos bloques y distribuirla a lo largo de los computadores disponibles. Esta es una diferencia substancial al procesamiento paralelo, donde no dividimos y asignamos la tarea a cada procesador, pero si nos aseguramos de correr múltiples tareas en paralelo.

Todos los computadores dependerá de un centro que definirá la tarea a dividir. Un aspecto clave es que los computadores pueden enviar y recibir mensajes entre sí, manteniendo la coordinación.

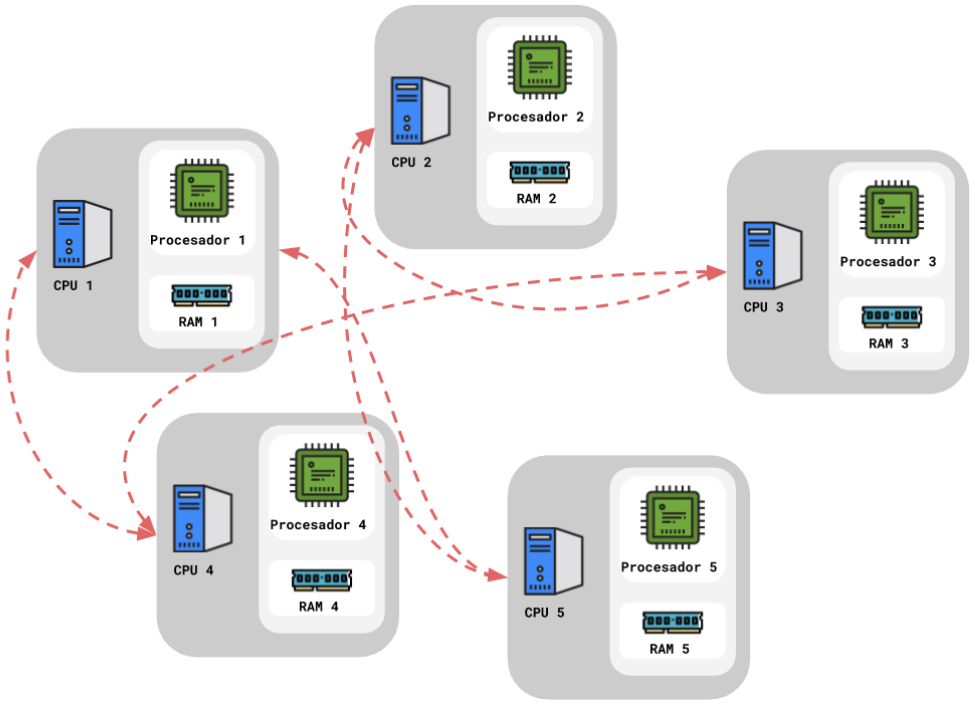

Imagen 3. Procesamiento distribuido.

Una de las principales ventajas del procesamiento distribuído es el hecho que permite generar soluciones escalables en la medida que nuestros objetivos se vuelvan más demandantes en recursos. Dado que estamos utilizando múltiples computadores, facilita el término de tareas de manera eficiente.

El procesamiento en paralelo en el contexto de Big Data

El procesamiento paralelo precede al surgimiento de las soluciones Big Data. De hecho, existe un código escrito en SQL que se ha ejecutado en paralelo por más de 20 años. Resulta que la necesidad por generar flujos de trabajo explícitamente paralelos tiene que ver con las limitaciones de SQL cuando las cantidades de datos son enormes, y la variedad de datos es tal que es imposible venir con un schema predefinido sobre cómo deben estar contenidos los datos.

En las siguientes unidades conoceremos sobre la principal variante de paralelismo implementada en Big Data: Paralelismo en las bases de datos. Más que estar interesados en parcelar el algoritmo de procesamiento en múltiples tareas, el objetivo es parcelar las bases de datos en partes pequeñas que hagan más agradable el procesamiento por el algoritmo.

Conteo de palabras

Probablemente el ejemplo más ubicuo en Big Data tiene que ser el conteo de ocurrencia de palabras en un texto. Este ejemplo es el equivalente a `print("Hello World")` en otros lenguajes. Partamos por generar nuestra primera interacción.

Para ello, vamos a descargar el libro Anna Karenina de Leon Tolstoi desde la página Gutenberg project. Esto lo vamos a lograr mediante el comando `wget` en la línea de comando. Este es un comando que permite descargar archivos desde la Web. La sintaxis a implementar se ejemplifica en la siguiente imagen:

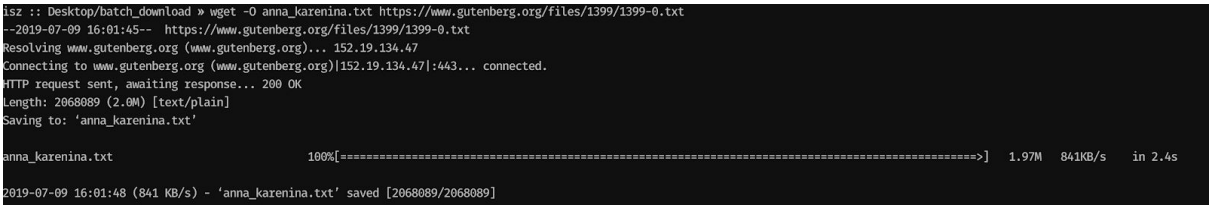

Imagen 4. Descarga del libro Anna Karenina.

Para mantener más ordenado nuestro proyecto, vamos a darle un nombre a la descarga del libro con la opción `-O anna_karenina.txt`. Si no especificamos esto, la descarga va a tomar el nombre del archivo alojado en el servidor. Una vez que la descarga termine, el documento se encontrará en la carpeta del proyecto. Podemos inspeccionar las primeras líneas del texto con el comando `head`.

```
isz :: Desktop/batch_download » head anna_karenina.txt

The Project Gutenberg EBook of Anna Karenina, by Leo Tolstoy

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org


Title: Anna Karenina
```

Imagen 5. Inspeccionado las primeras líneas del texto.

Nuestra primera iteración del conteo de palabras se realizará con un script simple de Python que alojaremos en la misma ruta donde se encuentra alojado nuestro texto a analizar. El script llamado `non_distributed.py` se detalla a continuación.

```
#non_distributed.py

#importamos las librerías de expresiones regulares y de sistema
import re, sys

# generamos un diccionario vacío donde las palabras serán las llaves
# y los valores serán la cantidad de ocurrencias en el texto
word_count_dict = {}

# vamos a leer anna_karenina.txt mediante el standard input
feed_document = sys.stdin

# para cada una de las líneas en anna_karenina.txt
for line_in_document in feed_document:
   # implementaremos una expresión regular que identifique todos
aquellos caracteres no alfanuméricos
   # y los reemplazaremos con un espacio vacío
   line_in_document = re.sub(r'^\W+|\W+$','', line_in_document)
   # Posterior a la limpieza de los caracteres no alfanuméricos, vamos a
separar cada palabra en la línea
   # devolviendo una lista de palabras
   words_in_line = re.split(r'\W+', line_in_document)

   # para cada palabra en nuestra lista de palabras
   for word in words_in_line:
       # vamos a convertir todas las palabras a minúsculas
       # para evitar que Python considere dos versiones de la misma
palabra como distintas
       word = word.lower()
       # Finalmente, vamos a ingresar la palabra como llave en el
diccionario
       # y asignaremos un 1 en su ocurrencia
       word_count_dict[word] = word_count_dict.get(word,0) + 1
   # imprimimos el resultado intermedio a nivel de línea
   print(word_count_dict)
# imprimimos el resultado final
print(word_count_dict)
```

El objetivo del script va a ser un tanto exhaustivo, con una complejidad algorítmica On^2, dado que recorrerá cada palabra dentro de cada línea del texto. Podemos ejecutar nuestro script de la siguiente manera:

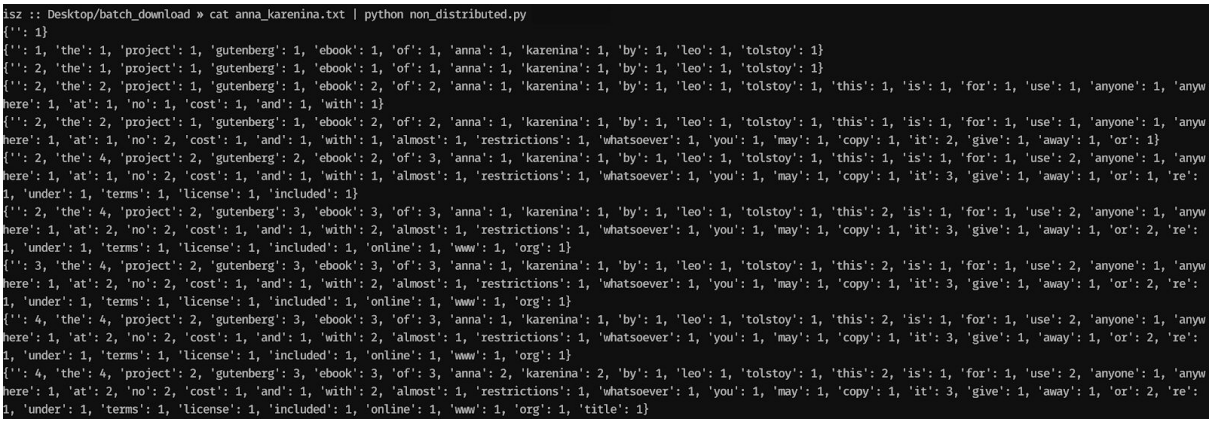

Imagen 6. Ejecutando script.

Donde mediante el comando `cat anna_karenina.txt` el sistema leerá todo el contenido del archivo. Posterior a este paso, vamos a ingresarlo dentro de un flujo con el comando |. Este comando también se conoce como pipe y permite concatenar acciones entre distintos comandos de la línea de comando. Dado que nuestro script `non_distributed.py` contiene la línea `feed_document = sys.stdin`, se pasará todo el contenido leído por el standard input de la consola a la variable `feed_document`. Al ejecutar el script, observamos que éste va evaluando por línea del texto y actualizando nuestro diccionario final. Por ejemplo, se observa la cantidad de veces que se menciona la palabra gutenberg en el documento.

Hasta ahora todo suena bien, pero tiene un percance en cuanto al tiempo de ejecución. Este script se demora aproximadamente 65 segundos. Este ejercicio se torna restrictivo cuando trabajamos con cantidades de datos gigantes. Necesitamos otro enfoque un poco más moderno.

Mapper y Reducer primitivos en Python

Vamos a refactorizar nuestro código en el script `non_distributed.py` mediante la separación de tareas. Generamos dos scripts, uno que asigne un contador `1` a cada una de las palabras existentes en el documento, y un script que sume los contadores a nivel de palabra. Estos se conocen como el `mapper` y el `reducer`, respectivamente. De manera adicional, existe un paso llamado sort implementado posterior al `mapper` y previo al `reducer` que facilita el proceso de suma por parte del reducer, el cual puede ser implementado directamente en la línea de comando.

Mapper

El paso de mapear valores es simple. Consiste en parsear el texto, extraer las palabras contenidas y para cada una de estas (irrestricta de su frecuencia de ocurrencia), asignamos un identificador de 1 con la nomenclatura `<palabra> 1`. Por ejemplo, si la palabra `sofa` se repite 2 veces, nuestro resultado del mapper sería:

```
sofa   1
sofa   1
```

Esto se conoce como un **resultado intermedio**, dado que debe ser procesado por las otras fases de nuestro mapreduce primitivo. A continuación vamos a generar el código del script `mapper.py` que alojaremos en la misma carpeta de proyecto. El código se encuentra comentado.

```
#mapper.py

# vamos a importar las librerías de expresiones regulares y de sistema
import sys,re

# vamos a leer los datos que provengan del standard input del sistema
feed_document = sys.stdin

# para cada una de las líneas en los datos
for line_in_document in feed_document:
   # reemplazamos los valores no alfanuméricos por espacios.
   line_in_document = re.sub(r'^\W+|\W+$', '', line_in_document)
   # convertimos el string en una lista de strings
   words_in_line = re.split(r'\W+',line_in_document)

   # para cada palabra en la lista de strings
   for word in words_in_line:
       # vamos a generar un print con la siguiente nomenclatura:
       # <palabra>   1
       print(word.lower() + '\t' + "1")
```

Así, las primeras líneas del `mapper.py` tendrán la siguiente estructura.

```
isz :: Desktop/batch_download » cat anna_karenina.txt| python mapper.py
        1
the     1
project 1
gutenberg       1
ebook   1
of      1
anna    1
karenina        1
by      1
leo     1
```

Imagen 7. Primeras líneas de `mapper.py`.

Shuffling y sort

Entre el resultado del `mapper` anteriormente visto y el `reducer` que realizará el conteo, existe el paso `sort` que permitirá el ordenamiento por palabra de los elementos del `mapper`. Este paso se implementa para reducir el tiempo de ejecución del `reducer`, dado que con las palabras ordenadas no tendrá que ir buscando en memoria su ocurrencia.

Si ejecutamos la línea `cat anna_karenina.txt | python mapper | sort -k1,1`, obtendremos una lista ordenada de todas las ocurrencias:

```
youth   1
youth   1
youthful        1
youthful        1
youthful        1
youthful        1
youthful        1
youthful        1
youthful        1
youthful        1
youthfulness    1
youths  1
```

Imagen 8. Ordenamiento `sort`.

La opción `-k1,1` de `sort` permite definir cuáles son los campos a ordenar. En este caso, tendremos dos campos correspondientes a la palabra y su indicador.

Reducer

Una vez que tengamos nuestros resultados ordenados mediante el paso `sort`, podemos implementar el `reducer` que permitirá contar la ocurrencia de cada una de las palabras. Lo que debemos considerar es que el input será el print del `mapper` ordenado por palabra. Así, estaremos sumando los indicadores para cada una de las palabras en el standard input.

```
#reducer.py
# importamos la librería de sistema
import sys
# vamos a leer los datos que provengan del standard input del paso
previo
feed_mapper_output = sys.stdin
# generamos un identificador de la palabra previa
previous_counter = None
# generamos un contador de la cantidad de palabras
total_word_count = 0
# para cada una de las líneas en el standard input del paso previo
for line_ocurrence in feed_mapper_output:
   #vamos a separar entre palabra e indicador
   #recordando que el standard input en esta etapa será
   # <palabra>   1
   word, ocurrence = line_ocurrence.split('\t')
   # si es que la palabra es distinta a la previa
   # procedemos a contarla
   if word != previous_counter:
       # si la palabra no es la primera
       if previous_counter is not None:
           # vamos a imprimir un resultado intermedio
           print(str(total_word_count) + '\t' + previous_counter)
       # asignamos al previous counter la palabra nueva
       previous_counter = word
       # reseteamos el contador para la palabra
       total_word_count = 0
   # contamos la cantida de ocurrencias
   total_word_count += int(ocurrence)
# imprimimos el resultado final
print(previous_counter  + '\t' + str(total_word_count))
```

Podemos implementar todo nuestro pipeline de trabajo con la línea `cat anna_karenina.txt | python mapper.py | sort -k1,1 | python reducer.py`. Cabe destacar que el paso `sort` permite reducir el tiempo de ejecución del reducer de 3 segundos a .4 segundos. El resultado se puede visualizar a continuación.

```
isz :: Desktop/batch_download » cat anna_karenina.txt| python mapper.py | sort -k1,1| python reducer.py
2       zoology
2       zu
22      à
1       âge_
1       ça
1       ça_
1       écoles
1       état
1       été
2       être
```

Imagen 9. Resultado final.

Así, podemos ilustrar y resumir el proceso en la siguiente imágen.

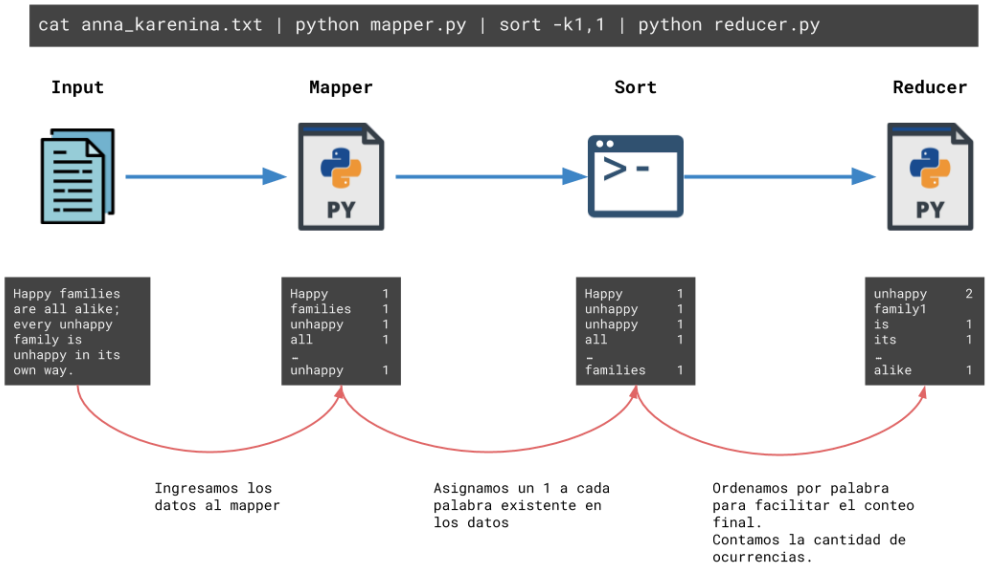

Imagen 10. Procesos de mapper reduce.

Conclusiones

A lo largo de esta lectura aprendimos sobre los distintos modos de agilizar nuestro trabajo. En específico, logramos diferenciar los procesos de paralelización y distribución de las tareas. Es bueno tener en consideración cómo operan a grandes rasgos para identificar la mejor solución a nuestro problema de negocio.

También implementamos nuestro primer flujo de trabajo en un MapReduce primitivo. Tengan en mente que los pasos `mapper`, `sort` y `reducer` estarán presentes de aquí en adelante, por lo que identificar bien qué hace cada uno dentro de nuestra línea de trabajo nos permitirá definir nuestras tareas.

Cabe destacar que esta implementación de MapReduce se apoya de manera subtancial en el ecosistema Hadoop. Veremos cómo implementar este modelo de trabajo cuando nuestros datos están distribuidos en HDFS y cómo logramos la asignación de tareas a múltiples fases del trabajo.