



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

**Implementación de casos de uso de patrones
de diseño MapReduce en Python, sobre
infraestructuras paralelas distribuidas
basadas en contenedores ligeros**

Realizado por: Manuel Rodríguez Sánchez

Dirigido por: Agustín Carlos Caminero Herráez

Curso: 2020-2021 Convocatoria de diciembre



**Implementación de casos de uso de patrones de diseño
MapReduce en Python, sobre infraestructuras paralelas
distribuidas basadas en contenedores ligeros**

**Proyecto de Fin de Grado en Ingeniería Informática
Modalidad específica**

Realizado por: Manuel Rodríguez Sánchez

Dirigido por: Agustín Carlos Caminero Herráez

Fecha de defensa: 18 de diciembre de 2020

Resumen

A lo largo de todo este trabajo, sentaremos la base de conocimiento para poder desarrollar y ejecutar ciertos patrones de diseño, que nos serán útiles para el manejo de grandes cantidades de datos originados desde distintos ámbitos. Pero antes hemos de hacer un estudio del contexto en el que nos vamos a mover, conociendo las características más destacables de las arquitecturas distribuidas, y las herramientas que permiten desplegarlas. A esto hay que sumarle el paradigma de programación en el que desarrollaremos nuestros patrones, y el lenguaje apropiado que se usará, incluyendo su entorno de trabajo.

Comenzaremos con una introducción a ciertos conceptos de los que hoy día oímos hablar en foros, noticias, redes sociales, etc. a la misma vez que explicaremos que nos lleva a sumergirnos en este campo de los macrodatos o "Big Data", marcando con ello los objetivos a los que queremos llegar.

Nos adentraremos en la virtualización y los avances en este campo que está revolucionando, entre otros, el concepto de computación y servicios, y tendremos oportunidad de ver cómo es posible virtualizar de forma ligera arquitecturas distribuidas, dándonos una idea de las ventajas que pueden llegar a ofrecernos.

Conoceremos Hadoop, su estructura, su ecosistema de aplicaciones, y su importancia en el campo de los datos masivos y extracción de información, junto con las herramientas y distribuciones que facilitan su despliegue. Nos sumergiremos en la forma que tiene de almacenar y gestionar los datos y, sobre todo, como los procesa, y precisamente en esta parte será donde más nos centraremos: el procesamiento de los datos con MapReduce.

Desarrollaremos cuatro tipos de patrones de diseño bajo este paradigma, en los que se explicará su funcionamiento, estructura, y desarrollos de los prototipos, que serán implementados en Python y su librería MRJob.

Lista de palabras: Hadoop, MapReduce, Docker, Contenedores ligeros, Big Data, Macrodatos, Python, MRJob, patrones de diseño, virtualización, computación distribuida.

Abstract

Implementation of use cases of MapReduce design patterns in Python, on distributed parallel infrastructures based on lightweight containers

Throughout all this work, we will establish the knowledge base to be able to develop and execute certain design patterns, which will be useful for handling large amounts of data originating from different areas. But first we have to do a study of the context in which we are going to move, knowing the most remarkable characteristics of distributed architectures, and the tools that allow them to be deployed. To this we must add the programming paradigm in which we will develop our patterns, and the appropriate language that will be used, including its work environment.

We will start with an introduction to certain concepts that we hear about today in forums, news, social networks, etc. at the same time we will explain what leads us to immerse ourselves in this field of big data, thereby marking the objectives we want to reach.

We will delve into virtualization and the advances in this field that is revolutionizing, among others, the concept of computing and services, and we will have the opportunity to see how it is possible to virtualize distributed architectures in a light way, giving us an idea of the advantages that can be achieved offer us.

We will learn about Hadoop, its structure, its ecosystem of applications, and its importance in the field of big data and information extraction, along with the tools and distributions that facilitate its deployment. We will immerse ourselves in the way it stores and manages data and, above all, how it processes it, and precisely in this part will be where we will focus the most: the data processing with MapReduce.

We will develop four types of design patterns under this paradigm, in which their operation, structure, and prototype developments will be explained, which will be implemented in Python and its MRJob library.

Key words: Hadoop, MapReduce, Yarn, Docker, light containers, Big Data, Python, MRJob, design patterns, virtualization, distributed computing

Índice

Resumen	2
Abstract	3
Capítulo 1. Introducción	10
1.1. Contextualización y motivación	10
1.2. Objetivos	12
1.3. Organización del trabajo	12
1.4. Resumen del capítulo	14
Capítulo 2. Hadoop	15
2.1. Introducción a Hadoop	15
2.2. Estructura de Hadoop	15
2.2.1. YARN	17
2.2.2. HDFS	18
2.2.3. MapReduce	22
2.3. Ecosistema de Hadoop	24
2.4. Resumen del capítulo	26
Capítulo 3. Estado del arte	27
3.1. Herramientas usadas en el marco de programación MapReduce	27
3.2. Tecnologías para el despliegue de infraestructuras <i>Big Data</i> .	34
3.3. Resumen de capítulo	40
Capítulo 4. Despliegue de la arquitectura	41
4.1. Secuencia de despliegue del <i>cluster</i>	41
4.2. Secuencia de arranque del cluster y acceso a sus aplicaciones	42
4.3. Ejemplo de ejecución de un programa MapReduce en el cluster	45
4.4. Resumen del capítulo	47
Capítulo 5. Conjuntos de datos (Datasets)	49
5.1. Foros	49
5.2. On line retail -Ventas Clientes.	50
5.3. Eventos alumnos	51
5.4. Resumen del capítulo	51
Capítulo 6. Patrones de diseño	53
6.1. Patrones de Resumen	53
6.1.1. Resúmenes numéricos	53
6.1.2. Índice invertido	61
6.1.3. Contando con contadores	65
6.2. Patrones de filtrado	69

6.2.1. Extracción de subconjuntos.....	69
6.2.2. Filtro valores más altos	73
6.2.3. Filtro de valores no repetidos.....	78
6.3. Patrones de Organización de datos	80
6.3.1. Estructuración jerárquica	80
6.3.2. Ordenación de registros	86
6.3.3. Mezclar y desordenar	89
6.4. Patrones de Unión	91
6.4.1. Estructura de los patrones	92
6.4.2. Unión interna	94
6.4.3. Unión externa.....	95
6.4.4. Antiunión	99
6.4.5. Producto cartesiano	100
6.4.6. Unión replicada	103
6.5. Resumen del capítulo	106
Capítulo 7. Planificación y presupuesto.....	107
7.1. Planificación	107
7.2. Presupuesto	108
7.2.1. Recursos empleados.....	108
7.2.2. Coste de los recursos	109
7.3. Resumen de capítulo.	110
Capítulo 8. Aportaciones, trabajo futuro y conclusiones	111
Bibliografía	113
Anexo – Patrones completos	117

Ilustración 1 - Arquitectura tradicional de un sistema distribuido [1]	16
Ilustración 2 - Arquitectura de Hadoop [1].....	16
Ilustración 3 - Esquema YARN. [12]	17
Ilustración 4 - Esquema de funcionamiento de YARN.....	18
Ilustración 5 - Ejemplo de "cluster" formado por un maestro y cuatro esclavos [14]	19
Ilustración 6 - Petición de lectura al NameNode	20
Ilustración 7 - Respuesta del NameNode al cliente	20
Ilustración 8 - Petición de bloques a los DataNode.	21
Ilustración 9 - Lectura y escritura en HDFS.	21
Ilustración 10 - Ejemplo de funcionamiento de MapReduce.	23
Ilustración 11 - Intérprete de Python y ejemplo de ejecución de una orden o instrucción.	28
Ilustración 12 - Programa en Python llamado "programa.py"	28
Ilustración 13 - Ejecución en una terminal de un programa en Python.....	28
Ilustración 14 - Ejemplo de uso de entorno Jupyter Notebook.	29
Ilustración 15 - Fichero "map.py", que contiene las tareas de Map para contar palabras.	30
Ilustración 16 - Fichero "reduce.py" que contiene las tareas que hay que hacer en el Reduce.....	30
Ilustración 17 - Ejemplo del contador de palabras con MRJob [14].	32
Ilustración 18 - Esquema que define un paso MapReduce con las MRJob.	32
Ilustración 19 - Ejecución de un programa MapReduce por pasos.	33
Ilustración 20 - Computador con máquinas virtuales (MV)	35
Ilustración 21 - Filosofía de los contenedores	36
Ilustración 22 - Computador con contenedores.....	37
Ilustración 23 - Esquema de un cluster Hadoop en contenedores Docker.	38
Ilustración 24 - Contenedor Docker con lo necesario para una plataforma Moodle.....	39
Ilustración 25 - Ejemplo de arquitectura de microservicios.....	39
Ilustración 26 - Arquitectura del "cluster" para desarrollo y pruebas de patrones que se va a desplegar.	41
Ilustración 27 - Contenedores levantados; el "cluster" está en funcionamiento.	42
Ilustración 28 - Arrancando el "cluster"	43
Ilustración 29 - Utilidades accesibles mediante el navegador. Éstas están dentro de los contenedores.	43
Ilustración 30 - Acceso a un contenedor.	44
Ilustración 31 - Directorio de intercambio con el "cluster"	44
Ilustración 32 - Ejecución en local del programa MapReduce.	46
Ilustración 33 - Subida del archivo al HDFS.	46
Ilustración 34 - Proceso de ejecución en el cluster. Observamos que ha terminado el paso 1.	46
Ilustración 35 - Continuación del proceso de ejecución y resultado final.	47
Ilustración 36 - Estructura de un patrón de resumen numérico [7, p. 16]	54
Ilustración 37 - Ejemplo de construcción de tabla de índice invertido.	62
Ilustración 38 - Esquema de patrón "Contando con contadores".	65
Ilustración 39 - Estructura de un patrón de filtrado [7, p. 45]	70
Ilustración 40 - Estructura del patrón "TopTen" [7, p. 60].....	73
Ilustración 41 - Cuerpo de archivo XML [36]	80
Ilustración 42 - Esquema del patrón de estructura jerárquica [7, p. 74].	81
Ilustración 43 - Muestra del ejemplo de la Tabla 40 después de añadirle la etiqueta <raíz> al principio y </raíz> al final	85
Ilustración 44 - Esquema de funcionamiento de la Unión Replicada [7].....	103

Tabla 1 - Campos que conforman la tabla Foros.	49
Tabla 2 - Campos de la tabla On-line Retail. La línea superior indica el índice con el que identificaremos los campos en los programas MapReduce.	50
Tabla 3 - Campos de la tabla eventos_alumnos. La línea superior indica el índice con el que identificaremos los campos en los programas MapReduce.	51
Tabla 4 - Mapper para el cálculo de la media aritmética.	54
Tabla 5 - Reducer para calcular la media aritmética.	55
Tabla 6 - Cálculo de la media en el Reducer usando la librería statistics.	55
Tabla 7 - Ejemplo de salida de la media aritmética.	55
Tabla 8 - Programa para contar artículos y mostrar el más caro y el más barato por cliente.	56
Tabla 9 - Segunda versión del Reducer para darnos la cuenta, valor máximo y valor mínimo.	57
Tabla 10 - Parte de la salida de contar, valor máximo y valor mínimo	57
Tabla 11 - Cálculo de la mediana	58
Tabla 12 - Segunda versión para el cálculo de la mediana usando la librería statistics.	59
Tabla 13 - Resultado parcial del cálculo de la mediana.	59
Tabla 14 - Programa para el cálculo de la desviación estándar.	60
Tabla 15 - Resultado de la ejecución del programa para la desviación estándar.	60
Tabla 16 - Contenido del archivo de ejemplo docejemplo.txt	63
Tabla 17 - Código del mapper para extraer las palabras y la línea donde se encuentran.	63
Tabla 18 - Muestra del código del Reducer para el índice invertido.	64
Tabla 19 - Resultado de la ejecución del patrón Índice invertido.	64
Tabla 20 - Versión 1 del patrón Contando con Contadores.	66
Tabla 21 - Resultado de la ejecución del modelo de la Tabla 20	67
Tabla 22 - Segunda versión del patrón Contando con Contadores, aplicando el Reducer.	67
Tabla 23 - Resultado de la ejecución del código de la Tabla 22	68
Tabla 24 - Versión tercera del patrón Contando con Contadores.	68
<i>Tabla 25 - Resultado de la ejecución del código de la Tabla 24</i>	68
Tabla 26 - Ejemplo de patrón de filtro de datos, en el que eliminamos los registros cuyo id sea texto.	71
Tabla 27 - Resultado parcial después de ejecutar el código de la Tabla 26	71
Tabla 28 - Código seguimiento de eventos.	72
Tabla 29 - Resultado parcial de la ejecución del código de la Tabla 28	72
Tabla 30 - Patrón TopN. Mapper.	74
Tabla 31 - Función para extraer los N valores más altos	76
Tabla 32 - Patron TopN. Reducer.	77
Tabla 33 - Resultado de la ejecución del patrón TopN.	77
Tabla 34 - Expresión regular y mapper para el patrón.	78
Tabla 35 - Reducer para el filtro de valores no repetidos.	79
Tabla 36 - Resultado de la ejecución del ejemplo descrito en la Tabla 34 y Tabla 35	79
Tabla 37 - Patrón Mapper estructuración jerárquica.	82
Tabla 38 - Patrón Reducer estructuración jerárquica.	83
Tabla 39 - Versión del Reducer del patrón estructuración jerárquica usando diccionarios	84
Tabla 40 - Ejemplo de salida en XML de la ejecución del prototipo: "estructuración jerárquica"	84
Tabla 41 - Mapper para la versión 1 del prototipo Ordenación de registros.	87
Tabla 42 - Reducer para la versión 1 del prototipo Ordenación de registros.	87

Tabla 43 – Resultado parcial de la ejecución del prototipo descrito en la Tabla 41 y Tabla 42	88
Tabla 44 - Prototipo 2 de Ordenación total de registros.	88
Tabla 45 – Resultado parcial de la ejecución del prototipo descrito en la Tabla 44	89
Tabla 46 - Prototipo mezclar registros	90
Tabla 47 - Mapper para el prototipo anonimizar y mezclar.	90
Tabla 48 - Código de la función para "limpiar" los nombres de los archivos que entran por stream.	92
Tabla 49 - Estructura condicional del mapper para controlar ficheros de entrada.	93
Tabla 50 - Estructura general del código del Reducer, antes de aplicar el algoritmo de unión.	93
Tabla 51 – artículos_stock	94
Tabla 52 – tiendas	94
Tabla 53 - Resultado de ejecutar una unión interna	95
Tabla 54 - Algoritmo para la Unión Interna	95
Tabla 55 - Salida parcial de la ejecución de la unión interna.	95
Tabla 56 - Unión por la izquierda de tabla tiendas y tabla_stock	96
Tabla 57 - Algoritmo para la Unión externa por la izquierda	96
Tabla 58 - Resultado parcial de la ejecución de una unión externa por la izquierda.	97
Tabla 59 - Resultado de la unión externa por la derecha	97
Tabla 60 - Algoritmo unión externa por la derecha	97
Tabla 61 - Resultado parcial de la salida de la unión externa por la derecha	98
Tabla 62 - Resultado de ejecutar una unión completa	98
Tabla 63 - Algoritmo unión completa	98
Tabla 64 - Resultado de la ejecución de la unión completa	99
Tabla 65 - Resultado de ejecución de la antiunión	99
Tabla 66 - Algoritmo antiunión.	99
Tabla 67 - Resultado de la antiunión.	99
Tabla 68 - Resultado de la operación Producto Cartesiano	101
Tabla 69 - Mapper del prototipo producto cartesiano.	101
Tabla 70 - Reducer y algoritmo de unión del prototipo producto cartesiano	102
Tabla 71 - Salida parcial de la ejecución del producto cartesiano.	102
Tabla 72 - Líneas de código para rellenar una estructura diccionario	105
Tabla 73 - Mapper para la unión replicada.	105
Tabla 74 - Coste en recursos humanos	109
Tabla 75 - Costes en recursos materiales	109
Tabla 76 - Costes en recursos inmateriales.	109
Tabla 77 - Coste total del proyecto	110

Patrón 1 - Resúmenes numéricos. Media aritmética, versión uno.	117
Patrón 2 - Resúmenes numéricos. Media aritmética, versión dos.	118
Patrón 3 - Resúmenes numéricos. Desviación típica.	118
Patrón 4 - Resúmenes numéricos. Mediana versión uno.	119
Patrón 5 - Resúmenes numéricos. Mediana versión dos.	120
Patrón 6 - Resúmenes numéricos. Cálculo del valor mínimo, máximo y contar. Versión uno.	120
Patrón 7 - Resúmenes numéricos. Cálculo del mínimo, máximo y contar. Versión dos.	121
Patrón 8 - Resúmenes numéricos. Contando con contadores. Versión uno.	121
Patrón 9 - Resúmenes numéricos. Contando con contadores. Versión dos con el Reducer.	122
Patrón 10 - Resúmenes numéricos. Contando con contadores. Versión tres. Declaración fuera de la clase.	122
Patrón 11 - Resúmenes numéricos. Índice invertido.	123
Patrón 12 - Filtrado. Extracción de subconjuntos	124
Patrón 13 - Filtrado. Seguimiento de hilo de eventos.	124
Patrón 14 - Filtrado. Extraer los N valores más altos.	125
Patrón 15 - Filtrado. Valores no repetidos.	126
Patrón 16 - Organización de datos. Estructuración jerárquica. Versión uno.	127
Patrón 17 - Organización de datos. Estructuración jerárquica. Versión dos usando diccionarios.	128
Patrón 18 - Organización de datos. Orden total.	129
Patrón 19 - Organización de datos. Orden total versión dos. La ordenación se hace en los Mapper.	130
Patrón 20 - Organización de datos. Orden total versión tres. Se ordenan registros usando la clave de cliente.	131
Patrón 21 - Organización de datos. Mezclar registros.	132
Patrón 22 - Organización de datos. Anonimizar y mezclar.	132
Patrón 23 - Unión interna.	133
Patrón 24 - Unión externa por la izquierda.	134
Patrón 25 - Unión externa por la derecha.	135
Patrón 26 - Unión completa.	136
Patrón 27 - Antiunión.	137
Patrón 28 - Producto cartesiano.	138
Patrón 29 - Unión replicada.	139

Capítulo 1. Introducción

1.1. Contextualización y motivación

Macrodatos o “Big Data”

Hoy día los sistemas y dispositivos recogen cantidades ingentes de datos. Estos son generados por las personas cuando acceden a redes sociales, hacen una compra con tarjeta (“on-line” o presencial), valoran una transacción, o pasan la tarjeta de fidelización en un supermercado. Pero también los datos son generados por la Internet de las Cosas (Internet of Things, IoT [1] [2]): desplazamiento de un vehículo, frecuencia de uso de un robot de limpieza, el altavoz que busca la música que le pides, o ese sensor que manda una señal cuando un cliente entra o sale de una tienda. Todos estos datos quedan almacenados a la espera que sean gestionados y analizados, y de eso trata cuando oímos hablar de datos masivos o “Big Data” [3], de la capacidad de gestionar estos grandes volúmenes de datos de todo tipo.

En consecuencia, los datos masivos o “Big Data”, han de cumplir una serie de requisitos. Éstos se conocen como las tres uves [4] [5]:

1. *Volumen*: la cantidad de información que generan las interacciones de las personas y de las cosas en el mundo digital. Pueden llegar a almacenarse terabytes e incluso petabytes de información.
2. *Variedad*: se refiere a que tenemos varios tipos de datos, que se pueden clasificar en datos estructurados, datos no estructurados y datos semi estructurados. Esto significa que los datos no están limitados solo a texto, si no que pueden ser imágenes, audios, vídeos, etc.
3. *Velocidad*: los datos se generan de una forma muy rápida y han de procesarse velozmente, para cumplir los objetivos en su análisis antes de que pierdan su vida útil o la calidad en la información. Prácticamente se han de analizar a tiempo real y con ello tomar decisiones también a tiempo real.

Se habla de que existen otras dos uves más, que creemos conveniente destacar. Estas son:

4. *Valor*, porque detrás del concepto llamado “Big Data”, lo que hay es un valor para el negocio. Este valor se traduce en una mayor productividad y/o rentabilidad, a

partir de las decisiones que se toman una vez analizada la información extraída de los datos recogidos.

5. *Veracidad* de los datos o lo que es lo mismo: *la calidad del dato*. Puede darse el caso que los datos no sean consistentes o estén desvirtuados, debido a sesgos o ruido en la recogida de éstos. De ahí la importancia que las fuentes de recogida de datos estén claras y bien definidas.

A partir de todas estas condiciones anteriores, deducimos que con los sistemas de almacenamiento y procesamiento tradicionales (como las bases de datos relacionales), se hace muy complicado procesar y extraer información. Esto nos lleva al uso de tecnologías que se han desarrollado con el objetivo de trabajar con datos masivos generados desde distintos canales, y en distintas formas. Una de las tecnologías más ampliamente conocida y usada es Hadoop [4].

Hadoop es un entorno distribuido de datos y procesos, es decir, implementa procesamiento en paralelo a través de nodos de datos en un sistema de ficheros distribuidos, lo que se conoce como grupo o conjunto de nodos; o con el término anglosajón *cluster*. Su estructura está compuesta por varios elementos que son [6]:

- ⇒ *Yarn*: se encarga de gestionar los recursos del conjunto de nodos o *cluster*.
- ⇒ *HDFS*: es el sistema de ficheros distribuido en el que se puede almacenar gran cantidad de datos.
- ⇒ *MapReduce*: es un algoritmo de procesamiento de datos en paralelo.

De estos tres componentes, el presente proyecto trata sobre MapReduce. Los programas que se desarrollan bajo este paradigma dividen los procesos de búsqueda y extracción de información en pasos *map* y *reduce*, y los manda a los nodos del *cluster* para su ejecución concurrente, de forma que es el código el que se envía a los nodos que almacenan los datos de entrada para su ejecución, y no al revés.

La programación *Mapreduce* requiere de un cambio drástico de pensamiento y abstracción, dada la complejidad si lo comparamos con otros paradigmas de programación. Para que la programación de aplicaciones *MapReduce* sea menos tediosa y su curva de aprendizaje sea más suave, es de utilidad disponer de patrones que proporcionen al desarrollador, guías sobre la forma en que debe estructurar su código en pasos *map* y *reduce*. Estos patrones permiten al desarrollador, tener una primera aproximación sobre cuántos pasos *map* y

reduce son necesarios para resolver un problema dado, qué debe hacer cada paso, y qué datos se deben enviar entre los pasos.

Existen patrones de diseño MapReduce para el lenguaje de programación Java [7], pero para otros lenguajes de programación como por ejemplo Python, no se han encontrado propuestas similares.

1.2. Objetivos

En vista de todo lo expuesto en el punto anterior, llegamos a la conclusión de que podemos marcarnos el siguiente objetivo: desarrollar patrones de diseño, bajo el paradigma de programación *MapReduce*, usando el lenguaje de programación Python.

Efectivamente, hemos podido comprobar que no existe bibliografía que recoja y clasifique distintos patrones de diseño con Python en el marco MapReduce, por tanto, estudiaremos y analizaremos el funcionamiento y comportamiento de varios tipos de patrones descritos en el libro *MapReduce Design Patterns* [7], y posteriormente implementaremos programas o prototipos *MapReduce* en Python, basándonos en los diseños previamente estudiados. Profundizaremos en la librería *MRJob* de Python para el desarrollo de aplicaciones *MapReduce*, ya que simplifica la implementación de éstas, y facilita las ejecuciones en local, en *clusters* Hadoop, y en proveedores de servicios en la nube como Amazon Web Services (AWS) o Azure.

Para poder poner en práctica y observar los resultados de la ejecución de estos programas, nos planteamos otro objetivo: hacer un despliegue de una infraestructura distribuida Hadoop en contenedores Docker [8], que es un mecanismo de virtualización ligera que será abordado con detalle en el capítulo 3.

Por último, necesitamos datos para probar todos los desarrollos que vayamos haciendo, teniendo en cuenta que dependiendo de los patrones que vayamos a utilizar o probar, será necesario disponer de datos adaptados a estos patrones. Por tanto, otra de las tareas u objetivos para la consecución de este trabajo, será la búsqueda y selección de diferentes bases de datos o tablas, que estarán disponibles públicamente en repositorios.

1.3. Organización del trabajo

Comenzaremos en el tema dos estudiando Hadoop, qué es, su estructura y su ecosistema de aplicaciones.

En el tema tres, abordaremos el estado del arte, donde en un primer apartado nos adentraremos en las herramientas, entornos y librerías que existen para el desarrollo de

programas en el marco MapReduce y en un segundo apartado de este tema, veremos las tecnologías para el despliegue de infraestructuras “*Big Data*” en entornos virtualizados; repasaremos lo que son las máquinas virtuales o “*hipervisores*”, y nos adentraremos en el concepto abstracto de virtualización ligera con contenedores.

A continuación, en el tema cuatro, haremos un despliegue de una arquitectura *Hadoop* basada en virtualización ligera, que será sobre la que ejecutemos todos los programas que vayamos desarrollando; veremos paso a paso las instalaciones de aplicaciones que son necesarias, y posteriormente, haremos los pasos necesarios para la descarga y puesta en funcionamiento de la infraestructura (que está previamente creada en Cloudera [9]). En este apartado, se desarrollará un ejemplo sencillo en Python, y veremos cómo ejecutarlo en el marco *MapReduce* dentro de la infraestructura desplegada.

En el tema cinco, se describirán los conjuntos de datos que se usarán en los desarrollos y pruebas de prototipos *MapReduce*.

En el tema seis entraremos de lleno en el objetivo principal de este trabajo: los tipos de patrones, subtipos, para que pueden usarse, y el desarrollo de prototipos. Trataremos cuatro tipos de patrones: resumen, filtrado, organización de datos y unión.

Dentro de los patrones de resumen, implementaremos los prototipos pertenecientes a los subtipos para el cálculo de distintas operaciones matemáticas y estadísticas, extracción de valores máximos o mínimos y desarrollaremos un ejemplo de índice invertido.

En los patrones de filtrado desarrollaremos varios subtipos: extracción de subconjuntos, seguimiento de un hilo de eventos de una web, extracción de los N valores más altos, y limpiar valores que están repetidos en una base de datos.

En el tercer apartado del tema seis, nos adentraremos en los patrones de organización de datos, donde veremos cómo a partir de un conjunto de datos podemos convertirlos en un formato estructurado como XML o JSON. También estudiaremos prototipos de este patrón, que permiten reorganizar las salidas de datos aplicando condiciones, junto con otros ejemplos para ordenar, desordenar, mezclar y barajar registros.

En el último apartado, entraremos en los patrones de unión, donde desarrollaremos ejemplos de los distintos tipos de uniones, y estudiaremos en detalle el patrón de unión replicada.

1.4. Resumen del capítulo

A lo largo de este capítulo se ha tratado de sentar las bases sobre las que se sustentará todo este trabajo, hemos marcado los objetivos y clasificado cada uno de los temas que vamos a desarrollar, con una introducción previa en los conceptos de macrodatos o “Big Data”, y las herramientas que nos permiten almacenar y procesar todos los datos que se generaran a nuestro alrededor. Estas herramientas las veremos con detalle en el siguiente capítulo.

Capítulo 2. Hadoop

2.1. Introducción a Hadoop

Hadoop se compone de un ecosistema de software para el almacenamiento, procesamiento y análisis de datos masivos. Hablamos de terabytes y petabytes, en un entorno de procesamiento paralelo a través de nodos de datos, en un sistema de ficheros distribuido, conformando una arquitectura de tipo *cluster* [10]. En éste, tendremos un conjunto de nodos maestros que gobernarán los nodos esclavos, que son los que realmente se van a encargar de realizar el procesamiento de la información.

Las características más importantes de Hadoop son [4] [6]:

1. *Es un entorno distribuido*: se ejecuta en un conjunto de ordenadores conectados entre sí. Implementa procesamiento en paralelo a través de nodos de datos en un sistema de ficheros distribuido.
2. *Es un sistema escalable*: cuantos más nodos se agreguen mayor será la capacidad de procesamiento y almacenaje. Está diseñado para escalar desde unos pocos nodos a miles de máquinas, cada una de ellas ofreciendo la lógica de negocio y el almacenamiento local.
3. *Es tolerante a fallos*: en los sistemas distribuidos los fallos son habituales, por tanto, si aquí falla un nodo, el sistema continúa funcionando, reasignando las tareas sin perder datos.
4. *Es de código abierto*: éste está disponible de forma abierta para quien quiera descargarlo, probarlo, modificarlo, etc.

2.2. Estructura de Hadoop

Destacamos que Hadoop es un sistema distribuido que descentraliza el almacenamiento de datos y la ejecución de los procesos, en contraposición a los grandes servidores centralizados que han sido tendencia en estos años atrás. A modo de ejemplo podemos ver en la Ilustración 1, que en los sistemas distribuidos tradicionales, los datos se almacenaban en una base de datos externa a los nodos, y éstos tenían que acceder a la base de datos, extraer la información y procesarla. Esto podía producir colapsos o cuellos de botella que provocaban la ralentización y bajada de productividad del sistema.

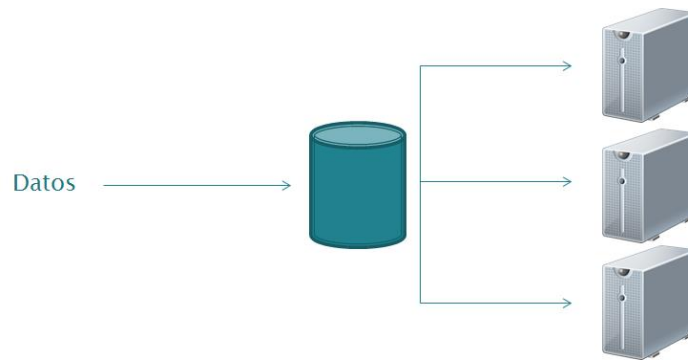


Ilustración 1 - Arquitectura tradicional de un sistema distribuido [4]

Sin embargo, con la arquitectura de *Hadoop*, que podemos verla en la Ilustración 2, los datos se almacenan en cada uno de los nodos del sistema, donde en éstos también se realizarán los cálculos y procesos.

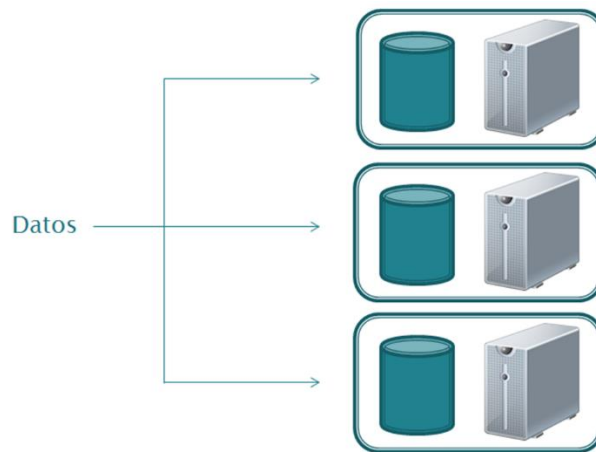


Ilustración 2 - Arquitectura de Hadoop [4]

Podemos decir que una agrupación de nodos o *cluster Hadoop*, va a estar formada como mínimo, por un nodo maestro y varios nodos esclavos, y nos permitirá implementar procesamiento paralelo usando los nodos esclavos del *cluster* en un sistema de ficheros distribuido. El componente de *Hadoop* para gestionar y almacenar los datos es el *Hadoop Distributed File System (HDFS)* [4] [11], el componente para el control y ejecución de procesos es *MapReduce*, y el componente para la gestión de todo el *cluster* y sus recursos será YARN [4] [6].

2.2.1. YARN

Podemos definir YARN (Yet Another Resource Negotiator) como la capa que se encarga de la planificación de trabajos y la gestión de los recursos del cluster Hadoop.

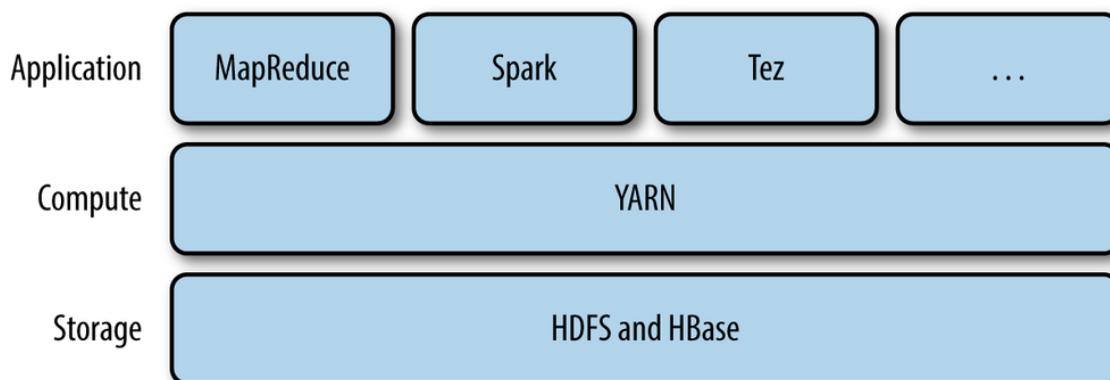


Ilustración 3 - Esquema YARN. [12]

Como podemos comprobar en el esquema de la Ilustración 3, todas las aplicaciones distribuidas asociadas a Hadoop (Ver apartado 2.3), serán gestionadas por el YARN. Podemos ver que MapReduce es una aplicación más, gestionada por éste.

La estructura de YARN se compone de [13]:

- *ResourceManager*: es la autoridad máxima que gestiona y planifica los recursos del cluster, y controla todas las aplicaciones del sistema. Este se encuentra en el nodo maestro (*namenode*). Tiene dos componentes principales:
 - o El planificador (*Scheduler*): determina como se planifican los trabajos, y decide los recursos que se van a utilizar dentro del *cluster*.
 - o El gestor de aplicaciones (*ApplicationManager*): este se encarga de arrancar el *ApplicationMaster* dentro de los nodos cuando una aplicación quiere ejecutarse, para que este último se encargue de esa aplicación.
- *NodeManager*: es el agente de marco por máquina, y es el responsable de los recursos del nodo hijo (*datanode*).
- *ApplicationMaster*: es una biblioteca específica cuya misión es negociar los recursos del *ResourceManager* y trabajar con los *NodeManager* para ejecutar y monitorear las tareas y aplicaciones.

En el siguiente esquema extraído de la web *Apache Software Foundation* [13], podemos ver el funcionamiento de la estructura de YARN.

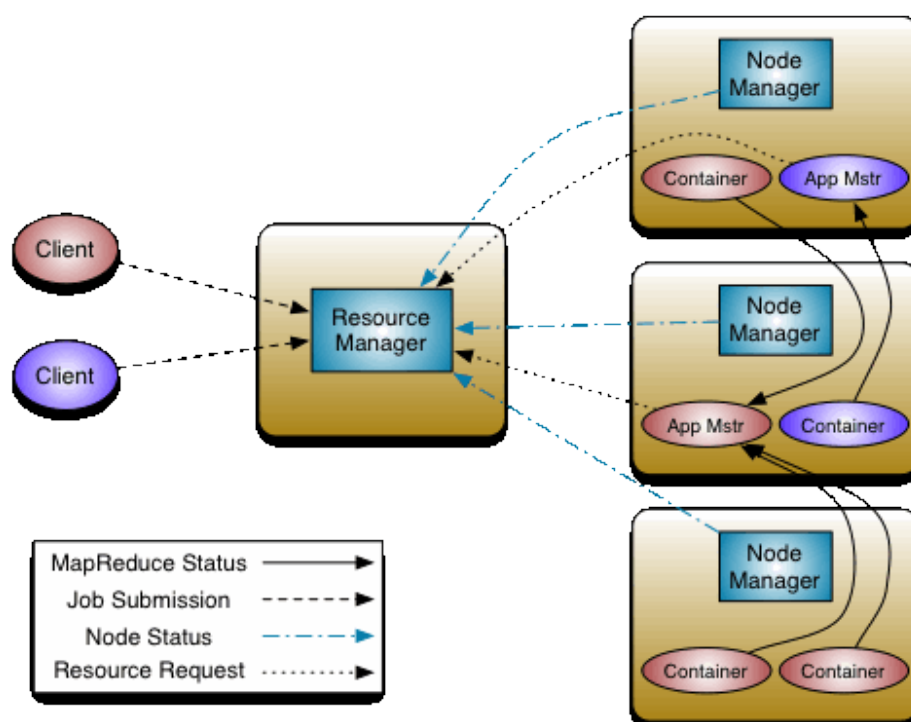


Ilustración 4 - Esquema de funcionamiento de YARN.

Cuando un cliente o aplicación quiere ejecutarse, el *ResourceManager* se encarga de activar un *ApplicationMaster* en un nodo, y éste se encargará de esa aplicación; se creará un *ApplicationMaster* por cada aplicación que quiera ejecutarse. En el esquema puede verse que hay dos aplicaciones cliente y por ende dos *ApplicationMaster* creados.

El *ApplicationMaster* junto con el *Schedule* (que está dentro del *ResourceManager*), irán arrancando y planificando en los distintos nodos, los *containers*, que son un recurso de memoria y CPU donde se ejecuta la aplicación cliente. El *ApplicationMaster* coordinará los *containers* de la aplicación, y solicitará recursos al *ResourceManager* cuando los necesite. Por otra parte, el *NodeManager* informará al *ResourceManager* del estado de nodo.

Como hemos podido comprobar, el YARN no es más que un gestor de recursos, que no hace ningún trabajo de aplicación, si no que se encarga de gestionar todos los recursos del *cluster*.

2.2.2. HDFS

El componente que Hadoop implementa para el almacenamiento y gestión de los datos es el *Hadoop Distributed File System (HDFS)* [4] [6]. Este sistema divide los ficheros en bloques, y almacenará copias duplicadas de esos bloques en los distintos nodos del *cluster*. Algunas de las características principales de HDFS son:

1. El almacenamiento es tolerante a fallos sobreviviendo a éstos sin perder datos. Si un nodo falla, al haber copias duplicadas, el trabajo se redistribuye entre los demás nodos del *cluster*.
2. Almacena gran cantidad de datos. Cada fichero se divide en bloques de 64 o 128 megabytes almacenándose varias copias de éstos, en varios nodos.
3. Es un sistema fácilmente escalable. Según va aumentando el volumen de datos, se pueden ir agregando nodos, sin tener que realizar ningún ajuste especial a la estructura inicial del *cluster*.

Una estructura Hadoop, se compone de un nodo maestro o *NameNode*, que actúa como maestro de datos; solo contiene metadatos. El resto de los nodos son los esclavos o *DataNode*, que contienen los datos propiamente dichos.

2.2.2.1. Escritura en HDFS

Para almacenar los datos, el nodo maestro (*NameNode*) divide estos datos en bloques de tamaño fijo, normalmente suelen ser de 64 o 128 MB, y posteriormente son almacenados de la forma más equilibrada posible, en cada uno de los nodos esclavos (*DataNode*).

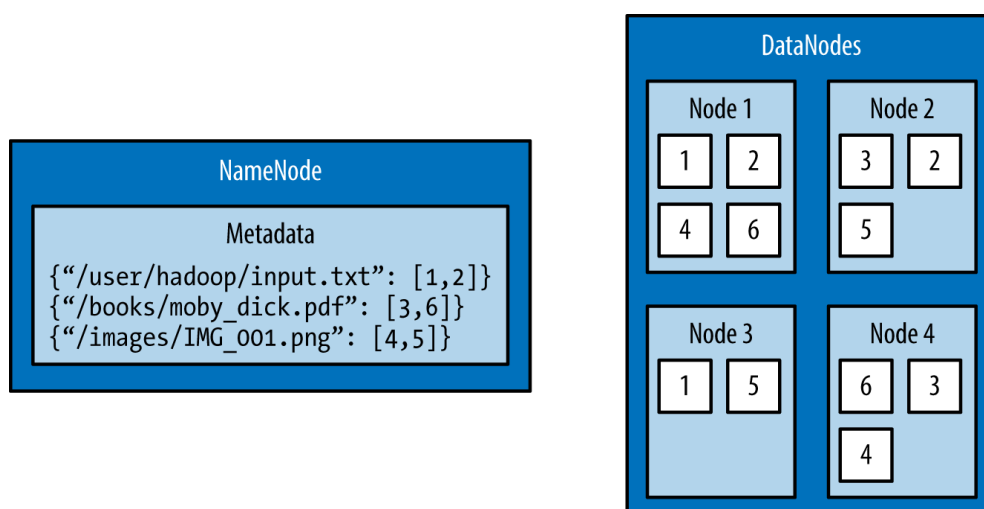


Ilustración 5 – Ejemplo de "cluster" formado por un maestro y cuatro esclavos [14]

En la Ilustración 5, podemos ver que el *NameNode* o nodo maestro contiene el mapeado de los ficheros con sus correspondientes números de bloques. Éstos son escritos en los *DataNode* o nodos esclavos. Vemos que, en este caso, se escriben dos copias de cada bloque repartidos en los cuatro nodos.

2.2.2.2. Lectura en HDFS

En primer lugar, el cliente hace una petición de lectura al *NameNode*; podemos verlo gráficamente en el siguiente esquema [4] .

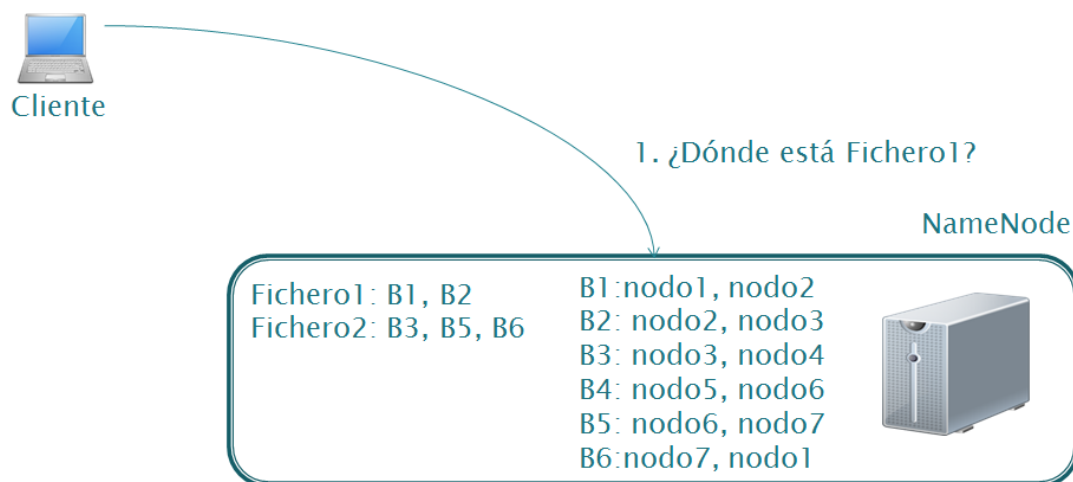


Ilustración 6 - Petición de lectura al NameNode

En segundo lugar, el *NameNode* comprueba en su mapa de bloques, donde se encuentran los que pertenecen al archivo que se está solicitando, devolviéndole el resultado al cliente, tal y como se muestra en la Ilustración 7 [4].

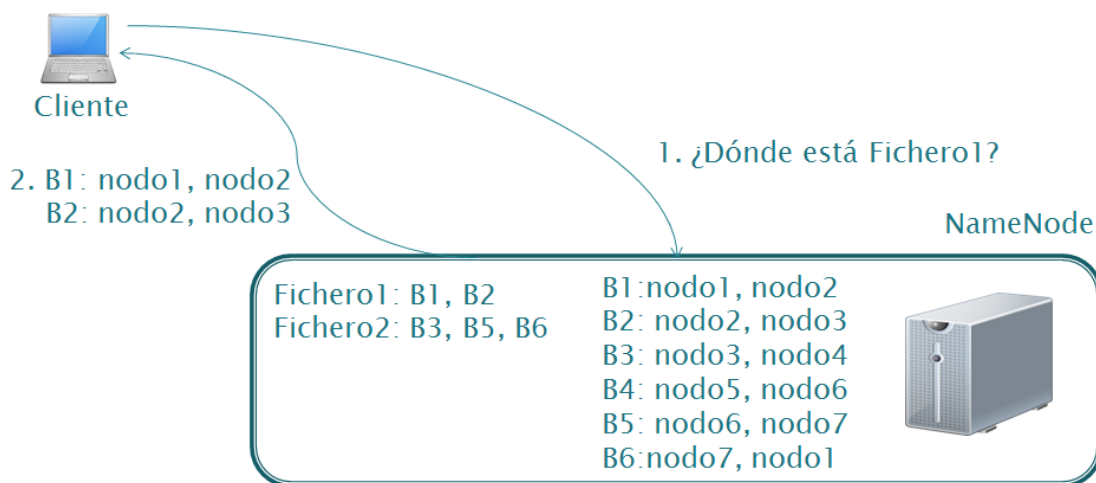


Ilustración 7 - Respuesta del NameNode al cliente

Por último, el cliente solicita a los nodos que contienen los bloques del fichero, que se los envíe. Los nodos (*DataNode*) le envían los bloques al cliente sin pasar por el *NameNode* tal y como se ve en la Ilustración 8 [4].

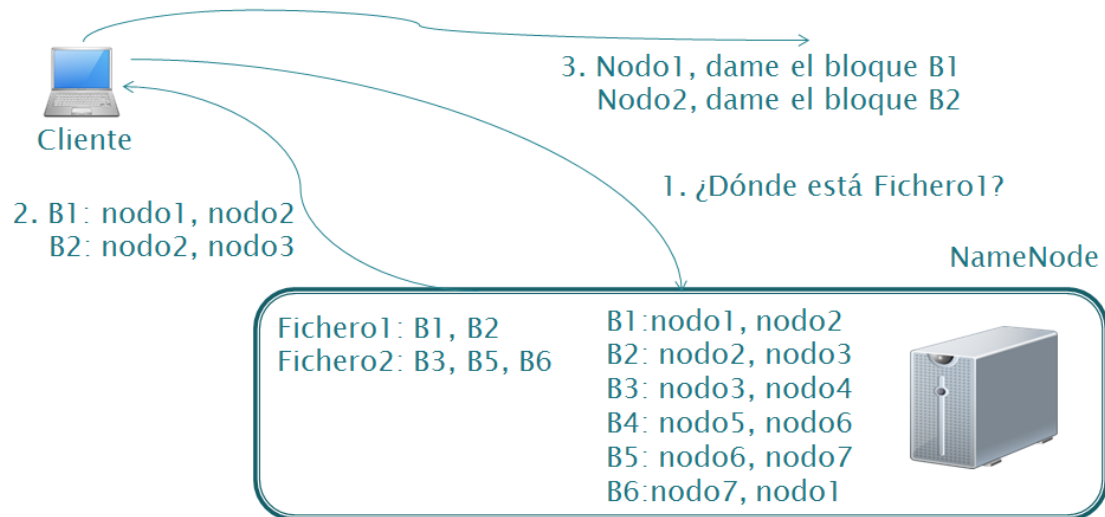


Ilustración 8 - Petición de bloques a los DataNode.

En resumen, el *NameNode* almacena los metadatos, que pueden ser por ejemplo el espacio de nombres o árbol de directorio del sistema, y el mapeo de bloques, entre otros. Este nodo maestro suele replicarse en un *cluster*, dada su importancia. Cuando una aplicación cliente quiere leer o escribir, el *NameNode* le indicará donde localizar la información en los *DataNodes*. A continuación, podemos ver un esquema que engloba todo lo visto anteriormente [15]:

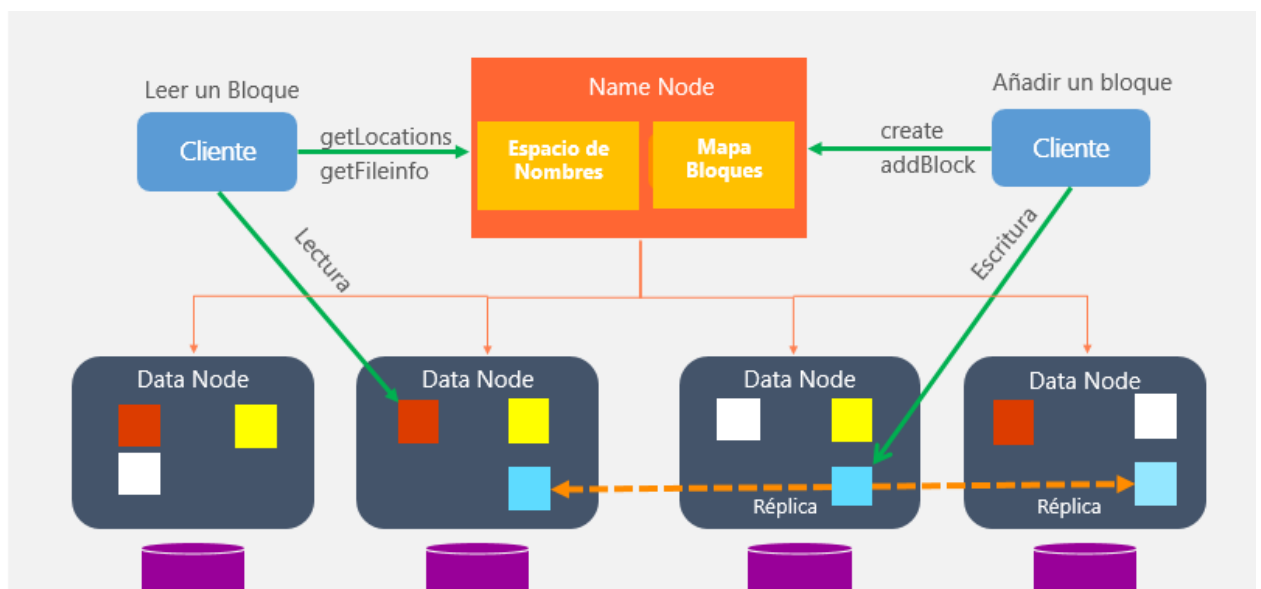


Ilustración 9 - Lectura y escritura en HDFS.

2.2.3. MapReduce

Entramos en la parte de procesamiento de los datos de Hadoop, y lo que va a ser el objeto principal de este trabajo. Pero antes veamos qué es y cómo funciona este paradigma.

La mayor parte de las herramientas de consulta están diseñadas para realizarlas con datos indexados y estructurados, pero esta solución no es útil cuando tenemos que manejar datos semiestructurados como pueden ser textos, o sin estructurar como pueden ser elementos multimedia (fotos, vídeos, sonidos, etc.). Para poder responder a una consulta o hacer un análisis exhaustivo con este tipo de datos, donde el tamaño de éstos puede llegar a ser de terabytes o petabytes, y que generalmente son cargados de forma masiva, usamos **MapReduce**.

MapReduce [4] [6] es un algoritmo de procesamiento paralelo y distribuido, que implementa procesos aplicando la filosofía del divide y vencerás. En vez de ejecutar una tarea de forma secuencial, lo que hace es que la trocea y la manda a cada uno de los nodos (*DataNodes*) del *cluster* para su procesamiento.

Cuando implementamos una aplicación dentro del marco *MapReduce*, hemos de tener en cuenta que éste, automatiza la paralelización de las ejecuciones y la distribución de las tareas en los nodos. Consta de varias fases en su ejecución que son:

- *Fase Map*: se pueden ejecutar varias tareas *Map* de forma paralela y cada una de ellas atacará un nodo de datos HDFS, ejecutándose ésta, en el nodo donde está el bloque de datos, siempre que sea posible. La salida del *Map* van a ser siempre parejas **<clave, valor>**.
- *Fase Shuffle & Sort (barajar y ordenar)*: ordena y consolida los datos intermedios o temporales que se generan en la fase *Map*.
- *Fase Reducer*: opera sobre los datos intermedios ordenados y barajados, es decir lo que ha salido del *Map*, produciendo los resultados finales también en formato **<clave, valor>**.

Podemos ver en la siguiente ilustración, un esquema de las fases del proceso *MapReduce* [15]:

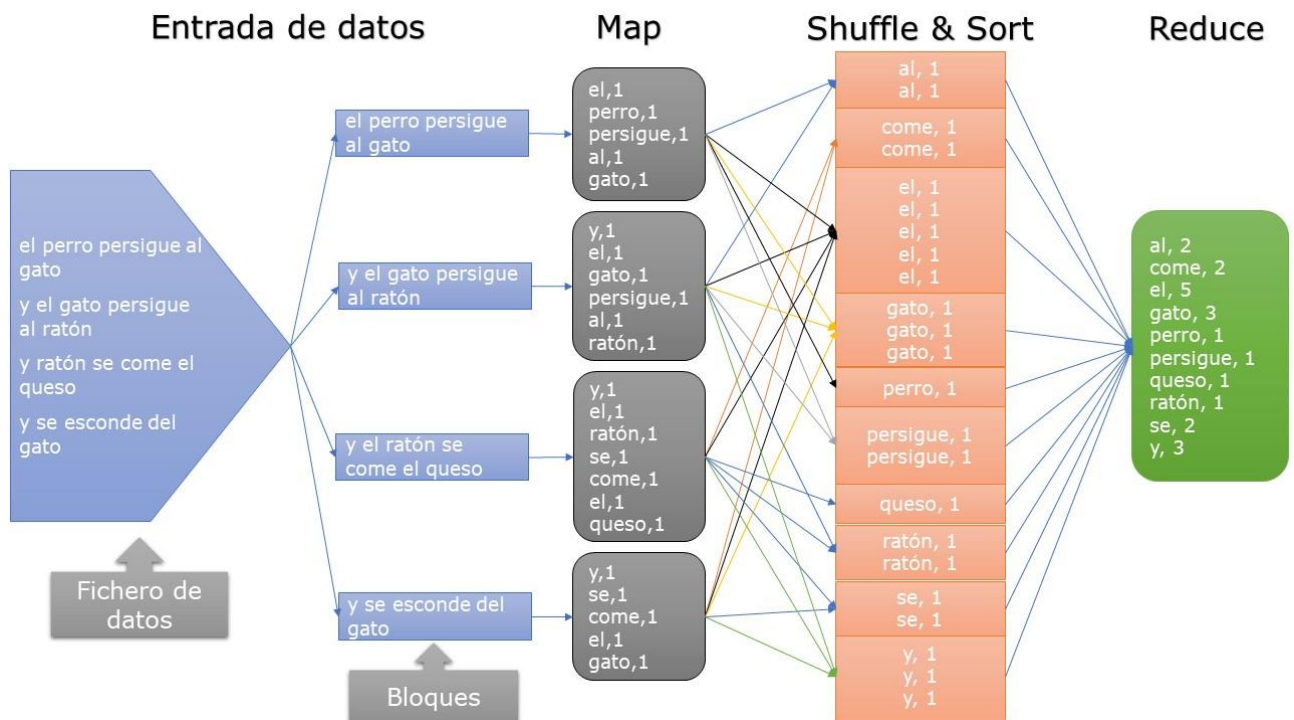


Ilustración 10 - Ejemplo de funcionamiento de MapReduce.

El esquema muestra el típico ejemplo para este paradigma: el contador de palabras. Este ejemplo trata de contar las palabras que tiene un fichero de texto siguiendo los siguientes pasos:

1. Primeramente, se ejecutan los procesos *Map* sobre cada bloque que forma el fichero. En la medida de lo posible, Hadoop se encarga de ejecutar el proceso en los mismos nodos donde se almacenan los bloques, con el objetivo de que haya menos trasiego de información a través de la red. Es importante destacar, que el código programado puede estar ejecutándose en más de una máquina al mismo tiempo. Una vez se ha realizado la fase de *Map*, se generan una serie de pares <clave, valor>, qué en el caso del esquema, la clave será la palabra y el valor será la unidad.
2. En segundo lugar, la fase *Shuffle & Sort* ordena todos los pares <clave, valor>, agrupando los que tengan la misma clave, que recordemos es la palabra.
3. Por último, la fase *Reduce* toma la salida de la anterior fase, y suma los valores de cada clave, obteniendo el resultado final, que es la suma del total de apariciones de la palabra que hay en el fichero.

Tan solo hemos visto un pequeño ejemplo, pero podemos imaginar lo extremadamente potente que puede llegar a ser este paradigma, pero no va a ser fácil encontrar soluciones a todos los problemas que se nos planteen. Se ha de tener claro que *MapReduce* es un marco de trabajo, y las soluciones que encontremos han de ajustarse a este marco, y esto a veces será bastante complicado, ya que descubrir cómo resolver un problema con las restricciones propias del entorno, requerirá de un cambio de pensamiento [7].

Podemos deducir con todo lo anteriormente expuesto, que programar en MapReduce no es fácil, puede llegar a ser confuso, tedioso y a veces caótico. Un desarrollo en este paradigma es un desafío en el sentido de que hay que intentar desarrollar un código limpio y fácil de mantener, sobre todo, porque este código va a ser ejecutado en cientos o miles de nodos para hacer cálculos sobre terabytes e incluso *petabytes* de datos. De ahí que elegir un diseño correcto para resolver un problema con MapReduce, sea muy importante si lo vemos desde el punto de vista del rendimiento del sistema.

Dicho todo esto, puede ser útil disponer de plantillas que ayuden y guíen a resolver problemas de manipulación de datos, y que estos problemas sean comunes en *MapReduce*. Esas plantillas son los patrones de diseño, que son herramientas reutilizables y generales, que permiten que el desarrollador no tenga que perder tiempo en descubrir cómo superar un obstáculo que suele ser habitual. Estos patrones de diseño los tenemos en Java [7], y tras una búsqueda diversa en libros, documentación, webs y repositorios, no ha sido posible encontrar estos patrones de diseño MapReduce desarrollados en Python.

2.3. Ecosistema de Hadoop

Hadoop dispone de un amplio abanico de aplicaciones que complementan y sacan el máximo partido a la arquitectura distribuida, ampliando las funcionalidades de este sistema. Conozcamos algunas de estas aplicaciones que podrían ser útiles, para nuestros objetivos en este trabajo.

Hive [16]

Esta herramienta facilita la lectura, escritura y administración de grandes conjuntos de datos que residen en el almacenamiento distribuido mediante SQL. Permite acceder a HDFS como si fuera una base de datos. Dispone de un lenguaje muy parecido a SQL llamado HiveSQL. Esto simplifica bastante el desarrollo y la gestión con Hadoop al poder hacer programas muy sencillos y rápidos, ya que cuando se ejecuta algo, esta herramienta

convierte las cláusulas SQL en instrucciones MapReduce, ya que esta capa está por debajo de Hive.

HBASE [17]

Es el sistema de almacenamiento no relacional de Hadoop, pensada para gestionar una gran cantidad de datos. Apache HBase es una base de datos de código abierto, distribuida, versionada, no relacional, de tipo columnas y escalable. Esta aplicación permite el almacenamiento de tablas muy grandes (miles de millones de filas X millones de columnas) sobre un *cluster* Hadoop, ajustándose a un sistema de almacenamiento distribuido para datos estructurados. Puede sernos útil cuando necesitemos acceso aleatorio de lectura / escritura en tiempo real a nuestra infraestructura *Big Data*.

PIG [18]

Pig es un lenguaje de alto nivel de tipo *scripting* para analizar grandes volúmenes de datos. Permite trabajar en paralelo, lo que lo hace ideal para gestionar gran cantidad de información. Podemos afirmar también que es como un compilador que genera comandos MapReduce.

Sqoop [19]

Sqoop es una herramienta diseñada para transferir de manera eficiente datos masivos entre Apache Hadoop y almacenes de datos estructurados como bases de datos relacionales Oracle, SQL Server, MySQL, etc.

Flume [20]

Flume trabaja con gran cantidad de datos de tipo texto. Es un servicio distribuido y eficiente para distribuir agregar y recolectar grandes cantidades de información. La característica principal es que es la alternativa al comando *put* de HDFS que usamos para subir ficheros. Con Flume es sencillo mover grandes cantidades de información al sistema de ficheros Hadoop. Es muy útil para cargar ficheros de logs, paquetes de twitter, etc. Dispone de una arquitectura adaptada al flujo de datos o *streaming*, que se caracteriza por su potencia y su capacidad de adaptabilidad según las necesidades del usuario. Esta herramienta junto con *Sqoop*, son las ideales para cargar datos en nuestra infraestructura *BigData*.

2.4. Resumen del capítulo

En este capítulo nos hemos adentrado en el mundo Hadoop, donde hemos tenido oportunidad de conocer sus características principales y el objetivo de su uso. Hemos conocido algunas de las aplicaciones del proyecto Apache que lo complementan, y los elementos principales que componen su estructura interna, entre ellos, hemos abordado con detalle el elemento MapReduce, que será el paradigma bajo el cual desarrollaremos nuestros patrones, pero antes de esto, en el próximo tema, nos adentraremos en las herramientas necesarias para desarrollar programas bajo ese paradigma, y estudiaremos las tecnologías para el despliegue de toda la arquitectura que implica Hadoop.

Capítulo 3. Estado del arte

A lo largo de este tema, veremos de que herramientas se disponen actualmente para trabajar las distintas tecnologías que nos permitirán conseguir los objetivos planteados. Por un lado, abordaremos el lenguaje de programación disponible junto con sus librerías, entornos de trabajo o *frameworks*, programas de gestión de versiones y repositorios de software. Por otro lado, nos adentraremos en las herramientas para realizar el despliegue de tecnologías *Big Data*, usando aplicaciones de virtualización existentes, y haciendo una comparativa de éstas.

3.1. Herramientas usadas en el marco de programación MapReduce

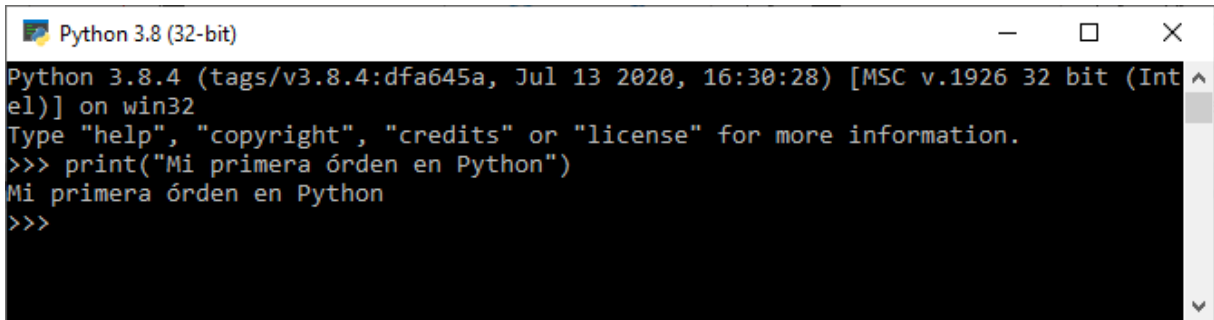
Las herramientas que se han usado para el desarrollo y pruebas de los patrones han sido: Python como lenguaje de programación, librerías MRJob, Git, GitHub, y para la instalación y preparación de todo el entorno de trabajo, nos hemos ayudado de la aplicación *Anaconda* [21] donde también lleva incluidos, entre otros, los editores Jupyter Lab y Jupyter Notebooks.

3.1.1. Lenguaje de programación Python [22]

El lenguaje de programación Python es un lenguaje de alto nivel que por su simplicidad, fácil aprendizaje y potencia, se ha extendido de una forma vertiginosa haciéndose muy popular en los últimos años, sobre todo en el área de ciencia de datos y el aprendizaje automático.

Entre sus características más destacables es que es multiplataforma, es decir, cualquier programa creado en este lenguaje puede funcionar en cualquier sistema operativo: Windows, Linux, MacOS, etc. Por otro lado, se trata de un lenguaje que soporta más de un paradigma de programación, ya que se puede programar de forma secuencial, funcional, orientado a objetos, etc. [23]

Para hacer la descarga del intérprete del lenguaje y su instalador correspondiente, podemos hacerlo a través de su página oficial [22]. Una vez finalizada la instalación, se pueden ejecutar sus programas mediante líneas de órdenes en su propio entorno, como puede apreciarse en la Ilustración 11.



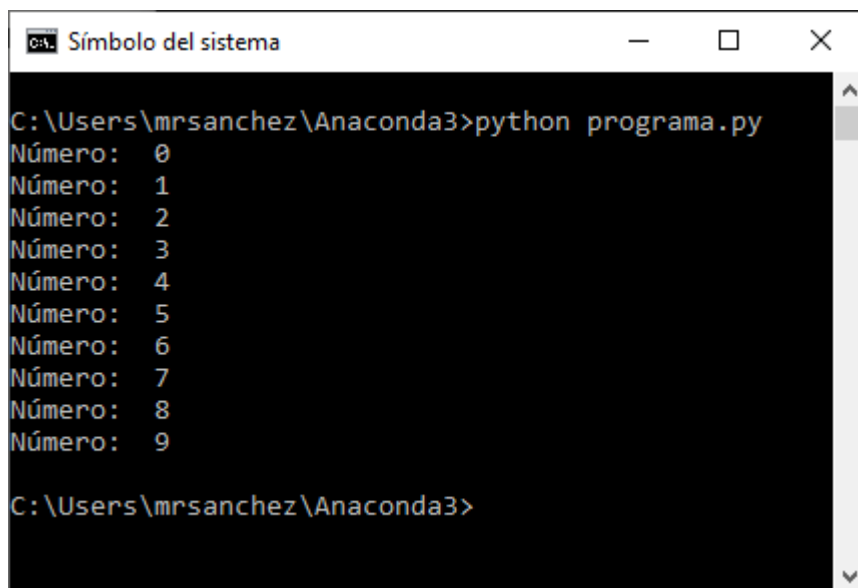
```
Python 3.8 (32-bit)
Python 3.8.4 (tags/v3.8.4:dfa645a, Jul 13 2020, 16:30:28) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Mi primera orden en Python")
Mi primera orden en Python
>>>
```

Ilustración 11 - Intérprete de Python y ejemplo de ejecución de una orden o instrucción.

Los programas también pueden ser escritos siguiendo una secuencia de órdenes o *script* (Ilustración 12), que puede ser posteriormente ejecutado mediante comando en una terminal como podemos ver en la Ilustración 13.

```
#Programa que imprime los 10 primeros números
for i in range(10):
    print("Número: ",i)
```

Ilustración 12 - Programa en Python llamado "programa.py"



```
Símbolo del sistema
C:\Users\mrsanchez\Anaconda3>python programa.py
Número: 0
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
Número: 6
Número: 7
Número: 8
Número: 9
C:\Users\mrsanchez\Anaconda3>
```

Ilustración 13 - Ejecución en una terminal de un programa en Python.

Existen diversos entornos o editores que facilitan el desarrollo de programas y aplicaciones en Python. Jupyter Notebook [24] es uno de estos entornos o *framework*, que está bastante extendido por su interfaz intuitiva, su facilidad de uso, y su comodidad a la hora de trabajar, ya que permite combinar distintas acciones como pueden ser insertar comentarios, ejecutar comandos del sistema operativo, escribir los programas y ver los resultados de su ejecución en un mismo entorno web. En la siguiente ilustración podemos ver un ejemplo.

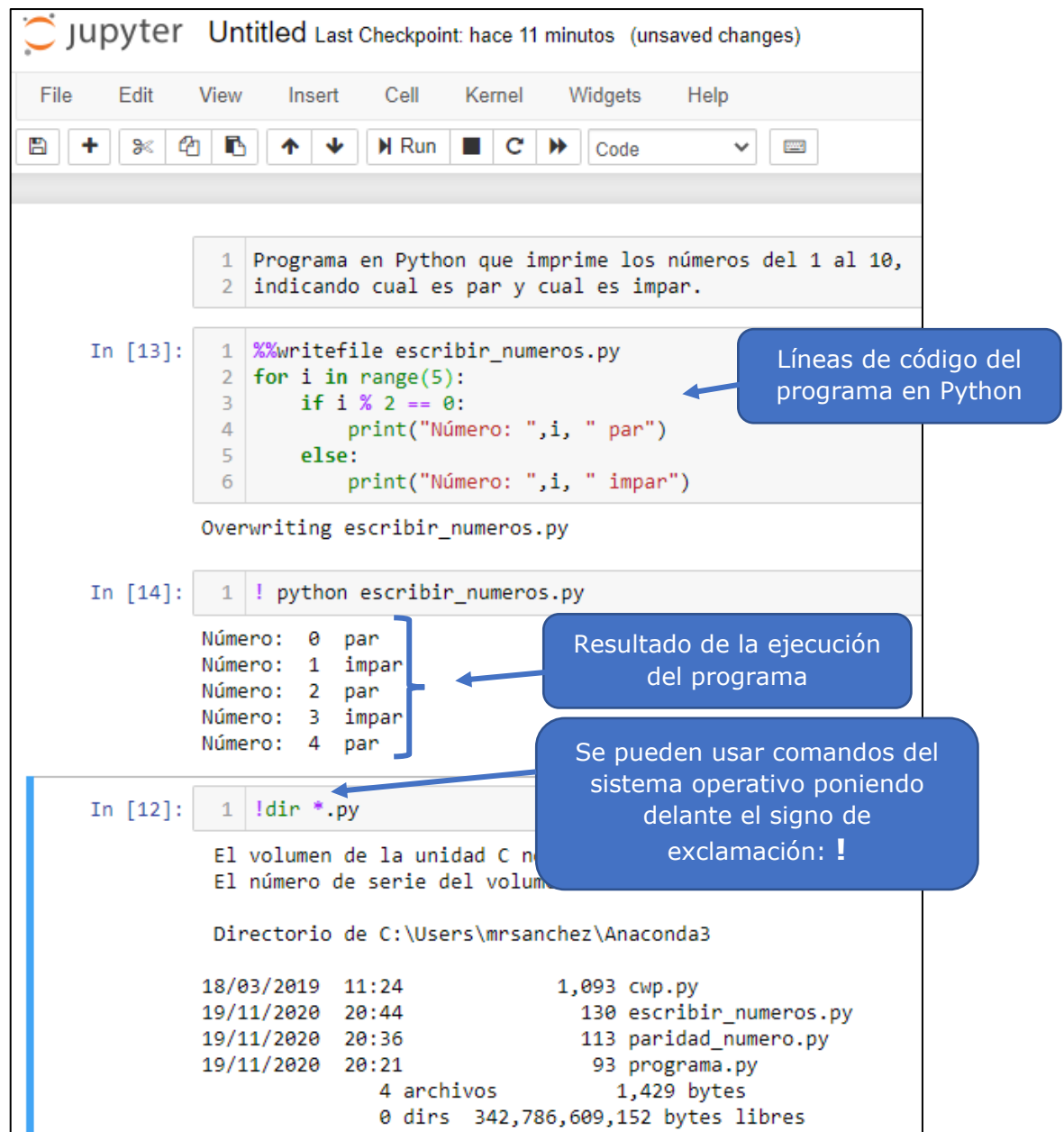


Ilustración 14 - Ejemplo de uso de entorno Jupyter Notebook.

3.1.2. Python: librerías MRJob

Cuando realizamos un programa bajo el paradigma *MapReduce*, hemos de recordar que tiene que ejecutar un primer trabajo que es el *Map*, y un segundo trabajo que es el *Reduce*. Para conseguir esto, se suele desarrollar dos archivos, uno con las tareas que hay que hacer en la parte del *Map*, y el otro con las tareas que se han de realizar en la parte del *Reduce*. Veámoslo con el típico ejemplo del contador de palabras [25].

⇒ Escribimos las líneas de código del Map:

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
    line=line.strip()
    words=line.split()
    for word in words:
        print ('%s\t%s' % (word,1))
```

Ilustración 15 - Fichero "map.py", que contiene las tareas de Map para contar palabras.

⇒ Escribimos las líneas de código del *Reduce*:

```
#!/usr/bin/env python

from operator import itemgetter
import sys

last_word = None
last_count = 0
cur_word = None

for line in sys.stdin:
    line = line.strip()

    cur_word, count = line.split('\t', 1)

    count = int(count)

    if last_word == cur_word:
        last_count += count
    else:
        if last_word:
            print '%s\t%s' % (last_word, last_count)
            last_count = count
            last_word = cur_word

if last_word == cur_word:
    print '%s\t%s' % (last_word, last_count)
```

Ilustración 16 - Fichero "reduce.py" que contiene las tareas que hay que hacer en el Reduce.

Para ejecutar esta tarea dentro del marco, hemos de tener en cuenta que la salida del *Map*, ha de ser la entrada del *reduce*. Podemos ejecutar fuera del *cluster* Hadoop mediante la instrucción:

```
cat fichero.txt | python map.py | sort | python reduce.py > resultado.txt
```

y si todo funciona correctamente, podemos probarlo en el mismo *cluster* mediante la instrucción:

```
hadoop jar /usr/lib/Hadoop-mapreduce/Hadoop-streaming.jar -file map.py -mapper
map.py -file reducer.py -reducer reducer.py -input /tmp/fichero.txt -output
/tmp/resultado
```

Es importante recordar que para poder ejecutar en el *cluster* los programas MapReduce, debemos subir al HDFS todos los ficheros necesarios [25].

En vista de todo lo anteriormente expuesto, para facilitarnos y hacernos algo más cómoda la implementación de programas bajo el paradigma *MapReduce*, existe la librería MRJob [26] de Python, donde algunas de sus principales ventajas son:

- 1- No es necesario programar el *Map* y el *Reduce* en distintos archivos. Se crea una clase donde se introducen, entre otros, los métodos *mapper* y *reducer* en un solo archivo donde podremos desarrollar todo el programa, o en nuestro caso, los prototipos de los patrones.
- 2- Podremos probar y depurar el código que desarrollemos sin necesidad de tener *Hadoop* instalado, es decir, desde una máquina local y desde el mismo *Jupyter Notebook* por ejemplo, podremos ver el comportamiento en la ejecución de los prototipos que desarrollemos.
- 3- La interfaz de *MRJob* es coherente con todos los entornos que admite. No importa si estamos trabajando localmente, en la nube o en el *cluster*, el código Python no variará. Lo que desarrollemos se ejecutará en cualquier entorno.
- 4- MRJob maneja los datos de entrada y de salida, desde y hacia el *cluster*, no es necesario implementar *scripts* para cargar archivos ni instalar dependencias.
- 5- MRJob facilita la depuración. De forma local se puede ejecutar una implementación *MapReduce* y obtener un rastreo en la consola en lugar de un archivo de registro.
- 6- MRJob nos permite variar el formato de los datos en cualquier punto del programa: cuando entran, durante la ejecución y/o cuando salen los datos.

Como desventaja, no resulta sencillo encontrar documentación o manuales que expresen con claridad y detalle el funcionamiento de esta librería. Sí que es cierto que nos podemos encontrar algunos apartados en libros [14], y que existe un manual básico [27] en el que nos muestra todo lo que se puede llegar hacer con las *MRJob*, pero no se llega a profundizar en detalle, y los ejemplos que podemos encontrar en repositorios, blogs y foros, no están muy claros ni organizados, y eso es precisamente lo que buscamos con este trabajo: proporcionar un repositorio de patrones de diseño Mapreduce con la librería MRJob.

Siguiendo con el ejemplo del contador de palabras, vamos ahora a desarrollarlo usando las MRJob:

```

from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()

```

Ilustración 17 - Ejemplo del contador de palabras con MRJob [14].

Como puede comprobarse, en pocas líneas se ha desarrollado el contador de palabras. El método *mapper* toma una clave y un valor como argumentos (que en este caso la clave se ignora, y una sola línea del texto se toma como valor) y mediante una iteración de los valores de la línea, producen tantos pares clave-valor como palabras haya en dicha línea, produciendo una clave: la palabra, y un valor: la unidad. El método *reducer* toma el par clave-valor producido por el *mapper*, agrupa las palabras que sean iguales, y las suma, devolviendo el número de palabras del texto.

Vemos que el trabajo está definido por una clase que hereda de *MRJob*. La clase contiene los métodos que definen los pasos de un programa *MapReduce*. Un paso consta de los siguientes métodos: *mapper*, *combiner* y *reducer*. Todos son opcionales, pero siempre ha de haber uno de ellos. Se deduce que podemos definir un programa *MapReduce* con uno o más pasos, siendo la salida de un paso, la entrada de otro.

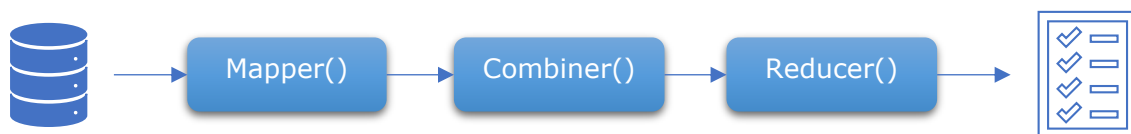


Ilustración 18 – Esquema que define un paso MapReduce con las MRJob.

Un paso puede estar compuesto por un *mapper* y un *reducer*, o solo un *reducer* o solo un *mapper*, o un *mapper* y un *combiner*, etc. [27, p. 5]

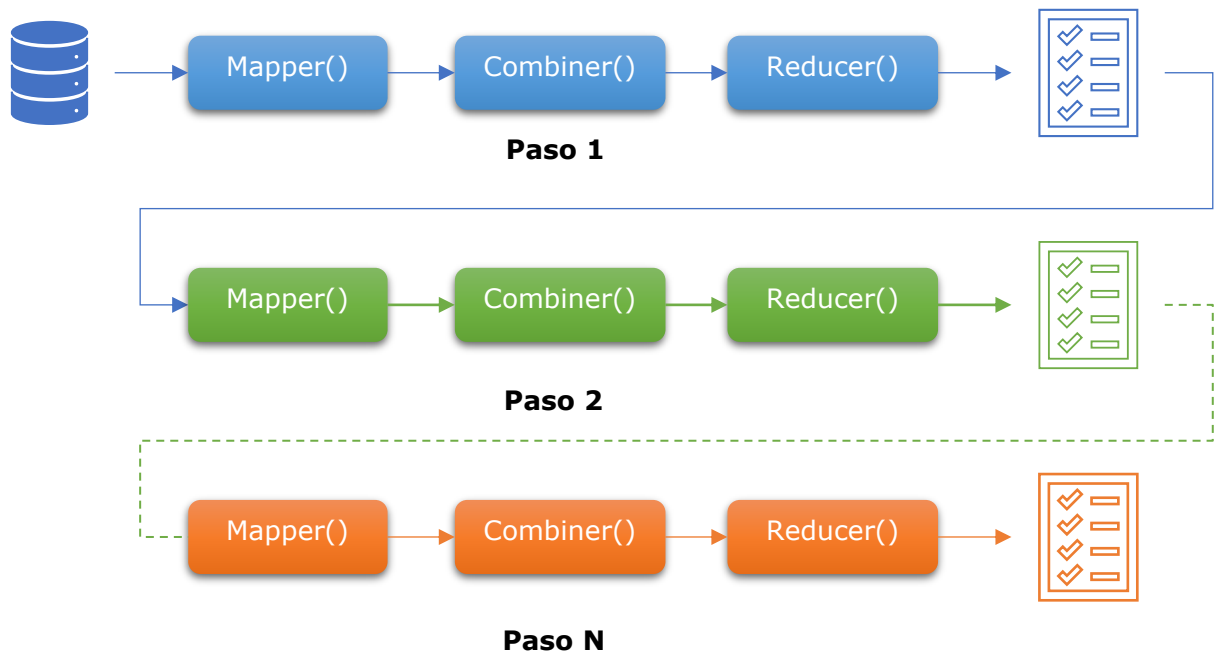


Ilustración 19 - Ejecución de un programa MapReduce por pasos.

El *combiner*, es un método que nos permite realizar operaciones intermedias entre el *mapper* y el *reducer*. Suele devolver los datos en formato lista si no se le indica lo contrario. Las últimas dos líneas de un programa *MapReduce* pasan el control de la ejecución a MRJob; es fundamental que se pongan siempre, si no, el programa no funcionará. [27, p. 5]

```
if __name__ == '__main__':
    MRWordCount.run()
```

Por defecto, la salida de los datos procesados, van a ser en formato JSON; una de las características más interesantes de las *MRJob* son los protocolos de entrada, protocolos internos y protocolos de salida [27, p. 14]. Lo interesante de esto, es que podemos modificar como queremos que entren los datos, en qué formato queremos manejarlos internamente, y como queremos que salgan. Para controlar esto, hemos de importarnos los protocolos que necesitamos e indicar justo después de la declaración de la clase, alguna de estas instrucciones:

INPUT_PROTOCOL = [protocolo de entrada]

INTERNAL_PROTOCOL = [protocolo interno]

OUTPUT_PROTOCOL = [protocolo de salida]

Por ejemplo, si necesitamos que del par <clave, valor>, una vez procesados, salgan en formato filas, tendríamos que establecer el protocolo *RawValueProtocol* de la siguiente forma:

```
from mrjob.protocol import RawValueProtocol
class nombre_clase(MRJob):
    OUTPUT_PROTOCOL = RawValueProtocol
.....
```

De esta forma, en la salida, los valores los mostrará en formato líneas y no en JSON o <clave-valor>.

3.1.3. Git y GitHub

Git [28] es un sistema de control de versiones, que nos permite auditar código, controlar los cambios en el tiempo, añadir etiquetas para las distintas versiones que vayamos creando, y para el trabajo colaborativo en proyectos de desarrollo.

Por otro lado, disponemos de GitHub [29], que es una red social que, mediante las herramientas de *Git*, podemos subir nuestros proyectos, creando y actualizando repositorios de código versionado. Estos repositorios podrán ser accesibles al público o de forma privada para que sean copiados, compartidos, o mejorar los desarrollos directamente en GitHub.

En este trabajo se ha creado un repositorio en GitHub en el que se han guardado los desarrollos de los patrones, junto a los conjuntos de datos usados para los prototipos. Puede ser accedido por cualquier usuario a la dirección:

https://github.com/manursanchez/TFG_Manuel_R

3.2. Tecnologías para el despliegue de infraestructuras *Big Data*.

En este tema sentaremos las bases de conocimiento en algunas de las tecnologías de virtualización, repasando lo que son las máquinas virtuales, y adentrándonos en el concepto y herramientas de lo que es la virtualización ligera.

Virtualización

Para poder sacar el máximo partido al soporte físico de un computador, se han creado aplicaciones como *VMWare* o *Virtual Box*, que permiten la creación de varios computadores virtuales donde se pueden hacer distintas configuraciones físicas, e instalar el software necesario. Estas máquinas virtuales pueden interactuar entre ellas y compartir recursos del ordenador anfitrión. La virtualización permite usar menos servidores, ayuda a la

escalabilidad, reduce los costos de infraestructura y mantenimiento, y también reduce el consumo de energía [30]. A los programas que permiten la virtualización también se les llama *hipervisores*.



Ilustración 20 - Computador con máquinas virtuales (MV)

Nos podemos encontrar con la desventaja de tener limitadas las máquinas virtuales debido a las capacidades del ordenador anfitrión, ya que a más número de máquinas, más han de compartir recursos, habrá más problemas de espacio, el rendimiento decaerá y la escalabilidad puede complicarse.

Podríamos plantearnos la siguiente reflexión: si pudiéramos reducir los recursos que las máquinas virtuales consumen en un computador (espacio del S.O., espacio en memoria, procesos, etc.) podríamos aprovechar mucho más estos recursos físicos, es decir, si disponemos de un espacio en disco, donde exista una aplicación con lo mínimo para funcionar e independiente del sistema operativo, tal vez se pueda sacar más partido al recurso físico, aumentando la seguridad, la escalabilidad y el rendimiento. La *virtualización ligera* puede ser una la solución a la reflexión planteada.

Virtualización ligera

Cuando hablamos de virtualización ligera, nos referimos al concepto abstracto de contenedor, o lo que es lo mismo, de Docker.

Docker es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones dentro de contenedores. Estos, empaquetan de forma ligera todo lo necesario

para que uno o varios procesos o aplicaciones funcionen independientemente del entorno donde se quiera desplegar, es decir, que podemos empaquetar una base de datos, un entorno de desarrollo, una aplicación o servidor web, etc. y montarlo y ejecutarlo en cualquier plataforma como puede ser Linux, Windows, Mac, AWS, Azure, etc. [31]

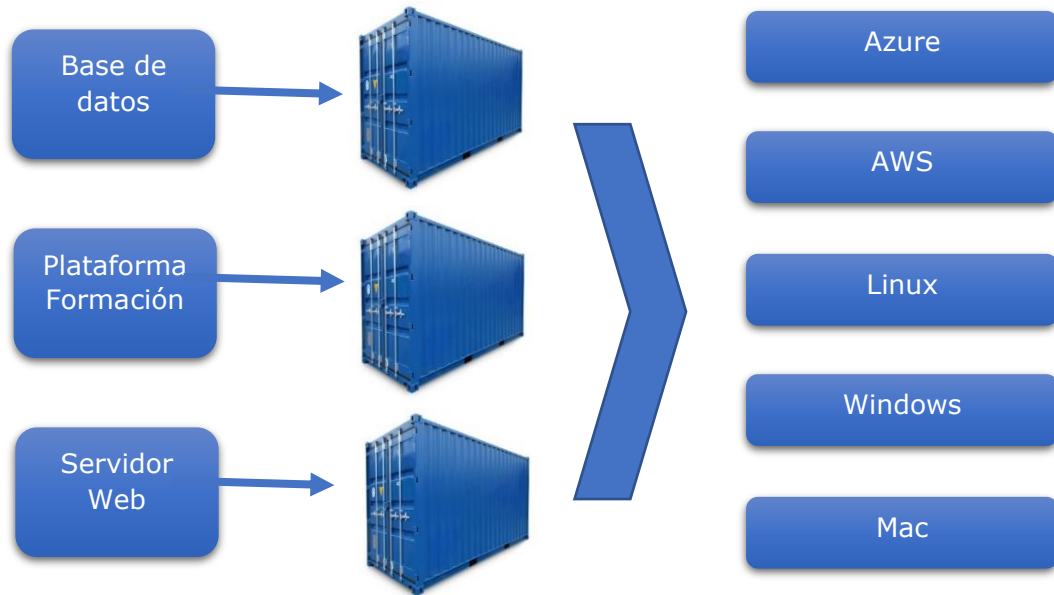


Ilustración 21 - Filosofía de los contenedores

Podemos pensar que para eso ya existen aplicaciones de virtualización vistas anteriormente, que permiten crear máquinas virtuales desde las que podemos virtualizar aplicaciones y/o servicios. La filosofía es parecida, pero la diferencia entre los contenedores y máquinas virtuales es que los primeros no necesitan un sistema operativo para funcionar, y tan solo tienen los archivos necesarios para que la aplicación funcione. Sin embargo, cuando usamos una aplicación de virtualización, hemos de instalar todas las aplicaciones y sistema operativo para que pueda funcionar.

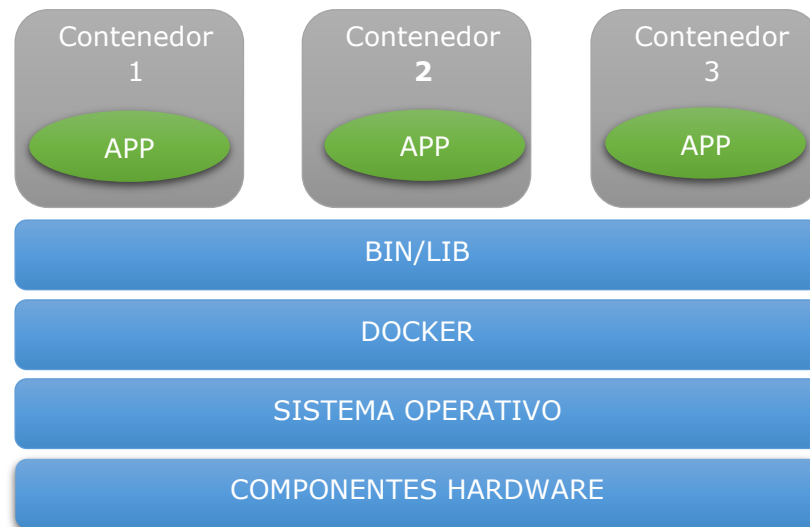


Ilustración 22 - Computador con contenedores

Algunas de las ventajas que tienen los contenedores son:

- Fácilmente portables e instalables, ya que se han estandarizado.
- Las aplicaciones que se ejecutan en un contenedor son más seguras, ya que entre otras cosas es más difícil acceder a éstos.
- Ahorro de espacio que puede ser dedicado para desplegar más contenedores con otras aplicaciones.
- Los contenedores son más ligeros que una máquina virtual, ya que solo contiene lo necesario para ejecutar una aplicación.

En vista de esto, es posible crear un *cluster* Hadoop, en el que dentro de una máquina física, podamos tener varios nodos, y cada uno de ellos dentro de un contenedor.

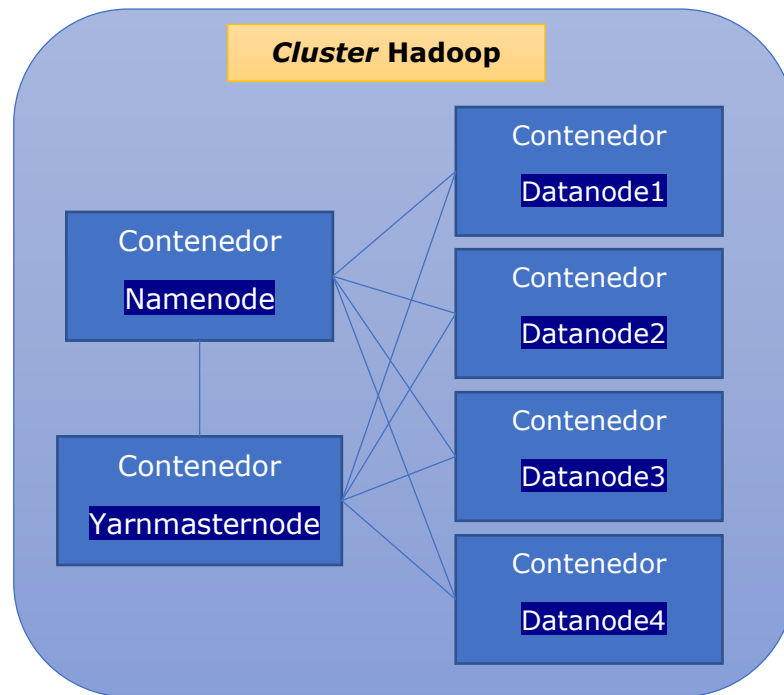


Ilustración 23 - Esquema de un cluster Hadoop en contenedores Docker.

Pongamos el ejemplo de que necesitemos crear un cluster Hadoop, éste va a estar limitado por el número de máquinas que podamos crear; tendremos que tener un sistema operativo con todos los programas de aplicación y con las configuraciones del cluster en cada una de las máquinas virtuales, pero si pudiéramos prescindir de todo lo que no necesitamos para la creación del cluster Hadoop, y que solo tengamos lo justo y necesario para que éste funcione. Mejoraría el rendimiento del sistema, y podríamos escalar más fácilmente entre otras ventajas si ya tenemos un sistema operativo con todo lo necesario para que el computador

Docker compose

Docker Compose permite definir y compartir aplicaciones de varios contenedores; se crea un archivo donde se definen los servicios, y con un solo comando se ponen en marcha, se detienen, o se realizan diversas funciones. [32]

Como ya vimos anteriormente, un contenedor Docker empaqueta todo lo necesario para que uno o más procesos funcionen: código, herramientas del sistema, bibliotecas dependencias, bases de datos, etc. [31]. Lo que hace *Docker Compose* es individualizar y separar en contenedores los servicios que conforman una aplicación, es lo que se llama la creación de microservicios. Estos son gestionados por *Docker Compose*, y éste se encarga de orquestar todos estos microservicios para que la aplicación funcione.

Pongamos el ejemplo de montar una plataforma de formación con *Moodle* [33]. Una de las formas es que desplaguemos en un contenedor, todo lo necesario para ponerla en marcha. Quedaría como se muestra en Ilustración 24

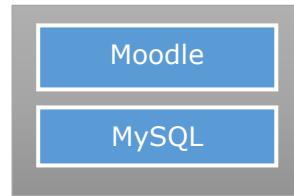


Ilustración 24 - Contenedor Docker con lo necesario para una plataforma Moodle.

Docker Compose nos permite dividir en dos contenedores: uno para *Moodle* y otro para *MySQL* convirtiéndolos en microservicios que serán gestionados y orquestados por la herramienta.

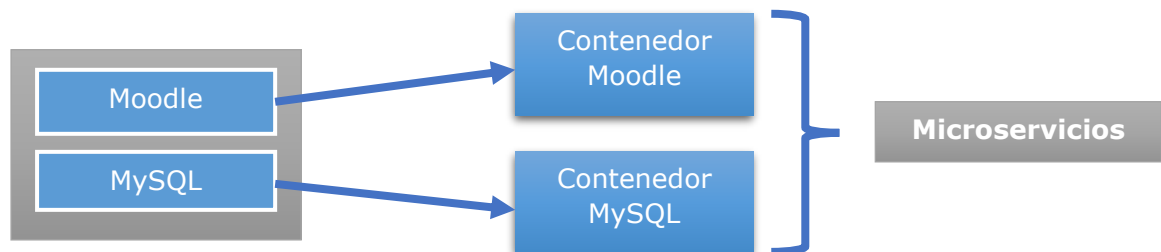


Ilustración 25 - Ejemplo de arquitectura de microservicios

Vemos que cada servicio es un contenedor y lo que se hace es que mediante el archivo de configuración *docker-compose.yml*, se definen las instrucciones para hacer los enlaces y las relaciones entre todos los contenedores de forma sencilla, más que si lo hiciéramos a mano. Por tanto, *Docker Compose* es un frontal contra *Docker* que nos permite gestionar este tipo de arquitecturas de microservicios de una forma simple. [31]

3.3. Resumen de capítulo

A lo largo de este capítulo nos hemos adentrado en las herramientas que usaremos para los desarrollos en MapReduce. Hemos conocido algunas de las características de Python, entre las que destaca su facilidad de adaptación a los diversos paradigmas de programación, y como gracias a su librería MRJob, y a los entornos de desarrollo como Jupyter Notebook, este lenguaje supone una herramienta útil y potente para la implementación de programas bajo el paradigma MapReduce.

Se ha hecho una pequeña incursión en aplicaciones de gestión de versiones como Git, y de desarrollo colaborativo y repositorios con GitHub

Por otro lado, se han estudiado tecnologías usadas para el despliegue de infraestructuras de macrodatos, donde se ha abordado el concepto de virtualización con hipervisores y comparado posteriormente con la virtualización ligera, entrando en los conceptos de contenedores Docker y microservicios con Docker-Compose.

Una vez vista la teoría, es hora de ponerlo en práctica en el próximo capítulo.

Capítulo 4. Despliegue de la arquitectura

Vamos a crear un cluster virtual de nodos, usando virtualización ligera con los contenedores Docker. Lo vamos a desplegar en un computador con Windows 10, y dentro de éste, en una máquina virtual Linux Ubuntu configurada y desplegada en Virtual Box. La agrupación se compondrá de un total de seis nodos. Estos son:

- 1 *namenode*: nodo que manejará todo el grupo de nodos.
- 1 *yarnmaster*: nodo que se encargará entre otros, de los procesos *MapReduce*.
- 4 *datanodes*: nodos donde estará el sistema de almacenamiento HDFS, y que serán usados para la ejecución de procesos *MapReduce*.

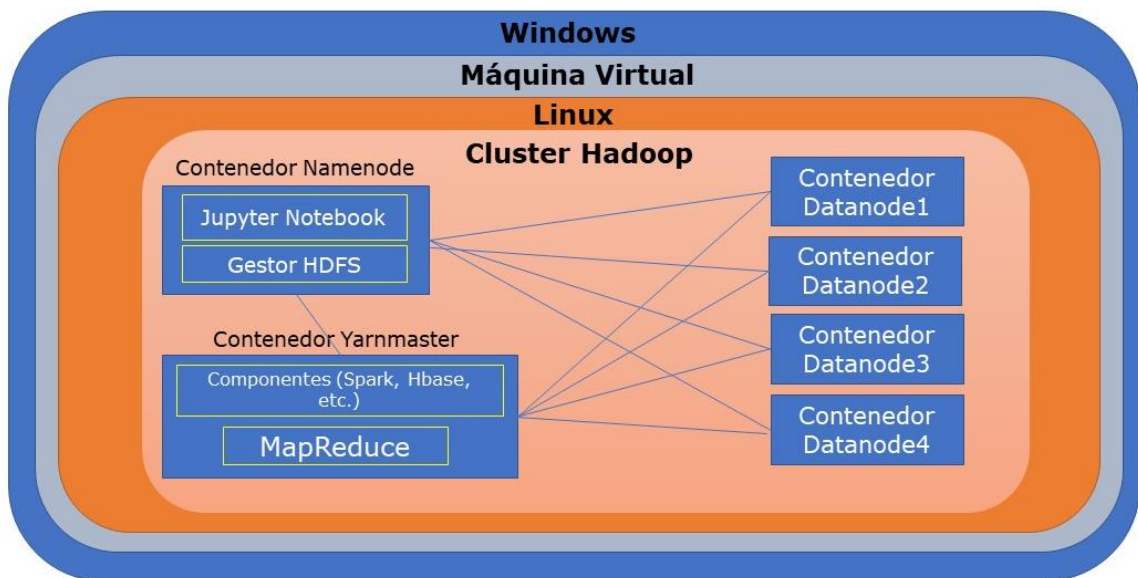


Ilustración 26 - Arquitectura del "cluster" para desarrollo y pruebas de patrones que se va a desplegar.

4.1. Secuencia de despliegue del cluster

Una vez estemos dentro de la máquina virtual, y por ende de Linux Ubuntu, desde una terminal tenemos que identificarnos como superusuario o *root*. Las aplicaciones que hemos de instalar previamente son *Docker* y *Docker-compose*, para ello seguimos los pasos siguientes [25]:

- 1- Comenzamos la instalación de *Docker*. En el enlace siguiente: <https://docs.docker.com/engine/install/ubuntu/>, se indica la secuencia de comandos a seguir.
- 2- Una vez finalizada la instalación, procedemos a seguir los pasos para instalar el orquestador *docker-compose* en: <https://docs.docker.com/compose/install/>

- 3- Descargamos y guardamos en un directorio los archivos necesarios para el despliegue del *cluster* desde GitHub: <https://github.com/accaminero/cluster-hadoop-docker>
- 4- Ejecutamos desde el directorio donde hemos guardado los archivos descargados el siguiente comando:

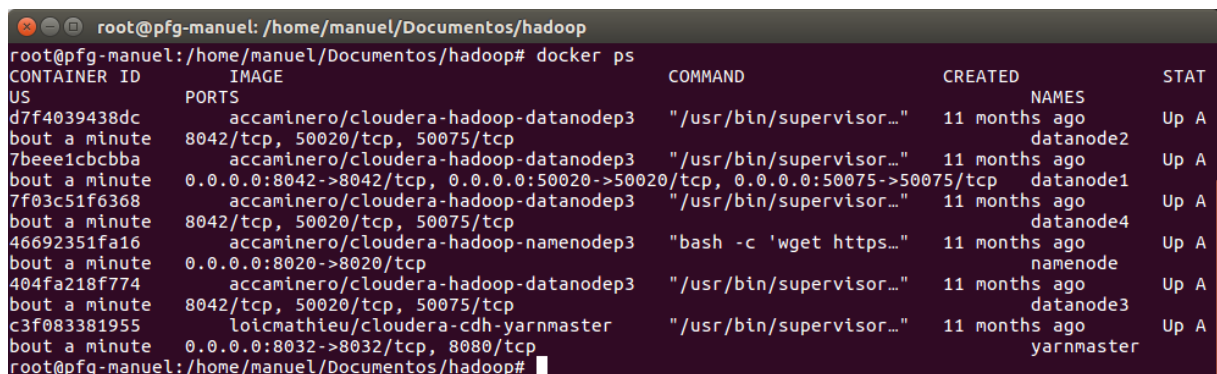
```
docker-compose up -d
```

Esta secuencia de despliegue solo se realiza una vez.

- 5- Una vez acabada la descarga y despliegue, el grupo de nodos estará arrancado. Podremos comprobarlo con cualquiera de estos dos comandos:

```
docker ps
```

```
docker-compose ps
```



```

root@pfg-manuel: /home/manuel/Documentos/hadoop
root@pfg-manuel:/home/manuel/Documentos/hadoop# docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
US             NAMES
d7f4039438dc   accaminero/cloudera-hadoop-datanode3  "/usr/bin/supervisor..." 11 months ago  Up A         8042/tcp, 50020/tcp, 50075/tcp
boute1minute   accaminero/cloudera-hadoop-datanode3  "/usr/bin/supervisor..." 11 months ago  Up A         0.0.0.0:8042->8042/tcp, 0.0.0.0:50020->50020/tcp, 0.0.0.0:50075->50075/tcp
7beee1cbbba    accaminero/cloudera-hadoop-datanode3  "/usr/bin/supervisor..." 11 months ago  Up A         8042/tcp, 50020/tcp, 50075/tcp
7f03c51f6368    accaminero/cloudera-hadoop-datanode3  "/usr/bin/supervisor..." 11 months ago  Up A         8042/tcp, 50020/tcp, 50075/tcp
46692351fa16    accaminero/cloudera-hadoop-namenode3  "bash -c 'wget https..." 11 months ago  Up A         0.0.0.0:8020->8020/tcp
404fa218f774    accaminero/cloudera-hadoop-datanode3  "/usr/bin/supervisor..." 11 months ago  Up A         8042/tcp, 50020/tcp, 50075/tcp
c3f083381955    loicmathieu/cloudera-cdh-yarnmaster   "/usr/bin/supervisor..." 11 months ago  Up A         0.0.0.0:8032->8032/tcp, 8080/tcp
root@pfg-manuel:/home/manuel/Documentos/hadoop#

```

Ilustración 27 - Contenedores levantados; el "cluster" está en funcionamiento.

4.2. Secuencia de arranque del cluster y acceso a sus aplicaciones

Para el desarrollo y ejecución de los patrones o programas *MapReduce* que vayamos desarrollando, cuando encendemos el ordenador o la máquina virtual, hemos de arrancar el *cluster*, ya que dentro de éste estarán todas las aplicaciones necesarias para el trabajo en este marco. Accederemos a la aplicación Jupyter Notebook para los desarrollos y ejecuciones de programas, y también al sistema HDFS donde podremos colocar todos los ficheros necesarios, estando disponibles y accesibles para su uso en el sistema, a la vez que podremos controlar en un entorno web, como trabajan todos los nodos del *cluster*, los balanceos de carga y rendimiento en general de éste y de cada trabajo que vayamos lanzando. Para preparar todo lo anteriormente descrito procedemos de la siguiente forma:

1. Desde una terminal, entramos como superusuario o root.
2. Nos dirigimos al directorio donde descargamos los archivos del despliegue de la arquitectura, que es donde está el archivo de configuración *docker-compose.yml*.
3. Arrancamos el *cluster*:

docker-compose start

```

root@pfg-manuel: /home/manuel/Documentos/hadoop
root@pfg-manuel: /home/manuel/Documentos/hadoop# docker-compose start
Starting namenode    ... done
Starting yarnmaster  ... done
Starting datanode1   ... done
Starting datanode2   ... done
Starting datanode3   ... done
Starting datanode4   ... done
root@pfg-manuel: /home/manuel/Documentos/hadoop#

```

Ilustración 28 - Arrancando el "cluster"

Si quisiéramos pararlo usamos el comando:

docker-compose stop

4. Una vez arrancado, es aconsejable extraer las direcciones IP del *namenode* y del *yarnmaster*, de esta forma podremos acceder a las aplicaciones web mediante URL. Para ello ejecutamos el siguiente comando:

docker inspect namenode | egrep IPAddress

docker inspect yarnmaster | egrep IPAddress

5. Con las IP ya podré entrar a las siguientes aplicaciones desde el navegador
 - a. *Jupyter Notebook* → **IP_namenode:8889**.
 - b. *Información del Namenode* y acceso HDFS → **IP_namenode:50070**
 - c. *Información del cluster Hadoop* → **IP_yarnmaster:8088**

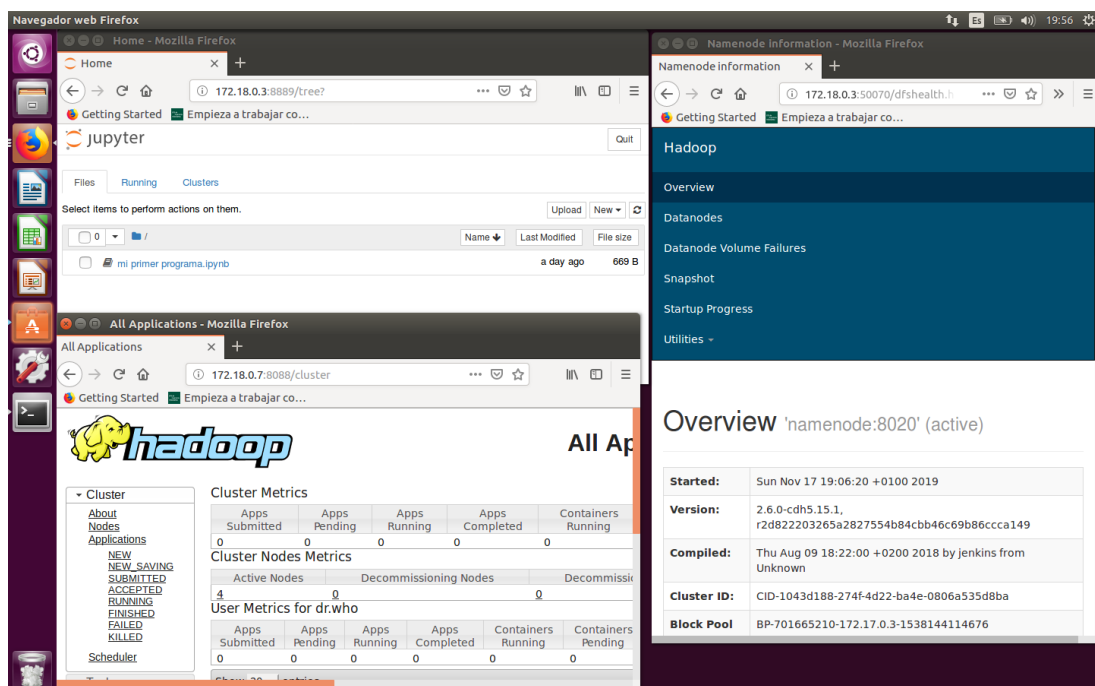


Ilustración 29 - Utilidades accesibles mediante el navegador. Éstas están dentro de los contenedores.

Hemos de tener en cuenta que cada nodo está en un contenedor, y lo más seguro es que en algún momento tengamos que entrar dentro de alguno de ellos para consultas, ajustes, algún proceso, etc. Para poder entrar en un contenedor ejecutamos el siguiente comando:

```
docker exec -it <nombre_contenedor> bash
```

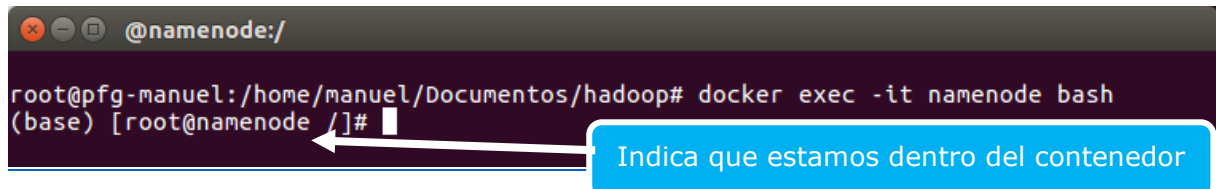


Ilustración 30 - Acceso a un contenedor.

Observamos que el contenedor tiene un sistema operativo con lo mínimo para funcionar. Mediante comandos podemos ejecutar algunas aplicaciones propias del sistema, y también nos permitirá gestionar archivos en el HDFS, con los comandos propios de éste.

Es importante destacar que disponemos de un directorio de intercambio entre el sistema operativo y los contenedores que conforman la agrupación de nodos, es decir, todo lo que se ponga en ese directorio, será accesible desde el *cluster*, por tanto, si tenemos un conjunto de archivos que necesitamos almacenarlos y procesarlos en el *cluster Hadoop*, la forma que tenemos de hacerlo es colocando éstos en */media/notebooks*, que está conectado a otro directorio con el mismo nombre dentro del contenedor *Namenode*.

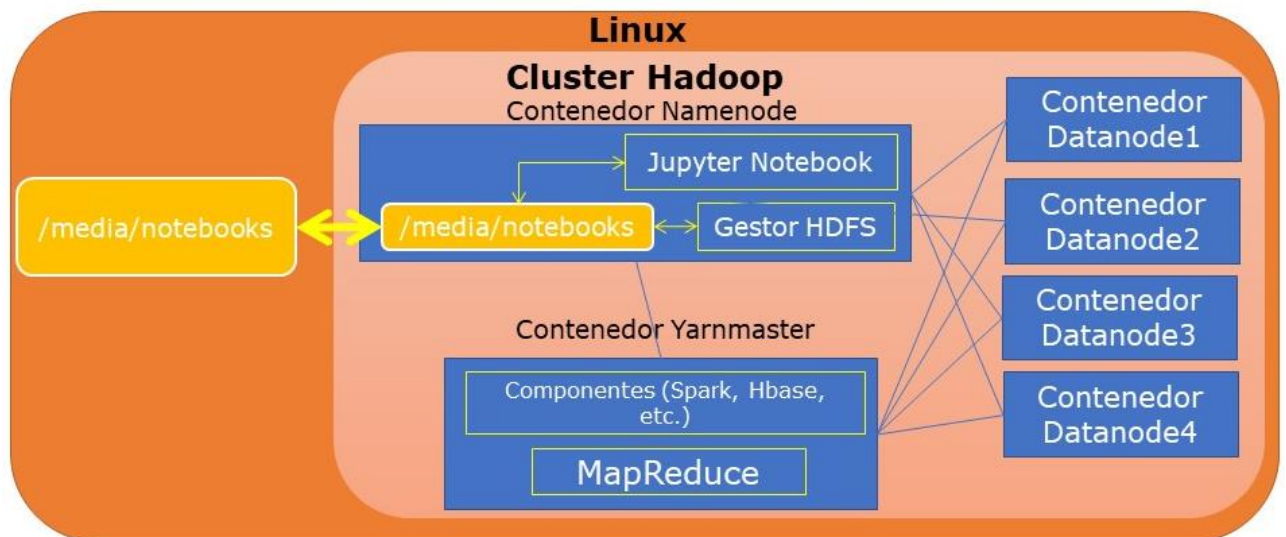


Ilustración 31 - Directorio de intercambio con el "cluster"

La arquitectura ha sido desplegada, y ya está todo preparado para desarrollar y ejecutar los programas *MapReduce* en el *cluster*.

4.3. Ejemplo de ejecución de un programa MapReduce en el cluster

Vamos a ver como se ejecuta paso a paso un programa que nos devuelve la palabra más usada en un texto [27, p. 6], tanto en un entorno local como en el *cluster* Hadoop desplegado.

- 1- Desde un cuaderno Jupyter Notebooks, escribimos el programa en una de sus celdas.

```
#directiva de Jupyter para guardar archivo:
%%writefile palabramasusada.py
#!/usr/bin/env python
from mrjob.job import MRJob
from mrjob.step import MRStep

import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_extrae_palabras,
                  combiner=self.combiner_cuenta_palabras,
                  reducer=self.reducer_cuenta_palabras),
            MRStep(reducer=self.reducer_encuentra_palabra_mas_usada)
        ]
    def mapper_extrae_palabras(self, _, line):
        # producimos cada palabra en la linea
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_cuenta_palabras(self, word, counts):
        # Sumamos las palabras vistas hasta ahora
        yield (word, sum(counts))

    def reducer_cuenta_palabras(self, word, counts):
        # Enviamos todo al siguiente reducer
        # para que encuentre la palabra más usada
        yield None, (sum(counts), word)

    # Descartamos la clave, nos hace falta la palabra y el número
    def reducer_encuentra_palabra_mas_usada(self, _, word_count_pairs):
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

- 2- Ejecutamos en local. Recordemos que MRJob emulaba un entorno distribuido.

```
In [22]: 1 !python palabramasusada.py archivos_datos/quijote.txt

20628      "que"

No configs found; falling back on auto-configuration
No configs specified for inline runner
Creating temp directory C:\Users\cyber\AppData\Local\Temp\palabramasusada.manuel.20201126.184903.433642
Running step 1 of 2...
Running step 2 of 2...
job output is in C:\Users\cyber\AppData\Local\Temp\palabramasusada.manuel.20201126.184903.433642\output
Streaming final output from C:\Users\cyber\AppData\Local\Temp\palabramasusada.manuel.20201126.184903.433642\output...
Removing temp directory C:\Users\cyber\AppData\Local\Temp\palabramasusada.manuel.20201126.184903.433642...
```

La palabra más usada en el Quijote es "que" con 20628 apariciones.

Ilustración 32 - Ejecución en local del programa MapReduce.

- 3- Para ejecutarlo en el cluster, tenemos que subir al HDFS el archivo de datos que vayamos a usar. En este caso es un fichero de texto: *quijote.txt*. Esta operación solo se hace una vez por cada archivo de datos que queramos subir.

```
In [5]: 1 !hdfs dfs -put archivos_datos/quijote.txt hdfs:///bbdd

In [6]: 1 !hdfs dfs -ls /bbdd

Found 3 items
-rw-r--r--  3 root supergroup  46622482 2020-11-25 11:59 /bbdd/Retail.csv
-rw-r--r--  3 root supergroup   2161066 2020-11-26 20:32 /bbdd/quijote.txt
-rw-r--r--  3 root supergroup   194550 2020-11-25 12:29 /bbdd/ventas_ESP.csv
```

Ilustración 33 - Subida del archivo al HDFS.

En la celda 5, en el comando indicamos el origen del archivo y el destino de éste en el HDFS.

En la celda 6, podemos comprobar que la subida ha sido un éxito cuando listamos los archivos del directorio */bbdd*.

- 4- Ejecutamos en el *cluster*.

```
In [4]: 1 !python palabramasusada.py hdfs:///tmp/quijote.txt -r hadoop --python-bin /opt/anaconda/bin/python3.7

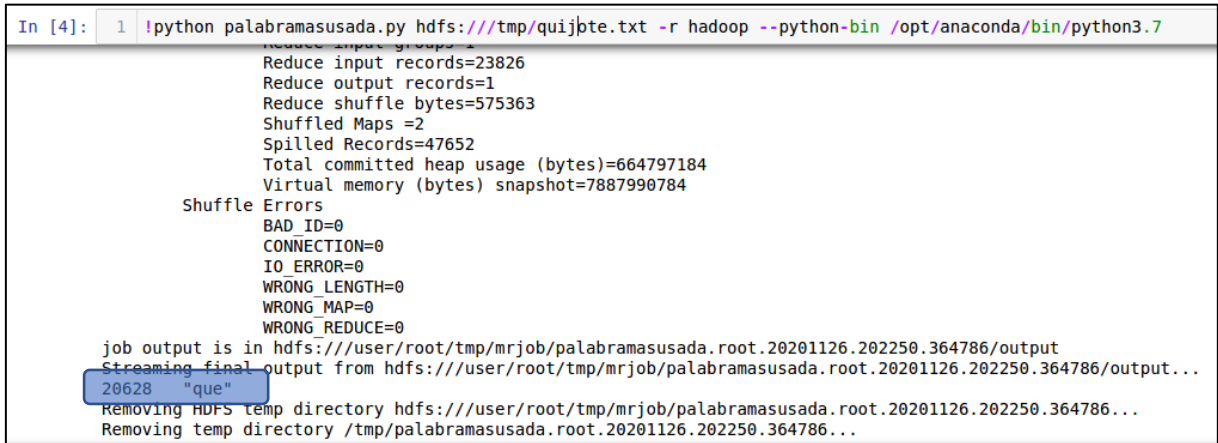
Copying other local files to hdfs:///user/root/tmp/mrjob/palabramasusada.root.20201126.202250.364786/files/
Running step 1 of 2...
packageJobJar: [] [/usr/lib/hadoop-mapreduce/hadoop-streaming-2.6.0-cdh5.15.1.jar] /tmp/streamjob32152099250942
29530.jar tmpDir=null
Connecting to ResourceManager at yarnmaster/172.18.0.4:8032
Connecting to ResourceManager at yarnmaster/172.18.0.4:8032
Total input paths to process : 1
number of splits:2
Submitting tokens for job: job_1606421218554_0001
Submitted application application_1606421218554_0001
The url to track the job: http://yarnmaster:8088/proxy/application_1606421218554_0001/
Running job: job_1606421218554_0001
Job job_1606421218554_0001 running in uber mode : false
map 0% reduce 0%
map 16% reduce 0%
map 44% reduce 0%
map 67% reduce 0%
map 100% reduce 0%
map 100% reduce 100%
```

Ejecución del paso 1

Ilustración 34 - Proceso de ejecución en el cluster. Observamos que ha terminado el paso 1.

Podemos ver en la celda 4, el comando con el que se ejecuta el programa *MapReduce* en el *cluster*.

5- Resultado de la ejecución.



```
In [4]: 1 !python palabramasusada.py hdfs:///tmp/quijote.txt -r hadoop --python-bin /opt/anaconda/bin/python3.7
Reduce input records=23826
Reduce output records=1
Reduce shuffle bytes=575363
Shuffled Maps =2
Spilled Records=47652
Total committed heap usage (bytes)=664797184
Virtual memory (bytes) snapshot=7887990784
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
job output is in hdfs:///user/root/tmp/mrjob/palabramasusada.root.20201126.202250.364786/output
Streaming final output from hdfs:///user/root/tmp/mrjob/palabramasusada.root.20201126.202250.364786/output...
20628 "que"
Removing HDFS temp directory hdfs:///user/root/tmp/mrjob/palabramasusada.root.20201126.202250.364786...
Removing temp directory /tmp/palabramasusada.root.20201126.202250.364786...
```

Ilustración 35 - Continuación del proceso de ejecución y resultado final.

Observamos en la zona marcada de la imagen, el resultado de la ejecución. Si la salida la queremos en otro fichero, ponemos al final de la orden:

--output-dir hdfs:///directorio_de_salida/archivo_de_salida

4.4. Resumen del capítulo

En este capítulo hemos podido estudiar paso a paso el despliegue de una arquitectura Big Data mediante virtualización ligera usando Docker. Hemos visto como se realiza una secuencia de arranque de un cluster Hadoop, y como acceder a todas sus aplicaciones y servicios que están virtualizados.

Con todo el sistema puesto en marcha, hemos tenido oportunidad de ejecutar un programa MapReduce en el entorno local, y posteriormente, ejecutar este mismo programa en el cluster, poniendo de manifiesto el funcionamiento del procesamiento distribuido.

Capítulo 5. Conjuntos de datos (Datasets)

En este capítulo se van a describir los conjuntos de datos que se han utilizado en los programas desarrollados a lo largo del capítulo 6. Se indicará su procedencia, número de campos, el tipo de dato, y número de registros que contienen. Los datos son ficticios, o en otros casos se han anonimizado, pero se ha procurado mantener éstos lo más cerca de la realidad.

5.1. Foros

Este conjunto de datos ha sido extraído de Udacity – Curso “Intro to Hadoop and MapReduce”, y concretamente desde el siguiente enlace:

http://content.udacity-data.com/course/hadoop/forum_data.tar.gz

A continuación, se muestran los campos de los que se compone esta tabla, mostrando los índices que usaremos para identificar los campos en programas MapReduce.

Índice	Nombre del campo
0	id
1	title
2	tagnames
3	author_id
4	body
5	node_type
6	parent_id
7	abs_parent_id
8	added_at
9	score
10	state_string
11	last_edited_id
12	last_activity_by_id
13	last_activity_at
14	active_revision_id
15	extra
16	extra_ref_id
17	extra_count
18	marked

Tabla 1 - Campos que conforman la tabla Foros.

Para los prototipos que desarrollemos en el próximo capítulo, no necesitaremos hacer uso de todos los campos, por tanto, vamos a describir los que vayan a ser de utilidad.

Id: Es el código único del mensaje. Tipo numérico.

Title: Título que se le da al mensaje en el foro. Tipo texto.

Autor_id: Código del usuario que ha creado el mensaje. Numérico

Body: Cuerpo del mensaje. Tipo texto.

Node_type: Tipo de mensaje. Identifica si es una pregunta, una respuesta o un simple comentario. Tipo texto

Parent_id: Código del mensaje padre, es decir, del mensaje del que cuelga. Numérico.

Added_at: Fecha y hora cuando se creó el mensaje. Tipo texto

El conjunto disponía de unos 200.000 registros, y se han eliminado líneas para hacer más sencillo la carga de datos para los ejemplos.

5.2. On line retail -Ventas Clientes.

Este es un conjunto de datos que contiene todas las transacciones ocurridas entre el 01/12/2010 y el 09/12/2011 para una venta minorista en Internet. La tienda vende principalmente regalos para cualquier ocasión. La tabla ha sido extraída de UCI, Machine Learning Repository [34]

Se compone de los siguientes campos:

InvoiceNo: Número de factura. Tipo numérico entero de 6 dígitos asignado de forma única a cada transacción. Si este código comienza con la letra 'c', indica una cancelación.

StockCode: Código de producto. Tipo numérico entero de 5 dígitos asignado de forma única a cada producto distinto.

Description: Nombre del producto. Tipo texto.

Quantity: Cantidad de cada producto por transacción. Tipo numérico entero.

InvoiceDate: Fecha y hora de entrada. Tipo fecha/hora. Indica el día y la hora en que se generó cada transacción.

UnitPrice: Precio unitario. Tipo numérico real. Es el precio del producto por unidad en libras.

CustomerID: Código de cliente. Tipo numérico entero de 5 dígitos asignado de forma exclusiva a cada cliente.

Country: nombre del país. Tipo texto. Contiene el nombre del país donde reside cada cliente.

0	1	2	3	4	5	6	7
InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country

Tabla 2 - Campos de la tabla On-line Retail. La línea superior indica el índice con el que identificaremos los campos en los programas MapReduce.

El número de registros que contiene este conjunto de datos es de 541.900, esto a la hora de probar desarrollos puede ser bastante pesado, por esta razón en algunas ocasiones

usaremos un subconjunto de esta tabla, para que sea más fácil probar los programas que hagan uso de ella. Por esta razón, nombraremos a esos archivos de subconjuntos como “*ventas_*” seguido de las tres primeras iniciales del país. Por ejemplo: *ventas_ESP.csv* correspondiente a las ventas en España o *ventas_AUS.csv*, para ventas en Australia.

5.3. Eventos alumnos

Este conjunto de datos contiene los eventos reales de los profesores y alumnos en un curso de teleformación de una plataforma de formación Moodle, que está actualmente en producción. Un evento puede ser la realización de una actividad, enviar un mensaje, extraer un informe, consultar una calificación, etc.

Los datos han sido anonimizados sustituyendo los nombres reales de los alumnos por nombres como *Alumno1*, *Alumno2*, *AlumnoN*, al igual que se ha hecho con los profesores; las direcciones IP también han sido modificadas. Esta tabla se generó a partir de un informe llamado *Registros*, que está ubicado en el menú *Administración del curso* en Moodle.

La tabla se compone de los siguientes campos:

Hora: Fecha y hora de cuando se produjo el evento. Tipo fecha/hora.

Nombre completo del usuario: usuario que provoca el evento. Tipo texto.

Usuario afectado: otro usuario que puede ser afectado por el evento provocado. Tipo texto.

Nombre del evento: nombre de la actividad o recurso que se ha tocado. Tipo texto.

Descripción: describe lo que el usuario ha realizado durante ese evento. Tipo texto.

Dirección IP: Indica la dirección IP desde donde se ha producido el evento.

0	1	2	3	4	5
Hora	Nombre completo del usuario	Usuario afectado	Nombre evento	Descripción	Dirección IP

Tabla 3 - Campos de la tabla eventos_alumnos. La línea superior indica el índice con el que identificaremos los campos en los programas MapReduce.

El número de registros de este conjunto de datos ronda los 15.500. Los datos se extrajeron en el intervalo de tiempo que va desde el 26/11/2019 al 14/03/2020.

5.4. Resumen del capítulo

En este capítulo hemos descrito los tres conjuntos de datos que van a ser usados para los prototipos que se van a desarrollar en el siguiente capítulo. Se han detallado los campos y las posiciones para identificar éstos en los programas MapReduce. Se añade una pequeña descripción de cada campo, que facilita la comprensión de su contenido.

Con este capítulo, ya lo tenemos todo preparado para iniciar la última parte de este trabajo: el desarrollo de los patrones de diseño.

Capítulo 6. Patrones de diseño

Comenzamos con el objetivo principal de este trabajo, y vamos a partir de la base que MapReduce es un paradigma de programación con la capacidad de usar cientos de computadores para procesar datos que estén almacenados en estos. El paradigma es bastante potente, pero veremos que en algunos problemas, será bastante complicado la aplicación de éste.

Cuando hablamos de patrones de diseño MapReduce, nos referimos a una idea, una plantilla o un enfoque general para resolver un problema en el que manipulamos una gran cantidad de datos. El uso de patrones de diseño nos permite usar artefactos ya probados para construir un mejor software *MapReduce*.

Nos centraremos en cuatro tipos de patrones y veremos ejemplos en todos ellos, y estudiando en mayor profundidad algunos casos particulares de especial relevancia e interés.

6.1. Patrones de Resumen

Este conjunto de patrones de diseño se centra en agrupar datos similares y posteriormente realizar alguna operación como: calcular una estadística, construir un índice invertido o simplemente hacer un conteo. Algunos ejemplos de casos reales donde podemos aplicar este tipo de patrones pueden ser: la suma de las ventas en un comercio, calcular la rotación de productos de los lineales de un supermercado, calcular el número de comentarios que un usuario ha hecho en un foro, o incluso ver cuantas veces aparece una palabra en uno o varios textos. Por tanto, los patrones que vamos a estudiar en este apartado serán de resúmenes numéricos, en los que incluiremos entre otros los siguientes modelos: la media aritmética, la desviación estándar, y el cálculo del índice invertido.

6.1.1. Resúmenes numéricos

Este patrón puede ser utilizado para calcular valores estadísticos, y para ello debemos agrupar por un campo clave los registros de la fuente de datos (generalmente una tabla), y realizar el cálculo correspondiente por grupo. Algunos ejemplos pueden ser la agrupación por franjas horarias del número de clientes que entran en una tienda, esto se puede trasladar a un histograma y reconocer en que horarios la tienda está más concurrida, o también clasificar por sección el número de productos que se venden de ésta, y ver si esa sección es rentable tenerla en la tienda.

En la siguiente figura se puede observar la estructura general de este patrón:

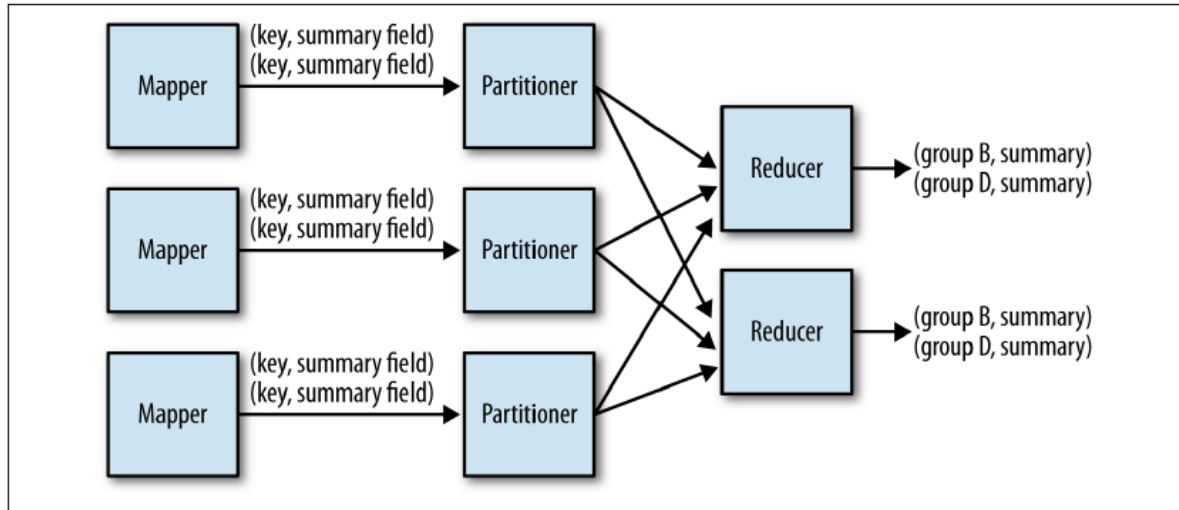


Ilustración 36 - Estructura de un patrón de resumen numérico [7, p. 16]

En el *Mapper* se van generando los pares clave-valor, posteriormente en el *Reducer* se terminan de agrupar y realizar los cálculos si procede.

6.1.1.1. Prototipos

Media aritmética

Para calcular la media aritmética, necesitamos para cada grupo de registros la suma de los valores y el número de valores sumados. Estos dos valores los podemos calcular en el lado del *Reducer* iterando a través de cada valor en el conjunto, de forma que a la vez que sumamos, vamos contando el número de elementos. Una vez finalizada la iteración tan solo hemos de dividir la suma entre el conteo para que nos de la media.

Con el conjunto de datos *On-Line Retail* descrito en la página 50, vamos a calcular la media de compra de cada cliente.

En el *Mapper* se produce el flujo de entrada de datos recogido por el parámetro *line*:

```

def mapper(self, _, line):
    #División del registro para tener la lista de campos
    linea=line.split(";")
    #Extraemos el precio del artículo
    valor=linea[5]
    #Extraemos el código del cliente
    clave=linea[6]
    #Cedemos la clave y el valor al Reducer
    yield clave, float(valor)
  
```

Tabla 4 - Mapper para el cálculo de la media aritmética.

Aquí viene lo apasionante de este paradigma, ya que no hemos tenido que escribir ninguna línea de código en la que busquemos el código del cliente y recojamos sus ventas asociadas, de eso ya se ha encargado el *mapper*. Por tanto, lo único que hacemos con el *Reducer* es recoger la clave y sus valores asociados y realizar las operaciones que estimemos.

```
def reducer(self, key, values):
    sumaValues=0 # Para la suma de los valores
    cuentaValues=0 # Para el número de valores
    # Iteramos a través del conjunto de valores
    # que han llegado desde el mapper
    for value in values:
        sumaValues+=value #Sumamos valor
        cuentaValues+=1 #Contamos valor
    #Dividimos
    mediaValues=sumaValues/cuentaValues

    #Entregamos la media de ventas
    #del cliente con clave linea[1]
    yield key, mediaValues
```

Tabla 5 - Reducer para calcular la media aritmética.

Si utilizamos la librería *statistics*, existe un método que nos permite realizar el cálculo de la media aritmética, la diferencia radica en que, en el *Reducer*, tendríamos que almacenar en una lista los valores que llegan del *Mapper*, y posteriormente llamar al método *mean*.

```
def reducer(self, key, values):
    listaValores=[]
    for valor in values:
        listaValores.append(valor)
    yield key, statistics.mean(listaValores)
```

Tabla 6 - Cálculo de la media en el Reducer usando la librería *statistics*.

El resultado de la ejecución es el siguiente (se muestran las cuatro primeras líneas):

"12346"	1.04
"12347"	2.644010989010989
"12348"	5.764838709677418
"12349"	8.289041095890411

Tabla 7 - Ejemplo de salida de la media aritmética.

Valor mínimo, valor máximo y contar valores

Siguiendo con el uso del conjunto de datos *Online_Retail*, cuando el *Mapper* cede el grupo de datos asociado a una clave, en el *Reducer* tan solo hemos de añadir los datos agrupados a una lista, y extraer de ésta el valor máximo, mínimo, o el total de valores que contiene.

Hay un punto interesante, y es que para agregar los valores que vienen del *Mapper* a una lista, podemos hacerlo iterando sobre estos y añadiéndola a ésta, o directamente usar la potencia de Python para trabajar con listas. Veámoslo con dos ejemplos:

```
def mapper(self, _, line):
    linea=line.split(";")

    #Controlamos que lo que entre sean datos numéricos
    encontrado=re.search('^[0-9]',linea[5])
    if encontrado!=None:
        valor=linea[5] #Precio del artículo
        clave=linea[6] #Código del cliente
        yield clave,float(valor)

def reducer(self, key, values):
    #Lista donde almacenaremos los valores
    valores=[]

    #Recorremos la lista de "values"
    #para agregarlos a la lista
    for value in values:
        valores.append(value)

    #Ordenamos los valores de la lista
    valores.sort()

    #Valor máximo
    vMax=valores[len(valores)-1]

    #Valor mínimo
    vMin=valores[0]

    #Número de valores del grupo
    cuenta=len(valores)

    yield key, (cuenta,vMax,vMin)
```

Tabla 8 - Programa para contar artículos y mostrar el más caro y el más barato por cliente.

En esta primera versión iteramos con un *for* para agregar todos los elementos del *values* a una variable de tipo lista; esto puede venirnos bien para el caso que queramos realizar cualquier tipo de operación elemento a elemento. En nuestro caso, se puede suprimir y usar una expresión de asignación:

```
valores=list(values)
```

Con esto sería suficiente para extraer lo que necesitamos. A continuación, se muestra el segundo ejemplo o versión del *Reducer*:

```
def reducer(self, key, values):
    valores=list(values)
    valores.sort()
    vMax=valores[len(valores)-1]
    vMin=valores[0]
    cuenta=len(valores)
    yield key, (cuenta,vMax,vMin)
```

Tabla 9 - Segunda versión del Reducer para darnos la cuenta, valor máximo y valor mínimo.

La extracción de los valores es muy sencilla, primeramente ordenamos la lista de menor a mayor, esto lo hacemos con la función *sort()*, que realiza esta operación sobre una lista. A continuación, extraemos el valor máximo yéndonos a la última posición de la lista. Es importante tener en cuenta que las posiciones de las listas empiezan desde 0, por tanto, la última posición será N-1 siendo N el número de elementos en una lista. Evidentemente, el valor mínimo será la posición 0 y el número total de elementos nos lo va a dar la función *len()*.

El resultado de este programa es el total de artículos comprados por el cliente, y el valor máximo y mínimo de los artículos de ese cliente. A continuación, puede verse parte de la salida después de una ejecución:

```
"12346" [2,1.04,1.04]
"12347" [182,12.75,0.25]
"12348" [31,40.0,0.29]
"12349" [73,300.0,0.42]
"12350" [17,40.0,0.85]
"12352" [95,376.5,0.65]
"12353" [4,9.95,1.45]
```

Tabla 10 - Parte de la salida de contar, valor máximo y valor mínimo

Cálculo de la Mediana

Una mediana es el valor numérico que separa la parte inferior y superior de un conjunto de datos, esto requiere que los datos estén ordenados, cosa que con *MapReduce* puede ser bastante complejo. Si el número de datos es impar, se coge el dato central, si el número de datos es par se cogen los dos centrales y se calcula la media de éstos. Ejemplo:

Tenemos la siguiente lista de datos ya ordenados:

valores = [3,5,8,10,25,52,65]

Al ser el número de datos impar, la mediana será:

$$M_e = 10$$

Valores = [4,6,9,21,35,41]

Al ser el número de datos par, cogemos los dos centrales. Por tanto, la mediana será:

$$M_e = \frac{9 + 21}{2} = \frac{29}{2} = 14.5$$

Veamos los desarrollos en el paradigma en dos versiones. Para ello vamos a usar el mismo conjunto de datos *Online_Retail* descrito en la página 50. Podremos observar que el *Mapper* solo recoge las líneas de datos y manda las agrupaciones por clave al *Reducer*, que es donde se hace todo el cálculo. Vamos a sacar la mediana de ventas por país.

```
def mapper(self, _, line):
    linea=line.split(";")
    valor=linea[5] #Precio del artículo
    clave=linea[7] #País
    yield clave,float(valor)
def reducer(self, key, values):
    valores=[]
    sumaValores=0
    contador=0
    for value in values:
        valores.append(value)
        sumaValores+=value
        contador=contador+1
    valores.sort()
    if contador%2==0: #Si es par
        indice_1=int(contador//2)
        indice_2=int(indice_1-1)
        #Extraemos los valores
        valor_1=valores[indice_1]
        valor_2=valores[indice_2]
        mediana=(valor_1+valor_2)/2
    else: #Si es impar
        indice=(contador-1)//2
        mediana=valores[indice]
    yield key, mediana
```

Tabla 11 - Cálculo de la mediana

Si usamos la librería *statistics*, existe un método que calcula la mediana, tan solo hemos de pasarle la lista con los datos y el resto se encargará Python. A continuación, vemos una segunda versión completa de este algoritmo.

```
from mrjob.job import MRJob
import statistics as st
class medianaV2(MRJob):
    def mapper(self, _, line):
        linea=line.split(";")
        valor=linea[5] #Precio del artículo
        clave=linea[7] #País
        yield clave,float(valor)
    def reducer(self, key, values):
        valores=list(values)
        yield key, st.median(valores)
if __name__ == '__main__':
    medianaV2.run()
```

Tabla 12 – Segunda versión para el cálculo de la mediana usando la librería statistics.

Evidentemente, las líneas de código se reducen considerablemente, siendo más eficiente esta última versión.

En la siguiente tabla se puede observar parte del resultado de la ejecución del programa expuesto en la Tabla 11 y la Tabla 12:

"Australia"	1.79	
"Austria"	1.95	
"Bahrain"	3.81	
"Belgium"	1.95	
"Brazil"	3.3200000000000003	
"Canada"	1.65	
"Channel Islands"	2.55	
"Cyprus"	2.95	
"Czech Republic"	1.45	
"Denmark"	1.95	

Tabla 13 - Resultado parcial del cálculo de la mediana.

Cálculo de la desviación estándar

La desviación estándar muestra cuanta variación existe en los datos con respecto del promedio. En este caso podemos aplicar dentro del *Reducer* la fórmula para su cálculo:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}}$$

Pero podemos directamente usar la librería *statistics* de Python, en la que tenemos el método *pstdev(lista)* que, pasándole la lista de valores, nos hace este cálculo. Usamos el mismo conjunto de datos de la página 50.

```
from mrjob.job import MRJob
import statistics as st
class devEstandar(MRJob):
    def mapper(self, _, line):
        linea=line.split(";")
        valor=linea[5] #Precio del artículo
        clave=linea[7] #País
        yield clave,float(valor)
    def reducer(self, key, values):
        valores=list(values)
        yield key, st.pstdev(valores)
if __name__ == '__main__':
    devEstandar.run()
```

Tabla 14 – Programa para el cálculo de la desviación estándar.

"Australia"	1.79
"Austria"	1.95
"Bahrain"	3.81
"Belgium"	1.95
"Brazil"	3.3200000000000003
"Canada"	1.65
"Channel Islands"	2.55
"Cyprus"	2.95
"Czech Republic"	1.45
"Denmark"	1.95

Tabla 15 - Resultado de la ejecución del programa para la desviación estándar.

6.1.1.2. Análisis y aplicabilidad

Podemos decir que los resúmenes numéricos se pueden aplicar:

- ➔ Cuando se está tratando con datos numéricos o contadores.
- ➔ Cuando los datos se pueden agrupar por campos específicos.

Hemos estudiado algunos de los casos de uso, como han sido el cálculo del máximo, el mínimo, recuento de valores y algunos cálculos estadísticos. Podríamos agregar también el análisis del flujo de datos en un determinado periodo de tiempo (horas, días, semanas, etc.).

Dentro de este tipo de patrones se encuentra el ejemplo típico del contador de palabras, para explicar el paradigma *MapReduce*. La aplicación genera cada palabra de un documento como la clave y 1 como el valor, agrupando de esta forma las palabras. En la fase de reducción suma los enteros y genera cada palabra única con la suma.

Como conclusión que podemos sacar de estos prototipos, es que dentro del *Reducer* podemos realizar todo tipo de cálculos y recuentos con el grupo de datos asociados a una clave. En definitiva, este tipo de patrones es para lo que *MapReduce* se creó.

6.1.2. Índice invertido

Un índice invertido es una estructura, generalmente de tipo tabla, donde se guarda un mapeo de la información contenida en uno o varios documentos en forma de palabras y/o números, con el objetivo de poder ser recuperada de la forma más eficiente y rápida posible. Cuando un usuario hace una petición de búsqueda, el buscador le indicará los documentos donde aparece esa información o término a partir de lo que se ha almacenado en esa tabla. Se le llama índice invertido por que éste se crea después, cuando el motor ya ha analizado los documentos donde se basará la búsqueda.

Vemos en el esquema de la Ilustración 37, que la tabla de índice invertido se forma con las palabras de cada uno de los documentos, y el número de documento donde aparecen cada una de esas palabras.

Nos podemos encontrar otras variantes como el número de veces que aparece en cada documento, e incluso la posición en donde aparece [35].

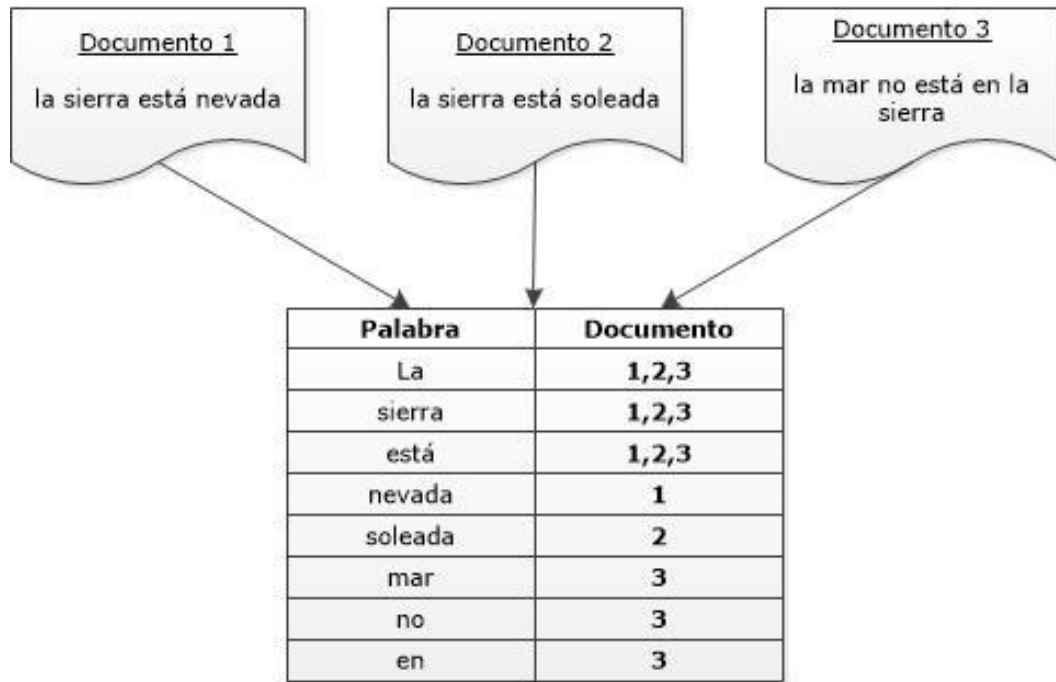


Ilustración 37 - Ejemplo de construcción de tabla de índice invertido.

6.1.2.1. Prototipo

Para el ejemplo que vamos a desarrollar, vamos a coger un documento con varias líneas, y nos va a devolver la palabra junto a las líneas del documento donde aparece, y el número de veces que aparece en cada línea. La clave será la palabra, y el valor es una lista con la línea y el número de veces que aparece la palabra en esa línea. Aunque este ejemplo ha sido desarrollado tomando como base un documento con las líneas numeradas, se puede extrapolar a la búsqueda en un conjunto de documentos, tan solo tendremos que controlar el flujo de archivos que entran al *mapper*.

Mapper

Comenzamos con el *Mapper* que es donde se produce el flujo de entrada de datos; en el documento que vamos a utilizar (docejemplo.txt) cada línea es recogida por el parámetro *line*, y posteriormente lo guardamos en la variable de tipo lista llamada *línea*. De cada línea extraemos su número o identificador, en nuestro ejemplo va a estar en la posición 0, este será el valor, y el resto (de la posición 1 en adelante) lo recogemos en otra variable de tipo lista llamada *palabras*. Esta última lista la recorreremos extrayendo cada palabra, con el objetivo de pasarla al *Reducer* en forma de clave.

En resumen, el *Mapper* genera la palabra como la clave, y el número de línea como el valor.

```

linea1 El perro corre rapido
linea2 El chavea corre con el perro que corre
linea3 El chavea nada rapidamente y corre
linea4 el chico de el camino es el mismo

```

Tabla 16 - Contenido del archivo de ejemplo *docejemplo.txt*

```

def mapper(self, _, line):
    linea=line.split(' ')
    nLinea=linea[0]
    palabras=linea[1:]
    for palabra in palabras:
        yield palabra.lower(),nLinea

```

Tabla 17 - Código del mapper para extraer las palabras y la línea donde se encuentran.

Reducer

La característica especial del *Reducer*, es que se va a encargar de hacer un tratamiento por cada palabra que le llegue, es decir, él recogerá la palabra como clave y todos los valores asociados a esa clave, que serán los números de línea donde aparece la palabra. Con esta información podremos averiguar cuantas veces aparece la palabra en cada una de las líneas del documento que estamos procesando.

Comenzamos introduciendo cada uno de los valores que vienen agrupados por clave, en una variable de tipo lista llamada *listaValores*. A continuación, hacemos un recorrido de cada una de las posiciones de la lista generada mediante la iteración “for”. Esta iteración nos irá dando los números de posición de los elementos de *listaValores*; la variable *posición* tomará valores de 0 a *longitud(listaValores)*. El elemento almacenado en la posición dada por la variable *posición* lo guardamos en la variable *elementoLista*, y éste lo usaremos para ver cuantos elementos iguales hay en la lista usando el método *count()*. El número de elementos que nos devuelva lo metemos en *cuentaElementos*. Una vez tenemos todos los datos necesarios, metemos en *subLista* el valor de la posición de *listaValores*, que será el número de línea o número de documento, y el número de veces que aparece esa palabra en esa línea o documento. Esto último lo hacemos para comprobar después, que en la *listaDef* no existe ese conteo, que si es el caso, agregamos la *subLista* a la lista definitiva, y continuamos con la iteración hasta finalizar el bloque, momento en el cual emitimos el resultado.

```

def reducer(self, key, values):
    listaValores=[]
    listaDef=[]
    for valor in values:
        listaValores.append(valor)

    for posicion in range(len(listaValores)):
        subLista=[]
        elementoLista=listaValores[posicion]
        cuentaElementos=listaValores.count(elementoLista)
        subLista=[listaValores[posicion],cuentaElementos]
        if subLista not in listaDef:
            listaDef.append(subLista)

    yield key, listaDef

```

Tabla 18 - Muestra del código del Reducer para el índice invertido.

"camino"	[["linea4",1]]
"chavea"	[["linea2",1],["linea3",1]]
"chico"	[["linea4",1]]
"con"	[["linea2",1]]
"corre"	[["linea1",1],["linea2",2],["linea3",1]]
"de"	[["linea4",1]]
"el"	[["linea1",1],["linea2",2],["linea3",1],["linea4",3]]
"es"	[["linea4",1]]
"mismo"	[["linea4",1]]
"nada"	[["linea3",1]]
"perro"	[["linea1",1],["linea2",1]]
"que"	[["linea2",1]]
"rapidamente"	[["linea3",1]]
"rapido"	[["linea1",1]]
"y"	[["linea3",1]]

Tabla 19 - Resultado de la ejecución del patrón Índice invertido.

6.1.2.2. Análisis y aplicabilidad

Generalmente los índices invertidos se han de usar cuando se necesitan respuestas rápidas a consultas. Algún caso más aparte del visto en la búsqueda de términos en documentos, puede ser la búsqueda de términos en comentarios en foros, y por supuesto la búsqueda de webs.

Puede parecer complicada la construcción de un índice invertido, pero es una aplicación bastante sencilla de *MapReduce*, por que el marco maneja la mayoría del trabajo. De hecho, este patrón se usa normalmente como un ejemplo para analítica *MapReduce*;

digamos que es el hermano mayor del famoso “*WordCount*”. Lo más importante aquí, es entender el concepto de qué es y cómo se forma el índice invertido. El marco MapReduce hará el resto.

6.1.3. Contando con contadores

Este patrón utiliza el mecanismo de conteo del marco de trabajo *MapReduce*, generalmente para extraer o contar el número de registros de una entrada de datos. Cuando se finaliza el trabajo de conteo, los resultados se pueden guardar en un registro, el sistema de archivos local, en HDFS, etc. Hadoop ya tiene sus propios contadores, y con este patrón el programador puede desarrollar sus contadores personalizados, para recopilar métricas de recuento o sumas de datos de sus conjuntos. El principal beneficio de usar contadores es que todo el conteo se puede hacer en la fase del *Mapper*.

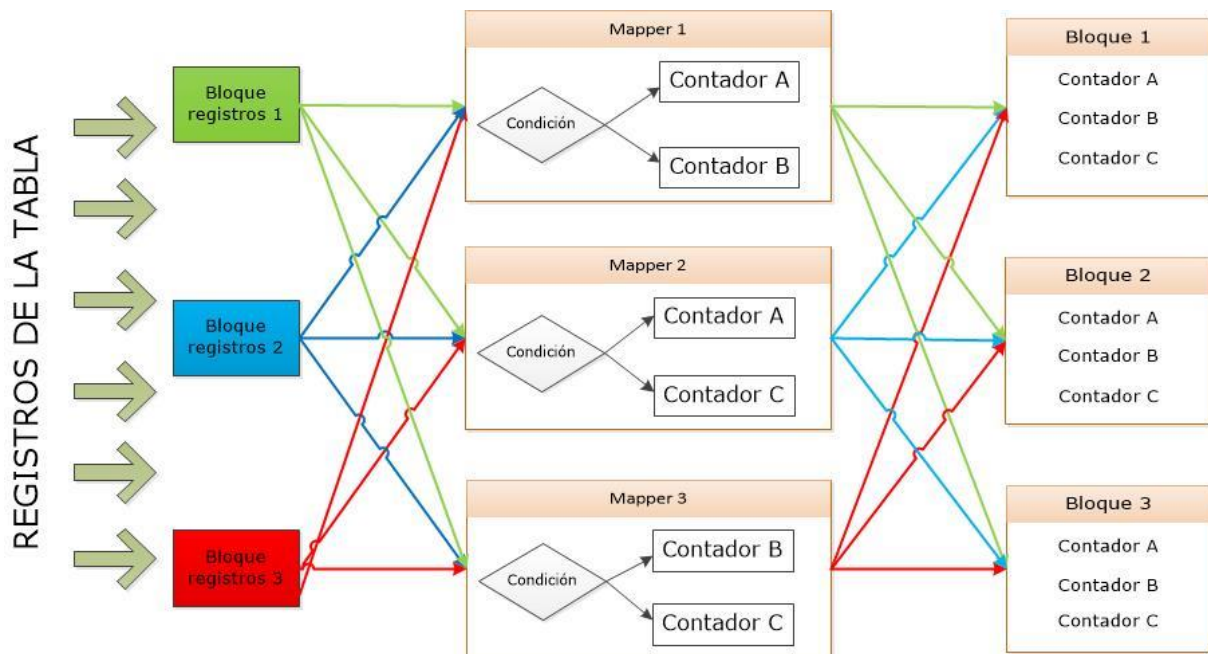


Ilustración 38 - Esquema de patrón "Contando con contadores".

6.1.3.1. Prototipo

Mapper

Para poder realizar la operación de conteo, necesitamos una estructura que nos almacene los resultados de los conteos de cada nodo, esto hará que cada *Mapper* que se ejecute, nos dará los conteos que le hayamos indicado. Las *MRJob* nos permite definir estructuras donde poder almacenar los datos que necesitemos para operar. Para ello usamos el método `mapper_init()` [27, p. 10], que permite definir una acción que se ejecutará antes de que el

Mapper procese cualquier entrada, generalmente se usa esta función para inicializar estructuras auxiliares específicas para el *Mapper*.

La estructura que definimos no es más que un diccionario donde vamos a meter los contadores, siendo estos la clave; el valor será el recuento del registro donde aparezca el contador definido en el diccionario.

Por ejemplo, vamos a utilizar el conjunto de datos Online_Retail descrito en la página 50. Usaremos el campo *Country* (*país*) donde se ha hecho la venta; vamos a utilizar varios países como contadores de registros, de forma que en cada registro que aparezca el país definido en el diccionario, se añadirá al contador de ese país.

```
from mrjob.job import MRJob
class counter(MRJob):
    def mapper_init(self):
        #Creamos un diccionario con los que serán los contadores a 0
        #El campo del que extraemos los contadores es: Country
        self.contadores={"Netherlands":0,"France":0,"Australia":0}

    def mapper(self, _, line):
        linea=line.split(";")
        # Estudiamos cada token de la línea recogido
        for token in linea:
            # Si el token está en el diccionario
            if token in self.contadores:
                # Contamos con el contador definido en diccionario
                self.contadores[token]=self.contadores[token]+1

    def mapper_final(self):
        yield "Bloque: ",self.contadores
```

Tabla 20 – Versión 1 del patrón Contando con Contadores

Observamos como metemos el método *mapper_final()*, que sirve para definir una acción que se ejecutará después de que el asignador llegue al final de la entrada. Una forma de usar esto es almacenar un total en una variable o estructura, y generarlo después de leer todos los datos de entrada.

Efectivamente se puede observar que una vez el *Mapper* ha finalizado su trabajo emitimos el resultado a través del método *mapper_final()*. La salida de este ejemplo será:

"Bloque: "	{"Netherlands":292,"France":970,"Australia":142}
"Bloque: "	{"Netherlands":269,"France":1048,"Australia":221}
"Bloque: "	{"Netherlands":279,"France":917,"Australia":137}
"Bloque: "	{"Netherlands":415,"France":1093,"Australia":367}
"Bloque: "	{"Netherlands":401,"France":1117,"Australia":146}
"Bloque: "	{"Netherlands":192,"France":1261,"Australia":201}
"Bloque: "	{"Netherlands":349,"France":1077,"Australia":37}
"Bloque: "	{"Netherlands":174,"France":1074,"Australia":8}

Tabla 21 - Resultado de la ejecución del modelo de la Tabla 20

Es importante entender que el *Mapper* no nos va a dar un resultado final, si no que prepara los contadores como un preprocesamiento de los datos.

Si por el contrario quisiéramos que nos diera un resultado final, no tenemos más que implementar el *Reducer*, tal y como observamos en el prototipo de la Tabla 22:

```
from mrjob.job import MRJob

class counterV2(MRJob):
    def mapper_init(self):
        #Creamos un diccionario con los que serán los contadores a 0
        self.contadores={"Netherlands":0,"France":0,"Australia":0}
    def mapper(self, _, line):
        linea=line.split(";")
        # Estudiamos cada token de la línea recogido
        for token in linea:
            # Si el token está en el diccionario
            if token in self.contadores:
                # Contamos con el contador definido en diccionario
                self.contadores[token]=self.contadores[token]+1
    def mapper_final(self):
        for clave, valor in self.contadores.items():
            yield clave,valor
    def reducer(self, key, values):
        yield key, sum(values)
```

Tabla 22 - Segunda versión del patrón Contando con Contadores, aplicando el Reducer.

Podemos ver que hemos insertado en el `mapper_final()` una estructura iterativa que recorre todos los elementos del diccionario, cediendo la clave y el valor al *Reducer*, el cual se encarga de sumar todos los valores agrupados.

La salida de la ejecución de este patrón será lo que se muestra en la Tabla 23:

"Australia"	1259
"France"	8557
"Netherlands"	2371

Tabla 23 - Resultado de la ejecución del código de la Tabla 22

Por otro lado, si sacamos la declaración del diccionario fuera de la clase, cada *Mapper* sumará su bloque de registros a esa variable global de tipo diccionario, dándonos en el último bloque el resultado de los contadores.

```
from mrjob.job import MRJob
contadores={"Netherlands":0,"France":0,"Australia":0}
class counterV3(MRJob):
    def mapper(self, _, line):
        linea=line.split(";")
        # Estudiamos cada token de la línea recogido#
        for token in linea:
            # Si el token está en el diccionario
            if token in contadores:
                # Contamos con el contador definido en diccionario
                contadores[token]=contadores[token]+1
    def mapper_final(self):
        yield "Bloque: ",contadores
```

Tabla 24 - Versión tercera del patrón Contando con Contadores.

"Bloque: "	{"Netherlands":292,"France":970,"Australia":142}
"Bloque: "	{"Netherlands":561,"France":2018,"Australia":363}
"Bloque: "	{"Netherlands":840,"France":2935,"Australia":500}
"Bloque: "	{"Netherlands":1255,"France":4028,"Australia":867}
"Bloque: "	{"Netherlands":1656,"France":5145,"Australia":1013}
"Bloque: "	{"Netherlands":1848,"France":6406,"Australia":1214}
"Bloque: "	{"Netherlands":2197,"France":7483,"Australia":1251}
"Bloque: "	{"Netherlands":2371,"France":8557,"Australia":1259}

Tabla 25 - Resultado de la ejecución del código de la Tabla 24

6.1.3.2. Análisis y aplicabilidad.

Resumiendo, podemos enumerar los casos en los que podemos aplicar este patrón:

- ➔ Contar el número de registros
- ➔ Contar distintos registros en base al contador configurado. Es importante indicar en este punto que el número de contadores no ha de ser muy grande, como mucho cien contadores.
- ➔ Realizar sumas.
- ➔ Analizar el rendimiento de los *Mapper*. Como el proceso de contar es muy rápido, al leerse solo a través del *Mapper*, se puede calcular el rendimiento en base al número de tareas y el tiempo que lleva ejecutarlas y procesarlas.

La implementación y uso de los contadores es bastante sencillo y rápido, ya que solo usamos el *mapper*, y lo único que debemos tener en cuenta es que el rendimiento dependerá del número de tareas de *Map* y de cuánto tiempo lleve el procesamiento de cada registro.

6.2. Patrones de filtrado

El objetivo de los patrones de filtrado es encontrar un subconjunto de datos de una fuente grande de información. El tamaño del subconjunto dependerá de para qué los necesitamos y de las condiciones aplicadas en su búsqueda. Este tipo de patrones no cambian los registros resultantes, al contrario de lo que pasaba con los patrones de resumen, que trataban de agrupar datos por campos similares, pudiendo realizar posteriormente todo tipo de operaciones. En resume, el filtrado es una forma más de búsqueda, que nos va a permitir mediante muestras comprender mejor los datos que se nos presentan.

En este tema, vamos a ver tres tipos de patrones de filtrado: extracción de subconjuntos, extraer los valores más altos y extracción de valores no repetidos. En algunos de estos patrones solo usaremos la parte del *Mapper*.

6.2.1. Extracción de subconjuntos

Ejemplo de patrón de filtrado en el que se extrae un subconjunto de datos a partir de una base de datos. El subconjunto de datos ha de cumplir ciertas características que el usuario o programador indicará antes del procesamiento *Map*. En este tipo de patrones solo se ejecuta el *Map*. La idea es que un *Map* proporcione información filtrada a otro proceso, ya sea un *Map*, un *Combinador* o un *Reduce*.

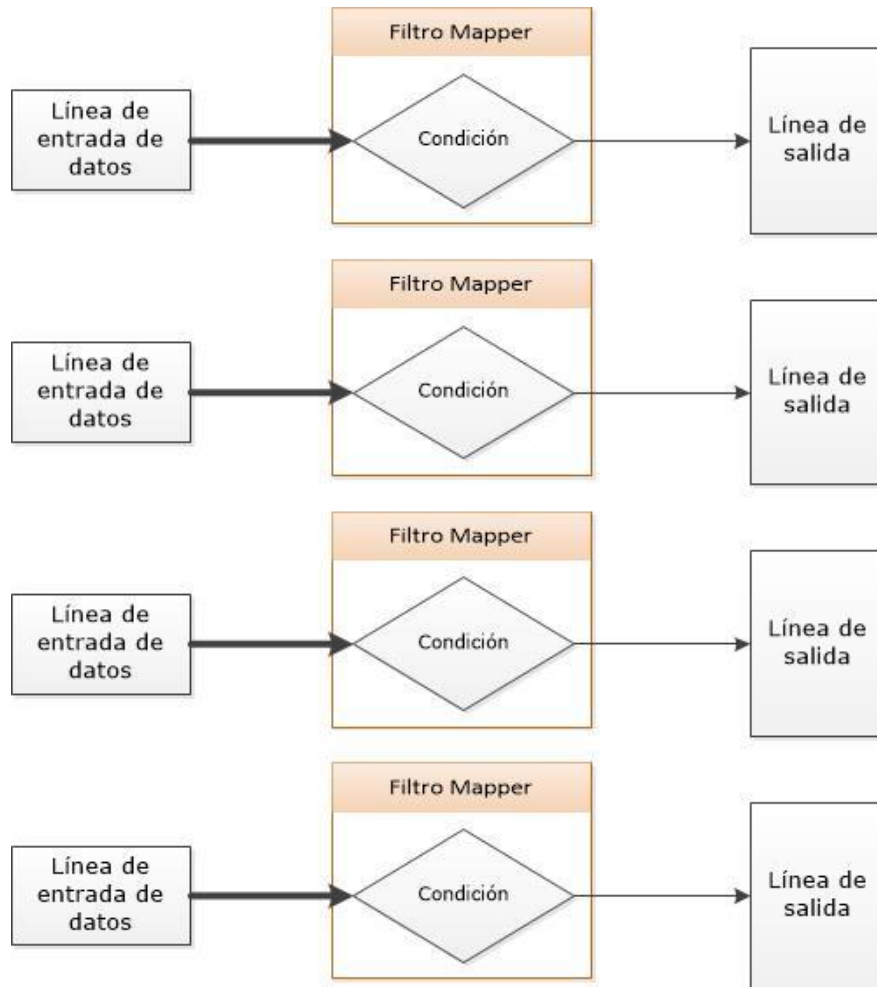


Ilustración 39 - Estructura de un patrón de filtrado [7, p. 45]

6.2.1.1. Prototipos

Extracción de un subconjunto

En el siguiente prototipo vamos a limpiar unos datos de entrada, concretamente los registros de un foro. Disponemos de un archivo llamado `foros_sucios.csv` (se describe en capítulo 5, página 49), que en el id del mensaje de algunas líneas, hay texto en vez de haber un dato numérico. Necesitamos que los datos estén perfectamente limpios antes de hacer un tratamiento de ellos. Comenzaremos eliminando los saltos de línea del comentario del foro, y dividiendo los campos de cada una de las líneas que nos llega y, usando posteriormente una expresión regular, filtraremos lo que entra como campo clave, que en nuestro caso tiene que ser un número, junto a una segunda condición donde el número de palabras extraídas de toda la línea de entrada tiene que ser mayor que uno.

```

from mrjob.job import MRJob
import re # Librería para expresiones regulares
class subconjunto(MRJob):
    def mapper(self,_, line):
        linea=line.rstrip("\n").split(";")
        #El campo de la columna 0 tiene que ser un número.
        encontrado=re.search('[0-9]',linea[0])
        if encontrado!=None and len(linea)>1:
            yield linea[0],linea

if __name__ == '__main__':
    subconjunto.run()

```

Tabla 26 - Ejemplo de patrón de filtro de datos, en el que eliminamos los registros cuyo id sea texto.

Con una expresión regular comprobamos si lo que entra es una cadena de caracteres o un número; si es un número nos quedamos con el registro, si no, lo deseamos. Este prototipo puede equipararse al comando “grep” de Linux, el cual extrae información de un archivo, a partir de una expresión regular.

Por último, aplicamos una condición, que en este caso es que la línea ha de tener más de una palabra, y cedemos con *yield* al siguiente proceso o tarea.

```

"11003345"      ["11003345","", "cs215 ", "100000197", "\"<p>First of all,
you have to give 'aliases' to all nodes in the graph, since string com
parisons are slower than, for instance, int comparisons.</p>"]

"11003365"      ["11003365","", "cs215 ", "100020078", "\"<p>Anmar, I foun
d an explicit formula for $x$x% using the derivative of the function w
e are trying to minimize with respect to $x$x%. </p>"]

"11003376"      ["11003376","", "cs215 ", "100025468", "\"<p><a href=\"\"h
ttp://forums.udacity.com/users/100023849/mike-stoessl\"\"><a href
= \"\"http

```

Tabla 27 - Resultado parcial después de ejecutar el código de la Tabla 26

Seguimiento de hilo de eventos

Con este prototipo pretendemos extraer un hilo de eventos consecutivos como parte de un estudio de un conjunto de datos más grande. Vamos a usar el conjunto de datos eventos_alumnos.csv descrito en la página 51, capítulo 5.

Vamos a extraer la información de un usuario en particular, que interactúa en un curso de una plataforma de formación. Analizamos todos los eventos producidos en la plataforma mediante el análisis de los registros almacenados en un fichero, quedándonos con los que cumplan la condición o condiciones definidas.

```
from mrjob.job import MRJob
class hiloeventos(MRJob):
    def mapper(self,_, line):
        linea=line.rstrip("\n").split(";")
        if linea[1]=="Alumno5" and linea[3]=="Formulario de entrega
visto.":
            yield (linea[1],linea[3]),1
if __name__ == '__main__':
    hiloeventos.run()
```

Tabla 28 - Código seguimiento de eventos.

Aquí cedemos el usuario y evento como clave, y como valor la unidad, para el tratamiento posterior en otro proceso.

CLAVE	VALOR
["Alumno5","Formulario de entrega visto."]	1
["Alumno5","Formulario de entrega visto."]	1
["Alumno5","Formulario de entrega visto."]	1
["Alumno5","Formulario de entrega visto."]	1
["Alumno5","Formulario de entrega visto."]	1
["Alumno5","Formulario de entrega visto."]	1

Tabla 29 - Resultado parcial de la ejecución del código de la Tabla 28

6.2.1.2. Análisis y aplicabilidad

Algunos de los casos, incluidos los dos prototipos vistos anteriormente, en los que podemos aplicar este tipo de patrón pueden ser:

- ➔ Extracción de un subconjunto de registros que tengan algún rasgo en común.
- ➔ Limpiar datos.
- ➔ Extracción de información mediante expresiones regulares.
- ➔ Eliminar o extraer datos que estén en un rango numérico o tipo fecha.

- ➔ Análisis de un subconjunto de datos con una característica común.
- ➔ Extracción de un hilo de eventos consecutivos para su estudio pormenorizado.
- ➔ Extracción de muestras aleatorias.

Este patrón es muy eficiente ya que el trabajo solo se hace en la parte del *Map*, por lo que los datos no se transmitirán al *Reduce*. La gran mayoría de las tareas de *Map*, extraen los datos de su sistema de ficheros local y vuelve a escribir en ese nodo local. Por otro lado, al no haber *Reduce*, no se ejecuta la fase de ordenación y clasificación (*Shuffle and sort*).

6.2.2. Filtro valores más altos

Este modelo de patrón trata de recuperar los k registros principales de un conjunto de datos, en base a un esquema de clasificación, es decir, en todo momento vamos a saber cuántos registros queremos obtener sin importar el tamaño de la entrada.

Encontrar valores atípicos es una parte importante en el análisis de datos, ya que estos registros que contienen estos valores, suelen ser piezas de datos que requieren de un estudio más en detalle. El objetivo de este patrón es encontrar los mejores registros como por ejemplo: que productos se venden más o cuales son las publicaciones con mayor puntuación en un foro. Aquí sí vamos a usar tanto el *Map* como el *Reducer*; cada *Mapper* aplicará su función de comparación o clasificación, y en un solo *Reducer* se producirá la extracción de los n mayores registros.

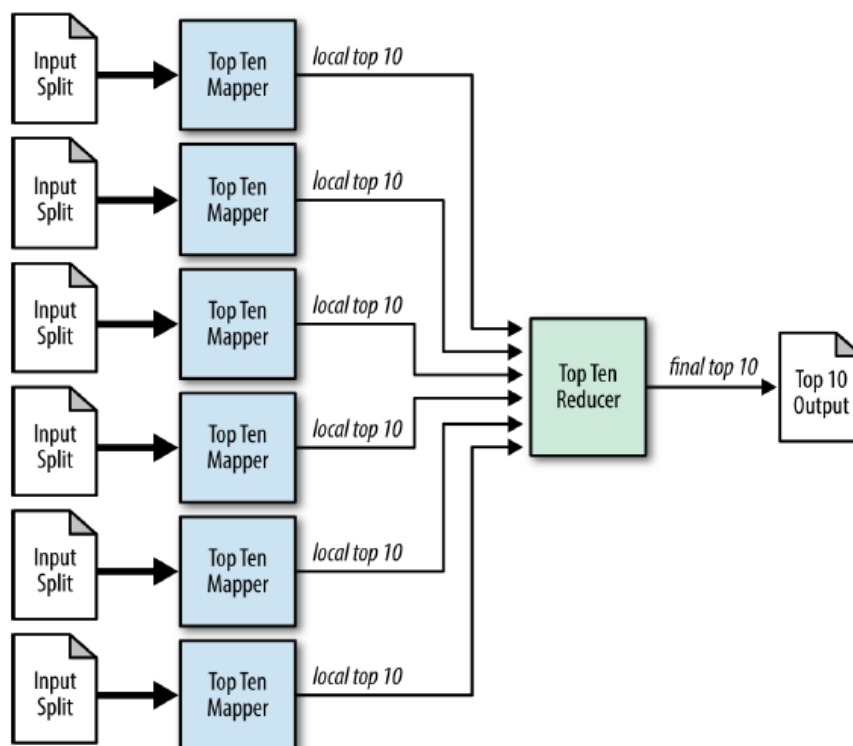


Ilustración 40 - Estructura del patrón "TopTen" [7, p. 60]

6.2.2.1. Prototipo

En este ejemplo desarrollado vamos a crear un método dentro de la clase llamado *extraeTopN()*, que será el que extraiga los N valores más altos de la lista que le mandaremos desde cada uno de los *Mapper* y desde el *Reducer* (Ilustración 40).

Comenzaremos inicializando una estructura de tipo lista llamada *topNmap* en el *mapper_init()*, en donde almacenaremos los valores que entran en el *Map*. En éste, cada registro que entra se va dividiendo, de esa forma podremos acceder a cada uno de los valores de los campos del registro de entrada. Si trabajamos con valores numéricos, nos podemos encontrar con un problema, y es que puede darse el caso que el valor del campo sea alfanumérico, bien por error en los datos, o por que vienen valores mezclados, o simplemente es el título de la columna en la tabla. Esto lo solucionamos mediante expresiones regulares; la librería **re** nos proporciona los artefactos necesarios para comprobar que tipo de valor es. En el ejemplo, buscamos la expresión regular indicada, y si no la encuentra, significa que el valor es numérico y puede agregarlo a la estructura de tipo lista *topNmap*, inicializada anteriormente en el *mapper_init()*.

Una vez que han entrado todas las líneas del bloque en el *mapper*, se ejecutará a continuación el *mapper_final()*, que como se explica en apartados anteriores, sirve para definir una acción que se ejecutará después de que el *mapper* termine de procesar su bloque de líneas.

```
def mapper_init(self):
    self.topNmap = []

def mapper(self, _, line):
    registro=line.rstrip('\n').split(';')

    #En la posición 5 del registro está el valor que necesitamos.
    encontrado=re.search('[a-zA-Z]',registro[5])
    #Si NO encuentra la expresión regular, procesamos dato.
    if encontrado==None:
        self.topNmap.append(float(registro[5]))

def mapper_final(self):
    self.topNmap.sort() #se ordena la lista
    # extraeTopN extrae los N elementos últimos de la lista.
    yield "TopNMap:", self.extraeTopN(self.topNmap)
```

Tabla 30 - Patrón TopN. Mapper.

Este es un buen ejemplo para comprobar cómo funciona este método de las *MRJob*, ya que cuando el *mapper* ha terminado de meter todos los valores en la lista definida en el *mapper_init()*, el *mapper_final()* mandará esta lista **ya ordenada de menor a mayor** con el método *sort()*, al método *extraeTopN()*, que será el que se encargue de extraer los N valores más altos. Veamos como lo hace.

Descripción del método *extraeTopN()*

Se define una variable global fuera de la clase, donde se indicará el número de valores que se quiere extraer. Pueden ser cinco, diez, veinte, etc.

Cuando llega la lista de valores *listaValores*, se ha de comprobar que el número de valores definidos por la variable *TopN* no sea mayor que el tamaño de la lista, si se da el caso, tendremos que limitar el tamaño de *TopN* al número de valores que hay en la lista. Por ejemplo, si hemos definido que se han de extraer los 10 valores más altos de la lista, y resulta que la lista solo trae 5 valores, aquí se da una incongruencia que puede provocar un error, y para que esto no ocurra, a la variable *nValores* se le asigna el tamaño de *listaValores* mediante el método *len()*. Si no se diera este caso especial, le asignamos al *nValores* lo que *TopN* tenga, o lo que es lo mismo, *nValores* contendrá el total de valores que queremos extraer, es decir N.

Comenzamos la búsqueda de los N valores distintos más altos empezando por el final de la lista que, al estar ordenada, **los valores más altos están al final**. Cada valor que encontremos lo añadimos a la lista *valoresMasAltos*.

Lo primero que hacemos es almacenar en la variable *posicion*, el número total de índices que tiene la *listaValores*, vamos a recorrer todos los valores desde el final de la lista, y recordemos que las posiciones o índices de las listas en Python van de 0 a n-1. A continuación, iniciamos una estructura iterativa que recorrerá los N valores últimos de la lista, pero debemos prever que pueden existir valores repetidos, si esto ocurre, la variable cuenta no se incrementará, ya que controla el número de iteraciones que hay que hacer. Evidentemente las posiciones a visitar, sí se tienen que ir decrementando para que podamos encontrar los diez valores NO repetidos más altos.

Puede darse el caso que lleguemos a una posición negativa o inexistente de *listaValores*, esto provocará una excepción de índice fuera de rango, que será recogida por try-catch, y devolverá los valores correctos por que ha terminado de recorrer toda la *listaValores*. Podemos deducir que las variables *cuenta* y *posicion* no van al unísono, sobre todo cuando hay valores repetidos.

```

from mrjob.job import MRJob
import re
TopN=10 #Los N valores más altos que vamos a extraer
class topN(MRJob):
    #Método que extrae los N valores más altos de una lista de valores.
    def extraeTopN(self, listaValores):
        valoresMasAltos = []
        cuenta=0
        #Comprobar tamaño de la lista
        if TopN>len(listaValores):
            nValores=len(listaValores)
        else:
            nValores=TopN
        #Almacenamos la última posición de la lista
        posicion=len(listaValores)-1
        while cuenta<nValores:
            try:
                # Si el valor no está en la lista de valores más altos,
                # lo agregamos.
                if listaValores[posicion] not in valoresMasAltos:
                    valoresMasAltos.append(listaValores[posicion])
                    cuenta+=1 # Si se agrega un valor aumentamos cuenta
                    posicion-=1 # Tanto si agregamos valor como si no,
                                # disminuimos la posición.
                #Si se produce excepción salimos y devolvemos resultado
            except:
                break
        return valoresMasAltos

```

Tabla 31 - Función para extraer los N valores más altos

Por último, si nos fijamos en el esquema de la Ilustración 40, observamos que el *reducer* recoge todas las listas de cada uno de los bloques *mapper*, es decir recoge una lista con los N valores más altos de cada *mapper*, y éstos valores los recogemos en *listaValores* del *reducer*; ordenamos la lista con *sort()*, y volvemos a llamar al método *extraeTopN()*, mandándole la nueva lista a analizar y devolviendo posteriormente los N valores más altos definitivos.


```

def reducer(self, key, values):
    listaValores=[]
    # metemos los valores en una lista
    for valores in values:
        listaValores.extend(valores)
    # ordenamos la lista
    listaValores.sort()
    yield "TopNReducer:", self.extraeTopN(listaValores)
if __name__ == '__main__':
    topN.run()

```

Tabla 32 – Patron TopN. Reducer.

Para probar los desarrollos descritos en: Tabla 30, Tabla 31 y Tabla 32 vamos a hacer uso del conjunto de datos descrito en la página 50 del capítulo 5, *Online_retail.csv* del que hemos derivado un subconjunto llamado *ventas_UK.csv* donde vamos a sacar los 10 productos más caros que han sido vendidos. En la Tabla 33 vemos el resultado de la ejecución.

```

"TopNReducer:" [38970.0,17836.46,16888.02,16453.71,13541.33,13474.79,
11586.5,11062.06,8286.22,8142.75]

```

Tabla 33 - Resultado de la ejecución del patrón TopN.

6.2.2.2. Análisis y aplicabilidad

Partimos que la salida ha de ser siempre de menor tamaño que la entrada, esto es lógico si hablamos de un patrón de filtrado, y de que se van a extraer un conjunto de valores que son los más altos. Otra cosa para tener en cuenta es que el tamaño de N que sale de los *Map*, por lo general será pequeño, esto hará que solo se haga uso de un solo *Reducer*.

Algunos de los casos de uso para este patrón pueden ser:

- ➔ Análisis de valores atípicos: como se indicaba al principio de este apartado, en el que nos podemos encontrar ejemplos tales como detectar los usuarios que más acceden a nuestra web, o analizar cuáles son los clientes que más nos compran, o ver que productos fluctúan más los precios, etc.
- ➔ Seleccionar datos que puedan ser de nuestro interés: si realizamos encuestas en las que cada ítem tienen una puntuación, podemos seleccionar los ítems que más

puntuación tengan (o menos puntuación) que nos permita aplicar medidas para mantener o mejorar esos ítems (calidades, precios, servicios, productos, etc.)

6.2.3. Filtro de valores no repetidos

Este modelo de patrón de filtro mostrará a partir de un conjunto de valores repetidos, una lista con valores filtrados no repetidos, es decir, eliminará los registros repetidos en base al dato del campo que le indiquemos.

6.2.3.1. Prototipo

En este ejemplo vamos a eliminar los registros cuya dirección IP sea igual, ya que necesitamos un informe del usuario, y las distintas direcciones IP desde las que se ha conectado. Vamos a usar la tabla descrita en la página 51 del capítulo 5, *eventos_alumnos.csv* donde la dirección IP se encuentra en la columna con índice 5.

Declaramos una expresión regular, que nos permitirá comprobar si lo que entra es una IP válida. El *mapper* dividirá el registro, y enviará la línea al *reducer* si cumple las condiciones. Observamos que la clave es nula, aunque podríamos usar cualquier campo como clave (nombre, la misma IP, etc.), esto no afectará al comportamiento del algoritmo.

```
from mrjob.job import MRJob
import re

#Expresion regular para validar ip
ip = ('^(?:25[0-5]|2[0-4][0-9]|'
      '[01]?[0-9][0-9]?\\.)'
      '(?:25[0-5]|2[0-4][0-9]|'
      '[01]?[0-9][0-9]?)$')
IP_RE = re.compile(ip)

class valoresNoRepetidos(MRJob):

    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search(IP_RE,linea[5])
        if encontrado!=None:
            yield _,linea
```

Tabla 34 - Expresión regular y mapper para el patrón.

En el *reducer* vamos a tener dos listas: una para los registros definitivos que vamos a mostrar, y otra para controlar los registros que son repetidos. Mediante un bucle revisamos cada uno de los registros que nos llegan del *mapper*, de forma que si el valor de control (en este caso la IP) no está en la lista, lo agregamos a ésta, y seguidamente el registro completo en la lista definitiva, que ésta es la que usaremos para sacar al sistema lo que necesitamos de los registros únicos y/o no repetidos.

```
def reducer(self, key, values):
    listaDef=[] #Lista para los registros únicos filtrados
    listaControl=[] #Lista para el control de los datos repetidos

    for registro in values:
        if registro[8] not in listaControl:
            listaControl.append(registro[5])
            listaDef.append(registro)

    for elemento in listaDef:
        yield elemento[1], (elemento[1],elemento[5])
```

Tabla 35 - Reducer para el filtro de valores no repetidos.

"Alumna1"	["Alumna1", "217.125.48.43"]
"Alumno5"	["Alumno5", "195.77.157.196"]
"Alumna3"	["Alumna3", "37.222.51.156"]
"Alumno6"	["Alumno6", "31.4.245.175"]
"Alumno7"	["Alumno7", "87.223.30.0"]
"Alumna8"	["Alumna8", "85.61.123.29"]
"Profesor1"	["Profesor1", "80.39.209.6"]
"Alumno21"	["Alumno21", "81.32.157.225"]
"Alumno4"	["Alumno4", "150.214.205.80"]

Tabla 36 - Resultado de la ejecución del ejemplo descrito en la Tabla 34 y Tabla 35

6.2.3.2. Análisis y aplicabilidad

Aplicar este patrón puede ser útil para limpiar conjuntos de datos grandes que tengan registros duplicados o similares, que puedan crear problemas como ocupar una cantidad significativa de espacio o sesgar resultados para análisis de datos en niveles superiores. Por ejemplo, cada vez que un usuario visita nuestro sitio web, almacenamos en nuestras bases de datos el navegador y dispositivo que utiliza. Puede darse el caso que ese usuario visite varias veces nuestra web, por tanto, esa información se registrará más de una vez. Si realizamos un análisis para calcular el porcentaje de los usuarios que usan un navegador

específico, la cantidad de veces que un mismo usuario ha entrado en nuestro sitio web desvirtuará los datos al estar repetidos. En este caso tendríamos que eliminar los datos duplicados.

Está claro que el único requisito para aplicar este patrón es tener datos duplicados, y entre los casos en los que podemos aplicarlo aparte de eliminación de registros, es en la obtención de valores distintos, y proteger de un colapso del sistema en uniones internas (se verá en el apartado 6.4), con claves externas repetidas en la tabla principal.

6.3. Patrones de Organización de datos

El objetivo de este tipo de patrones es la reorganización de los datos, ya que podemos encontrarnos que Hadoop y MapReduce sean solo piezas de un engranaje más amplio para el análisis de datos y extracción de información. Por otro lado, puede ser necesario que haya que transformar los datos de su estado original a un nuevo estado que facilite su tratamiento a la hora de extraer información. Estudiaremos tres tipos de patrones: estructuración jerárquica, ordenación de registros y mezclar y desordenar. En este último desarrollaremos un prototipo que nos permitirá anonimizar datos dentro del marco.

6.3.1. Estructuración jerárquica

Este ejemplo de patrón de organización de datos consiste en que dada una estructura de datos en formato tabla, ésta es convertida a otro tipo de estructura en un formato jerarquizado como puede ser JSON o XML. Recordemos que la estructura de un documento XML está formada por [36]:

- ➔ Prólogo (opcional)
 - Declaración de XML
 - Declaración del tipo de documento.
- ➔ Cuerpo

Estructura genérica XML

```

<raíz>
  <hijo1>
    <subhijo1_1>
      <subhijo1_1_1> ... </subhijo1_1_1>
      <subhijo1_1_2> ... </subhijo1_1_2>
      ...
    </subhijo1_1>
    <subhijo1_2> ... </subhijo1_2>
    ...
  </hijo1>
  <hijo2> ... </hijo2>
  ...
</raíz>

```

Ilustración 41 - Cuerpo de archivo XML [36]

Dada una o varias tablas, los *Map* se encargan de recoger las líneas de éstas, y analizará cada uno de los registros identificando a que parte o nivel de la jerarquía pertenecerá esa línea. A continuación, se le asocia una clave a ese registro, y este se pasa al *Reducer* el cual se encargará de agruparlos por su misma clave a la vez que, mediante una iteración, creará la estructura jerarquizada.

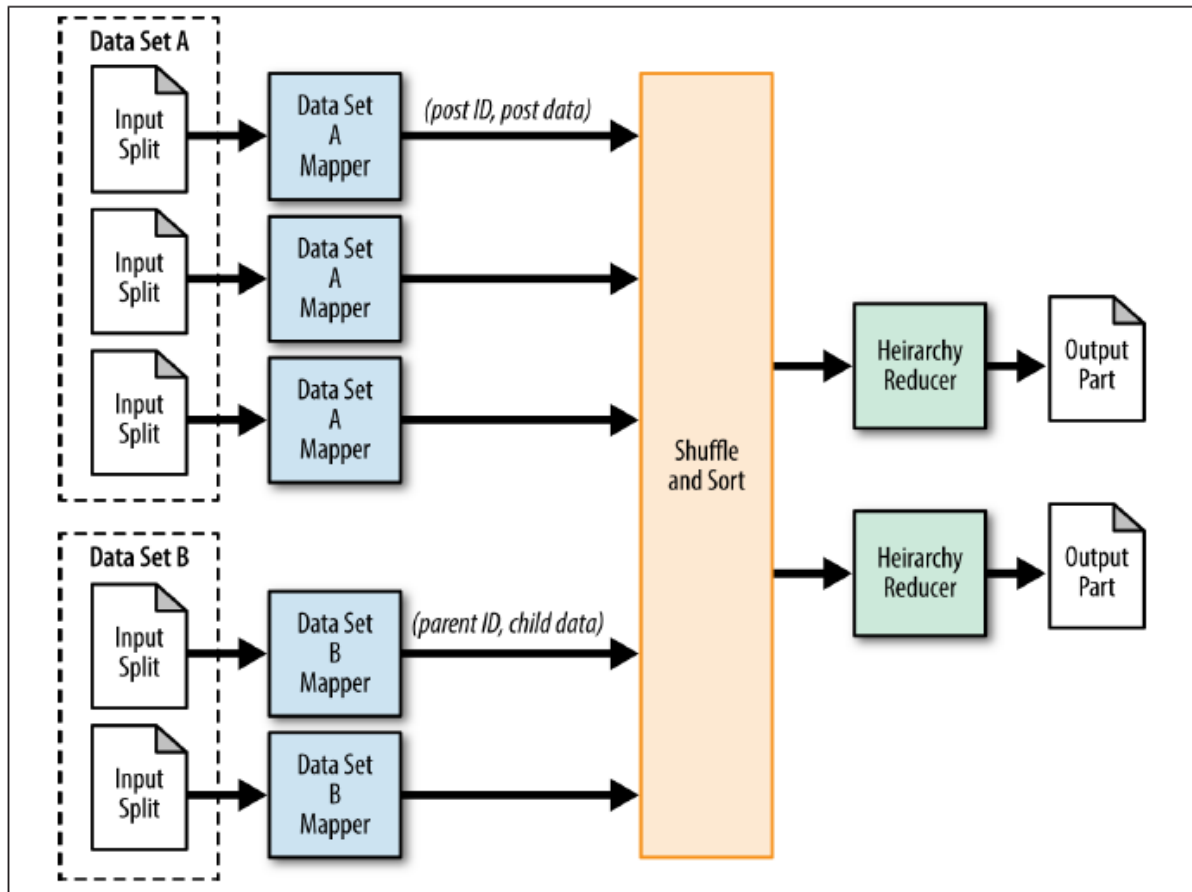


Ilustración 42 - Esquema del patrón de estructura jerárquica [7, p. 74].

Cuando el *Map* recibe los datos, que puede ser de una o varias tablas, el proceso *mapper* se encargará de establecer que tipo de registro va en un nivel u otro de la jerarquía. Como vemos en la Ilustración 42 se usan dos tablas con datos de entrada al Map, una con datos hijos (Data Set B), y otra con datos padre (Data Set A), pero el ID de ambos deben coincidir (parent ID=post ID) para establecer las agrupaciones por identificador en el *Reducer*, es decir la salida estará en forma jerárquica, agrupada por la clave ID.

6.3.1.1. Prototipo

Para este prototipo usaremos la tabla descrita en la página 49 del capítulo 5 *Foros.csv*, donde existen registros de preguntas y respuestas de un foro de consultas. Estas preguntas y respuestas están repartidas por toda la tabla, y lo que vamos a tratar de hacer es agrupar los mensajes padre (preguntas) con los mensajes hijos (respuestas o comentarios), usando

para ello el identificador principal del mensaje padre que será el *id* que esté guardado en el campo *parent_id*; éste será el que permita al *reducer* agrupar todos los mensajes de las publicaciones que contengan el mismo identificador, y que al producir las salidas del *reducer*, permita convertirlo a un formato XML ayudándonos de la librería *xmlify* [37].

Mapper

Vamos a guardar cada campo del registro (que es un mensaje publicado en el foro) en una lista, esto nos permitirá acceder al contenido de éste mediante su identificador en la lista. Del *mapper* necesitaremos el cuerpo del mensaje publicado en el foro, y el tipo de mensaje: pregunta o respuesta.

```
OUTPUT_PROTOCOL=RawValueProtocol
def mapper(self,_, line):
    linea=line.split(";") #Dividimos el registro

    mensaje=linea[4] # Recogemos el cuerpo del mensaje
    tipoMensaje=linea[5] #Recogemos el tipo de mensaje

    #Comprobamos si es una pregunta o respuesta/comentario
    if tipoMensaje=="question":
        idMensaje=linea[0] #Almacenamos el id padre de la
publicación
        yield idMensaje, (tipoMensaje,mensaje)
    else:
        idPadre=linea[7] #Almacenamos el id del mensaje padre
        yield idPadre, (tipoMensaje,mensaje)
```

Tabla 37 - Patrón Mapper estructuración jerárquica.

Para que el *reducer* pueda agrupar todos los mensajes de una publicación, necesitamos el identificador de esa publicación y todas las respuestas asociadas a esa primera publicación, que llevarán asociado ese identificador único. En el ejemplo que estamos usando, si la línea que ha entrado es una pregunta, el identificador único del mensaje estará en *linea[0]*, si no es una pregunta, significa que es una respuesta a una de las preguntas realizadas anteriormente, y cuyo identificador de la pregunta a la que está asociada (línea padre) la respuesta está en la *linea[7]*.

La gestión del identificador padre en el *mapper*, es fundamental por que es la clave que se usará en el *reducer* para agrupar preguntas y respuestas con el mismo identificador. Recordemos que estamos en procesamiento distribuido, y que los registros de la tabla entraran por bloques en los *mapper*, y por ejemplo una pregunta en el foro con ID=x podría ser procesada en el nodo 1 del *cluster*, pero una respuesta asociada a esa pregunta con ID=x puede ser procesada en el nodo 4 del *cluster*.

Por último, producimos la salida en formato clave-valor, donde la clave es el identificador padre del mensaje, y el valor es el tipo de mensaje y el cuerpo del mensaje.

Reducer

En el *reducer* vamos a usar tres listas:

- ⇒ listaValores=lista donde agregaremos cada tipo de mensaje y el mensaje asociado que han llegado del *mapper*.
- ⇒ listaPrincipal=lista donde formaremos la jerarquía para la construcción del XML. El primer elemento de la lista estará formado por la pregunta padre.
- ⇒ listaAuxiliar=lista donde almacenaremos las respuestas. Éstas serán agregadas a la lista principal.

Lo primero que se hace es recoger los valores que vienen del *mapper*; el marco de trabajo ya se encarga previamente de agrupar (ver Ilustración 10) los valores coincidentes, y mediante una estructura iterativa metemos en *listaValores* el tipo de mensaje y el mensaje, que se han pasado como parámetro desde el *mapper* y lo ha recogido *values*.

En una segunda estructura iterativa introducimos en *listaPrincipal* la pregunta que origina toda la jerarquía, y en la *listaAuxiliar* introducimos todos los mensajes asociados a la pregunta.

Para finalizar agregamos a la lista principal la lista auxiliar de manera que quede la siguiente estructura:

Key→["pregunta", ["respuesta1", "respuesta2"..."respuestaN"]]

```
def reducer(self, key, values):
    listaValores=[]
    listaPrincipal=[]
    listaAuxiliar=[]
    for v in values: #Metemos los valores que vienen en un matriz
        listaValores.append(v) #Lista con el tipo de mensaje y el
                               #mensaje asociado
    for valor in listaValores:
        if valor[0]=="question":#Si es una pregunta la metemos en la
                               #lista principal
            listaPrincipal.append(valor[1])
        else:
            listaAuxiliar.append(valor[1]) # Si son respuestas, las
            # vamos agregando a una lista auxiliar
    listaPrincipal.append(listaAuxiliar) #agregamos la lista de
    #respuestas a la lista principal
    yield key,xmlify.dumps(listaPrincipal,root = key) #Conversion a
    #XML indicando en el raíz el id del mensaje
```

Tabla 38 - Patrón Reducer estructuración jerárquica.

La *listaPrincipal* se la pasamos a la función *dumps*, que convierte la estructura en XML.

Podemos usar la estructura diccionarios de Python. En esta segunda versión del *Reducer*, podemos ver el ejemplo de uso en la siguiente tabla.

```
def reducer(self, key, values):
    diccionario=dict() #Para el caso que usemos diccionarios
    matrizParaXML=[]
    listaPrincipal=[]
    listaAuxiliar=[]
    for v in values:
        matrizParaXML.append(v)

    for valor in matrizParaXML:
        if valor[0]=="question":
            listaPrincipal.append(valor[1])
        else:
            listaAuxiliar.append(valor[1])
    listaPrincipal.append(listaAuxiliar)
    diccionario[key]=listaPrincipal #Caso de uso con diccionario
    yield key,xmlify.dumps(diccionario) # Conversion a XML
```

Tabla 39 - Versión del Reducer del patrón estructuración jerárquica usando diccionarios

Para las dos versiones, es importante destacar que hemos de configurar el protocolo de salida como:

OUTPUT_PROTOCOL = RawValueProtocol

Lo ideal en este patrón es que al ejecutarlo, la salida la derivemos directamente a un archivo y le añadamos la extensión xml; posteriormente editar ese archivo y agregarle por ejemplo la etiqueta **<raíz>contenido_generado_prototipo</raíz>**. Hay que tener en cuenta que el archivo creado, contendrá una línea por cada mensaje padre y sus mensajes hijos. Esto lo hemos definido con el protocolo de salida.

```
<n9426><n9426-item>&lt;p&gt;in 1st quiz of unit 2&lt;/p&gt;</n9426-
item><n9426-item /></n9426>

<n17209><n17209-item>"&lt;p&gt;Just reporting that an Error 500 page comes
up when I click on ""hottest"".&lt;/p&gt;"</n17209-item><n17209-item
/></n17209>

<n19203><n19203-item><n19203-item-item>&lt;p&gt;Python syntax and some
concepts about computer science.&lt;/p&gt;</n19203-item-item></n19203-
item></n19203>

<n19336><n19336-item>&lt;p&gt;who are from Bangladesh??&lt;/p&gt;</n19336-
item><n19336-item /></n19336>
```

Tabla 40 – Ejemplo de salida en XML de la ejecución del prototipo: "estructuración jerárquica".


```

-<raiz>
  -<n9426>
    <n9426-item><p>in 1st quiz of unit 2</p></n9426-item>
  </n9426>
  -<n17209>
    -<n17209-item>
      "<p>Just reporting that an Error 500 page comes up when I click on ""hottest"".</p>"
    </n17209-item>
  </n17209>
  -<n19203>
    -<n19203-item>
      -<n19203-item-item>
        <p>Python syntax and some concepts about computer science.</p>
      </n19203-item-item>
    </n19203-item>
  </n19203>
  -<n19336>
    <n19336-item><p>who are from Bangladesh??</p></n19336-item>
  </n19336>
</raiz>

```

Ilustración 43 – Muestra del ejemplo de la Tabla 40 después de añadirle la etiqueta <raíz> al principio y </raíz> al final

6.3.1.2. Análisis y aplicabilidad

Tal y como hemos indicado antes, lo más aconsejable es derivar a un archivo la salida para hacer un tratamiento a posteriori de éste. El marco *MapReduce* generará el formato de forma jerárquica agrupada por la clave que se indicó, pero puede darse el caso que haya que añadir un nivel superior a la jerarquía que no venga contemplada en los datos, o que haya que añadir el prólogo o algún encabezado, o si se genera algún error de caracteres extraños o de datos; aunque lo ideal es hacer un preprocesamiento de los datos para que no ocurra esto último.

Convertimos los datos a formatos jerarquizados para el almacenaje de documentos, intercambio y recuperación de datos entre sistemas y lenguajes, y además que este tipo de documentos suelen ser más ligeros y rápidos.

6.3.2. Ordenación de registros

La ordenación de registros suele ser un problema de fácil solución en la programación secuencial, pero ordenar registros en un marco de trabajo paralelo y distribuido como es *MapReduce*, no suele ser tarea sencilla. No obstante, las herramientas que el lenguaje nos proporciona nos permitirán realizar esta operación en pocas líneas; tan solo necesitamos un campo por el que poder ordenar y de lo demás, se encarga el marco de trabajo.

En este patrón lo que queremos hacer es ordenar todos los registros de una base de datos, el *Map* recoge bloques de registros que procesa en los nodos del *cluster* antes de llegar al *Reducer*; en el marco, necesitamos crear estructuras que almacenen cada conjunto de registros de cada *Mapper*, y se manden al *Reducer* para concatenarlos en una lista total de registros y posteriormente ordenarlos antes de emitirlos.

Vamos a utilizar estructuras de tipo listas, como hicimos en el patrón de filtrado *topN*. La diferencia es que en *topN* solo cogíamos unos datos numéricos, aquí cogemos todo el registro completo, y es importante destacar que necesitaremos un campo que permita hacer comparaciones para la ordenación, es decir, que lo más adecuado es que el campo sea de tipo numérico.

6.3.2.1. Prototipos

Prototipo 1 - Ordenación total

Vamos a desarrollar un ejemplo usando la tabla descrita en la página 50, capítulo 5, *Online_retail.csv*. Si queremos ordenar los registros por el importe de la venta (*unitPrice*), haremos uso de ese campo para este fin. Usaremos los métodos *mapper_init()*, *mapper()* y *mapper_final()* para el control de los bloques de la parte *Map*. En el *Reducer* uniremos todos los bloques y ordenaremos los registros.

Mapper

Dentro del *mapper_init()* inicializamos una estructura de tipo lista, que va a contener el bloque de registros del *mapper*. El método *mapper()*, recoge los registros de su bloque y los almacena en la estructura lista declarada en el *mapper_init()*. Por último, el método *mapper_final()* envía al *reducer* el bloque de registros. Se puede observar la secuencia de código en la Tabla 41.

```

def mapper_init(self):
    self.conjuntoRegistros = []

def mapper(self, _, line):
    linea=line.rstrip('/n').split(';')
    encontrado=re.search('[a-zA-Z]',linea[1])

    if encontrado==None:
        #Se agrega el registro completo a la lista
        self.conjuntoRegistros.append(linea)

def mapper_final(self):
    #Mandamos conjunto de registros al Reducer para reunirlos todos.
    yield None, self.conjuntoRegistros

```

Tabla 41 – Mapper para la versión 1 del prototipo Ordenación de registros.

Hasta ahora lo que se ha hecho es recoger los registros de cada bloque en el *mapper*; podemos observar que no usamos una clave específica, solamente el valor, que es el bloque de registros. Como lo que queremos es ordenar todos los registros, la clave será “null”, que como es la misma para todos los *map*, pues en la salida van al mismo proceso *reduce*, agrupándose todos por esa misma clave. Si solo ejecutamos el *mapper*, podremos ver como salen los bloques de cada *mapper* distribuido en su ejecución paralela en el *cluster*.

Reducer

Una vez llegan los valores (que son los bloques de registros que llegan de cada mapper) al *reducer*, hemos de agregarlos a una estructura de tipo lista, y mediante el método sort combinado con una expresión lambda [38], ordenamos y, con una estructura iterativa, emitimos cada elemento de la lista.

```

def reducer(self, _, values):
    listaCompletaRegistros=[]

    for valores in values:
        listaCompletaRegistros.extend(valores)
        #Expresión lambda para indicar
        # por qué campo tiene que ordenar la lista.
        listaCompletaRegistros.sort(key = lambda precio:
float(precio[5]))

    for registro in listaCompletaRegistros:
        yield _, registro

```

Tabla 42 – Reducer para la versión 1 del prototipo Ordenación de registros.

```

null    ["536552", "20950", "", "1", "01/12/2010 14:34", "0", "", "
United Kingdom"]

null    ["536553", "37461", "", "3", "01/12/2010 14:35", "0", "", "
United Kingdom"]

null    ["536554", "84670", "", "23", "01/12/2010 14:35", "0", "", "
United Kingdom"]

null    ["536589", "21777", "", "-10", "01/12/2010 16:50", "0", "", "
United Kingdom"]

```

Tabla 43 – Resultado parcial de la ejecución del prototipo descrito en la Tabla 41 y Tabla 42

Prototipo 2 - Ordenación agrupada con clave

En el caso que quisiéramos ordenar los registros por categorías, sí nos va a ser necesaria definir una clave. Vamos a trabajar con la misma tabla usada para el prototipo 1. En el ejemplo con el que hemos trabajado en el anterior apartado, si deseamos ordenar los registros en base a cada cliente necesitaremos el campo que contenga el nombre del cliente o el código del cliente.

Aquí el marco de trabajo hará su función y el *reducer* se encargará de agrupar los registros por el código del cliente, tan solo hemos de pasar desde el *mapper* el campo que lo contiene.

```

def mapper_init(self):
    self.conjuntoRegistros = []

def mapper(self, _, line):
    self.linea=line.rstrip('\n').split(';')
    encontrado=re.search('[a-zA-Z]',self.linea[1])
    if encontrado==None:
        self.conjuntoRegistros.append(self.linea)
def mapper_final(self):
    yield self.linea[6], self.conjuntoRegistros

#####Reducer#####

def reducer(self, cliente, values):
    listaCompletaRegistros=[]
    for valores in values:
        listaCompletaRegistros.extend(valores)
    listaCompletaRegistros.sort(key = lambda precio:
float(precio[5]))
    for registro in listaCompletaRegistros:
        yield cliente, registro

```

Tabla 44 - Prototipo 2 de Ordenación total de registros.

```

"13263"      ["548664", "22167", " OVAL WALL MIRROR DIAMANTE ", "5",
"01/04/2011 14:45", "9.95", "14911", "EIRE"]

"13263"      ["548664", "22796", "PHOTO FRAME 3 CLASSIC HANGING", "2
", "01/04/2011 14:45", "9.95", "14911", "EIRE"]

"13263"      ["548664", "22767", "TRIPLE PHOTO FRAME CORNICE ", "2",
"01/04/2011 14:45", "9.95", "14911", "EIRE"]

"13263"      ["548664", "22763", "KEY CABINET MA CAMPAGNE", "2", "01
/04/2011 14:45", "9.95", "14911", "EIRE"]

"13263"      ["548666", "22634", "CHILDS BREAKFAST SET SPACEBOY ", "
2", "01/04/2011 14:58", "9.95", "13124", "United Kingdom"]

```

Tabla 45 – Resultado parcial de la ejecución del prototipo descrito en la Tabla 44

6.3.2.2. Análisis y aplicabilidad

El hecho de poder ordenar los datos provee a estos de propiedades que pueden ser bastante útiles. Por ejemplo, si ordenamos los registros por fecha, nos proporciona una línea de tiempo en la que analizar tendencias o resultados. Por otro lado, puede mejorar la localización de elementos en conjuntos de datos ordenados, a la vez que se puede hacer mejores búsquedas binarias, entre otras ventajas.

6.3.3. Mezclar y desordenar

Durante todo este apartado hemos tratado de ordenar o clasificar los datos de alguna manera. En este apartado lo que vamos a tratar de hacer es lo contrario, es decir, vamos a desordenar o mezclar los datos.

6.3.3.1. Prototipos

Prototipo 1 - Mezclar

En este ejemplo lo que vamos a hacer es mezclar o barajar(shuffling) los datos que llegan ordenados. Primero los recogemos en el *Map*; no es necesaria la clave, solo el valor. Dividimos con *Split* y emitimos al *Reducer*.

Dentro del *Reducer*, vamos metiendo las líneas en una estructura de tipo lista, en la que haremos posteriormente la mezcla de todos los registros usando el método *shuffle* de la librería *random*, en el que le pasamos la lista de registros y el método para mezclar los registros de la tabla; emitiendo posteriormente.

```

def mapper(self, _, line):
    linea=line.split(";")
    yield _,linea

def reducer(self, _, values):
    listaRegistros=[]
    for registro in values:
        listaRegistros.append(registro)
        # Se mezclan o desordenan aleatoriamente
        random.shuffle(listaRegistros,random.random)
    for record in listaRegistros:
        yield _,record

```

Tabla 46 - Prototipo mezclar registros

Este prototipo es sobre todo para tablas o registros que estén ordenados previamente, ya sea desde el origen, o que se hayan ordenado para operar con ellos, y posteriormente se tengan que desordenar.

Prototipo 2 - Anonimizar y mezclar

En el siguiente ejemplo vamos a anonimizar los datos y desordenarlos posteriormente. Para anonimizar los registros hemos de eliminar o modificar los datos que identifiquen a los usuarios, clientes, pacientes, etc. Cuando entren los registros al *Map*, hacemos las tareas para anonimizar, que podemos hacerlo prescindiendo de los campos que contengan datos sensibles, o modificando o desvirtuando la información en algunos de los campos del registro. En el siguiente caso vamos a eliminar el código del cliente/usuario, y se va a modificar la fecha para que solo aparezca el año en que se hizo la operación. Para ello vamos a usar el conjunto de datos descrito en la página 50 del capítulo 5, *Online_retail.csv*.

```

def mapper_init(self):
    self.registros = []

def mapper(self, _, line):
    linea=line.split(";")
    encontrado=re.search('[a-zA-Z]',linea[0])
    if encontrado==None:
        # Truncamos la fecha
        fecha = datetime.strptime(linea[4], '%d/%m/%Y %H:%M')
        del(linea[6]) # Eliminamos el código del cliente
        linea[4]=fecha.year # Asignamos el año al campo fecha

        #agregamos a la estructura lista
        self.registros.append(linea)

def mapper_final(self):
    #Se manda al Reducer el bloque de registros
    for registro in self.registros:
        yield None, registro

```

Tabla 47 - Mapper para el prototipo anonimizar y mezclar.

Como podemos observar, tan solo hemos de retener en una estructura de tipo lista el bloque de datos en el *mapper*. Aquí vamos a usar el *mapper_init* para crear la estructura, el *mapper* para los procesos de eliminación o modificación de registros, y el *mapper_final* para mandar el resultado al *Reducer*, que este es el mismo que el de la Tabla 46.

El formato de salida de los resultados, son iguales que los que aparecen en la Tabla 43 y la Tabla 45.

6.3.3.2. Análisis y aplicabilidad

El caso de mezclar y anonimizar los datos, toma especial relevancia de cara a que muchas organizaciones necesiten salvaguardar la identidad o privacidad de los usuarios de los que han extraído éstos. Por ejemplo, para estudiar el comportamiento de los clientes en una tienda o supermercado al hacer uso de su tarjeta de fidelización, nos están dando sus datos personales, posteriormente éstos son analizados de forma interna o mediante entidades externas, que no deben tener acceso a información sensible. Sea como sea, hay que anonimizar esos datos da alguna forma, y quedarnos sólo con lo que necesitemos para extraer la información necesaria.

Por otro lado, a veces es necesario extraer un muestreo aleatorio del conjunto de datos. Lo ideal en estos casos, es barajar los datos y posteriormente extraer, por ejemplo, los cien o los mil primeros registros que nos permitan analizar ese conjunto de datos, simplemente cogiendo una muestra de éstos.

Por tanto, estos patrones nos permiten entre otras cosas, extraer muestreos aleatorios, y preparar los datos para evitar fugas de información sensible.

6.4. Patrones de Unión

Generalmente las uniones de tablas de datos suelen realizarse para enriquecer un conjunto de referencia más pequeño. Cuando hacemos uniones usando SQL tan solo hemos de lanzar la instrucción y el motor de base de datos se encargará de realizar el resto. En el marco *MapReduce* tenemos que trabajar con pares <clave, valor> por tanto, necesitaremos un campo que disponga de una clave común o clave externa de las dos tablas a unir, y un valor que será el registro completo. El *Map* se encargará de la operación fundamental de detección del origen del registro de entrada y extracción o asignación de la clave, y el *Reducer* se encargará de detectar de donde viene la <clave, valor>, y hacer la unión correspondiente.

Dentro de este último grupo de patrones, desarrollaremos prototipos similares para realizar la unión interna, externa, completa, antinión y producto cartesiano. Todos ellos

comparten la misma estructura y líneas de código, tan solo cambiará el algoritmo con el que se hace la unión dependiendo de su tipo.

El patrón de unión replicada cambia con respecto a los demás, sobre todo por su complejidad al usar la caché distribuida de Hadoop, para la carga de la tabla a la que irán unidas todas las que entren por el *stream*. Veremos varios de estos prototipos, donde estudiaremos sobre todo las formas que tendremos de cargar datos en memoria para la unión.

6.4.1. Estructura de los patrones

En todos los patrones excepto en el de *Unión Replicada*, cada tabla entrará por un archivo distinto, y aquí es donde está la operación fundamental para realizar las uniones con éxito: detectar de qué archivo es del que están entrando los registros por el *stream*. Esto será detectado en el *Mapper*, y lo haremos con el parámetro configurado: *map_input_file* [39], que nos devuelve el nombre del fichero que está siendo leído del *stream*.

El parámetro *map_input_file* devuelve una cadena de caracteres en formato: *file://...//nombre_fichero* si ejecutamos en local, o *hdfs://...//nombre_fichero* si ejecutamos en el *cluster*; como necesitaremos el nombre del archivo para ciertas operaciones condicionales con el objetivo de evitar errores, se ha creado un método: *limpiarNombreArchivo()*, que recibe la cadena y nos devuelve el nombre del archivo limpio.

```
def limpiarNombreArchivo(self,archivo):
    encontradaBarra=False
    tamaño=len(archivo)
    posición=tamaño-1
    while encontradaBarra==False or posición==0:
        if archivo[posición]=="\/":
            encontradaBarra=True
            return archivo[posición+1:tamaño]
        else:
            posición-=1
    if posición==0:
        return archivo
```

Tabla 48 - Código de la función para "limpiar" los nombres de los archivos que entran por *stream*.

Este método comienza a leer la cadena desde el final hasta que encuentra el carácter '\', momento en el cual extrae la subcadena desde la *posición+1*, que es donde encuentra la barra, hasta el final, que corresponde al nombre del archivo. El método es llamado desde el *mapper_init* y el resultado será guardado en *namefile*.

El *Mapper* controlará (como en todos los prototipos anteriores) que, mediante expresión regular, no procese los nombres de los campos de las tablas, y también verificará de qué fichero está entrando la línea o registro de datos.

```

if self.namefile=="clientes.csv":
    linea.append(self.namefile)
    clave=linea[0]
    yield clave,linea
else:
    linea.append(self.namefile)
    clave=linea[6]
    yield clave,linea

```

Tabla 49 - Estructura condicional del mapper para controlar ficheros de entrada.

Por último, se emite desde el Mapper la clave común de cada tabla según corresponda, y la línea como valor.

Para poder identificar en el *Reducer* de qué archivo ha venido cada línea, añadimos a ésta un campo cuyo contenido nos va a informar de ello. El contenido puede ser cualquier elemento identificativo; en los prototipos se ha optado por introducir el nombre del fichero.

En el *Reducer* se crean dos listas, en una almacenaremos los registros que vienen de un archivo, y en la otra lista los del otro archivo; como todo viene en un bloque de datos, para poder identificar donde va cada registro, usamos el campo que creamos en el *Mapper* donde almacenamos el nombre del archivo.

```

def reducer(self,key,values):
    listaA=[]
    listaB=[]
    for valor in values:
        if valor[len(valor)-1]=="tablaA.csv":
            listaA.append(valor)
        else:
            listaB.append(valor)

    #### Aquí insertamos en algoritmo de unión ####

```

Tabla 50 - Estructura general del código del Reducer, antes de aplicar el algoritmo de unión.

Por último, se procede a aplicar el algoritmo que corresponda dependiendo del tipo de unión. En todas las uniones nos aprovecharemos del marco MapReduce en el sentido que no será necesario ninguna línea de código para hacer comparaciones, el mismo contexto se encarga de ello. La salida del *Reducer* será la clave común, y el valor, el registro unido. Pasaremos a explicar cada relación y su correspondiente algoritmo en los siguientes apartados.

6.4.2. Unión interna

Esta unión, también llamada *unión natural*, es una operación binaria que permite combinar dos tablas mediante un campo clave común, que ha de estar presente en las dos relaciones. Selecciona los registros de cada tabla cuya clave común coincida, uniéndolos y formando un solo registro, y una sola tabla con los registros coincidentes unidos [40] [7, p. 103]. Veámoslo con un ejemplo. Disponemos de dos tablas de datos de una cadena de tiendas:

➔ Tabla *artículos_stock*: contiene tres campos

- *Cod_art*: código del artículo.
- *Cod_Tienda*: código con el que identificamos la tienda.
- *Stock_art*: unidades disponibles del artículo en la tienda.

➔ Tabla *tiendas*.

- *Cod_Tienda*: Código con el que identificamos la tienda
- Localización: ciudad donde se encuentra la tienda

Cod_art	Cod_Tienda	Stock_art
1001	1	20
1002	2	10
1003	1	15
1004	5	10
1005	2	5
1006	3	3
1007	1	8
1008	2	5
1009	4	1
1010	3	20

Tabla 51 – artículos_stock

Cod_Tienda	Localizacion
1	Santander
2	Albacete
3	Granada
6	Barcelona

Tabla 52 – tiendas

Pongamos que necesitamos saber el stock de artículos que tenemos en cada tienda, usamos como campo clave el código de la tienda identificado por el campo *Cod_Tienda*. Al hacer la operación unión interna, el resultado será el siguiente:

Cod_Tienda	Localizacion	Cod_art	Cod_Tienda	Stock_art
1	Santander	1001	1	20
2	Albacete	1002	2	10
1	Santander	1003	1	15
2	Albacete	1005	2	5
3	Granada	1006	3	3
1	Santander	1007	1	8
2	Albacete	1008	2	5
3	Granada	1010	3	20

Tabla 53 - Resultado de ejecutar una unión interna

Observamos los registros resultantes de unir la tabla *tiendas* con la tabla *artículos_stock* que, si hacemos una interpretación del resultado, la tabla resultante nos muestra la cantidad de distintos productos que quedan en cada una de las tiendas. Los registros no coincidentes por su campo clave, se descartan.

El prototipo de algoritmo aplicado en el marco *MapReduce* para realizar esta operación, es un recorrido bidimensional de estas tablas:

```
#Nos aseguramos que las listas están llenas
if listaA and listaB:
    for valor_A in listaA:
        for valor_B in listaB:
            yield key, (valor_A, valor_B)
```

Tabla 54 - Algoritmo para la Unión Interna

La *listaA* recoge los registros de la tabla *tiendas*, y la *listaB* recoge los registros de la tabla *artículos_stock*. Para cada valor en la *listaA*, comparamos con todos los valores de la *listaB*. Si encuentra una clave coincidente, emite la unión de los dos registros. Aquí podemos ver que no se hace ninguna comparación, de eso se encarga el marco *MapReduce*.

```
"1" [["1", "Santander", "tiendas.csv"], ["1001", "1", "20", "art.csv"]]
"1" [["1", "Santander", "tiendas.csv"], ["1003", "1", "15", "art.csv"]]
"1" [["1", "Santander", "tiendas.csv"], ["1007", "1", "8", "art.csv"]]
"2" [["2", "Albacete", "tiendas.csv"], ["1002", "2", "10", "art.csv"]]
"2" [["2", "Albacete", "tiendas.csv"], ["1005", "2", "5", "art.csv"]]
"2" [["2", "Albacete", "tiendas.csv"], ["1008", "2", "5", "art.csv"]]
```

Tabla 55 - Salida parcial de la ejecución de la unión interna.

6.4.3. Unión externa

Esta operación es una ampliación de la unión interna, que nos permite trabajar con información ausente [40]. Como hemos podido ver en el apartado anterior, cuando hemos

hecho la unión interna, en la tabla resultante, se perdió información, concretamente los artículos 1004 y 1009, al no haber tiendas que tuvieran esos artículos en su stock, y la tienda 6 tampoco aparece, al no tener ningún artículo en su almacén esa tienda.

Con la operación unión externa, tratamos de evitar la pérdida de información. Existen tres tipos de unión externa: unión por la izquierda, unión por la derecha y unión completa, que al realizar cualquiera de las tres operaciones, añaden campos adicionales (valores nulos) al resultado de estas.

6.4.3.1. Unión externa por la izquierda

Esta unión externa, toma los registros de la primera tabla que entra, cuya clave no coincide con ninguna clave de la segunda tabla, rellenando con valores nulos los campos de la segunda tabla [40, p. 52], y añadiendo esto al resultado de la unión interna.

Cod_Tienda	Localizacion	Cod_art	Cod_Tienda	Stock_art
1	Santander	1001	1	20
2	Albacete	1002	2	10
1	Santander	1003	1	15
2	Albacete	1005	2	5
3	Granada	1006	3	3
1	Santander	1007	1	8
2	Albacete	1008	2	5
3	Granada	1010	3	20
6	Barcelona	nulo	nulo	nulo

Tabla 56 - Unión por la izquierda de tabla tiendas y tabla_stock

La interpretación a este resultado es que la tienda 6, no tiene actualmente ningún artículo.

El algoritmo para el paradigma MapReduce se implementa con el siguiente prototipo:

```

for valorA in listaA:
    if listaB:
        for valorB in listaB:
            yield valorA, valorB
    else:
        #Si la listaB está vacía
        yield valorA, "null"

```

Tabla 57 - Algoritmo para la Unión externa por la izquierda

De cada valor de la primera tabla, comprobamos que exista en la segunda tabla, si así fuera, se hace la unión natural, si no, se hace la unión por la izquierda, añadiendo el valor nulo a los campos de la segunda tabla.

```
[ "2", "Albacete", "tiendas.csv" ] [ "1008", "2", "5", "artículos_stock.csv" ]
[ "3", "Granada", "tiendas.csv" ] [ "1006", "3", "3", "artículos_stock.csv" ]
[ "3", "Granada", "tiendas.csv" ] [ "1010", "3", "20", "artículos_stock.csv" ]
[ "6", "Barcelona", "tiendas.csv" ]      "null"
```

Tabla 58 - Resultado parcial de la ejecución de una unión externa por la izquierda.

6.4.3.2. Unión externa por la derecha

Esta unión toma los valores de la segunda tabla cuya clave no está en la primera tabla, y rellena con valores nulos la parte de ésta, agregando el resultado a la unión natural [40]. Es el proceso simétrico a la reunión por la izquierda.

Si ponemos como ejemplo la tabla *tiendas* como primera tabla, y la *tabla_stocks* como la segunda tabla, veremos que aparecen artículos que pertenecen a tiendas que ya no existen.

Cod_Tienda	Localizacion	Cod_art	Cod_Tienda	Stock_art
1	Santander	1001	1	20
2	Albacete	1002	2	10
1	Santander	1003	1	15
nulo	nulo	1004	5	10
2	Albacete	1005	2	5
3	Granada	1006	3	3
1	Santander	1007	1	8
2	Albacete	1008	2	5
nulo	nulo	1009	4	1
3	Granada	1010	3	20

Tabla 59 - Resultado de la unión externa por la derecha

El algoritmo usado en nuestro paradigma es el siguiente:

```
for valorB in listaB:
    if listaA:
        for valorA in listaA:
            yield valorA, valorB
        else:
            yield "null", valorB
```

Tabla 60 - Algoritmo unión externa por la derecha

Como hacemos la unión por la derecha, tenemos que empezar a recorrer la segunda tabla, que es la que manda, haciendo las comprobaciones por cada valor de ésta.

```
["2", "Albacete", "tiendas.csv"] ["1008", "2", "5", "art.csv"]
["3", "Granada", "tiendas.csv"]  ["1006", "3", "3", "art.csv"]
["3", "Granada", "tiendas.csv"]  ["1010", "3", "20", "art.csv"]
"null"                           ["1009", "4", "1", "art.csv"]
"null"                           ["1004", "5", "10", "art.csv"]
```

Tabla 61 - Resultado parcial de la salida de la unión externa por la derecha.

6.4.3.3. Unión completa

La unión externa completa realiza las dos operaciones anteriores en una sola. Rellena los registros de la primera tabla que no coinciden con ningún registro de la segunda tabla, y los registros de la segunda tabla que no coinciden con los de la primera, añadiendo estos resultados a la unión natural; en resumen, se unen todos los registros.

Cod_Tienda	Localizacion	Cod_art	Cod_Tienda	Stock_art
1	Santander	1001	1	20
2	Albacete	1002	2	10
1	Santander	1003	1	15
nulo	nulo	1004	5	10
2	Albacete	1005	2	5
3	Granada	1006	3	3
1	Santander	1007	1	8
2	Albacete	1008	2	5
nulo	nulo	1009	4	1
3	Granada	1010	3	20
6	Barcelona	nulo	nulo	nulo

Tabla 62 - Resultado de ejecutar una unión completa

Las líneas de código que definen el algoritmo de la unión completa en el *Reducer*, los podemos ver a continuación, pudiendo comprobar que este puede ser algo más complejo que los vistos anteriormente.

```
# Si listaA no está vacía, comprobamos cada una de sus entradas
if listaA:
    for valorA in listaA: # Por cada entrada en la listaA
        if listaB: #Si la listaB no está vacía, unimos A con B
            for valorB in listaB:
                yield valorA, valorB
            else: #Si no es el caso, sacamos A con union nula
                yield valorA, "null"
else:
    #Si la listaA está vacía, emitimos solo los elementos de listaB
    for valorB in listaB:
        yield "null", valor
```

Tabla 63 - Algoritmo unión completa.

["1", "Santander", "tiendas.csv"]	["1001", "1", "20", "art.csv"]
["1", "Santander", "tiendas.csv"]	["1003", "1", "15", "art.csv"]
["1", "Santander", "tiendas.csv"]	["1007", "1", "8", "art.csv"]
["2", "Albacete", "tiendas.csv"]	["1002", "2", "10", "art.csv"]
["2", "Albacete", "tiendas.csv"]	["1005", "2", "5", "art.csv"]
["2", "Albacete", "tiendas.csv"]	["1008", "2", "5", "art.csv"]
["3", "Granada", "tiendas.csv"]	["1006", "3", "3", "art.csv"]
["3", "Granada", "tiendas.csv"]	["1010", "3", "20", "art.csv"]
"null"	["1009", "4", "1", "art.csv"]
"null"	["1004", "5", "10", "art.csv"]
["6", "Barcelona", "tiendas.csv"]	"null"

Tabla 64 - Resultado de la ejecución de la unión completa.

6.4.4. Antiunión

La antiunión podemos definirla como una unión completa menos la unión interna. Unimos los registros cuya clave de la primera tabla, no está en la segunda y al contrario. Es lo opuesto a la unión externa completa.

Cod_Tienda	Localizacion	Cod_art	Cod_Tienda	Stock_art
nulo	nulo	1004	5	10
nulo	nulo	1009	4	1
6	Barcelona	nulo	nulo	nulo

Tabla 65 - Resultado de ejecución de la antiunión

Las líneas de código que añadiríamos al *Reducer*, podemos observarlas a continuación.

```
#Si la listaA o la listaB están vacías
if not listaA or not listaB:
    # Emitimos los valores vacíos de las dos listas
    for valorA in listaA:
        yield valorA, "null"
    for valorB in listaB:
        yield "null", valorB
```

Tabla 66 - Algoritmo antiunión.

"null"	["1009", "4", "1", "articulos_stock.csv"]
"null"	["1004", "5", "10", "articulos_stock.csv"]
["6", "Barcelona", "tiendas.csv"]	"null"

Tabla 67 - Resultado de la antiunión.

6.4.5. Producto cartesiano

6.4.5.1. Definición

El producto cartesiano o producto cruzado, toma cada registro de la primera tabla y lo une a todos los registros de la segunda tabla, a diferencia de las otras operaciones de unión, el producto cartesiano no tiene una clave común o clave externa, simplemente empareja cada registro de un conjunto de datos con los registros del otro conjunto de datos.

Dicho de otra forma, si disponemos de una primera tabla que contiene n registros y una segunda tabla que contiene m registros, el producto cruzado de la primera tabla con la segunda dará como resultado una tabla con $n \times m$ registros [7].

El resultado de realizar la operación cartesiana con las tablas *tiendas* y *artículos_stock* podemos verlo en la Tabla 68.

Cod_Tienda	Localizacion	Cod_art	Cod_Tienda	Stock_art
1	Santander	1001	1	20
1	Santander	1002	2	10
1	Santander	1003	1	15
1	Santander	1004	5	10
1	Santander	1005	2	5
1	Santander	1006	3	3
1	Santander	1007	1	8
1	Santander	1008	2	5
1	Santander	1009	4	1
1	Santander	1010	3	20
2	Albacete	1001	1	20
2	Albacete	1002	2	10
2	Albacete	1003	1	15
2	Albacete	1004	5	10
2	Albacete	1005	2	5
2	Albacete	1006	3	3
2	Albacete	1007	1	8
2	Albacete	1008	2	5
2	Albacete	1009	4	1
2	Albacete	1010	3	20
3	Granada	1001	1	20
3	Granada	1002	2	10
3	Granada	1003	1	15
3	Granada	1004	5	10
3	Granada	1005	2	5
3	Granada	1006	3	3
3	Granada	1007	1	8
3	Granada	1008	2	5

3	Granada	1009	4	1
3	Granada	1010	3	20
6	Barcelona	1001	1	20
6	Barcelona	1002	2	10
6	Barcelona	1003	1	15
6	Barcelona	1004	5	10
6	Barcelona	1005	2	5
6	Barcelona	1006	3	3
6	Barcelona	1007	1	8
6	Barcelona	1008	2	5
6	Barcelona	1009	4	1
6	Barcelona	1010	3	20

Tabla 68 - Resultado de la operación Producto Cartesiano

6.4.5.2. Prototipo

Una de las diferencias más destacables de esta unión con respecto a la unión interna y la unión externa, es que no va a ser necesaria una clave común o externa para las tablas de datos que entren por el *stream*. Esa diferencia es la que va a marcar el desarrollo del patrón de producto cartesiano.

Mapper

Cuando realizamos las operaciones en el *Mapper*, no extraemos ninguna clave externa de las tablas, simplemente definimos una clave igual para cada uno de los registros de cada tabla, y la lanzamos al *Reducer* junto con el valor, que será el registro extraído. Lo que no cambiará es el origen del registro, eso siempre hemos de controlarlo para saber que registro de la primera tabla es el que hay que unir con los demás registros de la segunda tabla.

```
def mapper_init(self):
    self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_
file'))
def mapper(self,_,line):
    linea=line.split(';')
    encontrado=re.search('[a-zA-Z]',linea[0])
    if encontrado==None:
        if self.namefile=="tiendas.csv":
            linea.append("file_1")#Añadimos identificador de archivo
            yield "llave",linea #Llave única
        else:
            linea.append("file_2")
            yield "llave",linea #Llave única
```

Tabla 69 - Mapper del prototipo producto cartesiano.

Observamos que la clave que definimos puede ser la que escojamos, teniendo en cuenta que tiene que ser igual tanto para los registros que vienen de la primera tabla, como para los que vienen de la segunda.

Reducer

El *Reducer* se encargará de recoger los registros de cada tabla, almacenarlos en sendas listas, y aplicar posteriormente el mismo algoritmo que para la unión interna.

```
def reducer (self, key, values):
    listaA=[]
    listaB=[]
    #Llenamos las dos listas
    for valor in values:
        if valor[len(valor)-1]=="file_1":
            listaA.append(valor)
        else:
            listaB.append(valor)

    ##### Producto Cartesiano #####
    if listaA and listaB:
        for valor_A in listaA:
            for valor_B in listaB:
                yield key, (valor_A, valor_B)
```

Tabla 70 - Reducer y algoritmo de unión del prototipo producto cartesiano

El marco se encargará de realizar la agrupación de los registros, siguiendo la definición del producto cartesiano.

```
"llave" [["1", "Santander", "file_1"], ["1001", "1", "20", "file_2"]]
"llave" [["1", "Santander", "file_1"], ["1002", "2", "10", "file_2"]]
"llave" [["1", "Santander", "file_1"], ["1003", "1", "15", "file_2"]]
"llave" [["1", "Santander", "file_1"], ["1004", "5", "10", "file_2"]]
"llave" [["1", "Santander", "file_1"], ["1005", "2", "5", "file_2"]]
```

Tabla 71 - Salida parcial de la ejecución del producto cartesiano.

6.4.5.3. Análisis y aplicabilidad

Esta unión produce una explosión masiva en el tamaño de los datos, debe usarse con moderación por que suele ser una operación muy costosa de realizar independientemente de donde se implemente, y *MapReduce* no es una excepción. Un trabajo que use este patrón puede tardar mucho en completarse, por lo que todo preprocesamiento de los datos que se haga puede ser beneficioso para mejorar el tiempo de ejecución. En la mayoría de los casos de uso, se suele usar esta unión para el análisis de similitud de documentos o publicaciones, y en todo caso aplicarlo cuando el tiempo de ejecución no sea un problema para lo que lo estemos aplicando.

6.4.6. Unión replicada

6.4.6.1. Definición

La unión replicada es un tipo especial de operación, que permite crear una combinación de registros a partir de uno o varios conjuntos de datos. La operación de unión se realiza solo en la parte del *Mapper*, no es necesario el *Reducer*, lo que permite aumentar la rapidez en la ejecución, al no hacer la fase de barajar y ordenar (shuffle and sort), ni la fase de reducción [7].

La idea es cargar la primera tabla en memoria y la segunda o más tablas introducirlas mediante el *stream*, realizando la unión en el *Mapper*. Es decir, al contrario que hacíamos en los patrones de unión interna, por la izquierda, derecha, etc. que metíamos las dos tablas que se iban a unir, aquí una tabla la metemos en memoria y la otra u otras, la metemos por el *stream* para unirlos en el *Mapper*.

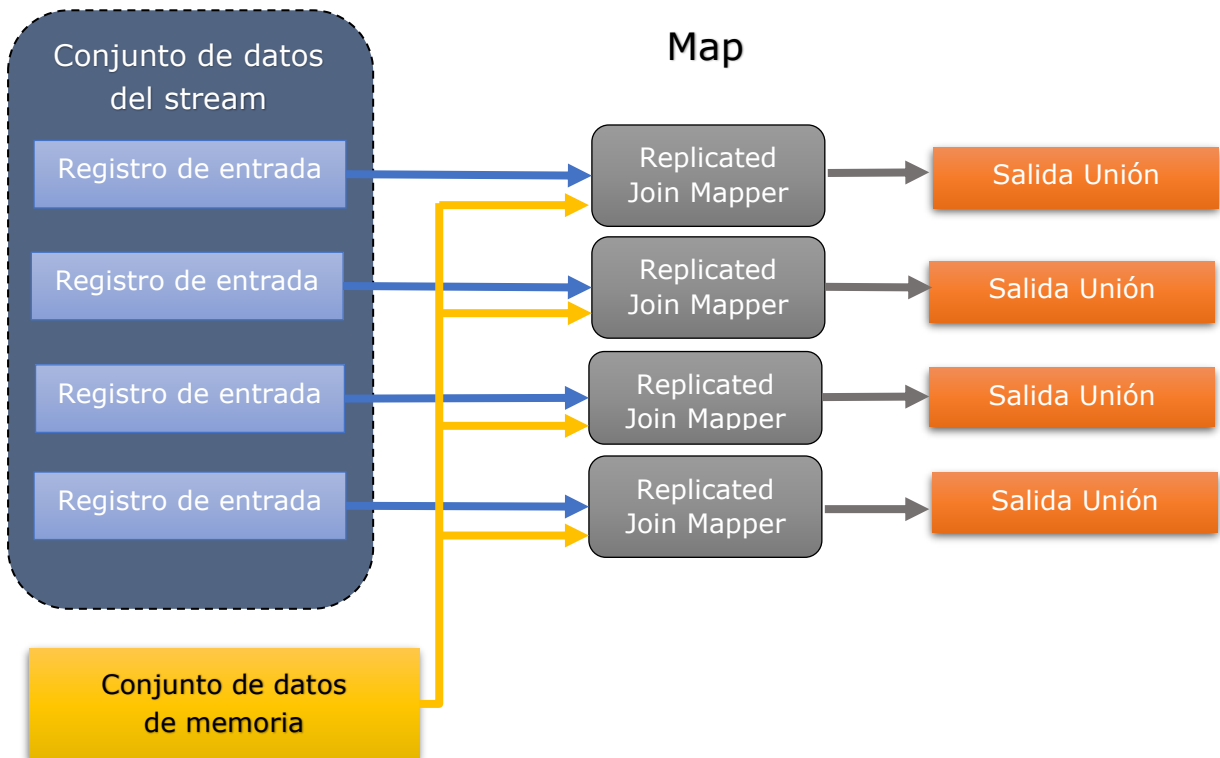


Ilustración 44 - Esquema de funcionamiento de la Unión Replicada [7].

En este caso, solo desarrollamos una función encargada de montar el fichero en memoria; esta función será llamada desde el *mapper_init*.

6.4.6.2. Prototipo

Este prototipo contendrá la clase que hereda de *MRJob* y la función que cargará en memoria la primera tabla. La función recibirá el archivo con la tabla, y devolverá una estructura diccionario en forma <clave, valor>, donde la clave será la clave externa y el valor el registro completo. De esta forma, la tabla o tablas que entren desde el *stream*, intervendrán en la unión con los registros que hay en el diccionario cargado en memoria.

Comenzamos el prototipo cargando en memoria el fichero con la primera tabla (tabla de la izquierda) desde HDFS. Para ello disponemos de la instrucción que nos proporciona MRJob llamada FILES [27] [41] con la que podemos cargar los ficheros que necesitemos trabajar. El formato sería:

FILES=['directorio/nombre_fichero.csv'] → Para cargar cuando estemos en local.

FILES=['hdfs:///directorio/nombre_fichero.csv'] → Para cargar cuando estemos en el cluster.

Una vez tenemos el fichero cargado en el entorno, lo almacenamos en una variable de tipo *file*:

```
Fichero='nombre_fichero.csv'
```

Desde el *mapper_init* cargamos la estructura diccionario en memoria, ayudándonos de la función *cargarFicheroEnMemoria*, que le pasamos como argumento el fichero con la tabla, y nos devolverá el diccionario.

```
def mapper_init(self):
    self.dicTablaA=cargarFicheroEnMemoria(self,self.fichero)
```

La función *cargarFicheroEnMemoria* recibe el fichero como argumento, y a continuación, cargamos cada línea del fichero en una estructura de tipo lista llamada *tablaEnMemoria*.

```
def cargarFicheroEnMemoria(self,fichero):
    diccionario={}
    with open(fichero) as f:
        self.tablaEnMemoria = set(line.strip() for line in f)
```

Como sabemos, el formato de la lista es de tipo registro (una lista de listas), y para que nos resulte más sencillo trabajar en el marco, vamos a crear una estructura diccionario para tener el contenido de memoria en formato <clave, valor>. Podemos ver esta estructura en la tabla

```

for linea in self.tablaEnMemoria:
    encontrado=re.search('[a-zA-Z]',linea[0])
    if encontrado==None:
        datos = linea.split(";")
        #Cambiar el INDICE_CLAVE dependiendo de la
        #posición en la que éste, esté en la tabla
        diccionario[datos[INDICE_CLAVE]]=datos
return diccionario

```

Tabla 72 - Líneas de código para rellenar una estructura diccionario

La clave será la que se utilice para la unión, y el valor será el registro completo. Es importante tener en cuenta donde se encuentra la clave en la fuente de datos, ya que dependiendo de donde esté, habrá que variar en la asignación del valor al diccionario, el índice con el que se identificará ese registro (que será la clave externa).

Ya disponemos de la primera tabla cargada en la memoria, por tanto, cuando comienza la ejecución de los *mapper* en los nodos, las comparaciones y uniones se hacen en estos, usando la tabla que está en memoria y todas las tablas o ficheros que van entrando por el *stream*.

```

def mapper(self,_,line):
    self.linea=line.split(';')
    encontrado=re.search('[a-zA-Z]',self.linea[0])
    if encontrado==None:
        self.clave=self.linea[1]
        if self.clave in self.dicTablaA:
            yield self.clave,
                (self.linea,self.dicTablaA.get(self.clave))
        else:
            yield self.clave,(self.linea,"null")

```

Tabla 73 - Mapper para la unión replicada.

En el fragmento de código anterior, podemos ver como extrae la clave (hay que cambiar el índice dependiendo de la fuente) de la tabla que entra por stream, y comprueba que la clave del fichero de entrada esté en el diccionario cargado en memoria. Si encuentra la clave en el diccionario, emitimos (yield) el resultado de la unión. Si no está, emitimos la línea del archivo con el valor nulo, al no tener valor complementario.

Por aclarar algo más el funcionamiento de esta unión, vamos a seguir con el ejemplo de las tablas de la cadena de supermercados, y vamos a suponer que disponemos de una sola tabla *tiendas*, y varias tablas *artículos_stock*, donde se recoge el inventario de artículos de todas esas tiendas. Una vez recogidas todas las tablas con los datos, es necesario asociarlas a las tiendas, y en vez de ir haciendo una unión por cada una de las tablas que tengamos, cargaríamos en memoria la tabla *tiendas*, y las demás tablas: *artículos_stock*,

artículos_stock2 y *artículos_stock3*, entrarían una detrás de otra por el *stream*, produciéndose una unión replicada, con una sola ejecución del programa *MapReduce*.

6.4.6.3. Análisis y aplicabilidad

Como hemos comentado, una combinación replicada puede ser el tipo de combinación más rápida en ejecutarse, al no requerir de la fase barajar y ordenar (*Shuffle and Sort*), ni la fase reductora (*Reducer*); podemos afirmar que es extremadamente útil de cara a realizar uniones de conjuntos de datos que vengan por distintos caminos. Sin embargo, también tiene sus restricciones, y son que solo podremos realizar la unión interna y la unión por la izquierda, y los límites propios del tamaño de la memoria al cargar un archivo o tabla.

6.5. Resumen del capítulo

A lo largo de este capítulo nos hemos adentrado en el conocimiento y desarrollo dentro del paradigma MapReduce de cuatro tipos de patrones: resumen, filtros, organización de datos y uniones.

Los patrones de resumen son patrones que, mediante algún cálculo o búsqueda, nos permiten extraer información resumida de un amplio conjunto de datos, mediante la realización de algún cálculo o búsqueda de elementos.

Los patrones de filtros nos permiten extraer subconjuntos de datos que cumplan una o varias condiciones.

Con los patrones de organización de datos se han desarrollado prototipos que nos permiten variar la organización de los datos de entrada, permitiéndonos hacer un tratamiento durante el procesamiento en MapReduce, para mostrar en la salida, los datos reorganizados e incluso desordenados.

En los patrones de unión, hemos podido comprobar como fuera del lenguaje estructurado de consultas (SQL), podemos realizar en el paradigma MapReduce operaciones similares de unión de conjuntos.

Dentro de cada uno de estos patrones, hemos implementado diversos prototipos con el lenguaje Python, con ejemplos y haciendo uso de los conjuntos de datos descritos en el capítulo 5. Estos desarrollos se pueden consultar en el repositorio GitHub [42] o en el *Anexo* de esta memoria.

Capítulo 7. Planificación y presupuesto

A lo largo de este capítulo vamos a exponer la planificación de este proyecto, y el establecimiento del presupuesto que conlleva llevarlo a cabo.

7.1. Planificación

El método de desarrollo va a consistir en la lectura, estudio, realización de pruebas empíricas de los despliegues de arquitecturas, y desarrollos de programas o prototipos. Se establecen distintas fases, desglosando en cada una de ellas el tiempo estimado y las tareas a realizar.

- ➔ *Fase 1.* Estudio de tecnologías de virtualización ligera con Docker.
 - Tiempo estimado: 25 horas
 - Tareas: lectura de bibliografía, blogs, artículos, formación en plataformas educativas, y realización de prácticas.
- ➔ *Fase 2.* Estudio de Hadoop y su ecosistema. Despliegue de un *cluster* sobre los contenedores Docker.
 - Tiempo estimado: 25 horas
 - Tareas: lectura de blogs, bibliografía, artículos, formación en plataformas educativas, y realización de prácticas.
- ➔ *Fase 3.* Selección, estudio y desarrollo de patrones.
 - Tiempo estimado: 250 horas
 - Tareas: seleccionar de entre los distintos grupos o categorías de patrones *MapReduce*, los más significativos o relevantes; estudiarlos, analizarlos, desarrollarlos y probar su funcionamiento en Python. Ejecutarlos en entorno local y en el *cluster* Hadoop.
- ➔ *Fase 4.* Elaboración de la memoria y organización de cuadernos Jupyter.
 - Tiempo estimado: 150 horas
 - Tareas: Documentar todo el proyecto mediante redacción de la memoria, a la vez que se van organizando todos los patrones en los cuadernos Jupyter. Trasladar todo a GitHub y elaborar documento de presentación o defensa de proyecto.

En la planificación se estimó un total de 450 horas de trabajo efectivo que corresponden a 18 créditos.

En los ocho meses que se le ha dedicado al Proyecto Fin de Grado, el montante de horas contabilizado ha sido de **592 horas**, 142 horas más por encima de lo previsto.

7.2. Presupuesto

Como cualquier proyecto que se precie, este siempre llevará unos costes asociados que establecerá su valor y el esfuerzo económico que pueda suponer llevarlo a cabo.

7.2.1. Recursos empleados

Los recursos a los que nos vamos a referir en este apartado corresponden a los humanos, materiales e inmateriales.

➔ *Recursos humanos*

- 1 Ingeniero junior.

➔ *Recursos materiales*

- 1 ordenador Toshiba Satellite L50B (i5 8GB, 250GB SSD)
- 2 monitores planos 17" conectados al portátil.
- Router.

➔ *Recursos inmateriales*

licencias de programas

- Windows 10
- Office 365
- Virtual Box
- Ubuntu 16.04
- Hadoop
- Docker
- Anaconda
- Jupyter Notebook

Conexión a Internet

- Línea cableada doméstica por RJ45 y conexión por fibra óptica.
- Conexión Wi-fi.

Cursos de formación

- Plataforma Formación Udemy – Curso Hadoop Big Data desde 0.
- Plataforma Formación Udemy – Aprende Docker desde cero a Swarm y Kubernetes.

7.2.2. Coste de los recursos

→ *Recursos humanos.* El coste hora de un ingeniero junior se puede establecer en unos 15€ brutos la hora.

Fases	Horas	Coste/Hora	Total
1	60	15€	900€
2	60	15€	900€
3	300	15€	4.500€
4	172	15€	2.580€
Total			8.880€

Tabla 74 - Coste en recursos humanos

→ *Recursos materiales.*

Dispositivo	Valor	Coefficiente de amortización [43]	Meses	Coste imputable
Ordenador Toshiba Satellite L50B	650€	26%	8	60,125€
Monitor 17" Samsung	84€	26%	8	7,77€
Monitor LG 17"	151,57€	26%	8	14,02€
Router	0€	26%	8	0€
Total				81,92€

Tabla 75 - Costes en recursos materiales.

→ *Recursos inmateriales*

- El coste de la licencia de Windows 10 va incluida en el computador.
- El coste de la licencia de Office 365, es asumido por la UNED.
- El resto de licencias es software libre, por lo que no lleva ningún coste asociado.
- La conexión a Internet se ha hecho una estimación de coste por hora, y se ha calculado en base a las 592 horas de dedicación al proyecto.

Concepto	Coste
Licencias Software	0 €
Conexión a Internet	59,02 €
Formación	25 €
Total	84,02 €

Tabla 76 - Costes en recursos inmateriales.

El coste total del proyecto ejecutado en 592 horas a lo largo de 8 meses se fija en 9.046,12 €. En la Tabla 77 se aprecia el resumen y coste total.

Concepto	Coste
Recursos Humanos	8.880€
Recursos materiales	81,92€
Recursos inmateriales	84,02
Total	9.046,12€

Tabla 77 - Coste total del proyecto

7.3. Resumen de capítulo.

Durante este capítulo hemos realizado un estudio de planificación y costes estimados, correspondientes a este proyecto. Se ha intentado plasmar lo más cerca de la realidad, en cuanto tiempo y forma se ha desarrollado, y cuanto dinero nos ha costado. Esto nos tiene que dejar claro que todo proyecto conlleva un proceso de planificación y unos costes asociados, que le dará valor al trabajo que se desarrolle y que será fundamental de cara a hacer rentable nuestro tiempo. Lo que se conoce como coste oportunidad.

Capítulo 8. Aportaciones, trabajo futuro y conclusiones

Aportaciones

Como aportación a la comunidad y complemento de esta memoria, todo el código desarrollado se encuentra en un repositorio en GitHub:

https://github.com/manursanchez/TFG_Manuel_R

En este, se encuentran clasificados por tipos de patrones, los conjuntos de datos (*Datasets*), los cuadernos *Jupyter* con los prototipos desarrollados en Python, y los resultados de la ejecución de éstos tanto en local, como en el *cluster Hadoop*.

Trabajo futuro

Este proyecto nos da la posibilidad de abrir varias líneas de trabajo futuras, como pueden ser:

- ➔ Adaptación de los patrones a otros conjuntos de datos (datasets) para desarrollar, observar y aprender nuevos ejemplos de uso.
- ➔ Creación de nuevos patrones. Tenemos la posibilidad de implementar algún tipo más de patrón, dentro de cada uno de los tipos definidos en este trabajo.
- ➔ Combinar patrones de distinto tipo para crear metapatrones [7], y desarrollar ejemplos de uso con conjuntos de datos adaptados.
- ➔ Implementar patrones de diseño en otras herramientas como puede ser Apache Spark [44], que es un entorno de procesamiento distribuido y paralelo que trabaja en memoria, siendo un poco más rápido que *MapReduce*.

Conclusiones

El desarrollo de todo este trabajo ha supuesto andar un camino en el que ha habido que detenerse en diversas áreas de conocimiento: sistemas operativos, sistemas distribuidos, sistemas de bases de datos, lenguajes de programación, ingeniería del software, redes, estadística, gestión de proyectos informáticos y alguna otra cosa del área de la inteligencia artificial. Ha habido que leer, estudiar, repasar y extraer información necesaria para poder cumplir con los objetivos marcados.

Se hubo de empezar por la parte de sistemas, practicando con distribuciones Linux (Ubuntu y CentOS) para la instalación y puesta en funcionamiento de los sistemas operativos que contendrían sistemas distribuidos de contenedores Docker y arquitectura Hadoop. Era necesario estudiar y repasar de una forma empírica estos elementos, para entender los despliegues con distribuciones Hadoop (Cloudera) y su funcionamiento. Si no entiendes la base, será más difícil entender lo que hay sobre ella.

El siguiente paso, fue el cambio de pensamiento para adaptarse a un nuevo paradigma de programación como es MapReduce, un reto importante por el aprendizaje de Python y su adaptación a este paradigma, y enriquecedor en el sentido de adaptar un lenguaje a otro idioma.

Otro de los puntos importantes, fue el reto que supuso entender que era un patrón en su concepto más abstracto. Fue determinante en la consecución de este trabajo; entendiendo un patrón como una idea o una base sobre la que construir diversas soluciones a problemas que puedan darse en distintos ámbitos. Una idea, que dependiendo del contexto, puede derivar en un desarrollo u otro.

En definitiva, este trabajo ha estado lleno de tareas enriquecedoras, al tocar las distintas disciplinas que durante los años de estudio se han visto, a la vez que ha sido sumamente interesante en cuanto al aprendizaje de nuevos contenidos y tecnologías.

Como se comentó en las motivaciones de este proyecto, existía poca documentación con respecto a las librerías MRJob y su uso, y este trabajo pretende suplir en cierta medida esa falta, y lo más reconfortante sería que fuera de utilidad a próximos estudiantes, o profesionales que decidan poner en producción estas tecnologías.

Cuando se planteó este proyecto, se estableció el objetivo de desarrollar programas MapReduce usando patrones de diseño sobre el lenguaje Python, y esto implicaba una serie de objetivos específicos consistentes en un despliegue de infraestructuras distribuidas sobre contenedores ligeros, estudio de patrones de diseño con MapReduce, y la implementación de éstos en Python.

Pues bien, a lo largo de este trabajo hemos aprendido en que consiste la virtualización ligera y hemos conocido la herramienta Docker. Hemos hecho un despliegue de una infraestructura distribuida con Hadoop que, entre otros, nos ha permitido conocer el entorno MapReduce. En este paradigma, hemos estudiado y analizado patrones de diseño, clasificándolos en cuatro tipos, para posteriormente desarrollarlos en Python. Por tanto, y como conclusión final, podemos afirmar que hemos cubierto los objetivos marcados para este proyecto.

Bibliografía

- [1] Santos, Paloma Recuero de los, «Internet de las cosas y Big Data,» 19 junio 2019. [En línea]. Available: <https://empresas.blogthinkbig.com/iot-y-big-data/>. [Último acceso: 15 11 2020].
- [2] Dígitum IoT, [En línea]. Available: <https://www.digitum.com/internet-of-things-retail-industry#>. [Último acceso: 07 12 2020].
- [3] Synergic Partners, «Curso: Big Data: el impacto de los datos masivos en la sociedad actual,» [En línea]. Available: <https://www.coursera.org/learn/impacto-datos-masivos/home/welcome>. [Último acceso: 13 12 2020].
- [4] Caminero Herráez, Agustín C.; Grau Fernández, Luis, Introducción al manejo de datos masivos con Hadoop y herramientas inspiradas en SQL, Práctica asignatura Sistemas de Bases de Datos del Grado en Ingeniería Informática de la UNED, Curso 2020/2021.
- [5] Santos, Paloma Recuero de los, «Tus datos más limpios casi sin frotar,» 13 junio 2017. [En línea]. Available: <https://empresas.blogthinkbig.com/tus-datos-mas-limpios-casi-sin-frotar/>.
- [6] Apasoft Training, «Curso Udemy: Hadoop Big Data desde cero,» [En línea]. Available: www.udemy.com. [Último acceso: 13 12 2020].
- [7] A. Shook y D. Miner, MapReduce Design Patterns, O'Reilly Media, Inc., 2012.
- [8] Docker, [En línea]. Available: <https://www.docker.com/>. [Último acceso: 07 12 2020].
- [9] Cloudera, [En línea]. Available: <https://es.cloudera.com/>. [Último acceso: 07 12 2020].
- [10] Cambridge Dictionary, «Significado CLUSTER,» [En línea]. Available: <https://dictionary.cambridge.org/es/diccionario/ingles-espanol/cluster>. [Último acceso: 07 12 2020].
- [11] Apache Hadoop - HDFS Architecture, [En línea]. Available: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>. [Último acceso: 07 12 2020].
- [12] T. White, Hadoop: The definitive guide. 4th Edition, O'Reilly Media, Inc., 2015, p. Chapter I Hadoop fundamentals. Chapter II MapReduce. Chapter IV YARN.
- [13] Apache Software Foundation, «Apache Hadoop,» [En línea]. Available: <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html>. [Último acceso: 08 12 2020].
- [14] Z. Radtka y D. Miner, Hadoop with Python, O'Reilly Media, Inc., 2015, p. Chapter 2: MapReduce with Python.

- [15] Santos, Paloma Recuero de los, «Hadoop por dentro (II): HDFS y MapReduce,» [En línea]. Available: <https://empresas.blogthinkbig.com/hadoop-por-dentro-ii-hdfs-y-mapreduce/>. [Último acceso: 30 11 2020].
- [16] Apache Hive, [En línea]. Available: <https://hive.apache.org/>. [Último acceso: 9 12 2020].
- [17] Apache HBASE, [En línea]. Available: <https://hbase.apache.org/>. [Último acceso: 9 12 2020].
- [18] Apache PIG, [En línea]. Available: <https://pig.apache.org/>. [Último acceso: 9 12 2020].
- [19] Apache Sqoop, [En línea]. Available: <https://sqoop.apache.org/>. [Último acceso: 9 12 2020].
- [20] Apache FLUME, [En línea]. Available: <https://flume.apache.org/>. [Último acceso: 9 12 2020].
- [21] Anaconda Inc., [En línea]. Available: <https://www.anaconda.com/>. [Último acceso: 07 12 2020].
- [22] Python Software Foundation, [En línea]. Available: <https://www.python.org/>. [Último acceso: 15 11 2020].
- [23] Pontificia Universidad Católica de Chile, «Curso Introducción a la programación en Python,» [En línea]. Available: <https://www.coursera.org/learn/aprendiendo-programar-python/home/welcome>. [Último acceso: 15 9 2020].
- [24] Project Jupyter, [En línea]. Available: <https://jupyter.org/>. [Último acceso: 07 12 2020].
- [25] Caminero Herráez, Agustín Carlos, «Empieza a trabajar con tecnologías Big Data con Hadoop y Docker,» Septiembre 2019. [En línea]. Available: <https://informaticaunedblog.wordpress.com/2019/11/11/empieza-a-trabajar-con-tecnologias-big-data-con-hadoop-y-docker/>. [Último acceso: 11 11 2020].
- [26] MRJOB, [En línea]. Available: <https://mrjob.readthedocs.io/en/latest/>. [Último acceso: 13 12 2020].
- [27] S. Johnson, mrjob Documentation. Release 0.7.4, Junio 2020.
- [28] GIT, [En línea]. Available: <https://git-scm.com/>. [Último acceso: 28 11 2020].
- [29] GitHub, [En línea]. Available: <https://github.com/>. [Último acceso: 07 12 2020].
- [30] Microsoft Azure, [En línea]. Available: <https://azure.microsoft.com/es-es/overview/what-is-virtualization/>. [Último acceso: 30 11 2020].
- [31] Apasoft Training, «Curso Udemy: Aprende Docker desde cero,» [En línea]. Available: www.udemy.com. [Último acceso: 11 12 2020].
- [32] Microsoft, «Uso de Docker Compose,» [En línea]. Available: <https://docs.microsoft.com/es-es/visualstudio/docker/tutorials/use-docker-compose>. [Último acceso: 11 2020].

- [33] Moodle, [En línea]. Available: <https://moodle.org/>. [Último acceso: 29 11 2020].
- [34] UCI Machine Learning Repository, [En línea]. Available: <https://archive.ics.uci.edu/ml/datasets/Online%20Retail#>. [Último acceso: 9 12 2020].
- [35] Wikipedia - Índice invertido, [En línea]. Available: https://es.qaz.wiki/wiki/Inverted_index. [Último acceso: 07 12 2020].
- [36] «TodoXML,» [En línea]. Available: <https://sites.google.com/site/todoxmltd/home>. [Último acceso: 11 2020].
- [37] <https://heptapod.host/saajns/xmlify>. [En línea]. Available: <https://pypi.org/project/Xmlify/>. [Último acceso: 30 11 2020].
- [38] R. Moya, «<https://jarroba.com/ordenar-listas-python-ejemplos/>,» [En línea]. [Último acceso: 28 11 2020].
- [39] MapReduce Tutorial, [En línea]. Available: http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Configured+Parameters. [Último acceso: 1 11 2020].
- [40] A. Silberschatz, H. F. Korth, S. Sudarshan, Fundamentos de base de datos. 5ª Ed., McGraw-Hill, 2007, p. 44 a 53; 90 a 93.
- [41] <https://github.com/Yelp/mrjob>. [En línea]. Available: <https://github.com/manursanchez/mrjob/tree/master/mrjob/examples>. [Último acceso: 06 12 2020].
- [42] M. R. Sánchez, «TFG_ Manuel_R,» [En línea]. Available: https://github.com/manursanchez/TFG_Manuel_R.
- [43] Agencia Tributaria, «Tabla de amortización simplificada,» [En línea]. Available: www.agenciatributaria.es. [Último acceso: 12 12 2020].
- [44] Apache Spark, [En línea]. Available: <https://spark.apache.org/>. [Último acceso: 13 12 2020].
- [45] «Librería MRJob,» [En línea]. Available: <https://pypi.org/project/mrjob/>. [Último acceso: 07 12 2020].
- [46] M. Manoochehri, Data Just Right: Introduction to large-scale data & analytics, Addison-Wesley Professional, 2015, pp. Part IV: Data Pipelines and Real Time Data. 8-Putting it together: MapReduce Data Pipelines.
- [47] University of Michigan, «Curso Python for everybody,» [En línea]. Available: <https://www.coursera.org/learn/python/home/welcome>. [Último acceso: 1 11 2020].

Anexo – Patrones completos

```

%%writefile average.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re
class average(MRJob):

    def mapper(self,_, line):
        #División del registro para tener la lista de campos
        linea=line.split(";")
        #El valor del campo [5] tiene que ser un número.
        encontrado=re.search('^[0-9]',linea[5])

        if encontrado!=None:
            #Extraemos el valor de la lista con el que operaremos
            valor=linea[5]
            #Extraemos la clave asociada al valor
            clave=linea[6]
            #Cedemos la clave y el valor al Reducer
            yield clave,float(valor)

    def reducer(self, key, values):
        sumaValues=0 # Para guardar la suma de los valores
        cuentaValues=0 # Para guardar el número de valores
        # Iteramos a través del conjunto de valores que han
        #llegado desde el mapper
        for value in values:
            sumaValues+=value #Sumamos valor
            cuentaValues+=1 #Contamos valor
        #Dividimos
        mediaValues=sumaValues/cuentaValues
        #Entregamos la media de ventas
        #por país
        yield key, mediaValues

if __name__ == '__main__':
    average.run()

```

Patrón 1 - Resúmenes numéricos. Media aritmética, versión uno.

```

%%writefile averageV2.py
#!/usr/bin/env python
#Segunda versión del patrón de resumen numérico
from mrjob.job import MRJob
import statistics
import re
class averageV2(MRJob):

    def mapper(self, _, line):

        linea=line.split(";")
        encontrado=re.search('^[0-9]',linea[5])
        if encontrado!=None:
            valor=linea[5] #UnitPrice
            clave=linea[7] #Country
            yield clave,float(valor)

    def reducer(self, key, values):
        listaValores=[]
        for valor in values:
            listaValores.append(valor)
        yield key, statistics.mean(listaValores)

if __name__ == '__main__':
    averageV2.run()

```

Patrón 2 - Resúmenes numéricos. Media aritmética, versión dos.

```

%%writefile desviacion.py
#!/usr/bin/env python

from mrjob.job import MRJob
import statistics as st
import re
class desviacion(MRJob):

    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search('^[0-9]',linea[5])
        if encontrado!=None:
            valor=linea[5] #UnitPrice
            clave=linea[7] #Country
            yield clave,float(valor)

    def reducer(self, key, values):
        valores=[]
        for value in values:
            valores.append(value)
        desviacionEstandar=st.pstdev(valores)
        yield key, desviacionEstandar

if __name__ == '__main__':
    desviacion.run()

```

Patrón 3 - Resúmenes numéricos. Desviación típica.

```

%%writefile mediana.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re

class mediana(MRJob):

    def mapper(self, _, line):

        linea=line.split(";")
        encontrado=re.search('^[0-9]',linea[5])
        if encontrado!=None:
            valor=linea[5] # UnitPrice
            clave=linea[7] # Country
            yield clave,float(valor)

    def reducer(self, key, values):
        valores=[]
        sumaValores=0
        contador=0
        #Recorremos los valores
        for value in values:
            valores.append(value)
            sumaValores+=value
            contador=contador+1
        #Ordenamos los valores
        valores.sort()

        #Comprobamos si el total de
        #los valores es par o impar

        if contador%2==0:
            #Para el número de Claves(key) pares

            #Calculamos los índices
            #donde se encuentran los valores
            #centrales
            indice_1=int(contador//2)
            indice_2=int(indice_1-1)

            #Extraemos los valores
            valor_1=valores[indice_1]
            valor_2=valores[indice_2]
            mediana=(valor_1+valor_2)/2

        else:
            #Para el número de claves(key) impares
            indice=(contador-1)//2
            mediana=valores[indice]

        yield key, mediana

if __name__ == '__main__':
    mediana.run()

```

Patrón 4 - Resúmenes numéricos. Mediana versión uno.

```

%%writefile medianaV2.py
#!/usr/bin/env python

from mrjob.job import MRJob
import statistics as st
import re

class medianaV2(MRJob):

    def mapper(self, _, line):

        linea=line.split(";")
        encontrado=re.search('^[0-9]',linea[5])
        if encontrado!=None:
            valor=linea[5] #UnitPrice
            clave=linea[7] #Country
            yield clave,float(valor)

    def reducer(self, key, values):
        valores=list(values)
        yield key, st.median(valores) #Mediana

if __name__ == '__main__':
    medianaV2.run()

```

Patrón 5 - Resúmenes numéricos. Mediana versión dos.

```

%%writefile minmaxcount.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re
class minmaxcount(MRJob):

    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search('^[0-9]',linea[5])
        if encontrado!=None:
            valor=linea[5]
            clave=linea[6]
            yield clave,float(valor)

    def reducer(self, key, values):
        valores=[] #Lista donde almacenaremos los valores
        #Recorremos la lista de "values"
        #para agregarlos a la lista
        for value in values:
            valores.append(value)
        valores.sort()#Ordenamos los valores de la lista
        vMax=valores[len(valores)-1] #Valor máximo
        vMin=valores[0] #Valor mínimo
        cuenta=len(valores) #Número de valores del grupo
        yield key, (cuenta,vMax,vMin)

if __name__ == '__main__':
    minmaxcount.run()

```

Patrón 6 - Resúmenes numéricos. Cálculo del valor mínimo, máximo y contar. Versión uno.

```

%%writefile minmaxcountV2.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re

class minmaxcountV2(MRJob):

    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search('^[0-9]',linea[5])
        if encontrado!=None:
            valor=linea[5]
            clave=linea[6]
            yield clave,float(valor)

    def reducer(self, key, values):
        valores=list(values)
        valores.sort()
        vMax=valores[len(valores)-1]
        vMin=valores[0]
        cuenta=len(valores)
        yield key, (cuenta,vMax,vMin)

if __name__ == '__main__':
    minmaxcountV2.run()

```

Patrón 7 - Resúmenes numéricos. Cálculo del mínimo, máximo y contar. Versión dos.

```

%%writefile counter.py
#!/usr/bin/env python
from mrjob.job import MRJob

class counter(MRJob):
    def mapper_init(self):
        #Creamos un diccionario con los que serán los contadores a 0
        self.contadores={"Netherlands":0,"France":0,"Australia":0}

    def mapper(self, _, line):
        linea=line.split(";")
        # Estudiamos cada token de la línea recogido
        for token in linea:
            # Si el token está en el diccionario
            if token in self.contadores:
                # Contamos con el contador definido en diccionario
                self.contadores[token]=self.contadores[token]+1

    def mapper_final(self):
        yield "Bloque: ",self.contadores

if __name__ == '__main__':
    counter.run()

```

Patrón 8 - Resúmenes numéricos. Contando con contadores. Versión uno.

```

%%writefile counterV2.py
#!/usr/bin/env python
from mrjob.job import MRJob

class counterV2(MRJob):

    def mapper_init(self):
        #Creamos un diccionario con los que serán los contadores a 0
        self.contadores={"Netherlands":0,"France":0,"Australia":0}

    def mapper(self, _, line):
        linea=line.split(";")
        # Estudiamos cada token de la línea recogido
        for token in linea:
            # Si el token está en el diccionario
            if token in self.contadores:
                # Contamos con el contador definido en diccionario
                self.contadores[token]=self.contadores[token]+1

    def mapper_final(self):
        for clave, valor in self.contadores.items():
            yield clave,valor

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    counterV2.run()

```

Patrón 9 - Resúmenes numéricos. Contando con contadores. Versión dos con el Reducer.

```

%%writefile counterV3.py
#!/usr/bin/env python
from mrjob.job import MRJob

contadores={"Netherlands":0,"France":0,"Australia":0}
lista=[]
class counterV3(MRJob):

    def mapper(self, _, line):
        linea=line.split(";")
        # Estudiamos cada token de la línea recogido#
        for token in linea:
            # Si el token está en el diccionario
            if token in contadores:
                # Contamos con el contador definido en diccionario
                contadores[token]=contadores[token]+1

    def mapper_final(self):
        yield "Bloque: ",contadores

if __name__ == '__main__':
    counterV3.run()

```

Patrón 10 - Resúmenes numéricos. Contando con contadores. Versión tres. Declaración fuera de la clase.

```

%%writefile indexinverted.py
#!/usr/bin/env python
from mrjob.job import MRJob

class indexinverted(MRJob):

    def mapper(self, _, line):

        linea=line.split(' ')

        nLinea=linea[0]

        palabras=linea[1:]
        for palabra in palabras:
            yield palabra.lower(),nLinea

    def reducer(self, key, values):
        listaValores=[]
        listaDef=[]

        for valor in values:
            listaValores.append(valor)

        for posicion in range(len(listaValores)):
            subLista=[]
            elementoLista=listaValores[posicion]

            cuentaElementos=listaValores.count(elementoLista)

            subLista=[listaValores[posicion],cuentaElementos]
            if subLista not in listaDef:
                listaDef.append(subLista)

        yield key, listaDef

if __name__ == '__main__':
    indexinverted.run()

```

Patrón 11 - Resúmenes numéricos. Índice invertido.

```

%%writefile subconjunto.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re

class subconjunto(MRJob):

    def mapper(self,_, line):

        #linea=line.rstrip("\n").split(";")
        linea=line.split(";")

        #Con una expresión regular comprobamos si lo que entra es una
        #cadena de caracteres o un número
        #Si es un número, nos quedamos con el registro, si no, lo
        #desechamos.
        encontrado=re.search('^[0-9]',linea[0]) #La columna 0 tiene que
        #ser un número.

        #Aplicamos una condición y lanzamos con yield los que la cumplan

        if encontrado!=None and len(linea)>1:
            yield linea[0],linea

if __name__ == '__main__':
    subconjunto.run()

```

Patrón 12 - Filtrado. Extracción de subconjuntos

```

%%writefile hiloeventos.py
#!/usr/bin/env python
from mrjob.job import MRJob

class hiloeventos(MRJob):

    def mapper(self,_, line):
        linea=line.rstrip("\n").split(";")
        if linea[1]=="Alumno5" and linea[3]=="Formulario de entrega
visto.":
            yield (linea[1],linea[3]),1

if __name__ == '__main__':
    hiloeventos.run()

```

Patrón 13 - Filtrado. Seguimiento de hilo de eventos.


```

%%writefile topN.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re

TopN=10 #Los N valores más altos que vamos a extraer

class topN(MRJob):

    def extraeTopN(self, listaValores):
        valoresMasAltos = []
        cuenta=0
        if TopN>len(listaValores):
            nValores=len(listaValores)
        else:
            nValores=TopN

        posicion=len(listaValores)-1

        while cuenta<nValores:
            try:
                if listaValores[posicion] not in valoresMasAltos:
                    valoresMasAltos.append(listaValores[posicion])
                    cuenta+=1

                posicion-=1
            except:
                break
        return valoresMasAltos

    def mapper_init(self):
        self.topNmap = []

    def mapper(self, _, line):
        self.registro=line.rstrip('/n').split(';')
        encontrado=re.search('[a-zA-Z]',self.registro[5])
        if encontrado==None:
            self.topNmap.append(float(self.registro[5]))

    def mapper_final(self):
        self.topNmap.sort()
        yield "TopNmap:", self.extraeTopN(self.topNmap)

    def reducer(self, key, values):
        listaValores=[]

        for valores in values:
            listaValores.extend(valores)

        listaValores.sort()
        yield "TopNreducer:", self.extraeTopN(listaValores)

if __name__ == '__main__':
    topN.run()

```

Patrón 14 - Filtrado. Extraer los N valores más altos.

```

%%writefile valoresNoRepetidos.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re

#Expresion regular para validar ip
ip = ('^(?:25[0-5]|2[0-4][0-9]|'
      '[01]?[0-9][0-9]?)\.){3}'
      '(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$')
IP_RE = re.compile(ip)

class valoresNoRepetidos(MRJob):

    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search(IP_RE,linea[5])
        if encontrado!=None:
            yield _,linea

    def reducer(self, key, values):
        listaDef=[] #Lista para los registros únicos filtrados
        listaControl=[] #Lista para el control de los registros que son
repetidos

        # metemos los valores en las listas con lo que nos viene del
mapper.
        for valor in values:
            #Si el valor de control (en este caso la IP) no está en la
lista de control
            #agregamos el valor de control en la lista de control, y el
registro completo en la
            #lista definitiva.
            if valor[5] not in listaControl:
                listaControl.append(valor[5])
                listaDef.append(valor)

        #Podemos sacar por el reducer lo que estimemos
        for p in listaDef:
            yield p[1], (p[1],p[5])

if __name__ == '__main__':
    valoresNoRepetidos.run()

```

Patrón 15 - Filtrado. Valores no repetidos.

```

%%writefile jerarquico.py
#!/usr/bin/env python
#Usamos el archivo foros.csv

from mrjob.job import MRJob
from mrjob.protocol import RawValueProtocol
import xmlify

class jerarquico(MRJob):

    OUTPUT_PROTOCOL = RawValueProtocol

    def mapper(self, _, line):
        linea=line.split(";")

        mensaje=linea[4] # Recogemos el mensaje de la posición 4 de la
        línea
        tipoMensaje=linea[5] #Recogemos de la posición 5, si es una
        pregunta o respuesta

        if tipoMensaje=="question":
            idMensaje=linea[0] #Almacenamos el id único del mensaje
            yield idMensaje,(tipoMensaje,mensaje)
        else:
            idMensaje=linea[7] #Almacenamos el identificador del mensaje
            idMensaje
            yield idMensaje,(tipoMensaje,mensaje)

    def reducer(self, key, values):
        listaValores=[]
        listaPrincipal=[]
        listaAuxiliar=[]

        for v in values: #Metemos los valores que vienen en un matriz
            listaValores.append(v) #Matriz que contiene el tipo de
            mensaje y el mensaje asociado

        for valor in listaValores:
            if valor[0]=="question":#Si es una pregunta la metemos en la
            lista principal
                listaPrincipal.append(valor[1])
            else:
                listaAuxiliar.append(valor[1]) # Si son respuestas, las
            vamos agregando a una lista

        listaPrincipal.append(listaAuxiliar) #agregamos la lista de
        respuestas a la lista principal

        #Conversion a XML indicando en el raiz el id del mensaje
        yield "Creada linea XML: " ,xmlify.dumps(listaPrincipal,root =
        key)

if __name__ == '__main__':
    jerarquico.run()

```

Patrón 16 - Organización de datos. Estructuración jerárquica. Versión uno.

```

%%writefile jerarquicoV2.py
#!/usr/bin/env python
#Usamos el archivo foros.csv

from mrjob.job import MRJob
from mrjob.protocol import RawValueProtocol
import xmlify

class jerarquicoV2(MRJob):
    OUTPUT_PROTOCOL = RawValueProtocol
    def mapper(self, _, line):
        linea=line.split(";") # Cada línea es un mensaje del foro
        (pregunta, respuesta o comentario)

        mensaje=linea[4] # Recogemos el mensaje de la posición 4 de la
        línea
        tipoMensaje=linea[5] #Recogemos de la posición 5, si es una
        pregunta, respuesta o comentario

        if tipoMensaje=="question":
            idMensaje=linea[0] #Almacenamos el id único del mensaje
            yield idMensaje,(tipoMensaje,mensaje)
        else:
            idPadre=linea[7] #Almacenamos el identificador del mensaje
            idMensaje
            yield idPadre,(tipoMensaje,mensaje)

    def reducer(self, key, values):
        diccionario=dict() #Para el caso que usemos diccionarios
        matrizParaXML=[]
        listaPrincipal=[]
        listaAuxiliar=[]

        for v in values: #Metemos los valores que vienen en un matriz
            matrizParaXML.append(v) #Matriz que contiene el tipo de
            mensaje y el mensaje asociado

            for valor in matrizParaXML:
                if valor[0]=="question":#Si es una pregunta la metemos en la
                lista principal
                    listaPrincipal.append(valor[1])
                else:
                    listaAuxiliar.append(valor[1]) # Si son respuestas, las
                    vamos agregando a una lista

            listaPrincipal.append(listaAuxiliar) #agregamos la lista de
            respuestas a la lista principal
            diccionario[key]=listaPrincipal #Para el caso que usemos
            diccionarios
            yield key,xmlify.dumps(diccionario) # Conversion a XML para el
            caso que usemos diccionarios

if __name__ == '__main__':
    jerarquicoV2.run()

```

Patrón 17 - Organización de datos. Estructuración jerárquica. Versión dos usando diccionarios.

```

%%writefile ordenTotal.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re
class ordenTotal(MRJob):

    def mapper_init(self):
        self.conjuntoRegistros = []

    def mapper(self, _, line):
        linea=line.rstrip('/n').split(';')

        encontrado=re.search('[a-zA-Z]',linea[1])

        if encontrado==None:
            #Se agrega el registro completo a la lista
            self.conjuntoRegistros.append(linea)

    def mapper_final(self):
        # mando el conjunto de registros al Reducer para reunirlos
        todos.
        yield None, self.conjuntoRegistros

    def reducer(self, _, values):
        listaCompletaRegistros=[]

        for valores in values:
            listaCompletaRegistros.extend(valores)
            #Expresión Lambda para indicar
            # por qué campo tiene que ordenar la lista.
            listaCompletaRegistros.sort(key = lambda precio:
float(precio[5]))

        for registro in listaCompletaRegistros:
            yield _, registro

if __name__ == '__main__':
    ordenTotal.run()

```

Patrón 18 - Organización de datos. Orden total.

```

%%writefile ordenTotalV2.py
#!/usr/bin/env python

from mrjob.job import MRJob
import sys, os, re

class ordenTotalV2(MRJob):

    #Inicializamos la estructura que nos permitirá contener los valores
    #que nos llegan del Map
    def mapper_init(self):
        self.conjuntoRegistros = []

    def mapper(self, _, line):
        linea=line.rstrip('/n').split(';')
        encontrado=re.search('[a-zA-Z]',linea[1])
        if encontrado==None:
            self.conjuntoRegistros.append(linea)

    def mapper_final(self):
        #Otra versión
        self.conjuntoRegistros.sort(key = lambda x: float(x[5]))
        for registro in self.conjuntoRegistros:
            yield None, registro

    def reducer(self, key, values):
        listaCompletaRegistros=[]
        listaCompletaRegistros.extend(values)

        # ordenamos la lista
        listaCompletaRegistros.sort(key = lambda x: float(x[5]))

        for registro in listaCompletaRegistros:
            yield None, registro

if __name__ == '__main__':
    ordenTotalV2.run()

```

Patrón 19 - Organización de datos. Orden total versión dos. La ordenación se hace en los Mapper.

```

%%writefile ordenTotalV3.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re

class ordenTotalV3(MRJob):

    #Inicializamos la estructura que nos permitirá contener los valores
    #que nos llegan del Map
    def mapper_init(self):
        self.conjuntoRegistros = []

    def mapper(self, _, line):
        self.linea=line.rstrip('/n').split(';')

        encontrado=re.search('[a-zA-Z]',self.linea[1])

        if encontrado==None:
            #Se agrega el registro completo a la lista
            self.conjuntoRegistros.append(self.linea)

    def mapper_final(self):
        yield self.linea[6], self.conjuntoRegistros

    def reducer(self, cliente, values):
        listaCompletaRegistros=[]

        for valores in values:
            listaCompletaRegistros.extend(valores)

        # ordenamos la lista
        listaCompletaRegistros.sort(key = lambda x: float(x[5]))

        for registro in listaCompletaRegistros:
            yield cliente, registro

if __name__ == '__main__':
    ordenTotalV3.run()

```

Patrón 20 - Organización de datos. Orden total versión tres. Se ordenan registros usando la clave de cliente.

```

%%writefile mezclar.py
#!/usr/bin/env python

from mrjob.job import MRJob
import random, re

class mezclar(MRJob):
    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search('[a-zA-Z]',linea[3])
        if encontrado==None:
            yield _,linea

    def reducer(self, _, values):
        listaRegistros=[]
        for registro in values:
            listaRegistros.append(registro)
        random.shuffle(listaRegistros,random.random)
        for record in listaRegistros:
            yield _,record

if __name__ == '__main__':
    mezclar.run()

```

Patrón 21 - Organización de datos. Mezclar registros.

```

%%writefile anonimizar_y_mezclar.py
#!/usr/bin/env python
from mrjob.job import MRJob
from datetime import datetime
import random, re
class anonimizar_y_mezclar(MRJob):

    def mapper_init(self):
        self.registros = []
    def mapper(self, _, line):
        linea=line.split(";")
        encontrado=re.search('[a-zA-Z]',linea[3])
        if encontrado==None:
            fecha = datetime.strptime(linea[4], '%d/%m/%Y %H:%M')
            del(linea[6]) # Eliminamos el código del cliente
            linea[4]=fecha.year # Asignamos el año al campo fecha
            self.registros.append(linea)
    def mapper_final(self):
        for registro in self.registros:
            yield None, registro

    def reducer(self, _, values):
        listaRegistros=[]
        for registro in values:
            listaRegistros.append(registro)
        random.shuffle(listaRegistros,random.random)
        for record in listaRegistros:
            yield _,record
if __name__ == '__main__':
    anonimizar_y_mezclar.run()

```

Patrón 22 - Organización de datos. Anonimizar y mezclar.


```

%%writefile unionInterna.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re,os

class unionInterna(MRJob):
    def limpiarNombreArchivo(self,archivo):
        encontradaBarra=False
        tamano=len(archivo)
        posicion=tamano-1
        while encontradaBarra==False or posicion==0:
            if archivo[posicion]=="\/":
                encontradaBarra=True
                return archivo[posicion+1:tamano]
            else:
                posicion-=1
        if posicion==0:
            return archivo

    def mapper_init(self):

        self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_file'
        ))

    def mapper(self,_,line):
        clave=""
        linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            if self.namefile=="tiendas.csv":
                linea.append(self.namefile)
                clave=linea[0]
                yield clave,linea
            else:
                linea.append('art.csv')
                clave=linea[1]
                yield clave,linea

    def reducer(self,key,values):
        listaA=[]
        listaB=[]
        #Llenamos las dos listas
        for valor in values:
            if valor[len(valor)-1]=="tiendas.csv":
                listaA.append(valor)
            else:
                listaB.append(valor)
        ##### Union interna #####
        if listaA and listaB:
            for valor_A in listaA:
                for valor_B in listaB:
                    yield key,(valor_A, valor_B)

if __name__ == '__main__':
    unionInterna.run()

```

Patrón 23 - Unión interna.

```

%%writefile unionIzquierda.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re,sys,os

class unionIzquierda(MRJob):

    def limpiarNombreArchivo(self,archivo):
        encontradaBarra=False
        tamaño=len(archivo)
        posición=tamaño-1
        while encontradaBarra==False or posición==0:
            if archivo[posición]=="/":
                encontradaBarra=True
                return archivo[posición+1:tamaño]
            else:
                posición-=1
        if posición==0:
            return archivo

    def mapper_init(self):
self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_file'))

    def mapper(self,_,line):
        clave=""
        linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            if self.namefile=="tiendas.csv":
                linea.append(self.namefile)
                clave=linea[0] #Esta clave es la común de la tabla 1
                yield clave,linea
            else:
                linea.append('art.csv')
                clave=linea[1] #Clave común de la tabla 2
                yield clave,linea

    def reducer(self,key,values):
        listaA=[]
        listaB=[]
        #Llenamos las dos listas
        for valor in values:
            if valor[len(valor)-1]=="tiendas.csv":
                listaA.append(valor)
            else:
                listaB.append(valor)
        ##### Union por la izquierda #####
        for valorA in listaA:
            if listaB:
                for valorB in listaB:
                    yield valorA, valorB
            else:
                #Si la listaB está vacía
                yield valorA, "null"

if __name__ == '__main__':
    unionIzquierda.run()

```

Patrón 24 - Unión externa por la izquierda.

```

%%writefile unionDerecha.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re,sys,os

class unionDerecha(MRJob):

    def limpiarNombreArchivo(self,archivo):
        encontradaBarra=False
        tamaño=len(archivo)
        posición=tamaño-1
        while encontradaBarra==False or posición==0:
            if archivo[posición]=="/":
                encontradaBarra=True
                return archivo[posición+1:tamaño]
            else:
                posición-=1
        if posición==0:
            return archivo

    def mapper_init(self):
self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_file'))

    def mapper(self,_,line):
        clave=""
        linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            if self.namefile=="tiendas.csv":
                linea.append(self.namefile)
                clave=linea[0]
                yield clave,linea
            else:
                linea.append('art.csv')
                clave=linea[1]
                yield clave,linea

    def reducer(self,key,values):
        listaA=[]
        listaB=[]
        #Llenamos las dos listas
        for valor in values:
            if valor[len(valor)-1]=="tiendas.csv":
                listaA.append(valor)
            else:
                listaB.append(valor)
        ##### Union por la derecha #####
        for valorB in listaB:
            if listaA:
                for valorA in listaA:
                    yield valorA, valorB
            else:
                yield "null",valorB

if __name__ == '__main__':
    unionDerecha.run()

```

Patrón 25 - Unión externa por la derecha.

```

%%writefile unionCompleta.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re,os

class unionCompleta(MRJob):
    def limpiarNombreArchivo(self,archivo):
        encontradaBarra=False
        tamano=len(archivo)
        posicion=tamano-1
        while encontradaBarra==False or posicion==0:
            if archivo[posicion]=="\/":
                encontradaBarra=True
                return archivo[posicion+1:tamano]
            else:
                posicion-=1
        if posicion==0:
            return archivo

    def mapper_init(self):
self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_file'))

    def mapper(self,_,line):
        clave=""
        linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            if self.namefile=="tiendas.csv":
                linea.append(self.namefile)
                clave=linea[0]
                yield clave,linea
            else:
                linea.append('art.csv')
                clave=linea[1]
                yield clave,linea

    def reducer(self,key,values):
        listaA=[]
        listaB=[]
        for valor in values:
            if valor[len(valor)-1]=="tiendas.csv":
                listaA.append(valor)
            else:
                listaB.append(valor)
        if listaA:
            for valorA in listaA:
                if listaB:
                    for valorB in listaB:
                        yield valorA, valorB
                    else:
                        yield valorA, "null"
        else:
            for valorB in listaB:
                yield "null", valorB

if __name__ == '__main__':
    unionCompleta.run()

```

Patrón 26 - Unión completa.

```

%%writefile antiunion.py
#!/usr/bin/env python
from mrjob.job import MRJob
import re,os

class antiunion(MRJob):

    def limpiarNombreArchivo(self,archivo):
        encontradaBarra=False
        tamaño=len(archivo)
        posición=tamaño-1
        while encontradaBarra==False or posición==0:
            if archivo[posición]=="\/":
                encontradaBarra=True
                return archivo[posición+1:tamaño]
            else:
                posición-=1
        if posición==0:
            return archivo

    def mapper_init(self):

self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_file'))

    def mapper(self,_,line):
        clave=""
        linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            if self.namefile=="tiendas.csv":
                linea.append(self.namefile)
                clave=linea[0]
                yield clave,linea
            else:
                linea.append(self.namefile)
                clave=linea[1]
                yield clave,linea

    def reducer(self,key,values):
        listaA=[]
        listaB=[]
        #Llenamos las dos listas
        for valor in values:
            if valor[len(valor)-1]=="tiendas.csv":
                listaA.append(valor)
            else:
                listaB.append(valor)

        ##### Antiunion #####
        if not listaA or not listaB:
            for valorA in listaA:
                yield valorA, "null"
            for valorB in listaB:
                yield "null", valorB

if __name__ == '__main__':
    antiunion.run()

```

Patrón 27 - Antiunión.

```

%%writefile productoCartesiano.py
#!/usr/bin/env python

from mrjob.job import MRJob
import re,os

class productoCartesiano(MRJob):

    def limpiarNombreArchivo(self,archivo):
        encontradaBarra=False
        tamano=len(archivo)
        posicion=tamano-1
        while encontradaBarra==False or posicion==0:
            if archivo[posicion]=="\/":
                encontradaBarra=True
                return archivo[posicion+1:tamano]
            else:
                posicion-=1
        if posicion==0:
            return archivo

    def mapper_init(self):
self.namefile=self.limpiarNombreArchivo(os.getenv('map_input_file'))

    def mapper(self,_,line):
        linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            if self.namefile=="tiendas.csv":
                linea.append("file_1")#Añadimos identificador de archivo
                yield "llave",linea #Llave única
            else:
                linea.append("file_2")
                yield "llave",linea #Llave única

    def reducer(self,key,values):
        listaA=[]
        listaB=[]
        #Llenamos las dos listas
        for valor in values:
            if valor[len(valor)-1]=="file_1":
                listaA.append(valor)
            else:
                listaB.append(valor)

        ##### Producto Cartesiano #####
        if listaA and listaB:
            for valor_A in listaA:
                for valor_B in listaB:
                    yield key,(valor_A, valor_B)

if __name__ == '__main__':
    productoCartesiano.run()

```

Patrón 28 - Producto cartesiano.

```

%%writefile unionReplicada.py
#!/usr/bin/python
from mrjob.job import MRJob
import re

def cargarFicheroEnMemoria(self,fichero):
    diccionario={}
    with open(fichero) as f:
        self.tablaEnMemoria = set(line.strip() for line in f)
    for linea in self.tablaEnMemoria:
        #Para que no tenga en cuenta las cabeceras de las tablas
        encontrado=re.search('[a-zA-Z]',linea[0])
        if encontrado==None:
            datos = linea.split(";")
            diccionario[datos[0]]=datos
    return diccionario

class unionReplicada(MRJob):

    FILES = ['archivos_datos/tiendas-articulos/tiendas.csv']
    fichero='tiendas.csv'
    def mapper_init(self):
        #Nos devuelve la estructura diccionario rellena con los datos
del fichero
        self.dicTablaA=cargarFicheroEnMemoria(self,self.fichero)

    def mapper(self,_,line):
        self.linea=line.split(';')
        encontrado=re.search('[a-zA-Z]',self.linea[0])
        if encontrado==None:
            self.clave=self.linea[1] #Compara Clave de la tabla/s del
stream
            if self.clave in self.dicTablaA:
                yield
self.clave,(self.linea,self.dicTablaA.get(self.clave) )
            else:
                yield self.clave,(self.linea,"null")

if __name__ == '__main__':
    unionReplicada.run()

```

Patrón 29 - Unión replicada.