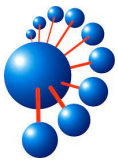
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE CONCEPCIÓN



ESTUDIO EMPÍRICO DEL USO DE DATOS CODIFICADOS PARA LA APLICACIÓN WORDCOUNT EN EL AMBIENTE DE PROCESAMIENTO DISTRIBUIDO HADOOP

POR
Leonardo Dimitri Aravena Cuevas

Memoria presentada para la obtención del título de
Ingeniero Civil Informático

Patrocinante: Dr. José Sebastián Fuentes Sepúlveda
Co-patrocinante: Dr. Diego Seco Naveiras

Concepción, 20 de marzo de 2023

La educación es el gran motor del desarrollo personal. Sólo a través de ella la hija de un campesino puede convertirse en médico, el hijo de un minero puede llegar a ser director de la mina, o el de un granjero puede llegar a ser presidente de una gran nación.

Nelson Mandela

Agradecimientos

La realización de este reporte no solo representa la conclusión de seis meses de trabajo, sino también es la conclusión de seis años de estudio universitario. No habría podido llegar a este punto sin el inmenso apoyo de mis padres, Leonor Cuevas Cid y Armando Aravena Gómez, a quienes agradezco profundamente el respaldo que me brindaron cuando decidí retomar los estudios a los 28 años. Igualmente agradezco a mi familia, en especial a mis hermanas Evelyn y María Paz, por su constante aliento y motivación.

Agradezco a mi profesor patrocinante, José Fuentes, por su valiosa ayuda en la realización de este trabajo y por su constante comunicación durante estos meses, lo cual permitió concluir esta memoria con resultados satisfactorios. También quiero agradecer al profesor copatrocinante, Diego Seco, por sus valiosas ideas y críticas constructivas que contribuyeron significativamente en el desarrollo de este trabajo. Asimismo, quiero agradecer al profesor Zheng Li por su generosa documentación sobre la instalación de Hadoop, la cual fue de gran ayuda en la fase inicial de esta memoria. La colaboración y el apoyo de estos docentes han sido fundamentales en el éxito de este proyecto.

A Mauricio Echavarría le agradezco facilitarme materiales necesarios para la instalación del clúster y también por su ayuda durante mis años de estudiante. Siempre estuvo dispuesto a brindarme su ayuda sin problema alguno cuando la necesité.

También quiero expresar mi más sincero agradecimiento a mis amigas Nany y Jazmín, quienes han sido una parte crucial de mi vida. Siempre han estado allí para apoyarme, escucharme y alegrarme los días más difíciles. Me han brindado su amistad incondicional, lo cual valoro profundamente.

Gracias a todos los profesores que me han guiado y apoyado durante estos seis años, en especial a los docentes del Departamento de Ingeniería Civil Informática y Ciencias de la Computación. Gracias a su dedicación y enseñanzas, he logrado adquirir conocimientos sólidos y he desarrollado habilidades importantes que me permitirán enfrentar los retos futuros. Les agradezco su paciencia, sabiduría y compromiso en mi formación académica y personal.

Gracias a mis compañeros y amigos más cercanos de la UdeC, quienes han sido un soporte importante en mi vida: Ivonne, Matías, Patricio, Sergio y en especial a Felipe, quién también me aportó con materiales para montar la red local del clúster.

Finalmente, quiero expresar mi agradecimiento especial a la ex presidenta Michelle Bachelet por su gestión durante su segundo gobierno, en el cual se instauró la gratuidad en la educación superior. Gracias a esta iniciativa pude cumplir mi sueño de estudiar Informática en la UdeC. ♡

Resumen

En los entornos de procesamiento distribuido se manejan archivos de datos que a menudo son de gran tamaño. Esto puede generar dificultades en el procesamiento de los datos en entornos con recursos limitados, cuando se incrementa el tamaño de estos. Para superar este problema, se pueden aplicar técnicas de codificación para reducir el tamaño de los datos.

En esta memoria de título se evaluó el rendimiento de la utilización de códigos de largo fijo y de largo variable en la aplicación WordCount en entornos de procesamiento distribuido.

Los resultados obtenidos fueron alentadores, ya que se logró reducir el espacio de almacenamiento de manera considerable mediante la aplicación de ambas técnicas de codificación, utilizando entre un 30 % y 70 % menos espacio. La aplicación que procesa los códigos de largo fijo obtuvo mejores resultados que la aplicación original WordCount, aproximadamente un 12 % menos de tiempo de ejecución. Sin embargo, la aplicación que procesa los códigos de largo variable presentó tiempos más altos, comparándolo con WordCount (aproximadamente un 12 % más), a pesar de reducir considerablemente el tamaño de los datos. Para mejorar los tiempos de ejecución se sugiere continuar con el desarrollo de esta aplicación.

Índice

Índice de figuras

Índice de cuadros

1. Introducción

Resulta evidente cómo el avance tecnológico de las últimas décadas ha permitido la proliferación de dispositivos conectados a internet, que van desde pequeños aparatos y poco potentes como cámaras de video, electrodomésticos o sensores hasta las grandes granjas de servidores y *data centers* que forman el núcleo de la red mundial. La abundancia y diversidad de estos dispositivos, junto con la enorme cantidad de datos generados, ha fomentado el desarrollo de distintos paradigmas de procesamiento de estos datos, como el *cloud computing* y el *edge computing*, cuya relación se observa en la figura 1.

Mientras que en el *cloud computing* la información generada por los nodos más alejados de la red es transmitida hacia los núcleos (la *nube*) para ser procesada, en el *edge computing* la información es procesada, parcial o totalmente, más cerca de los nodos en donde se genera esta información (el *edge*), para luego ser enviada a la nube de ser necesario. Este último enfoque tiene la ventaja de reducir la cantidad de información que se transmite y obtener respuestas más rápidas [2].

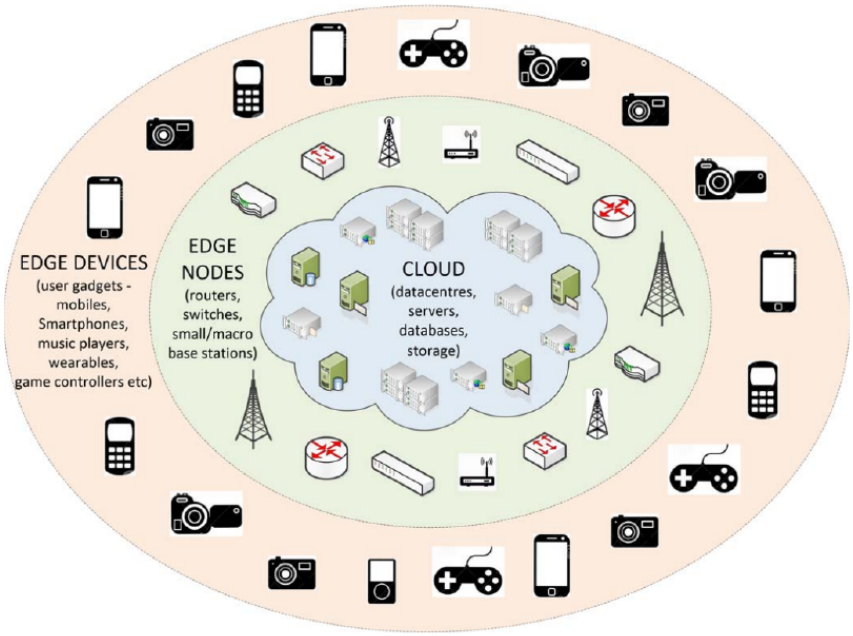


Figura 1: Dispositivos *Edge* y Nodos *Edge* en relación con el *Cloud*. [1]

Una gran parte de los dispositivos que forman parte del *edge* tienen limitaciones de recursos, ya sea de memoria, almacenamiento, velocidad de transmisión y capacidad de procesamiento, por lo que reducir el espacio utilizado por los datos a procesar puede aportar una mejora en los tiempos de respuesta.

La forma más común de reducir el espacio utilizado por los datos es usando algoritmos de compresión, pero se requiere que estos datos sean descomprimidos para su procesamiento. Como alternativa surgen las estructuras de datos compactas (CDS por sus siglas en inglés), las que permiten una disminución del espacio usado por los datos y permiten realizar opera-

ciones de consultas y manipulación directamente en su forma compacta [3]. Aunque las CDS ya cuentan con un cuerpo sólido de conocimiento, el estudio de su uso en el edge computing es reciente y no se ha abordado en profundidad [4].

El objetivo de esta memoria de título es la realización de un estudio que entregue resultados del impacto que tiene codificar los datos, usando técnicas de las CDS, en un ambiente de procesamiento distribuido. El estudio será realizado en el ambiente de procesamiento distribuido Hadoop [8], el cual será implementado en un clúster de computadores de recursos limitados.

Para el cumplimiento del objetivo de este estudio se deben cumplir las tres etapas mostradas en la figura 2.

Figura 2: Etapas para completar el objetivo de esta memoria de título.

No obstante, este estudio se enfoca en la etapa central, correspondiente al procesamiento de los datos codificados en el ambiente de procesamiento distribuido Hadoop (de ahora en adelante será llamado solamente *Hadoop*). La etapa inicial es necesaria para generar estos datos codificados. La etapa final es necesaria para comprobar que los datos codificados procesados sean equivalentes a los datos originales. Solamente la etapa central es la que se realiza en Hadoop. Para procesar estos datos se utilizará la aplicación de MapReduce **WordCount**.

Para lograr este estudio se deben cumplir los siguientes objetivos específicos:

1. Implementar un clúster de computadores de recursos limitados. Este clúster debe configurarse con el ambiente de procesamiento distribuido Hadoop para la realización de pruebas.

2. Implementar la aplicación de MapReduce WordCount para que sirva como aplicación base de los experimentos. Esta aplicación procesa archivos de texto plano.

3. Desarrollar software para generar archivos codificados, usando técnicas de CDS, a partir de archivos de texto plano con el objetivo de reducir su espacio de almacenamiento.

4. Desarrollar aplicaciones de MapReduce que procesen estos archivos codificados y que entreguen resultados equivalentes a los entregados por los archivos de texto plano procesados.

5. Realizar experimentaciones para determinar si el menor tamaño de los archivos codificados impacta en el rendimiento al ejecutarse en Hadoop.

6. Escribir un reporte con los conocimientos adquiridos al realizar este estudio y los resultados obtenidos en la experimentación.

Para la comprensión del estudio realizado es necesario entregar antecedentes que detallen los conceptos, algoritmos y herramientas que fueron utilizados en este estudio. En el siguiente capítulo se reportan estos antecedentes.

2. Antecedentes

El estudio empírico sobre el uso de datos codificados para la aplicación WordCount en Hadoop, requiere de conocer varios antecedentes. Primero se necesita conocer sobre la codificación que se utilizó sobre los datos para reducir su tamaño. También es necesario comprender cómo funciona Hadoop, cuáles son sus componentes y cómo funcionan las aplicaciones de MapReduce. Finalmente se debe conocer la aplicación de MapReduce WordCount, la cual es la base que permitió desarrollar nuevas aplicaciones para procesar los datos codificados.

2.1. Codificación

La información almacenada en los computadores se puede definir como agrupaciones de bytes, que a su vez están formados por grupos de 8 bits, donde cada bit puede tener el valor de 1 o 0. Ya sean archivos de texto, imágenes, música o videos todo se almacena como bytes. La forma cómo se interpretan estos bytes es lo que da sentido y utilidad a la información.

Una de las formas más simples de codificar conjuntos de datos para reducir el espacio que utilizan, es cambiar la cantidad de bits o bytes usados por estos datos. Una forma de hacer esto es generar nuevas formas de codificación, las cuáles pueden tener una cantidad de bits o bytes fija o variable.

2.1.1. Códigos de largo fijo

La codificación de largo fijo consiste en asignar nuevos códigos a los datos, donde cada código generado tiene un tamaño fijo, ya sea de bits o de bytes. En la figura 3 se muestra un ejemplo de codificación de largo fijo a nivel de bytes. En *a)* se tiene un arreglo de 4 números que utilizan 3 bytes cada uno para ser representados. En *b)* estos números se codifican de tal manera que ahora solamente utilizan 1 byte para ser represantados. En *c)* se tiene una tabla que indica la equivalencia entre estos números. La información de esta tabla permite codificar y decodificar estos números. Llamaremos a los números de las figuras *b)* y *c)* **códigos**, los cuales representan a los valores originales.

El objetivo de esta codificación es utilizar menos espacio de memoria o almacenamiento. Por ejemplo, si se tuviese un arreglo de números enteros positivos, con valores pequeños (menores a 255), se podrían codificar usando solamente un byte por valor. Esta forma de codificación se ve bastante beneficiada cuando los datos originales presentan números que se repiten con frecuencia. Si los números originales son todos únicos, la codificación de largo fijo no es tan efectiva.

2.1.2. Códigos de largo variable

Los códigos variables se pueden dividir en dos categorías: códigos de bits variables y códigos de bytes variables.

En el primer caso se generan códigos donde la cantidad de bits para representar los datos originales es variable. Uno de los algoritmos utilizados para codificar un conjunto de datos

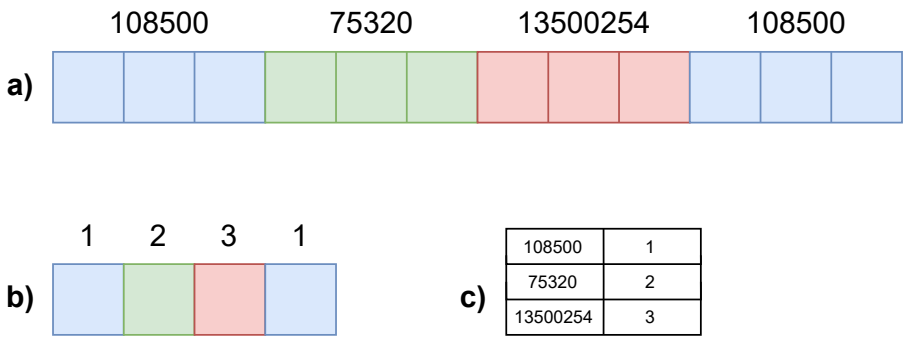


Figura 3: Representación de un arreglo de números enteros y su codificación de tamaño fijo.

como códigos de bits de largo variable es la **Codificación de Huffman** [5]. En este algoritmo se generan códigos más pequeños para los símbolos más frecuentes y códigos más largos para los menos frecuentes. La codificación se inicia con el análisis del conjunto de datos para determinar la frecuencia de aparición de cada símbolo. Luego se construye un árbol binario en el que cada hoja representa un símbolo y la longitud desde la raíz del árbol hasta la hoja es el código binario asignado a este símbolo.

Para construir el árbol, el algoritmo de Huffman utiliza una cola de prioridad que inicialmente contiene un nodo para cada símbolo, con su frecuencia de aparición como prioridad. En cada iteración, el algoritmo saca los dos nodos con menor prioridad de la cola, los combina en un nuevo nodo cuya prioridad es la suma de las prioridades de los nodos combinados, y agrega el nuevo nodo a la cola. Este proceso se repite hasta que solo queda un nodo en la cola, que es la raíz del árbol.

El árbol resultante tiene la propiedad de que los códigos binarios asignados a cada símbolo son prefijos únicos. Esto significa que ningún código asignado a un símbolo es un prefijo de otro código asignado a un símbolo diferente. Esta propiedad permite decodificar el archivo comprimido utilizando el árbol de Huffman de manera eficiente, simplemente leyendo los bits del archivo uno a uno y siguiendo el camino apropiado en el árbol hasta encontrar una hoja. El símbolo correspondiente a la hoja es el siguiente símbolo decodificado.

En la figura 4 se muestra un ejemplo de la codificación de Huffman, donde se codifica la palabra *abracadabra*. En *a)* se tiene el árbol generado. En cada nodo se representa la suma de las frecuencias de sus nodos hijos. La tabla *b)* indica la letra, su frecuencia y la codificación binaria obtenida. En *c)* se tiene la palabra codificada. Los códigos están separados por puntos para facilitar su lectura en este ejemplo. Se puede observar el cumplimiento de la propiedad de que cada código no es prefijo de ningún otro.

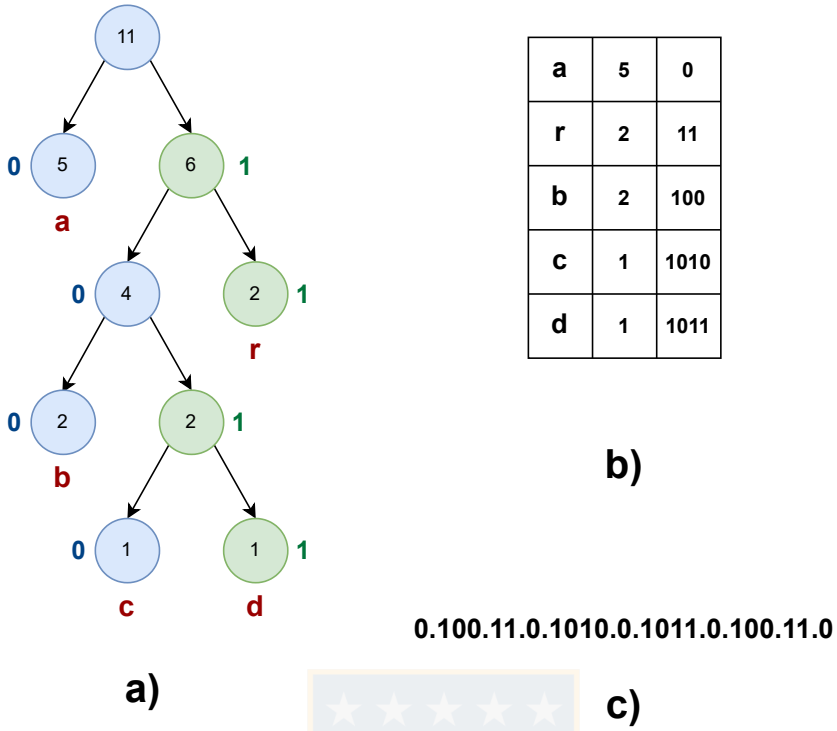


Figura 4: Codificación de Huffman de la palabra *abracadabra*.

VByte (*Variable Byte*) es un algoritmo de codificación que permite codificar números enteros usando cantidades variables de bytes. Este algoritmo de codificación se utiliza para reducir el espacio de almacenamiento de números enteros positivos. Se puede considerar que cada número codificado es represantado por un arreglo de bytes de tamaño variable, en vez de la codificación tradicional de 4 bytes. Por cada byte de este arreglo se utilizan los 7 bits menos significativos para codificar el número. El bit más significativo de cada byte es llamado *bit de continuación* e indica si este byte es el último del arreglo o no. Existen variaciones en cómo se utiliza este bit de continuación. Algunas versiones consideran que el bit más significativo con valor 0 indica que este byte es el último del arreglo [6] y otras consideran que esto se define cuando el bit es 1 [7]. En este reporte se utilizará la segunda versión, como se muestra en la figura 5, donde en *a)* se tiene un número codificado que utiliza un byte y en *b)* hay otro número codificado que utiliza dos bytes. El bit de continuación está marcado con un rectángulo verde.

En la figura 6 se muestra un ejemplo de cómo se realiza la codificación VByte del número entero 2808 representado en forma binaria en *a)*. En este esquema se omiten los dos bytes más significativos del entero, puesto que solamente contienen bits con valor cero. El número entero se divide en secciones de 7 bits para generar el número codificado *b)*. En este caso se puede representar este número con dos bytes en vez de los cuatro originales. Para números

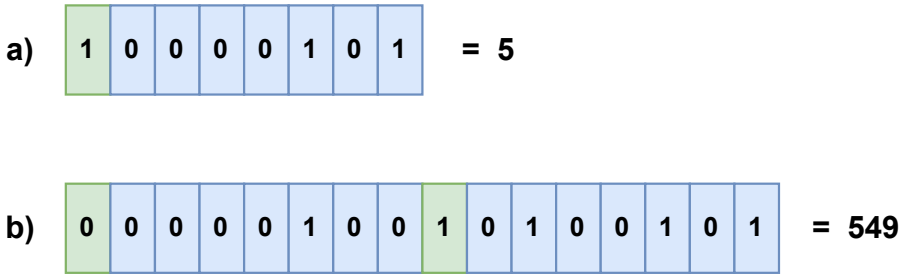


Figura 5: Representación de dos números enteros codificados con VByte.

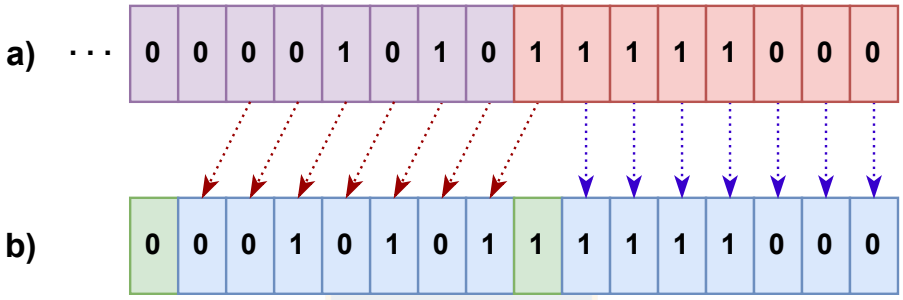


Figura 6: Codificación del número entero 2808 usando VByte.

mayores a 2^{28} se necesitan 5 bytes para realizar la codificación VByte, porque en cada byte se tienen 7 bits para almacenar la información.

La decodificación se realiza de manera inversa. Desde el número codificado se genera un número entero obteniendo la información de los 7 bits menos significativos de cada byte, como se observa en la figura 7. El bit de continuación indica la cantidad de bytes que utiliza el número codificado. El resto de los bits más significativos del número decodificado se establecen en 0.

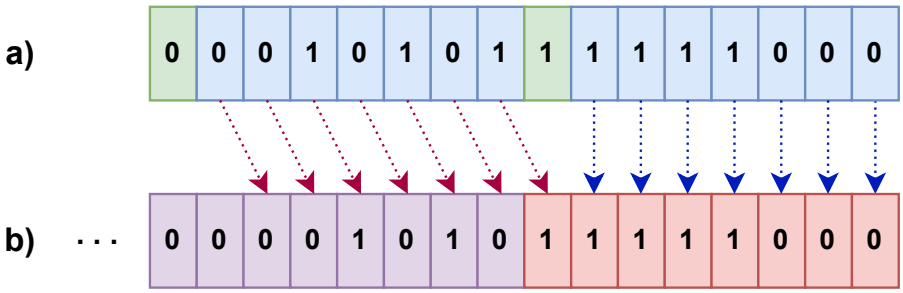


Figura 7: Decodificación de un número codificado con VByte a número entero.

2.2. Apache Hadoop

Apache Hadoop es un *framework* de software que permite el procesamiento distribuido de grandes volúmenes de datos en clústeres de computadores. Está desarrollado en Java, es de código abierto y es mantenido por la Apache Software Foundation. [8]

El framework Hadoop tiene varios componentes, siendo los principales el sistema de almacenamiento distribuido **Hadoop Distributed File System** (HDFS), el gestor de recursos y aplicaciones **Hadoop YARN** y el modelo de procesamiento distribuido **Hadoop MapReduce**.

HDFS permite el almacenamiento de grandes volúmenes de datos en distintas máquinas (conocidos como nodos) garantizando la alta disponibilidad e integridad de estos gracias a su diseño basado en la redundancia, detección y control de fallas. Esto permite a los usuarios montar ambientes de procesamiento distribuido con máquinas que sean menos fiables y de menor costo en vez de tener máquinas de alta fiabilidad con costos muy elevados.

Hadoop YARN *(Yet Another Resource Negotiator)* es el componente que se encarga de la gestión de los recursos y de la planificación de las tareas en los clústeres de datos. De esta manera los usuarios pueden ejecutar diferentes aplicaciones asignando los recursos del clúster de manera eficiente y monitoreando la ejecución, informando el estado en que se encuentra y si se ha completado de manera exitosa o con errores.

MapReduce se utiliza para el procesamiento de los datos distribuidos de manera eficiente y escalable, dividiendo las tareas en varias partes y ejecutándolas en paralelo en los diferentes nodos que forman parte del clúster.

Además existen otras herramientas que forman parte del ecosistema de Hadoop:

- Apache HBase: Es una base de datos no relacional y distribuida que funciona sobre el sistema de datos distribuidos HDFS [9].
- Apache Hive: Es un motor de procesamiento de datos que permite realizar consultas y análisis de grandes conjuntos de datos mediante una interfaz similar a SQL llamada HiveQL [10].
- Apache Spark: Es otro motor de procesamiento de grandes volúmenes de datos en tiempo real. Proporciona bibliotecas para el procesamiento de datos en tiempo real, aprendizaje automático y análisis de grafos [11].

A continuación se detallan los componentes principales de Hadoop que formaron parte importante del trabajo de esta memoria de título.

2.2.1. Hadoop Distributed File System (HDFS)

El sistema de archivos distribuido Hadoop (HDFS) utiliza una arquitectura maestro/esclavo [12]. En el nodo maestro del clúster se define el **NameNode**, el cual es responsable de la administración del sistema de archivos. Los nodos esclavos, conocidos como **DataNodes**, son los encargados de almacenar físicamente los datos en el HDFS.

Los archivos en HDFS pueden considerarse como conjuntos de registros o *records*, que son las unidades básicas de información que son procesadas por las aplicaciones de MapReduce. Los archivos se dividen en uno o más bloques de tamaño fijo (128 MiB por defecto) y se almacenan en los DataNodes. En la figura 8 se observa una representación de un archivo que se divide en tres bloques, que pueden estar ubicados en nodos distintos. Los registros pueden sufrir de cortes en los límites de los bloques, por lo que es necesario definir algoritmos para evitar la pérdida de información en estos casos.

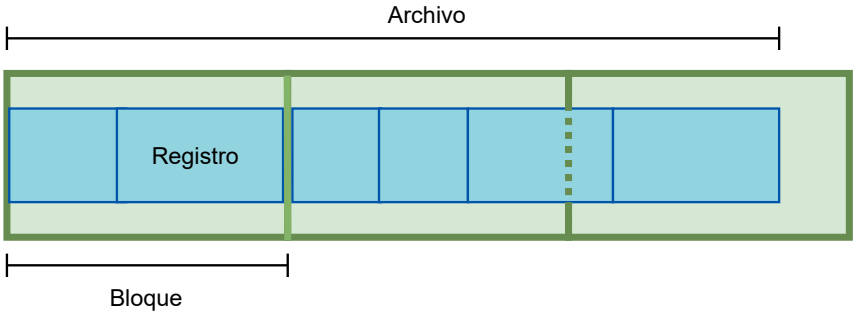


Figura 8: Representación de los archivos en HDFS. Los archivos están formados por registros (rectángulos azules) y son divididos en bloques (rectángulos verdes).

El NameNode es responsable de mantener un registro de los bloques, incluyendo su ubicación en el clúster, metadatos, tamaños y permisos de acceso.

Para garantizar la disponibilidad de los datos, se realiza la replicación de los bloques en diferentes nodos. De esta forma, si un nodo falla y no está disponible, los datos que se almacenaban en ese nodo se pueden obtener desde otros nodos que contienen copias de los mismos bloques, como se observa en el ejemplo de la figura 9, donde los DataNodes contienen los bloques de tres archivos: el archivo rojo que utiliza un bloque, el archivo verde que ocupa dos bloques y el archivo amarillo que usa tres bloques. Si un DataNode falla y se vuelve inaccesible aún es posible acceder a los datos debido a que cada bloque está replicado dos veces.

Esto permite el uso de máquinas de menor costo que no sean completamente fiables. En caso de que alguna de estas máquinas falle, se puede reemplazar y el NameNode se encarga de replicar los bloques de datos que se encontraban almacenados en la máquina descartada en la nueva máquina.

Debido a que solo hay un NameNode en el clúster de Hadoop, este se convierte en un punto vulnerable. Si llegara a fallar, no sería posible acceder a los archivos almacenados en el HDFS. El NameNode es el encargado de mantener la información de los archivos y las ubicaciones de los bloques que los componen. Los DataNodes no pueden comunicarse directamente con el cliente, sino que necesitan la intervención del NameNode. Para evitar estos problemas, se define un **Secondary NameNode**, que es un nodo de respaldo con copias exactas de los componentes vitales del NameNode y se actualiza constantemente con los cambios que ocurren en este. Si el NameNode falla, se puede recuperar el estado previo

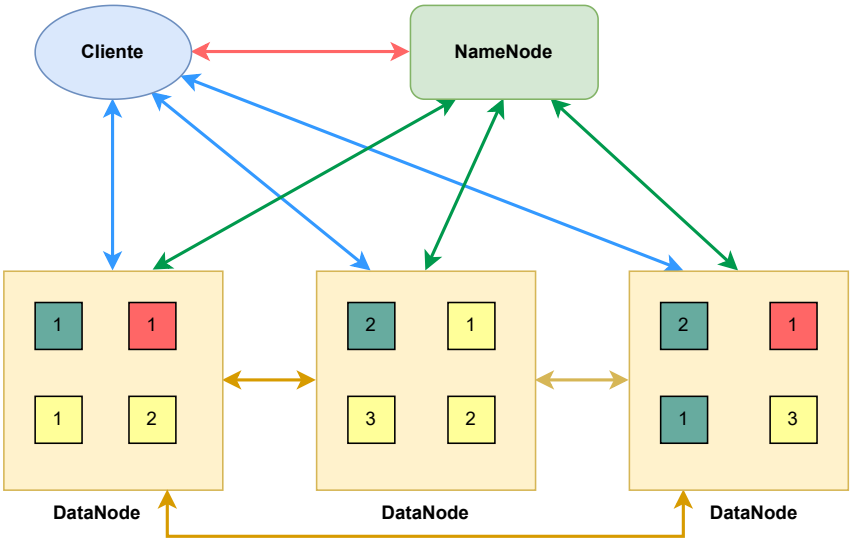


Figura 9: Esquema simplificado de la arquitectura del HDFS.

recurriendo al Secondary NameNode para establecer un nuevo NameNode y mantener la continuidad operativa del HDFS.

Para leer un archivo almacenado en el HDFS, el cliente debe enviar una solicitud al NameNode para obtener la ubicación de los bloques de datos que componen el archivo. El NameNode también verifica que el cliente tenga los permisos necesarios para leer el archivo. Luego, el cliente se conecta a los DataNodes correspondientes para leer el contenido del archivo.

Por otro lado, para escribir un archivo en el HDFS, el cliente envía una petición al NameNode, quien consulta el registro de bloques y determina qué DataNodes almacenarán los bloques del archivo. El archivo se divide en bloques y se distribuyen a los DataNodes correspondientes, creando la cantidad de réplicas que estén configuradas. Una vez que el archivo se ha creado en el HDFS, se notifica al NameNode para que actualice el registro de bloques y metadatos del archivo.

2.2.2. Yet Another Resource Negotiator (YARN)

Hadoop YARN se encarga de la administración de los recursos y planificación de los trabajos que se ejecutan en el clúster. Cuando el cliente solicita la ejecución de un trabajo (*job*) en Hadoop, YARN se encarga de asignar los recursos necesarios para el cumplimiento del trabajo, monitoreando el estado de la ejecución e informando al cliente cuando se completa exitosamente o se producen errores.

Al igual que en HDFS, YARN tiene una arquitectura maestro/esclavo [13], donde el nodo maestro define el **Resource Manager** que se encarga de asignar recursos para la ejecución de trabajos solicitados por el cliente según su prioridad. Los recursos se asignan mediante

contenedores (*containers*), que son entidades abstractas de procesamiento que representan un conjunto de recursos computacionales, como núcleos de CPU, memoria RAM, disco y red. Un contenedor es un entorno aislado que puede ejecutar procesos de una aplicación de manera independiente a otros contenedores. Cuando se inicia una aplicación el Resource Manager asigna un *Application Master* o Maestro de Aplicación que es un contenedor responsable de solicitar recursos al Resource Manager para la ejecución del trabajo y monitorear su estado.

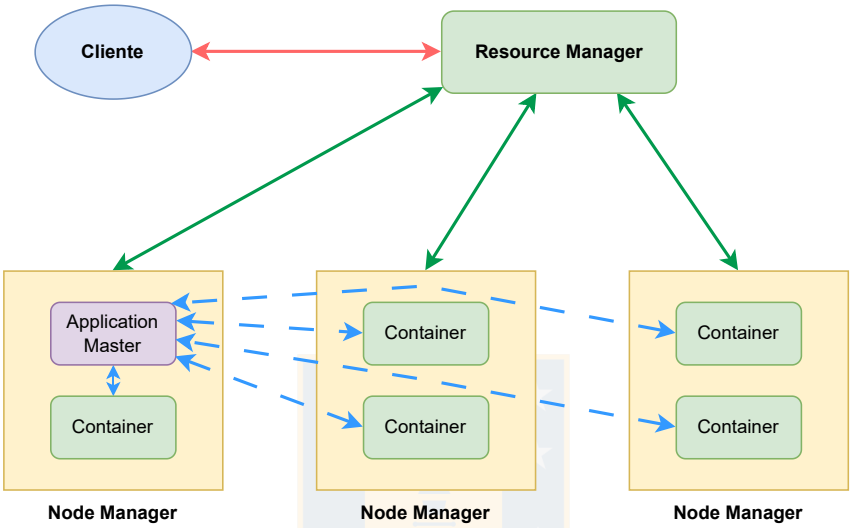


Figura 10: Esquema simplificado de la arquitectura de Hadoop YARN.

En los nodos esclavo se definen los **Node Manager**, quiénes son responsables de administrar los recursos locales, mediante los contenedores, y ejecutar los trabajos asignados por el Resource Manager. En la figura 10 se observa un ejemplo de la arquitectura con tres Node Manager, donde cada uno tiene 2 contenedores. Al momento de ejecutar una aplicación uno de los contenedores es asignado como Application Master quien monitorea al resto de los contenedores.

2.2.3. MapReduce

Hadoop MapReduce es un modelo de programación y un sistema de procesamiento distribuido diseñado para procesar grandes volúmenes de datos que se basa en la idea de que el procesamiento de grandes conjuntos de datos se puede realizar de manera eficiente distribuyendo las tareas en múltiples nodos de un clúster, aprovechando la localidad de los datos en estos nodos para reducir la sobrecarga de transmisión a través de la red.

MapReduce divide los datos de entrada en fragmentos pequeños que tienen la estructura clave-valor para crear *tareas* que se procesan de manera independiente en los contenedores

de los nodos *Node Manager*. El procesamiento pasa por varias tareas, siendo las principales las de **Map** y **Reduce**.

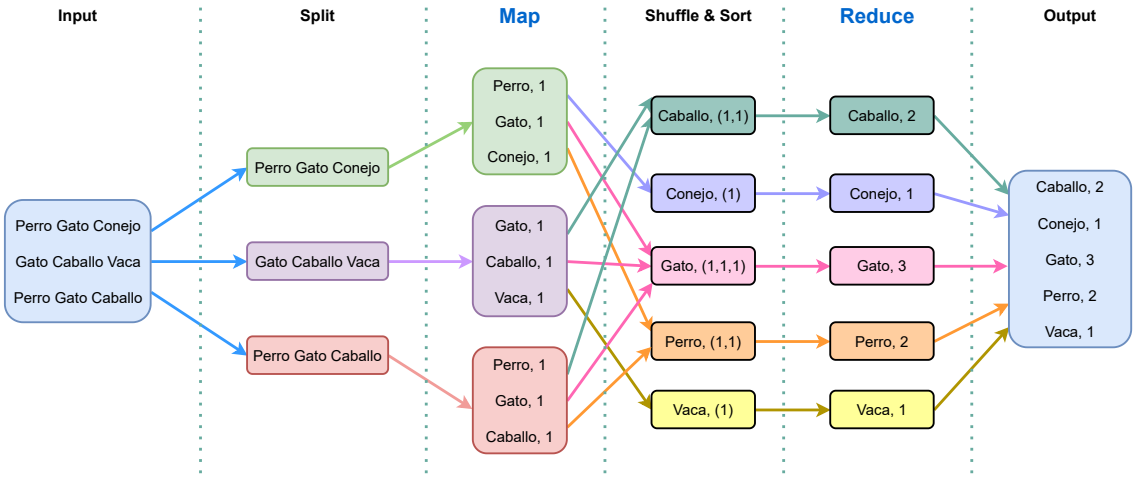


Figura 11: Etapas de una aplicación MapReduce para contar las palabras de un texto.

En la figura 11 se observa un ejemplo de una aplicación MapReduce para contar palabras. Como entrada se recibe un archivo de texto de tres líneas. En la etapa *Split* se separan las líneas y cada línea es procesada por tareas de *Map* distintas que se ejecutan de manera paralela en contenedores independientes. En esta tarea se forman pares clave-valor donde la clave es cada palabra de la línea a la cual se le asigna el valor 1. En la tarea de *Shuffle and Sort* se organizan los datos de tal manera que ahora la clave es la palabra y el valor es una lista de sus valores asignados en la tarea de Map, donde también se ordenan las claves, en este caso el orden es alfabético. En la tarea de *Reduce* se suman los valores de cada palabra para finalmente generar la salida donde se listan las palabras con la cantidad de ocurrencias en el texto.

Este modelo de programación permite la resolución de problemas con datos que sean paralelizables y que permitan procesarlos como pares clave-valor, por lo que no cualquier problema puede ser resuelto usando MapReduce.

2.2.4. Interfaces web de Hadoop

Hadoop también proporciona servidores web para informar sobre el estado del sistema de almacenamiento distribuido (HDFS) y para monitorear el estado actual de la ejecución de una aplicación de MapReduce mediante YARN. Además, el servidor *HistoryServer* proporciona una interfaz web sobre el historial de las ejecuciones de los trabajos (jobs) donde se entrega información muy completa sobre el uso de los recursos que se utilizaron en ese trabajo. Se puede acceder desde cualquier navegador web, desde máquinas conectadas en la misma red local que el clúster, ingresando la dirección IP local del nodo Master y el puerto

correspondiente al servicio. Desde el puerto 9870 se accede al servicio web que informa sobre el estado del HDFS. En la figura 12 se muestra la información que entrega este servicio sobre el estado de almacenamiento de los DataNodes. También se proporciona un navegador de archivos para el HDFS (figura 13). Al pulsar el nombre del archivo se muestra información sobre los bloques que componen el archivo y en cuáles DataNodes están almacenados físicamente.

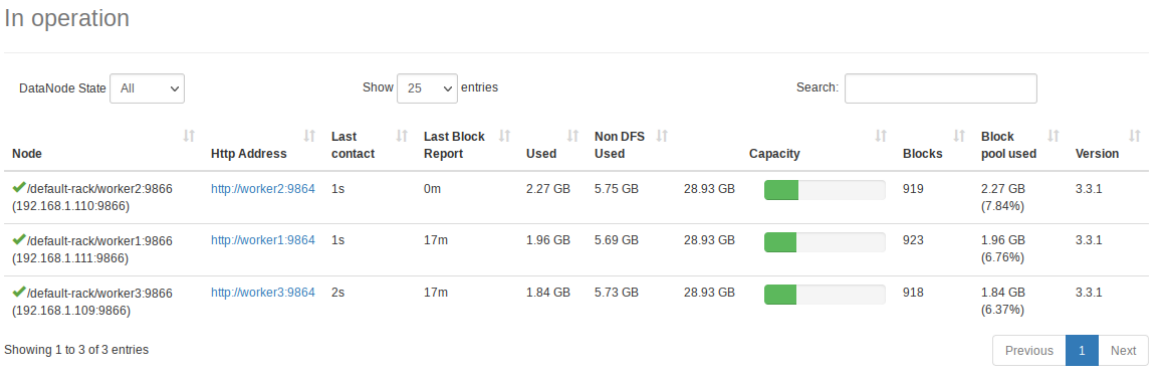


Figura 12: Estado de almacenamiento de los DataNodes del HDFS.

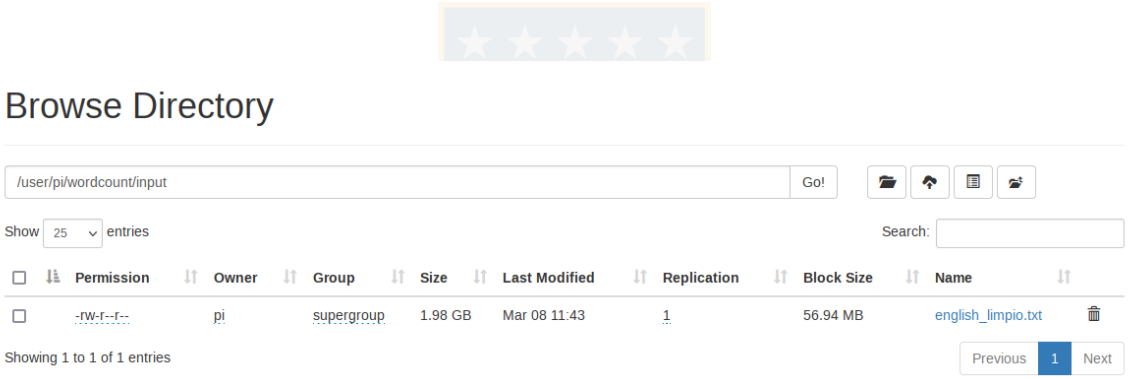


Figura 13: Navegador de archivos para el HDFS.

Si accede al puerto 8088 se puede observar el estado de los trabajos que están ejecutándose mediante YARN. En la figura 14 se muestra la información general de la ejecución. En la figura 15 se puede observar la información de los recursos asignados para la aplicación. En la parte superior se observa que se utilizarán 28 containers en total, de 1 GiB de memoria y 1 Core de CPU cada uno. En la parte inferior se muestran los containers que actualmente se están utilizando de manera paralela, que corresponden a la capacidad configurada del clúster de 9 containers.

Al historial de ejecuciones de los trabajos se accede desde el puerto 19888. En la figura 16 se muestra la vista inicial de esta página, donde se listan los trabajos pasados. Al seleccionar uno de estos trabajos se muestra información muy detallada de los recursos utilizados, los tiempos de ejecución de las distintas tareas, la cantidad de bytes leídos y escritos en el

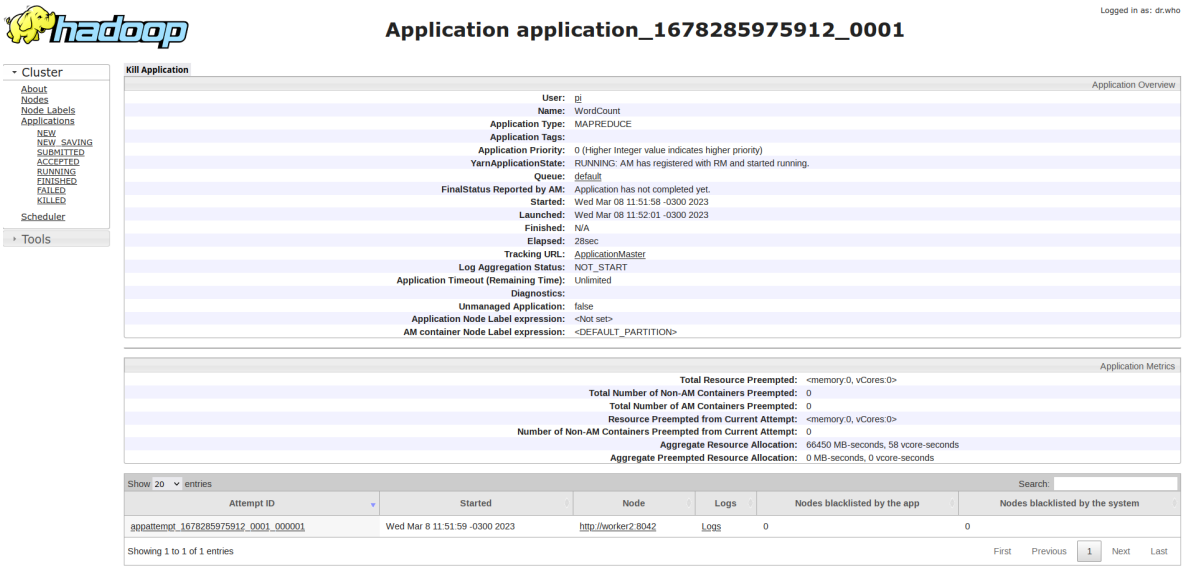


Figura 14: Vista general de la ejecución de un trabajo en YARN.

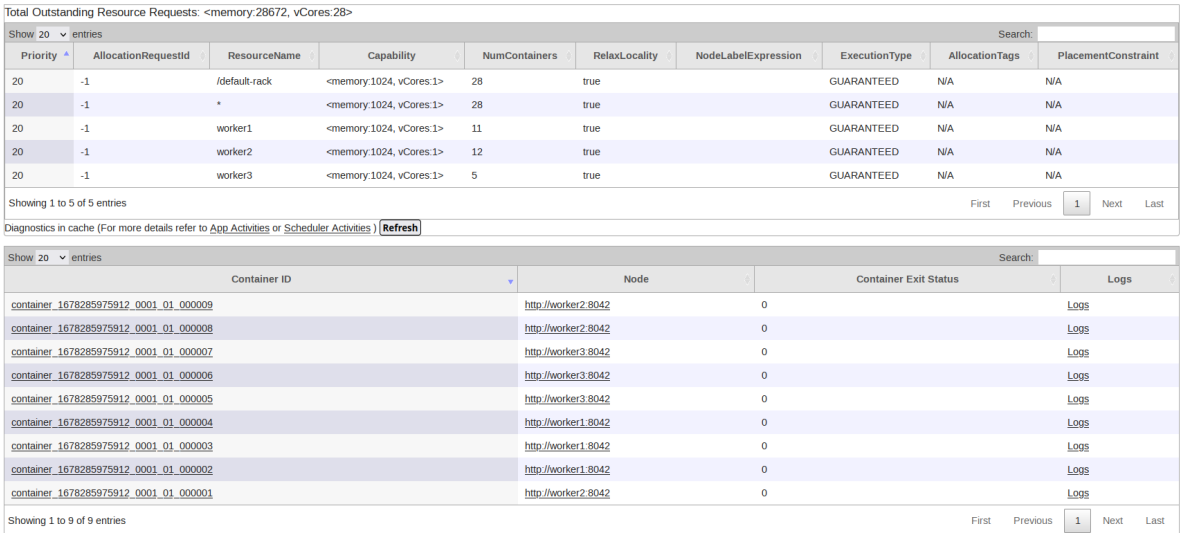
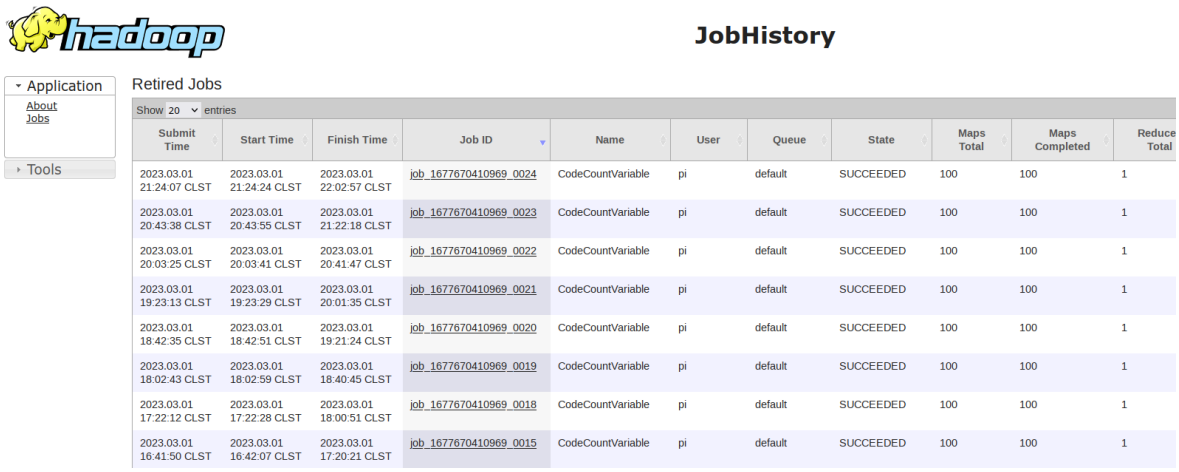


Figura 15: Vista de los recursos asignados por YARN.

HDFS, el uso de la CPU, el tiempo utilizado por el Garbage Collector (GC) de Java, entre otros datos. En la figura 17 se muestra una parte de los datos que entrega el HistoryServer, correspondiente a las tareas de Map y Reduce.



Submit Time	Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduce Total
2023.03.01 21:24:07 CLST	2023.03.01 21:24:24 CLST	2023.03.01 22:02:57 CLST	job_1677670410969_0024	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 20:43:38 CLST	2023.03.01 20:43:55 CLST	2023.03.01 21:22:18 CLST	job_1677670410969_0023	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 20:03:25 CLST	2023.03.01 20:03:41 CLST	2023.03.01 20:41:47 CLST	job_1677670410969_0022	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 19:23:13 CLST	2023.03.01 19:23:29 CLST	2023.03.01 20:01:35 CLST	job_1677670410969_0021	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 18:42:35 CLST	2023.03.01 18:42:51 CLST	2023.03.01 19:21:24 CLST	job_1677670410969_0020	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 18:02:43 CLST	2023.03.01 18:02:59 CLST	2023.03.01 18:40:45 CLST	job_1677670410969_0019	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 17:22:12 CLST	2023.03.01 17:22:28 CLST	2023.03.01 18:00:51 CLST	job_1677670410969_0018	CodeCountVariable	pi	default	SUCCEEDED	100	100	1
2023.03.01 16:41:50 CLST	2023.03.01 16:42:07 CLST	2023.03.01 17:20:21 CLST	job_1677670410969_0015	CodeCountVariable	pi	default	SUCCEEDED	100	100	1

Figura 16: Listado del historial de trabajos.

	Name	Map	Reduce	Total
	Combine input records	3,490,332,824	0	3,490,332,824
	Combine output records	263,528,628	0	263,528,628
	CPU time spent (ms)	15,134,940	289,770	15,424,710
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	86,448	2,904	89,352
	Input split bytes	15,700	0	15,700
	Map input records	3,330,515,209	0	3,330,515,209
	Map output bytes	26,644,121,672	0	26,644,121,672
	Map output materialized bytes	1,037,110,730	0	1,037,110,730
	Map output records	3,330,515,209	0	3,330,515,209
	Merged Map outputs	0	100	100
Map-Reduce Framework	Peak Map Physical memory (bytes)	485,933,056	0	485,933,056
	Peak Map Virtual memory (bytes)	2,017,337,344	0	2,017,337,344
	Peak Reduce Physical memory (bytes)	0	792,207,360	792,207,360
	Peak Reduce Virtual memory (bytes)	0	2,471,292,928	2,471,292,928
	Physical memory (bytes) snapshot	47,639,420,928	779,988,992	48,419,409,920

Figura 17: Algunos datos del historial correspondientes al Framework MapReduce.

2.2.5. Tipos de Datos de Hadoop

Los tipos de datos utilizados en las aplicaciones de MapReduce deben heredar de la interfaz Writable de Hadoop [14]. Una interfaz es un conjunto de métodos que proporciona un contrato que describe cómo otros componentes de software pueden interactuar con la clase que implementa la interfaz. En este caso, la interfaz Writable proporciona métodos que permiten la serialización y deserialización de los datos para ser transmitidos y usados de manera distribuida. En el cuadro 1 se enumeran algunos de los tipos de datos más importantes en Hadoop que heredan de la interfaz Writable, junto con sus equivalencias en los tipos de datos de Java.

IntWritable	Integer
FloatWritable	Float
LongWritable	Long
DoubleWritable	Double
Text	String
ByteWritable	Byte
NullWritable	Null

Cuadro 1: Algunos tipos de datos de Hadoop y su equivalencia.

Es posible crear nuevos tipos de datos para ser utilizados en las aplicaciones MapReduce. Para esto se deben desarrollar clases que implementen la interfaz Writable, definiendo los métodos:

- **write**: En este método se define cómo se realiza la serialización de los datos.
- **readFields**: La deserialización de los datos es definida en este método.

2.2.6. Creación de aplicaciones MapReduce con Java

Se pueden desarrollar aplicaciones de MapReduce utilizando lenguajes de programación como Python y C++, aunque la manera más directa es hacerlo en Java, debido a que Hadoop está diseñado en ese lenguaje. No obstante, la creación de aplicaciones para ser ejecutadas en Hadoop presenta algunas diferencias con la programación convencional. En vez de crear una aplicación completa, se deben definir clases y métodos para procesar archivos de entrada de manera paralela en diferentes nodos y obtener el resultado esperado. [15].

Es necesario definir ciertas clases obligatorias y opcionalmente otras más para adaptarse a las necesidades específicas de la aplicación. Entre las clases obligatorias se encuentran la clase *Mapper*, que define cómo se procesarán los registros en la fase de Map, y la clase *Reducer*, que especifica cómo se procesarán los datos en la fase de Reduce. Adicionalmente se deben establecer las configuraciones de la aplicación, las clases que se usarán y los tipos de dato que componen los pares clave-valor. Estas configuraciones se pueden definir en el método *main* de la aplicación o creando la clase *Driver* para este propósito.

Por otro lado, también se pueden definir clases opcionales, como la clase *Combiner*, que actúa como un mini-Reducer en la fase de Map y ayuda a reducir el tráfico en la red y mejorar la eficiencia del proceso. La clase *InputFormat* permite definir cómo se leerán los datos de entrada y la clase *OutputFormat* es utilizada para especificar el formato de salida de la aplicación. A continuación se describen las clases más importantes con sus métodos.

Mapper: Esta clase tiene la responsabilidad de procesar los datos en la etapa de Map. Recibe los registros de entrada como pares de clave-valor y utiliza el método *map* para producir otros pares clave-valor que serán agrupados y ordenados en la fase de *Shuffle and Sort*.

Reducer: Esta clase se encarga de procesar y reducir los datos recibidos (pares clave-valor) desde la etapa *Shuffle and Sort*. El método *reduce* es donde se establece cómo se realiza la reducción de los valores para cada par clave-valor recibido. Entrega los resultados (pares clave-valor) a la etapa de *Output*.

InputFormat: Es la clase que define el formato que tiene el archivo o archivos a procesar. Sus métodos principales son:

- *createRecordReader:* Este método se utiliza para crear los objetos de la clase *RecordReader*, que es la responsable de leer los registros de los archivos para ser procesados por las tareas de Map.
- *isSplitable:* En este método se define si el archivo de entrada se puede dividir en *splits* o particiones más pequeñas para ser procesadas por tareas de Map distintas y en forma paralela. Si el archivo no es particionable solamente una tarea de Map se encarga de procesar el archivo de manera secuencial.
- *getSplits:* Si el archivo es particionable se debe definir este método que devuelve una lista de *splits* para que sean procesados de manera paralela.

RecordReader: Esta clase es la encargada de leer los registros de cada split en forma secuencial para que sean procesados por las tareas de Map. Contiene los siguientes métodos:

- *nextKeyValue:* Este método recibe un split y obtiene el par clave-valor del registro según en la posición de lectura que se encuentre el RecordReader.
- *getCurrentKey:* Método para devolver la clave actualmente leída por nextKeyValue.
- *getCurrentValue:* Método que devuelve el valor leído por nextKeyValue.
- *getProgress:* Con este método se informa el progreso de lectura de los registros del split.

OutputFormat: Clase responsable de recibir los datos desde la etapa de *Reduce* y escribirlos como archivo en el HDFS. Tiene un método *getRecordWriter* que llama a la clase **RecordWriter** que se encarga de escribir los resultados usando el método *write*. Generalmente los resultados se escriben como pares clave-valor en formato de texto, aunque también se pueden definir formatos personalizados.

En la biblioteca de Hadoop vienen incluidas clases para leer distintos tipos de archivos, algunas de ellas son:

FileInputFormat: Es la clase base para todas las clases que utilizan archivos como entrada para las aplicaciones de MapReduce. Provee los métodos necesarios para crear formatos de entradas personalizados para adaptarse a distintos tipos de archivo [16]. De esta clase se heredan las siguientes:

- **TextInputFormat:** Es la clase por defecto para la lectura de los archivos de entrada de tipo texto en Hadoop. Esta clase considera que cada línea del texto corresponde a un registro, donde el par clave-valor esta formado por el número byte de inicio (*byte offset*) de la línea y el valor corresponde a la línea de texto. Como se observa en el ejemplo de la figura 8 es altamente probable que los registros queden divididos en bloques disintos. Para evitar perder registros por esta situación es que en el método *getSplits* se define el siguiente algoritmo para crear los splits:

  1. En el primer bloque el split comienza en el inicio del archivo y termina en el primer salto de línea del siguiente bloque.
  2. En todos los bloques siguientes, excepto el último bloque, el split comienza desde el segundo registro del bloque (después de encontrar el primer salto de línea) hasta el primer salto de línea del siguiente bloque.
  3. En el último bloque el split comienza desde el segundo registro del bloque (después de encontrar el primer salto de línea) hasta llegar al final del archivo.

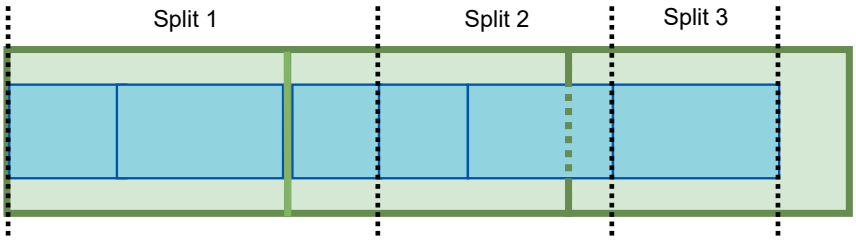


Figura 18: Ejemplo de *splits* generados por la clase *TextInputFormat*. Los rectángulos verdes representan a los bloques y los azules a los registros.

Con este algoritmo se generan splits como los ejemplificados en la figura 18, donde se asegura que no existirán pérdidas de registros.

Para aprovechar la localidad de los datos en los nodos, MapReduce concede los splits a los containers que están asignados en el mismo nodo que los bloques. Siguiendo con el ejemplo de la figura 18, si cada bloque está en un nodo distinto, por ejemplo en los nodos 1, 2 y 3 respectivamente y cada bloque tiene el tamaño por defecto de 128 MiB, MapReduce asigna el primer split al container que está en el nodo 1, el segundo split al container que está en el nodo 2 y el tercero se lo asigna al container del nodo 3. Como el último registro del split 1 está en el nodo 2 se debe transmitir por red hasta el container del nodo 1, lo mismo ocurre con el split 2, que el registro está incompleto y debe transmitirse parte de la información desde el nodo 3 al nodo 2 para su procesamiento. Esta transmisión puede ser de unos pocos bytes o kilobytes (dependiendo de que tan larga sea la línea de texto que forma el registro), lo cual es prácticamente despreciable en comparación con los 128 MiB que están accesibles de manera local.

- **FixedLengthInputFormat:** Esta clase permite procesar archivos binarios que tengan registros de una longitud fija y donde no existan delimitadores entre los registros. En la configuración de la aplicación MapReduce se debe definir de cuántos bytes es esta longitud.

- **SequenceFileInputFormat:** Esta clase se utiliza para procesar archivos que contenga registros secuenciales que forman pares clave-valor. Los archivos secuenciales (*Sequence File*) también pueden contener información de configuración adicional y metadatos. De esta clase derivan **SequenceFileAsTextInputFormat** para archivos secuenciales de texto y **SequenceFileAsBinaryInputFormat** para archivos secuenciales binarios.

- **KeyValueTextInputFormat:** Esta clase es una versión más sofisticada de TextInput-

Format, donde cada registro (cada línea) contiene una clave y valor diferenciados por un separador predefinido, que puede ser un símbolo especial, comas, punto, guiones, etc.

- **CombineFileInputFormat:** Hadoop trabaja más eficiente con la información almacenada en pocos archivos grandes que en muchos archivos pequeños (se consideran archivos pequeños a aquellos de menor tamaño que el bloque), razón por la cual existe esta clase que permite combinar varios archivos pequeños en splits más grandes para que sean procesados por pocas tareas de Map. También existe la clase **CombineTextInputFormat** que permite combinar archivos de texto y **CombineSequenceFileInputFormat** que se utiliza con archivos que contengan el formato Sequence File.

2.2.7. Ejecución de aplicaciones MapReduce

Habiendo descrito los componentes que forman parte de una aplicación de MapReduce, es posible definir con mayor detalle cómo es la ejecución de estas aplicaciones en Hadoop.

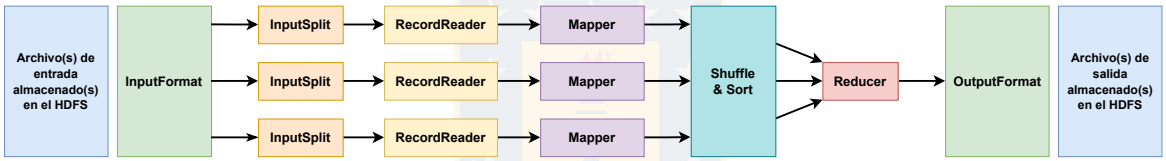


Figura 19: Esquema de ejecución de una aplicación MapReduce en Hadoop.

En la figura 19 se muestra el esquema de ejecución de una aplicación MapReduce en Hadoop. El proceso comienza con los datos de entrada almacenados en uno o más archivos, que son procesados por un objeto de la clase **InputFormat**. Este objeto genera los **InputSplit**, que representan la división de los archivos de entrada, según su composición en bloques, en fragmentos manejables para el procesamiento distribuido. Luego cada InputSplit se comunica con objetos del tipo **RecordReader**, quienes se encargan de leer los registros y generar los pares clave-valor que son entregados a los objetos **Mapper** quienes, a su vez, procesan los datos y entregan pares clave-valor a la etapa de **Shuffle and Sort**, para que sean agrupados, ordenados y entregados al objeto **Reducer**, quien realiza el último procesamiento a los datos y entrega los pares clave-valor resultantes al objeto **OutputFormat** quien es responsable de escribir el archivo o archivos (mediante el *RecordWriter*) en el Sistema de Archivos Distribuidos de Hadoop (HDFS).

Durante la ejecución de la aplicación, la comunicación entre los distintos objetos (representada por las flechas) se realiza a través de un objeto de la clase **Context**, quien permite la interacción de los distintos objetos. En este ejemplo se observa que cada split va asociado a un objeto Mapper, lo que es común en la ejecución de aplicaciones MapReduce. La cantidad

de splits determina la cantidad de tareas de Map que se ejecutan en paralelo. Después de la fase de Shuffle and Sort se tiene una sola tarea de Reduce, lo que también puede ocurrir al ejecutar las aplicaciones, donde se tienen varias tareas de Map y solamente una de Reduce.

2.3. Aplicación WordCount

Se podría decir que la aplicación WordCount es el equivalente al *"Hola Mundo"* para Hadoop. Cuando se realiza la instalación se recomienda ejecutar esta aplicación con algún texto básico y pequeño para verificar que el framework está funcionando correctamente. Es el primer ejemplo de aplicación MapReduce que se presenta en la documentación oficial de Hadoop [17].

Esta aplicación recibe como entrada uno o varios archivos de texto y entrega como salida un archivo de texto que contiene un listado con las palabras, ordenadas alfabéticamente, y la cantidad de ocurrencias de esa palabra en los archivos de entrada. A continuación se muestran las subclases de la clase *WordCount* que compone la aplicación. El código fuente completo se encuentra en [18].

```
1  public static class TokenizerMapper
2       extends Mapper<Object, Text, Text, IntWritable>{
3
4       private final static IntWritable one = new IntWritable(1);
5       private Text word = new Text();
6
7       public void map(Object key, Text value, Context context
8                       ) throws IOException, InterruptedException {
9
10          StringTokenizer itr = new StringTokenizer(value.toString());
11
12          while (itr.hasMoreTokens()) {
13              word.set(itr.nextToken());
14              context.write(word, one);
15          }
16      }
17 }
```

Figura 20: Clase Mapper de la aplicación WordCount.

En la figura 20 se muestra el código de la clase **TokenizerMapper** que hereda de la clase *Mapper*. Esta clase recibe cuatro párametros, los primeros dos corresponden a los tipos de datos del par clave-valor del registro de entrada y los siguientes dos corresponden a los tipos de datos del par clave-valor que devuelve la clase. El registro de entrada es entregado por la clase TextInputFormat, donde la clave del registro es el byteoffset de la línea de texto y el valor es la línea de texto. Como en esta aplicación no se utilizará la clave se define como el tipo genérico *Object*. El valor de entrada corresponde a la línea de texto, por eso es del tipo *Text*. La salida de esta clase es el par («palabra», 1), por eso sus tipos de datos son *Text* y *IntWritable*.

El método **map** (línea 7) recibe como parámetros el par clave-valor del registro de entrada y la referencia al objeto del tipo *Context*, que se encarga de realizar las transferencias de

datos entre las distintas etapas de la ejecución de la aplicación. Usando la clase *StringTokenizer* de Java [19] (línea 10) se separa la línea de texto en palabras individuales. Esta clase separa las palabras por espacios, tabulaciones o saltos de línea. StringTokenizer recibe como argumento de su constructor objetos del tipo *String*, por eso se utiliza el método *toString* del objeto *value*, que es de tipo *Text*, para realizar la separación de las palabras.

A cada palabra que compone la línea de texto (la clave) se le asigna el valor 1, del tipo *IntWritable* y se escribe en el Contexto el par clave-valor: («palabra», 1).

Luego viene la etapa de Shuffle and Sort, la cual es transparente para el usuario y generalmente no es necesario intervenirla, a menos que se requiera un orden distinto al establecido por defecto.

```
1   public static class IntSumReducer
2        extends Reducer<Text, IntWritable, Text, IntWritable> {
3
4     private IntWritable result = new IntWritable();
5
6     public void reduce(Text key, Iterable<IntWritable> values,
7                        Context context
8                        ) throws IOException, InterruptedException {
9       int sum = 0;
10      for (IntWritable val : values) {
11        sum += val.get();
12      }
13      result.set(sum);
14      context.write(key, result);
15    }
16  }
```

Figura 21: Clase Reducer de la aplicación WordCount.

En la figura 21 se muestra la clase **IntSumReducer**, la cual hereda de la clase *Reducer*. Esta clase recibe desde el objeto *Context* el par clave-valor con tipo de dato *Text* para clave y *IntWritable* para el valor y devuelve un par clave-valor con los mismos tipos de datos (línea 2). El método *Reduce* recibe como párametros la clave del par, de tipo *Text* y el valor, el cual corresponde a un objeto *Iterable* del tipo *IntWritable*. El objeto *values* puede considerarse como una lista que permite la iteración por todos los valores que están asociados a la clave *key*. El tercer parámetro corresponde al objeto *Context*. En la línea 10 se realiza la suma de todos los valores del objeto *values*. La suma se realiza con el valor *int* de cada *IntWritable*. En la línea 13 se construye el resultado del tipo *IntWritable* con el entero *sum*. Finalmente se escribe el par clave-valor en el contexto.

Para completar la aplicación WordCount es necesario definir el método *main* donde se establecen las configuraciones de los componentes de la aplicación. En la línea 2 de la figura 22 se crea el objeto de configuración de nombre *conf*. En la línea 3 se crea la instancia de trabajo *job* y se establece un nombre para representar el trabajo que se va a realizar. La línea

```
1   public static void main(String[] args) throws Exception {
2     Configuration conf = new Configuration();
3     Job job = Job.getInstance(conf, "WordCount");
4     job.setJarByClass(WordCount.class);
5     job.setMapperClass(TokenizerMapper.class);
6     job.setCombinerClass(IntSumReducer.class);
7     job.setReducerClass(IntSumReducer.class);
8     job.setOutputKeyClass(Text.class);
9     job.setOutputValueClass(IntWritable.class);
10    FileInputFormat.addInputPath(job, new Path(args[0]));
11    FileOutputFormat.setOutputPath(job, new Path(args[1]));
12    System.exit(job.waitForCompletion(true) ? 0 : 1);
13  }
```

Figura 22: Método *main* de la aplicación WordCount.

4 define la clase principal (WordCount) del *job*. En la línea 5 se establece la clase Mapper. En la línea 6 se define la clase Combiner, que en este caso corresponde a la misma clase Reducer. La línea 7 define la clase Reducer. Las líneas 7 y 8 corresponde a los tipos de datos de salida que tendrán el par clave-valor. En la línea 10 se establece la ruta de los archivos de entrada en el HDFS. En este caso la ruta se recibe por parámetro al momento de ejecutar la aplicación. La ruta donde se almacenará el archivos salida, también en el HDFS, se recibe por parámetro (línea 11). Finalmente en la línea 12 se establece que la salida de la aplicación se realice una vez se complete el trabajo *job*, donde el código de salida 0 indica que la ejecución fue correcta, en caso contrario se termina con código 1. Como el archivo (o archivos) de entrada son de tipo texto y el archivo de salida también es de tipo texto no es necesario definir la configuración de *job.setInputFormatClass* y *job.setOutputFormatClass*.

3. Solución propuesta

3.1. Instalación del ambiente de procesamiento distribuido de datos

El primer paso para realizar el estudio empírico del uso de datos codificados para la aplicación WordCount en el ambiente Hadoop, fue montar el clúster de computadores para crear el ambiente de procesamiento distribuido. La instalación física se realizó en el domicilio del autor. Los computadores utilizados fueron cuatro **Raspberry Pi 4 model B** [20] (un nodo master y tres workers) con las siguientes características de hardware:

- *Procesador:* ARM Quad Core de 64 bits a 1.5 GHz.
- *Ram:* 4 GB para los nodos workers y 8 GB para el nodo master.
- *Red:* Ethernet Gigabit y WiFi 802.11ac de 2.4 GHz y 5.0 GHz.
- *Almacenamiento:* Memoria Flash MicroSD Sandisk Extreme 32 GB con velocidad de escritura de hasta 60 MB/s y de lectura de hasta 100 MB/s para los nodos workers y Memoria Flash MicroSD Samsung EVO de 32 GB con velocidad de transferencia de hasta 95 MB/s para el nodo master.

La instalación de la red se realizó con cable UTP categoría 6 lo que permite velocidad de hasta 1 Gbps. Se utilizó un switch ethernet Mercusys MS105G V1 de cinco bocas con velocidad de hasta 1 Gbps. El switch se conectó al router otorgado por el ISP Movistar, de marca Mitrastar modelo GPT-2541GNAC el cual también tiene velocidad ethernet de hasta 1 Gbps, por lo que se puede considerar que el clúster tiene velocidad de red Gigabit. En la figura 23 se presenta una fotografía con la disposición física del clúster.

A los cuatros nodos se les instaló el sistema operativo Ubuntu Server 22.04.1 LTS para arquitecturas ARM de 64 bits [21]. La instalación se realizó usando la aplicación *Raspberry Pi Imager* [22], la cual permite instalar sistemas operativos para Raspberry Pi directamente en las tarjetas MicroSD y realizar la configuración básica del sistema operativo, como es el nombre del Host, instalación de SSH, nombre de usuario y contraseña, distribución del teclado y zona horaria.

Una vez se instalaron las tarjetas MicroSD en las cuatro Raspberry Pi se configuró el acceso mediante SSH vía WiFi desde el computador de trabajo del autor para proceder con la instalación y configuración de Hadoop y así poner en línea el clúster. En la figura 24 se muestra un esquema de la conexión de red del clúster con el computador de trabajo del autor.

Figura 23: Fotografía del clúster de Raspberry Pi 4.

3.1.1. Instalación de Hadoop

Antes de instalar Hadoop en el clúster es necesario instalar Java, por lo cual se instaló *Java Development Kit* **Open JDK 8** para arquitecturas ARM de 64 bits en cada una de las máquinas [23]. Luego se procede a instalar la versión 3.3.1 de Hadoop, la cual era la versión más reciente para arquitecturas ARM 64 (la versión más reciente para arquitecturas x86 era la 3.3.4) [24]. Se realizan las configuraciones correspondientes para establecer el nodo Master como NameNode en el HDFS y ResourceManager en YARN y a los nodos workers se les establece como DataNode en HDFS y NodeManager en YARN. La configuración de Hadoop se realiza siguiendo la estructura XML, en cuatro archivos [1]:

- **core-site.xml**: Configura qué nodo corresponde al Master.
- **hdf-site.xml**: Contiene la configuración del sistema de archivos distribuidos de Hadoop (HDFS).

[1]La configuración completa utilizada en el clúster se encuentra en [18].

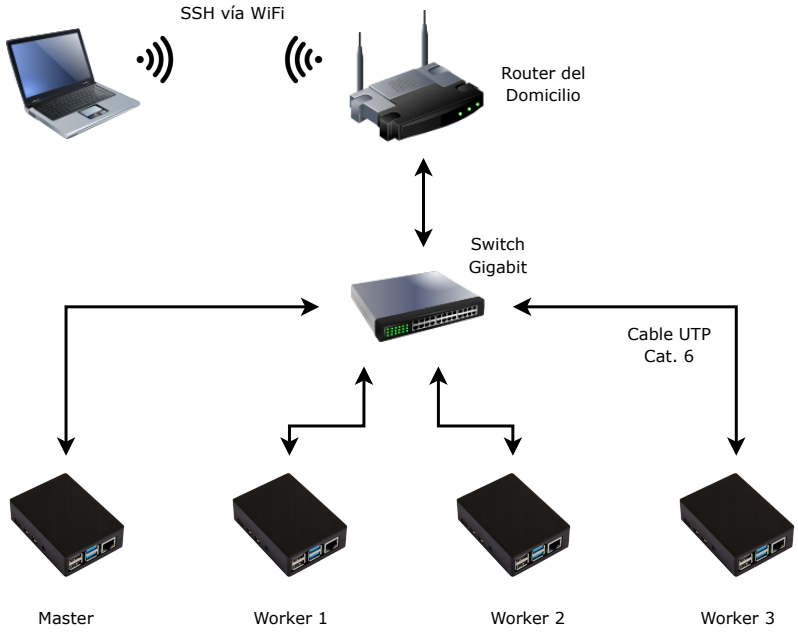


Figura 24: Esquema de la conexión de red del clúster.

- **mapred-site.xml**: En este archivo se configuran los recursos para la ejecución de las aplicaciones MapReduce.
- **yarn-site.xml**: Configuración del administrador de recursos YARN.

Originalmente estos archivos no tienen configuraciones establecidas y es responsabilidad del usuario definir aquellos valores relevantes en los archivos para sobreescribir los valores por defecto. Es necesario replicar estas configuraciones en todos los nodos del clúster.

Como este ambiente es para realizar pruebas y no para producción, además de tener un almacenamiento limitado, se estableció a los bloques del HDFS con replicación 1, es decir, no hay copias de los bloques.

Para configurar los contenedores se siguieron las recomendaciones de [25]. En este sitio se explica cómo configurar los recursos disponibles de acuerdo a las características de hardware de los nodos. Para facilitar los cálculos se utilizó el software *yarn-utils.py* [26] y así se obtuvo la configuración de contenederos recomendada. Esta aplicación recibe como parámetros la cantidad de cores de CPU, la cantidad de memoria RAM y la cantidad de discos de almacenamiento de cada máquina con lo cual entrega las configuraciones recomendadas que deben ingresarse en los archivos *mapred-site.xml* y *yarn-site.xml*. Cada contenedor quedó asignado con un core de CPU y 1 GB de memoria RAM, donde cada nodo worker tiene 3 contenedores. En total se disponen de 9 contenedores, por lo cual se pueden realizar hasta 8 tareas paralelas de Map o Reduce (un contenedor se asigna como Application Master y este se encarga de coordinar y reportar el estado del resto de los contenedores).

3.2. CodeCountFixed

Inicialmente, se propuso la idea de reducir el tamaño de los datasets de texto plano asignando códigos numéricos a las palabras. Es importante tener en cuenta que las palabras son arreglos de caracteres del tipo de dato *char*, los cuales tienen una longitud de 1 byte cada uno. De este modo, una palabra formada por cinco letras utilizará 5 bytes de espacio. En los textos en inglés, las palabras constan, en promedio, de 5 caracteres [27]. Además, los separadores entre palabras, como espacios, tabulaciones, saltos de línea y otros símbolos, también ocupan 1 byte de espacio cada uno (en la codificación ASCII). Con estas consideraciones, se generaron códigos numéricos de longitud fija para cada palabra, con el fin de codificar los textos y reducir el espacio utilizado.

3.2.1. Creación de los códigos de largo fijo

La codificación de las palabras del archivo de texto de entrada se realizó con este sencillo algoritmo:

```
1  generateFixedCode(InputFile)
2      foreach line l ∈ InputFile do
3          words = getWords(l)
4          foreach word w ∈ words do
5              if dictionary.exists(w) == True then
6                  code = dictionary.get(w)
7                  writeToFile(code)
8              else
9                  code = code + 1
10                 dictionary.put(w) = code
11                 writeToFile(code)
```

La implementación de este algoritmo se realiza en el lenguaje Java. Se recibe como entrada un archivo de texto *InputFile* el cual se lee línea por línea usando la clase **BufferedReader**. Las palabras de la línea leída se separan usando la clase **StringTokenizer** (de la misma manera que se utiliza en la clase Mapper de la aplicación WordCount). Como diccionario para almacenar las palabras con su correspondiente código numérico se utiliza la estructura de datos **HashMap** donde la clave corresponde al tipo de dato **String** y el valor al tipo de dato **Integer**. Los códigos corresponden a números enteros secuenciales que inician con el número 0. Por cada palabra de la línea de texto se verifica si existe en el HashMap. En caso afirmativo se obtiene el código (número entero) asociado a la palabra y se escribe en el archivo de salida como datos binarios usando la clase **DataOutputStream**. Si la palabra no existe en el diccionario se aumenta el valor del código en 1, se guarda en el diccionario asociándolo con la palabra y se escribe el código en el archivo binario de salida.

Una vez codificado todo el texto se crea un diccionario de decodificación a partir del diccionario utilizado para almacenar las palabras con sus códigos correspondientes. Este dic-

cionario de decodificación tiene como clave el código número y como valor la palabra, con esto se facilita la decodificación del resultado que entrega la aplicación CodeCountFixed. Este diccionario, que es un objeto de la clase HashMap, se escribe a archivo como objeto de Java para ser utilizado en el programa de decodificación de resultados.

Si el archivo de texto de entrada tiene una cantidad de palabras distintas menor a 16.777.216 (2^{24}) es posible representar los códigos con 3 bytes, en vez de usar enteros de 4 bytes, lo cual disminuye el tamaño del archivo binario de salida. Como se verá en el capítulo 4, dos datasets en lenguaje natural cumplen con tener menos de 2^{24} palabras distintas, por lo que se trata de una situación posible de encontrar en la práctica.

Para utilizar códigos de largo fijo con datasets que contengan una cantidad de palabras distintas superior a 2^{24} se desarrolló una variante de CodeTextFixed llamada **CodeTextFixed4B**, donde los códigos utilizan números enteros de 4 bytes. Al igual que en CodeTextFixed, los códigos númericos se escriben como datos binarios.

En la siguiente sección se detalla cómo la aplicación CodeCountFixed y CodeCountFixed4B leen estos códigos de 3 bytes y 4 bytes respectivamente, para contar la cantidad de ocurrencias y entregar un resultado que sea equivalente al entregado por la aplicación WordCount.

3.2.2. Implementación de la aplicación CodeCountFixed

La aplicación de MapReduce CodeCountFixed recibe como entrada archivos binarios donde las palabras están codificadas como números enteros de 3 bytes de longitud. Para poder leer estos códigos se utiliza la clase **FixedLengthInputFormat** de la biblioteca de Hadoop, la cual se utiliza para leer registros binarios de tamaño fijo. Para simular las líneas de texto se definió que los registros del archivo binario de entrada corresponden a un conjunto de 60 bytes consecutivos, es decir, 20 códigos. Con esto se pretende simular a una aplicación WordCount que lee archivos de texto que contienen 20 palabras por línea.

El método main de la aplicación CodeCountFixed es mostrado en la figura 25 donde se configura el formato de entrada en la línea número 9, la cual indica que se utilizará la clase FixedLengthInputFormat. Además, se debe definir la cantidad de bytes que utilizan los registros. En este caso se tienen 20 códigos de 3 bytes cada uno por cada registro, por lo que el valor es de 60 bytes. Esto se observa en la línea número 3. El archivo de salida que genera esta aplicación es un archivo de texto con los códigos numéricos y su frecuencia de aparición.

La clase Mapper mostrada en la figura 26 puede parecer compleja en comparación con la clase Mapper de la aplicación WordCount, sin embargo lo que se realiza entre las líneas número 10 a la 19 es convertir el arreglo de tres bytes en un entero de cuatro bytes del tipo IntWritable. Esta clase recibe el par clave-valor del tipo Object y BitesWritable, y devuelve el par clave-valor con los tipos IntWritable. Al igual que en la aplicación WordCount, la clave de entrada no se utiliza. El tipo de dato BytesWritable equivale a un arreglo de bytes. Se utiliza un ciclo *for* para leer los 20 códigos del registro de 60 bytes. En la línea número

```
1     public static void main(String[] args) throws Exception {
2      Configuration conf = new Configuration();
3      conf.setInt(FixedLengthInputFormat.FIXED_RECORD_LENGTH, 60);
4      Job job = Job.getInstance(conf, "CodeCountFixed");
5      job.setJarByClass(CodeCountFixed.class);
6      job.setMapperClass(CodedMapper.class);
7      job.setCombinerClass(IntSumReducer.class);
8      job.setReducerClass(IntSumReducer.class);
9      job.setInputFormatClass(FixedLengthInputFormat.class);
10     job.setOutputKeyClass(IntWritable.class);
11     job.setOutputValueClass(IntWritable.class);
12     FileInputFormat.addInputPath(job, new Path(args[0]));
13     FileOutputFormat.setOutputPath(job, new Path(args[1]));
14     System.exit(job.waitForCompletion(true) ? 0 : 1);
15    }
```

Figura 25: Método main de la aplicación CodeCountFixed.

10 y 11 se leen 3 bytes, correspondientes a un código, para almacenarlos en un arreglo de byte llamado **aux**. Entre las líneas 12 y 17 se agrega un byte con valor 0 al inicio del arreglo aux para obtener un arreglo de 4 bytes. En las líneas número 18 y 19 se convierte este arreglo de bytes a un objeto del tipo **ByteBuffer**. Esta conversión es necesaria para crear el par clave-valor, donde la clave es el número entero del tipo *IntWritable* de nombre *word* y el valor corresponde al número **1** del tipo IntWritable. Este par se escribe en el Contexto. En resumen, se sigue el mismo algoritmo que la clase Mapper de la aplicación WordCount, pero utilizando números en vez de cadenas de texto.

La clase Reducer de esta aplicación es prácticamente igual a la clase Reducer de WordCount. Los únicos cambios realizados son los tipos de datos de los pares clave-valor de entrada y salida, donde todos son del tipo IntWritable, como se observa en la figura 27.

La clase FixedLengthInputFormat es exigente en cuanto al tamaño de los registros y puede reportar errores o interrumpir la ejecución de la aplicación si el tamaño de los bytes configurados no se cumple. Para evitar estos problemas, el programa de codificación CodeTextFixed agrega bytes de relleno con valor **0xFF** al final del archivo binario de salida si la cantidad de bytes escritos, correspondientes a la codificación del archivo de texto de entrada, no es un múltiplo de 60, que es el tamaño del registro configurado para su lectura.

```java
  public static class CodedMapper
       extends Mapper<Object, BytesWritable, IntWritable, IntWritable>{

    private final static IntWritable one = new IntWritable(1);

    public void map(Object key, BytesWritable value, Context context
                    ) throws IOException, InterruptedException {

      for (int i = 0; i < 60; i += 3) {
          ByteBuffer byteBuffer = ByteBuffer.wrap(value.getBytes(), i, 3);
          byte[] aux = new byte[byteBuffer.remaining()];
          byteBuffer.get(aux);
          byte byteZero[] = new byte[1];
          byteZero[0] = 0;
          ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
          outputStream.write(byteZero);
          outputStream.write(aux);
          byte intBytes[] = outputStream.toByteArray();
          ByteBuffer intBuffer = ByteBuffer.wrap(intBytes);

          IntWritable word = new IntWritable(intBuffer.getInt());
          context.write(word, one);
      }
    }
  }
```

Figura 26: Clase Mapper de la aplicación CodeCountFixed.

```java
public static class IntSumReducer
       extends Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(IntWritable key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

Figura 27: Clase Reducer de la aplicación CodeCountFixed.

3.2.3. Implementación de la aplicación CodeCountFixed4B

Esta variante de la aplicación CodeCountFixed se utiliza para contar códigos de largo fijo de 4 bytes. En la figura 28 se muestra la implementación de la clase Mapper, la cual es más sencilla que la versión anterior. En la línea número 3 se observa que se siguen utilizando los mismos tipos de datos para los pares clave-valor de entrada y salida. En el ciclo *for* que inicia en la línea número 7 se muestra que la lectura se realiza cada 4 bytes, a diferencia de la versión anterior que se realizaba cada 3 bytes. En la línea número 8 se crea el objeto del tipo **ByteBuffer** con esos 4 bytes leídos para luego, en la línea número 8, crear el número del tipo IntWritable. Finalmente en la línea número 10 se escribe el par clave-valor de salida en el Contexto.

```
1  public class CodeCountFixed4B{
2    public static class CodedMapper
3         extends Mapper<Object, BytesWritable, IntWritable, IntWritable>{
4      private final static IntWritable one = new IntWritable(1);
5      public void map(Object key, BytesWritable value, Context context
6                      ) throws IOException, InterruptedException {
7        for (int i = 0; i < 80; i += 4) {
8            ByteBuffer byteBuffer = ByteBuffer.wrap(value.getBytes(), i, 4);
9            IntWritable word = new IntWritable(byteBuffer.getInt());
10           context.write(word, one);
11       }
12     }
13   }
```

Figura 28: Clase Mapper de la aplicación CodeCountFixed4B.

La clase Reducer es idéntica a la de la aplicación CodeCountFixed. El método *main* solamente cambia en la configuración de la cantidad de bytes que utiliza el registro. Ahora se leen 20 códigos de tamaño 4 bytes, por lo cual el valor de la configuración **FIXED RECORD LENGTH** es 80.

3.3. CodeCountVariable

La principal motivación para usar códigos de largo variable en la codificación de archivos de texto es disminuir aún más el tamaño del archivo generado. Para optimizar la generación de estos se deben crear códigos pequeños para las palabras más frecuentes. A continuación se describe cómo se realizó esta codificación y cómo se creo la aplicación de MapReduce **CodeCountVariable**.

3.3.1. Creación de los códigos de largo variable

La generación de códigos de largo variable es más compleja que la de códigos de largo fijo y requiere leer el archivo de texto de entrada dos veces. Su funcionamiento principal se puede observar en el siguiente algoritmo:

```
1  generateVariableCode(InputFile)
2      foreach line l ∈ InputFile do
3          words = getWords(l)
4          foreach word w ∈ words do
5              if freqDictionary.exists(w) == True then
6                  freq = freqDictionary.get(w) + 1
7                  freqDictionary.put(w) = freq
8              else
9                  freqDictionary.put(w) = 1
10     sortDictionary = invertedSortByValue(freqDictionary)
11     foreach key k ∈ sortDictionary do
12         rankingDictionary.put(k) = i
13         i = i + 1
14     foreach line l ∈ InputFile do
15         words = getWords(l)
16         foreach word w ∈ words do
17             intCode = rankingDictionary.get(w)
18             variableCode = encodeInt(intCode)
19             writeToFile(variableCode)
```

Este algoritmo lee el archivo de texto completo para crear un diccionario donde la clave es la palabra y el valor es la frecuencia (cantidad de repeticiones) de esa palabra en el texto (líneas número 2 a la 9). Luego se ordena el diccionario por el valor para que las palabras más frecuentes queden al inicio del diccionario (línea número 10). A partir de este diccionario ordenado se crea un nuevo diccionario, donde las palabras más frecuentes tienen números más pequeños (líneas número 12 y 13). Después se lee nuevamente el archivo de texto de entrada para crear la codificación variable, donde las palabras más frecuentes tendrán códigos numericos más pequeños. Por cada palabra leída se obtiene su código numérico asociado desde el diccionario *rankingDictionary* y se crea el código de largo variable. La creación de

este código se realiza utilizando el algoritmo VByte. Luego se escribe el código en el archivo binario de salida.

La implementación de esta algoritmo se realiza en Java. Al igual que el programa CodeTextFixed, el archivo de texto de entrada se lee línea a línea usando la clase BufferedReader, mientras que la clase StringTokenizer se usa para la separación de las palabras. Se utiliza un HashMap con par clave-valor del tipo String-Integer como diccionario para almacenar las frecuencias de las palabras. Luego de leer todo el archivo se ordena este diccionario por el valor, de mayor a menor. Luego se utiliza otro HashMap, también del tipo String-Integer, para generar un *ranking* de palabras. Cada palabra tiene un código número único donde la más frecuente tiene valor 0, y la menos frecuente tiene valor $n - 1$, donde n es la cantidad total de palabras distintas dentro del archivo de texto de entrada. El siguiente paso es volver a leer el archivo de texto de entrada, para generar el archivo binario de salida con los códigos de largo variable. Se vuelve a leer línea por línea, separando las palabras de cada línea. Se obtiene el código númerico asociado a la palabra usando el diccionario de ranking y a ese número se le aplica el algoritmo **VByte** para generar el código de largo variable. El código obtenido puede tener longitud de 1 hasta 5 bytes, dependiendo del número entero a codificar. Luego de obtener el código variable se escribe en el archivo binario de salida.

3.3.2. Implementación del algoritmo VByte

La implementación del algoritmo de codificación VByte se muestra en la figura 29 y fue obtenida desde [28]. Este algoritmo recibe un número entero del tipo **int** y se devuelve un arreglo de bytes del tipo **byte[]**. Esta codificación funciona solamente con enteros positivos, por lo que en la línea número 2 se verifica esta condición. En la línea número 5 se crea un arreglo de bytes de tamaño 5, que es la cantidad máxima de bytes necesaria para codificar un entero de 4 bytes. El ciclo *while* de la línea número 8 se ejecuta mientras el número recibido siga siendo mayor a 127, lo que quiere decir que utiliza al menos 2 bytes. Dentro de este ciclo se almacenan en el arreglo **bytes** los 7 bits menos significativos del número. Se incrementa el índice **i** de la posición del arreglo y el número **value** se desplaza 7 bits a la derecha. Una vez terminado este ciclo, en la línea número 13, se almacenan los últimos 7 bits del número *value* y el bit más significativo se establece en 1 para indicar que este byte es el último del arreglo. Se crea un nuevo arreglo de bytes que tenga el tamaño exacto de los bytes utilizados para codificar el número. De esta forma los números más pequeños utilizarán menos bytes que los números más grandes. En la línea número 15 se copia el contenido del arreglo *bytes* a este nuevo arreglo *result* para ser retornado.

3.3.3. Implementación de la aplicación CodeCountVariable

Esta aplicación de MapReduce es más compleja que las anteriores, debido a que requiere la definición de clases para leer los registros del archivo binario que contiene códigos de largo variable. Hadoop no provee bibliotecas para este tipo de dato tan específico, por lo que se desarrolló la clase **VByteInputFormat** que hereda de FileInputFormat para leer las

```java
public static byte[] encodeByteArray(int value) throws IOException {
    if (value < 0) {
        throw new IllegalArgumentException("Value must be positive");
    }
    byte[] bytes = new byte[5];
    int i = 0;

    while(value > 127) {
        bytes[i++] = (byte)(value & 0x7F);
        value>>>=7;
    }

    bytes[i++] = (byte)(value | 0x80);
    byte[] result = new byte[i];
    System.arraycopy(bytes, 0, result, 0, i);
    return result;
}
```

Figura 29: Método para codificar un número entero como arreglo de bytes usando VByte.

palabras codificadas con VByte. Uno de los métodos que se deben sobreescribir (*@override*) es **getSplits**, el cual define los splits (o particiones) que son asignados a cada tarea paralela de Map durante la ejecución de la aplicación. Se define el registro como un código VByte, es decir, cada registro es una palabra codificada. Para crear los splits de manera que no se pierdan los registros, debido a la división del archivo de entrada en bloques, se utilizó la misma idea que se usa en la clase *TextInputFormat*, donde los splits terminan al final del primer registro del siguiente bloque, siguiendo el algoritmo:

1. En el primer bloque el split comienza al inicio del archivo y termina en el primer registro del siguiente bloque.

2. En todos los bloques siguientes, excepto el último bloque, el split comienza desde el segundo registro del bloque hasta el final del primer registro del siguiente bloque.

3. En el último bloque el split comienza desde el segundo registro del bloque hasta llegar al final del archivo.

En la figura 30 se muestra un ejemplo de cómo se generan los splits en un archivo que utiliza 3 bloques (rectángulos verdes) y tiene 9 registros (rectángulos celestes). En cada registro se identifican los bytes que lo componen y el último byte está representado por un rectángulo azul. Este byte tiene el bit de continuación (bit más significativo) en valor 1, lo que indica que este byte es el último del registro.

Para implementar el método *getSplits* se deben identificar los bytes que limitan los splits, almacenando estas posiciones en una lista del tipo *List<long>*. Para eso se siguió el algoritmo descrito anteriormente, donde el primer split comienza al inicio del archivo binario

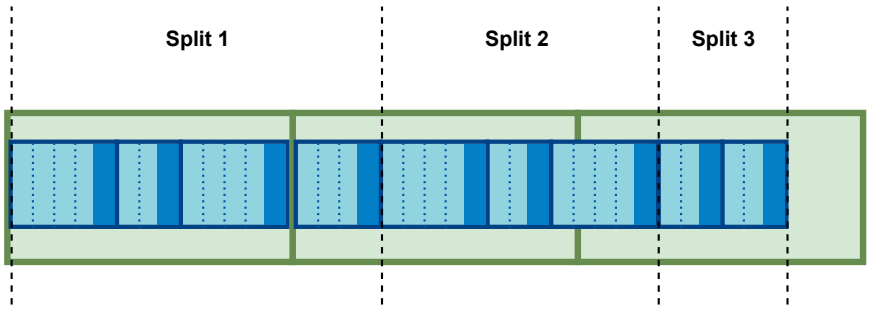


Figura 30: Ejemplo de splits generados por la clase VByteInputFormat.

de entrada. El término de este primer split ocurre en el segundo bloque, en el primer byte que tenga el bit más significativo en 1. Se utilizan operaciones a nivel de bits para identificar este byte. Por medio de un ciclo se identifican estas posiciones en cada bloque del archivo. Finalmente se agrega la posición del último byte del archivo binario.

Con esta lista de posiciones de bytes, que indican los límites de los splits, se generan los splits usando el constructor de la clase **FileSplit** que recibe como argumento la ubicación del archivo, la posición de inicio del split y el tamaño del split. El código fuente completo de esta clase se encuentra en [18].

La otra clase implementada para la lectura de estos registros fue **VByteRecordReader** que hereda de la clase *RecordReader*. Esta clase se encarga de leer los registros de cada split y enviarlos al Contexto para que sean recibidos por las tareas de Map. El método más importante de esta clase es **nextKeyValue** el cual lee los registros para generar el par clave-valor. En este caso la clave se estableció como *NullWritable* porque no se utiliza y el valor es del tipo *IntWritable*. La implementación se muestra en la figura 31.

La condición de la línea número 2 es para realizar la lectura del byte actual **pos** hasta antes de llegar al final **end**. El arreglo de bytes de la línea número 4 tiene tamaño **MAX RECORD SIZE**, que es un número obtenido de la configuración del método *main* de la aplicación. Este número indica el número máximo de bytes de los códigos de largo variable. En esta aplicación se utilizaron hasta 5 bytes. En el ciclo que inicia en la línea número 8 se leen los bytes del registro. La variable entera **b** corresponde a la lectura del byte actual, la cual se almacena como tipo de dato byte en el arreglo *bytes*, en la posición **length**. En la línea 12 se decodifica este byte a número entero en la variable **number** escribiendo los 7 bits menos significativos. En la condición de la línea número 14 se comprueba si el byte leído es el último del registro. Si es el último se establece el valor de **value**, que es del tipo IntWritable, a partir del valor de *number*. La condición de la línea número 18 es para asegurar de que el archivo de entrada está codificado correctamente y no hay registros más largos de los permitidos. En la línea número 22 se actualiza la posición de lectura y se retorna *true* para indicar que la lectura fue realizada.

```
1       public boolean nextKeyValue() {
2           if (pos < end) {
3               int length = 0;
4               byte[] bytes = new byte[MAX_RECORD_SIZE];
5               int number = 0;
6               int i = 0;
7
8               while (true) {
9                   int b = in.read();
10                  bytes[length] = (byte)b;
11                  length++;
12                  number |= (bytes[i] & 0x7F) << (7 * i);
13                  i++;
14                  if ((b & 0x80) != 0) {
15                      value.set(number);
16                      break;
17                  }
18                  if (length == MAX_RECORD_SIZE) {
19                      throw new IOException("Registro muy largo");
20                  }
21              }
22              pos = in.getPos();
23              return true;
24          }
25          return false;
26      }
27
```

Figura 31: Método *nextKeyValue* de la clase VByteRecordReader.

La implementación de la clase Mapper es la más sencilla de todas. Puesto que cada registro corresponde a una palabra codificada, simplemente se obtiene el valor del par clave-valor leído y se le asigna el número 1, como se observa en la figura 32. La implementación de la clase Reduce es idéntica a la utilizada en la aplicación CodeCountFixed, donde se suman la lista de valores correspondientes a cada clave.

La implementación del método main se muestra en la figura 33, donde en la línea número 3 se estable la configuración del máximo número de bytes de cada registro. En la línea número 11 se establece que la clase del formato de entrada será la clase implementada **VByteInputFormat**. En la línea número 12 se define que el formato de salida será del tipo texto, aunque esta configuración se puede omitir porque es la clase definida por defecto.

```
    public static class CodedVByteMapper extends
            Mapper<NullWritable, IntWritable, IntWritable, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        public void map(NullWritable key, IntWritable value,
                Context context) throws IOException, InterruptedException {

            context.write(value, one);
        }
    }

    public static class CodedVByteReducer extends
            Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(IntWritable key, Iterable<IntWritable> values,
                Context context) throws IOException, InterruptedException {

            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

```

Figura 32: Implementación de las classe Mapper y Reducer de CodeCountVariable.

```
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("max.vbyte.record.size", "5");
        Job job = Job.getInstance(conf, "CodeCountVariable");
        job.setJarByClass(CodeCountVariable.class);
        job.setMapperClass(CodedVByteMapper.class);
        job.setCombinerClass(CodedVByteReducer.class);
        job.setReducerClass(CodedVByteReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
        job.setInputFormatClass(VByteInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }

```

Figura 33: Implementación del método Main de CodeCountVariable.

4. Experimentos y resultados

4.1. Datasets utilizados

El framework Hadoop fue desarrollado para trabajar con grandes volúmenes de datos. Es necesario considerar esto al momento de elegir cuáles datasets se utilizan para realizar los experimentos en las distintas aplicaciones. A continuación se describen los tres datasets utilizados en las experimentaciones con las aplicaciones MapReduce implementadas.

4.1.1. English

El primer dataset utilizado fue el denominado *english*, obtenido desde el corpus **Pizza & Chili** [29]. Este dataset contiene textos en inglés de varias fuentes, pero también contiene caracteres Unicode, los cuales utilizan más de un byte para ser representados. Al realizar pruebas con la aplicación CodeCountFixed, usando la versión codificada de este dataset, se obtuvo un resultado que no era equivalente al entregado por la aplicación WordCount. Esta diferencia de resultados es producida por estos caracteres Unicode. Para evitar estos problemas se decidió eliminar estos caracteres del dataset, además de permitir solamente los caracteres de letras y números, además de los espacios, tabulaciones y saltos de línea. También se convirtieron a minúsculas todas las letras. De esta forma se generó un nuevo dataset que se denominó **English Limpio**. El tamaño de este dataset es de 2.126.798.185 bytes, aproximadamente 2 GiB. Este tamaño de dataset es apropiado para utilizarlo en Hadoop puesto que utiliza varios bloques (16) repartidos por el clúster.

Al ejecutar la aplicación WordCount con este dataset como entrada se obtuvieron 1.692.835 palabras distintas. Esta cantidad puede ser representada por 3 bytes, por esta razón la aplicación CodeCountFixed utiliza códigos de 3 bytes. El dataset *English Limpio* se codifica con 3 bytes para generar el dataset denominado **English Limpio Encoded**. El tamaño de este dataset es de 1.164.999.960 bytes, aproximadamente 1,1 GiB. Al momento de codificar este dataset también se genera el diccionario de decodificación, el cual es utilizado para decodificar el resultado entregado por CodeCountFixed y comprobar que sea el mismo resultado generado por WordCount. Este diccionario tiene tamaño 38.182.979 bytes, aproximadamente 36,4 MiB.

También se realizó la codificación de *English Limpio* usando códigos de 4 bytes, para ser ejecutados por la aplicación CodeCountFixed4B. Este dataset se denominó **English Limpio Encoded4B** y su tamaño es de 1.553.333.280, aproximadamente 1,4 GiB. El diccionario de decodificación generado para decodificar el resultado entregado por la ejecución de la aplicación CodeCountFixed4B es exactamente el mismo que el diccionario del dataset **English Limpio Encoded**.

Para utilizar este dataset con la aplicación CodeCountVariable se generaron los códigos de largo variable usando el algoritmo VByte y se obtuvo el dataset **English Limpio EncodedVByte**. Este dataset tiene tamaño de 588.758.131 bytes, aproximadamente 561,5 MiB. El

diccionario de decodificación para este dataset tiene el mismo tamaño que los diccionarios de los otros datasets codificados, 36,4 MiB. Sin embargo no son iguales. Aunque los tres diccionarios tienen pares clave-valor del tipo Integer-String, el diccionario de este dataset tiene claves enteras distintas.

En la figura 34 se muestra un gráfico donde se comparan de manera visual el tamaño de estos datasets, donde también se incluye el diccionario de decodificación. En cada barra de los datasets codificados se muestra el porcentaje de espacio utilizado en comparación al texto original.

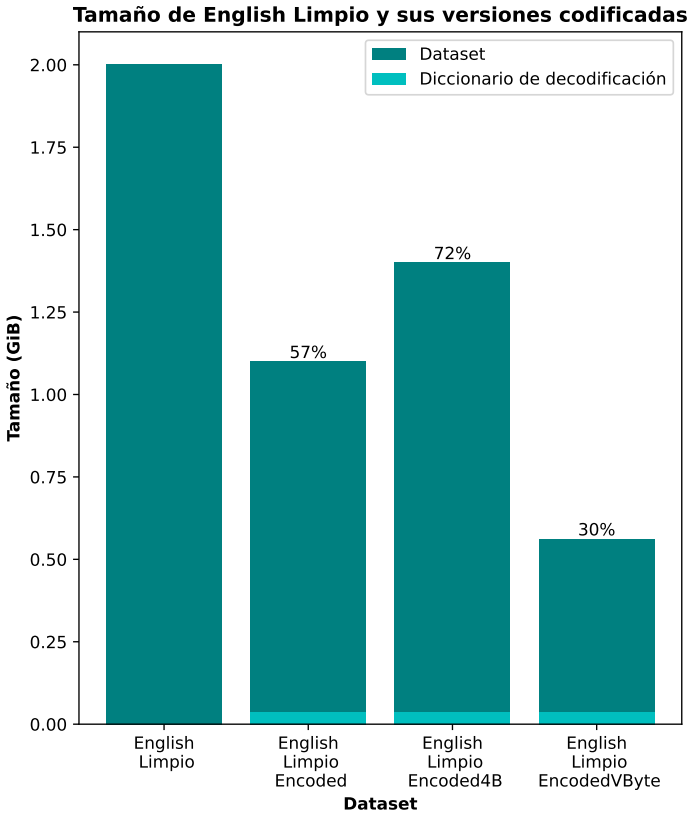


Figura 34: Gráfico de comparación de los tamaños de los datasets *English Limpio*.

En cada ejecución de las cuatro aplicaciones con sus correspondientes cuatro datasets se verificó que el resultado fuera equivalente. Es decir, se comprobó que los resultados decodificados fueran exactamente los mismos que los resultados entregados por la aplicación WordCount.

4.1.2. English Wikipedia 20 GiB

Este dataset fue obtenido de [30]. Se implementó un script para extraer los artículos de la estructura JSON y generar archivos de texto plano. Este script genera un artículo en cada línea de texto, con lo cual se obtuvieron líneas de texto muy largas. Se desarrolló un programa en C++ para quitarle los caracteres que no fueran letras ni números y convertir todas las letras a minúsculas, como se hizo con el dataset *english*. Además este programa permite establecer la cantidad de palabras por línea. En el capítulo 4.2 se describe la razón por la que se eligieron 20 palabras por línea de texto.

El dataset generado se denominó **EnWiki 20GiB** y tiene un tamaño de 21.581.379.460, aproximadamente 20,1 GiB. Fue el dataset más grande utilizado en los experimentos. La ejecución de la aplicación WordCount con este dataset entregó como resultado 30.113.341 palabras. Esta cantidad no puede ser representada en 3 bytes, por lo tanto no se puede codificar este dataset de manera que sea utilizado por la aplicación CodeCountFixed. Se realizaron dos codificaciones, una para la aplicación CodeCountFixed4B y otra para la aplicación CodeCountVariable.

El dataset codificado con códigos de largo fijo de 4 bytes, para ser utilizado por la aplicación CodeCountFixed4B, se denominó **EnWiki 20GiB Encoded4B** y tiene un tamaño de 13.322.060.880 bytes, aproximadamente 12,4 GiB. El diccionario de decodificación tiene un tamaño de 814.971.541 bytes, aproximadamente 777,2 MiB.

El dataset para la aplicación CodeCountVariable se denominó **EnWiki 20GiB EncodedVByte** y tiene un tamaño de 5.970.482.573 bytes, aproximadamente 5,6 GiB. El diccionario de decodificación tiene el mismo tamaño que el diccionario del dataset *EnWiki 20GiB Encoded4B*. En la figura 35 se muestra un gráfico donde se comparan de manera visual el tamaño de estos datasets.

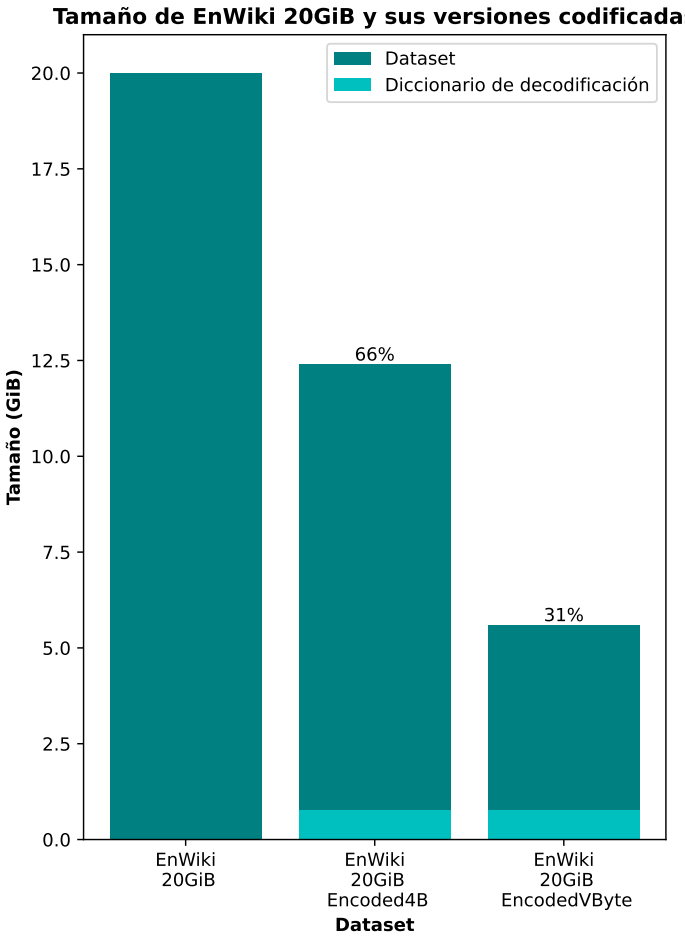


Figura 35: Gráfico de comparación de los tamaños de los datasets *EnWiki 20GiB*.

4.1.3. English Wikipedia 5 GiB

El último dataset utilizado es una versión reducida de *EnWiki 20GiB*, donde se utilizaron los primeros 5 GiB de este dataset para generar el dataset denominado **EnWiki 5GiB**. El tamaño de este dataset es de 5.368.709.120 bytes. La cantidad de palabras distintas que tiene este dataset es de 10.888.968, por lo que puede ser representada con 3 bytes.

Para la aplicación CodeCountFixed se generaron las palabras codificadas con 3 bytes. El dataset obtenido se denominó **EnWiki 5GiB Encoded**. Su tamaño es de 2.481.429.180 bytes, aproximadamente 2,3 GiB. El diccionario de decodificación tiene un tamaño de 278.188.293 bytes, aproximadamente 265,3 MiB.

El dataset generado para la aplicación CodeCountFixed4B tiene palabras codificadas con 4 bytes. Este dataset se denominó **EnWiki 5GiB Encoded4B**. Su tamaño es de 3.308.572.080 bytes, aproximadamente 3,1 GiB. El diccionario es idéntico al de *EnWiki 5GiB Encoded*.

Para la aplicación CodeCountVariable se creó el dataset con los códigos de largo variable.

Este dataset se nombra **EnWiki 5GiB EncodedVByte**. Su tamaño es de 1.482.667.626 bytes, aproximadamente 1,4 GiB. El diccionario de codificación tiene el mismo tamaño que los otros, aproximadamente 265,3 MiB. En la figura 36 se muestra un gráfico donde se comparan de manera visual el tamaño de estos datasets.

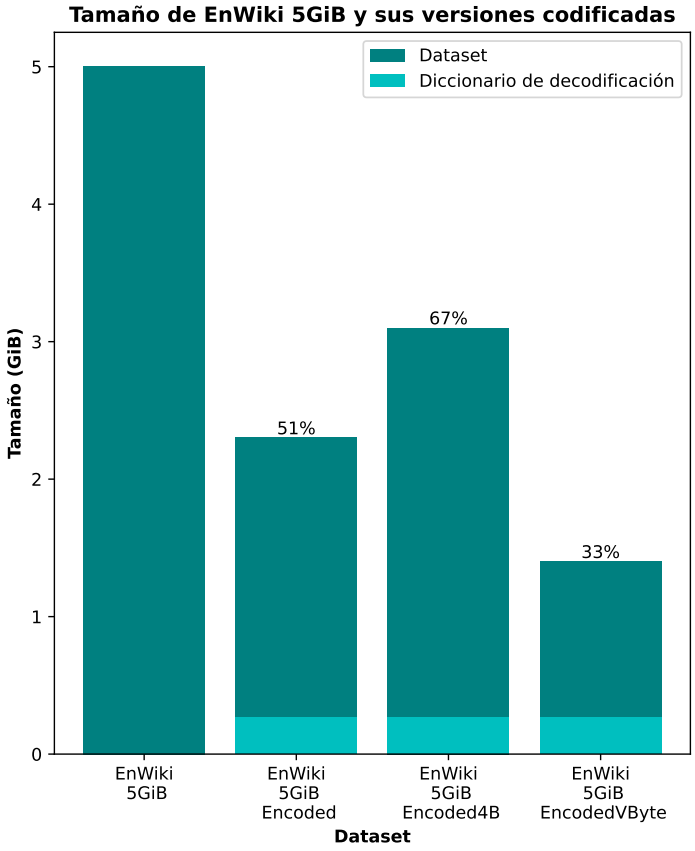


Figura 36: Gráfico de comparación de los tamaños de los datasets *EnWiki 5GiB*.

4.2. Ajustes previos

Antes de llevar a cabo los experimentos con el fin de obtener resultados sobre el rendimiento de las aplicaciones con los datasets mencionados anteriormente, se llevaron a cabo algunos ajustes tanto en los datasets como en las aplicaciones. Estos ajustes se realizaron con el objetivo de obtener resultados óptimos en cada aplicación.

4.2.1. Cantidad de palabras por línea de texto

La aplicación WordCount utiliza la clase TextInputFormat para leer los archivos de texto de entrada. Esta clase considera que cada registro está formado por una línea de texto. En la clase Mapper de WordCount esta línea de texto recibida es divida en palabras usando la clase StringTokenizer de java. Por lo cual es adecuado pensar en la pregunta: **¿Influye la cantidad de palabras por línea de texto en la ejecución de la aplicación WordCount?**.

Se creó un script en bash que recibe un archivo de texto de entrada y genera un archivo de texto de salida con la cantidad de palabras por línea que el usuario determine [18]. Se utilizó el dataset *English Limpio* para generar un nuevo dataset denominado **English 1G** el cual corresponde al primer gigabyte de este dataset. Se ejecutó el script para generar datasets que tuvieran 3, 5, 10, 20, 30, 40, 50 y 60 palabras por línea. Se realizaron 10 tests con cada uno de estos datasets y se promediaron los resultados, los cuales se muestran en la tabla 2.

La primera columna de la tabla, **Palabras por línea**, indica la cantidad de palabras por línea del dataset. La columna **Tareas de Map** muestra la cantidad de tareas de Map que se generaron al ejecutar la aplicación WordCount con cada dataset. El número de tareas de Map depende de la cantidad de splits. Generalmente la cantidad de splits y de bloques que componen los archivos de entrada es la misma. Debido a las características de hardware del clúster, es posible ejecutar 8 tareas de Map de manera simultánea. La columna **Tiempo total** indica el tiempo total de ejecución con el dataset de entrada correspondiente. Este tiempo no considera la generación de los splits ni la asignación de los containers para el trabajo. El tiempo comienza cuando se inicia la lectura del primero registro. La columna **Prom. Tareas de Map** corresponde al promedio del tiempo utilizado por las tareas de Map. **Prom. Tareas de Shuffle** indica el promedio de todas las tareas de *Shuffle and Sort*, que corresponde a la etapa intermedia entre Map y Reduce. La cantidad de estas tareas es la misma que las tareas de Map. Finalmente **Tiempo Tarea de Reduce** indica el tiempo utilizado por la única tarea de Reduce. Estos tiempos fueron entregados por Hadoop a través del HistoryServer.

Generalmente, el tiempo total no corresponde a la suma directa de los tiempos promedio de las tareas de Map, Shuffle and Sort y Reduce. Esto es porque las tareas se ejecutan de manera paralela en diferentes containers, no de manera secuencial. En todos los experimentos realizados con aplicaciones MapReduce, en esta memoria de título, siempre se ejecutó una sola tarea de Reduce.

Con estos resultados se responde a la pregunta **¿Influye la cantidad de palabras por línea de texto en la ejecución de la aplicación WordCount?:** Se observa que con 3

Palabras por línea	Tareas de Map	Tiempo total	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
3 palabras	8	109.5s	91.3s	16.3s	6.1s
5 palabras	8	105s	88.9s	14.5s	6s
10 palabras	8	103.7s	87.3s	15.3s	6.1s
20 palabras	8	103.7s	86.6s	20.7s	6s
30 palabras	8	103.5s	86.3s	22.4s	6s
40 palabras	8	103.7s	86.1s	21.2s	6s
50 palabras	8	103.3s	85.3s	22.7s	6.2s
60 palabras	8	102.7s	85.2s	22.6s	6s

Cuadro 2: Resultados de las pruebas de variación de cantidad de palabras por línea de texto.

palabras por línea el tiempo de ejecución es un poco mayor que el resto (aproximadamente un 5 %), pero no es demasiado para ser concluyente. Aunque el menor tiempo se obtuvo con el dataset de 60 palabras por línea, no es posible concluir que a mayor cantidad de palabras se tiene mejor rendimiento puesto que la diferencia entre los resultados es pequeño. Se requieren realizar mayores experimentos y con datasets más diversos para contestar esta pregunta de manera más concreta.

Luego de estos resultados se decidió utilizar la cantidad de 20 palabras por línea de texto en el dataset obtenido de la Wikipedia, porque a juicio del autor de esta memoria, es una cantidad razonable de palabras por línea. La cantidad de palabras por línea de texto en el dataset *English* no se modificó.

4.2.2. Variaciones de CodeCountFixed

La aplicación CodeCountFixed utiliza la clase FixedLengthInputFormat para el archivo binario codificado, donde cada palabra es representada con un número entero que se representa con 3 bytes. En la configuración de esta clase se debe definir el tamaño del registro que es entregado a las tareas de Map. Para simular líneas de texto se configuró para que cada registro tuviera la longitud de 60 bytes, es decir, en cada registro hay 20 códigos que deben ser separados por la clase Mapper. Se creó una versión alternativa de esta aplicación donde se lee un código a la vez. En este caso el registro es de 3 bytes. Esta versión se denominó **CodeCountFixed2**. Se realizó el experimento de comparar estas dos versiones para concluir cuál de las dos es más eficiente en tiempo de ejecución. El dataset utilizado fue la versión codificada de *EnWiki 5GiB*: **Enwiki 5GiB Encoded**. Este dataset fue utilizado en las dos versiones. Se realizaron 10 tests con cada variante y se promediaron los resultados, los cuales se muestran en el cuadro 3. La columna **Diferencia porcentual relativa** indica en cuánto mejora (valores negativos) o empeora (valores positivos) la versión de la aplicación respecto a la versión original.

Estos resultados muestran claramente que la versión CodeCountFixed2 es menos eficiente en cuanto al tiempo en comparación con la versión que lee 20 registros a la vez. Específicamente es un 16.6 % más lenta. Se puede asumir que la forma de leer los registros tiene un impacto signficativo en los tiempos de ejecución de las aplicaciones MapReduce, en las

Aplicación	Tareas de Map	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
CodeCountFixed	19	507.2s	-	145.6s	271.8s	59.2s
CodeCountFixed2	19	591.4s	+16.6 %	174.3s	338.4s	57.5s

Cuadro 3: Resultados del dataset *EnWiki 5GiB Encoded* en dos versiones de la aplicación CodeCountFixed

cuales se ven incrementados sus tiempos promedios de las tareas de Map y Shuffle.

En la aplicación CodeCountFixed, se puede modificar otro factor, que es la conversión de los 3 bytes que forman la palabra codificada a un número entero del tipo IntWritable, que tiene 4 bytes. Esta conversión se realiza en la clase Mapper de CodeCountFixed. Se consideró la opción de usar BytesWritable para evitar esta conversión. Con esta opción, la clase Mapper devuelve el par clave-valor con tipos de datos BytesWritable-IntWritable, y la clase Reducer recibe y devuelve estos mismos tipos de datos. Al evitar esta conversión de datos **¿Se mejoran los tiempos de ejecución?**.

Se implementó esta variante y se denominó **CodeCountFixedBytes**. Se realizaron experimentos con el dataset *English Limpio* versión codificada, **English Limpio Encoded**, para comparar este rendimiento con el de CodeCountFixed. Se realizaron 10 tests con cada variante y se promediaron los resultados, los cuales se muestran en el cuadro 4.

Aplicación	Tareas de Map	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
CodeCountFixed	9	227.3s	-	72.2s	125s	10.7s
CodeCountFixedBytes	9	231.9s	+2 %	74.6s	98.7s	11.6s

Cuadro 4: Resultados del dataset *English Limpio Encoded* en dos versiones de la aplicación CodeCountFixed

Los tiempos de ejecución son ligeramente mejores con la versión CodeCountFixed. Como la diferencia es muy pequeña (2 %) no se puede determinar que esta versión es más eficiente. Sería necesario hacer más experimentos, con distintos datasets, para determinar si efectivamente esta versión es mejor. Sin embargo, es posible concluir que el proceso de conversión de BytesWritable a IntWritable en la clase Mapper de CodeCountFixed no es un factor que perjudique la eficiencia de la aplicación. La versión definitiva de CodeCountFixed es la que realiza esta conversión a IntWritable, donde cada registro está compuesto por 20 códigos y con tamaño de 60 bytes.

4.2.3. Variaciones de CodeCountVariable

Se desarrollaron 4 variantes de CodeCountVariable con pequeñas modificaciones en cómo se realiza la decodificación de los registros que están codificados con el algoritmo VByte a número entero del tipo IntWritable:

- La versión **CodeCountVariableV1** realiza la decodificación a IntWritable en la clase Mapper. Esta clase recibe el par clave-valor del tipo NullWritable-BytesWritable y entrega el par del tipo IntWritable-IntWritable. La clase *RecordReader* lee los bytes correspondientes al registro y los entrega como tipo de dato BytesWritable.
- La versión **CodeCountVariableV2** no realiza decodificación a número entero. La clase Mapper recibe el par clave-valor del tipo NullWritable-BytesWritable y entrega los tipos de dato BytesWritable-IntWritable. La clase Reducer también entrega como resultado estos tipos de dato.
- La versión **CodeCountVariableV3** realiza la decodificación en la clase *RecordReader*. Primero lee todos los bytes del registro para luego decodificarlos y enviarlos al Contexto. La clase Mapper recibe los tipos de dato NullWritable-IntWritable.
- La versión **CodeCountVariableV4** también realiza la decodificación en la clase *RecordReader*. A diferencia de la versión 3, en este caso la decodificación se realiza byte a byte. Es decir, se lee un byte e inmediatamente se decodifica a número entero. Este proceso se realiza con todos los bytes que forman el registro.

Se utiliza el dataset *Enwiki 5GiB Encoded4B* para realizar estos experimentos. Cada aplicación se ejecuta 10 veces y se promedian los resultados, los cuales se muestran en el cuadro 5.

Aplicación	Tareas de Map	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
CodeCountVariableV1	12	668.1s	-	285.5s	290.2s	53.6s
CodeCountVariableV2	12	694.9s	+4 %	292.1s	300.4s	62.2s
CodeCountVariableV3	12	664.1s	-0.6 %	283.7s	286.6s	54.5s
CodeCountVariableV4	12	666.1s	-0.3 %	284.2s	288.4s	54s

Cuadro 5: Resultados del dataset *EnWiki 5GiB Encoded4B* en cuatro versiones de la aplicación CodeCountVariable

Es notable que la versión 2 sea más lenta en un 4.6 %, comparándola con la versión más rápida, a pesar de no realizar el proceso de decodificación del código VByte del tipo de dato BytesWritable a IntWritable. Aunque no es un porcentaje muy signficativo, se podría asumir que la comunicación de valores del tipo de dato BytesWritable es un poco más lenta que al realizarla con el tipo de dato IntWritable, pero que no afecta de manera importante. Entre las versiones 1, 3 y 4 no hay diferencias significativas de tiempo de ejecución. Siendo ligeramente mejor la versión 3 (un 0.6 % más rápida). El autor de esta memoria escogió la versión 4 como la versión definitiva para la aplicación CodeCountVariable porque consideró que la decodificación del registro se realizaba de manera más elegante en esta versión.

Todas estas versiones de las aplicaciones MapReduce implementadas, además de las versiones definitivas, se encuentran en [18].

4.3. Experimentos con el dataset *English Limpio*

Se realizaron experimentos con este dataset en cada una de las cuatro aplicaciones Map Reduce implementadas. El archivo de texto *English Limpio* se usó como entrada para la aplicación **WordCount**. El archivo binario donde cada palabra está codificada como un número entero de 3 bytes, *English Limpio Encoded*, se utilizó en la aplicación **CodeCountFixed**. El archivo binario con codificación de 4 bytes para cada palabra, *English Limpio Encoded4B* fue utilizado por la aplicación **CodeCountFixed4B**. El archivo binario con códigos de largo variable, *English Limpio EncondedVByte*, se utilizó con la aplicación **CodeCountVariable**. En cada experimento se realizaron 20 ejecuciones por cada aplicación y se promediaron los resultados de los tiempos de ejecución.

El primer conjunto de experimentos se realizó con la configuración de tamaño de bloque en el HDFS de 128 MiB, que es la configuración por defecto. Los resultados de los tiempos de ejecución se muestran en el cuadro 6.

Aplicación	Tamaño Dataset	Tareas de Map	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	2 GiB	16	253.4s	-	89.15s	139.25s	12.05s
CodeCountFixed	1.1 GiB	9	228.2s	-10 %	133.4s	83.9s	9.05s
CodeCountFixed4B	1.45 GiB	12	220.6s	-13 %	99.6s	97.35s	10.05s
CodeCountVariable	0.56 GiB	5	332.3s	+31 %	275.35s	188.55s	8.35s

Cuadro 6: Resultados del dataset *English Limpio* en las cuatro aplicaciones MapReduce con el tamaño de bloque por defecto.

El rendimiento de las aplicaciones que utilizan los datasets codificados se comparan con la aplicación WordCount que utiliza el archivo de texto plano. Es notable cómo la cantidad de tareas de Map depende del tamaño del dataset, ya que los bloques que componen el archivo tienen un tamaño de 128 MiB. El tiempo de ejecución de CodeCountFixed es 10 % menor que WordCount. CodeCountFixed4B es quien tiene mejor rendimiento de tiempo, al ser 13 % más rápido que WordCount. A pesar de que CodeCounVariable recibió el dataset de menor tamaño, utilizando solamente un 30 % del tamaño del dataset original, su tiempo total de ejecución es 31 % más lento que WordCount. El menor rendimiento de esta aplicación se debe, en parte, a que solamente se realizan 5 tareas de Map, ejecutadas por 5 containers de manera paralela. Los recursos de hardware del clúster están configurados para tener 9 containers, de los cuales 8 pueden realizar tareas de manera paralela (un container se define como Application Master y se encarga de gestionar al resto). Por lo cual no se utilizan todos los recursos al momento de ejecutar la aplicación CodeCounVariable. Para poder comparar las aplicaciones de tal manera que se utilicen todos los recursos del clúster se diseñaron los siguientes experimentos, en los cuales se establece una cantidad de bloques iguales para cada dataset, para tratar de lograr que cada tarea de Map en cada una de las aplicaciones, reciba la misma cantidad de datos de entrada. Esto se logra cambiando el tamaño del bloque en la configuración del HDFS. Se probaron con 9, 18, 27 y 36 bloques por cada dataset.

En el cuadro 7 se muestran los resultados obtenidos al normalizar la cantidad de bloques en 9.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	225.4 MiB	273.8s	-	144.65s	123.65s	10.15s
CodeCountFixed	123.5 MiB	255.1s	-6.8 %	133.45s	114.75s	9.15s
CodeCountFixed4B	164.6 MiB	249.4s	-8.9 %	128.75s	111.95s	9.4s
CodeCountVariable	62.4 MiB	323.15s	+18 %	167.55s	147.7s	9.9s

Cuadro 7: Resultados del dataset *English Limpio* en las cuatro aplicaciones MapReduce normalizadas a 9 bloques.

En los datasets más grandes se utilizan tamaños de bloques mayores. En todas las ejecuciones se tuvieron 9 tareas de Map, 9 tareas de Shuffle and Sort y 1 tarea de Reduce. Al realizar la normalización de bloques se tiene que la aplicación CodeCountFixed4B sigue siendo más rápida que WordCount en un 8.9 %. La aplicación CodeCountFixed también es más rápida que WordCount, en aproximadamente un 6.8 %. Al igual que el experimento anterior, la aplicación CodeCountVariable sigue siendo más lenta que WordCount en un 18 %. Se mantiene la tendencia del experimento anterior, pero los porcentajes de diferencias de los tiempos disminuyeron.

El siguiente experimento se realizó al establecer la cantidad de bloques de cada dataset en 18. Los resultados se muestran en el cuadro 8.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	112.7 MiB	251.65s	-	79.95s	128.65s	12.25s
CodeCountFixed	61.7 MiB	233.05s	-7.3 %	74s	133.55s	10.95s
CodeCountFixed4B	82.3 MiB	225.05s	-10.5 %	71.35s	114.45s	11.5s
CodeCountVariable	31.2 MiB	283.95s	+12.8 %	91s	165.35s	11.4s

Cuadro 8: Resultados del dataset *English Limpio* en las cuatro aplicaciones MapReduce normalizadas a 18 bloques.

Con esta configuración, donde cada dataset se divide en 18 bloques, se obtuvieron los mejores tiempos de ejecución en las aplicaciones de WordCount y CodeCountVariable. También se mantiene la tendencia de los rendimientos de las aplicaciones, donde CodeCountFixed4B es la más rápida, seguida de CodeCountFixed. La aplicación más lenta sigue siendo CodeCountVariable.

En los cuadros 9 y 10 se observan los resultados de ejecutar las aplicaciones con cantidad de 27 y 36 bloques respectivamente. En ambos casos se mantienen las tendencias. Se destaca que con la configuración de 36 bloques se obtienen los peores rendimientos para las aplicaciones WordCount, CodeCountFixed y CodeCountFixed4B.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	75.1 MiB	270.2s	-	59.7s	148.6s	13.95s
CodeCountFixed	41.2 MiB	248.4s	-8 %	55.1s	135.55s	12.35s
CodeCountFixed4B	54.9 MiB	241.3s	-11 %	53.35s	131.2s	12.3s
CodeCountVariable	20.8 MiB	293.4s	+8.6 %	66.35s	166.4s	13.7s

Cuadro 9: Resultados del dataset *English Limpio* en las cuatro aplicaciones MapReduce normalizadas a 27 bloques.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	56.3 MiB	288.65s		48.5s	168.3s	18.5s
CodeCountFixed	30.9 MiB	264.35s	-8.4 %	44.9s	152.75s	15.95s
CodeCountFixed4B	41.2 MiB	250.8s	-13 %	42.1s	143.95s	16.1s
CodeCountVariable	15.6 MiB	300.6s	+4.1 %	52.4s	175.55s	16.25s

Cuadro 10: Resultados del dataset *English Limpio* en las cuatro aplicaciones MapReduce normalizadas a 36 bloques.

En el gráfico de la figura 37 se muestra el resultado de estos cinco conjuntos de experimentos, donde en el eje x se tiene el tamaño del dataset utilizado y en el eje y. El tiempo promedio de las 20 ejecuciones de cada aplicación. Los colores representan a las aplicaciones y las formas representan la configuración de cantidad de bloques, donde *Default Block Size* es la configuración con tamaño de bloque de 128 MiB y cada dataset varía en cantidad de bloques. Esto es por sus tamaños distintos. La variación entre los mejores y peores tiempos en cada aplicación es aproximadamente 15 %. Es notable también que los mejores tiempos de las aplicaciones WordCount y CodeCountVariable se consiguen con los datasets que utilizan 18 bloques y que los mejores tiempos de CodeCountFixed y CodeCountFixed4B se obtienen con la configuración de tamaño de bloque por defecto de 128 MiB, seguida de cerca por la configuración de 18 bloques.

En resumen, la configuración de bloques puede afectar significativamente el rendimiento de las aplicaciones. Si se quiere optimizar el tiempo se recomienda utilizar la aplicación CodeCountFixed4B. Si lo necesario a optimizar es el espacio, la aplicación CodeCountVariable es la mejor opción. CodeCountFixed ofrece un menor uso de espacio que CodeCountFixed4B a cambio de un pequeño aumento del tiempo de ejecución.

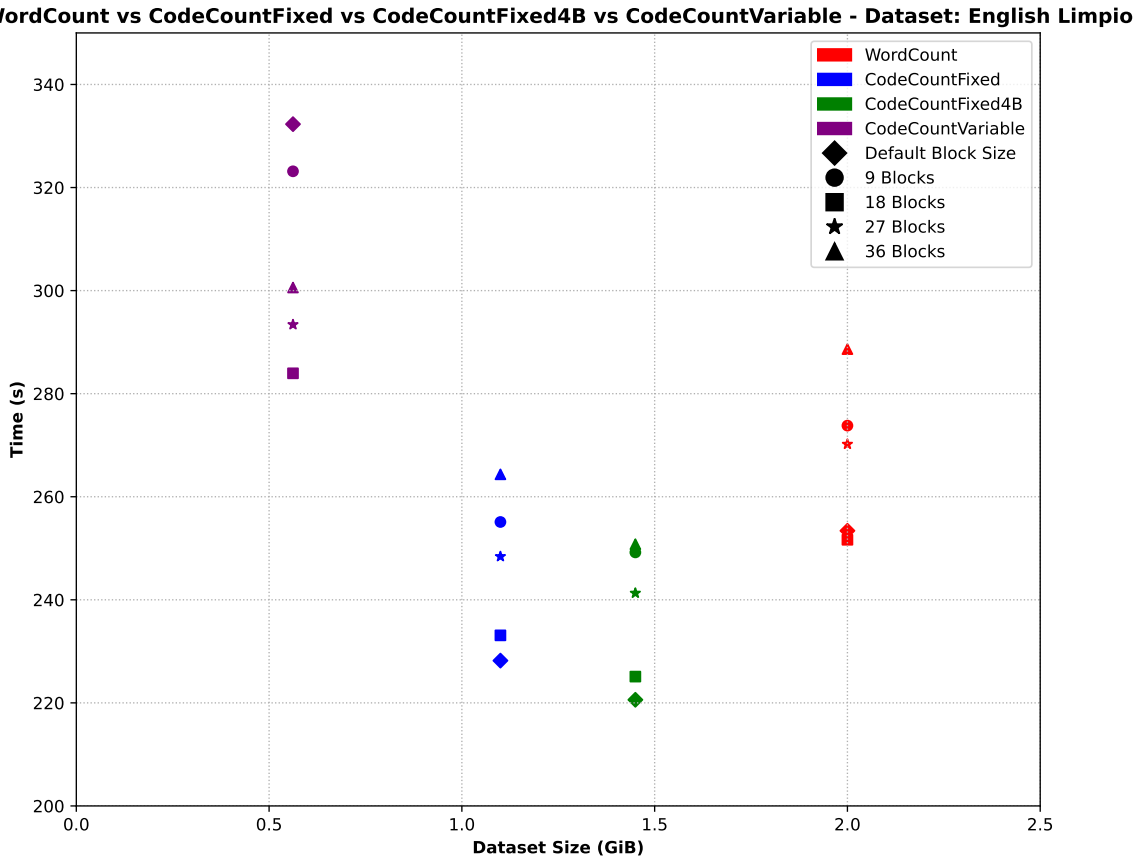


Figura 37: Gráfico de comparación de los tiempos de ejecución de las 4 aplicaciones MapReduce con el dataset *English Limpio*.

4.4. Experimentos con el dataset *EnWiki 5GiB*

En esta serie de experimentos se utilizó el dataset **EnWiki 5GiB** que tiene un tamaño mayor al dataset anterior. La cantidad de palabras distintas también es superior. Se utilizaron las versiones codificadas de este dataset con sus correspondientes aplicaciones. El primer experimento realizado fue con la configuración de bloque por defecto. Los resultados de rendimiento se muestran en el cuadro 11.

Aplicación	Tamaño Dataset	Tareas de Map	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	5 GiB	40	580.45s	-	87.85s	335.4s	52.3s
CodeCountFixed	2.3 GiB	19	512.05s	-11.7 %	147.7s	276.15s	56.85s
CodeCountFixed4B	3.1 GiB	25	498.9s	-14 %	113.5s	244.8s	61s
CodeCountVariable	1.4 GiB	12	672.8s	+16 %	287.55s	292.35s	52.65s

Cuadro 11: Resultados del dataset *EnWiki 5GiB* en las cuatro aplicaciones MapReduce con el tamaño de bloque por defecto.

Una vez más la aplicación CodeCountFixed4B es la más rápida, superando en un 14 % a WordCount. La aplicación CodeCountVariable es la más lenta, utilizando 16 % más de tiempo que WordCount para completar la ejecución del trabajo. La aplicación CodeCountFixed es más rápida que WordCount en un 11.7 %. Se observa cómo el promedio de las tareas de Map de WordCount es el menor y su promedio de las tareas de Shuffle and Sort es el mayor.

Al igual que se hizo con el dataset anterior, se normalizó la cantidad de bloques para cada dataset. Los resultados de configurar el tamaño de bloque de manera que el dataset quedara dividido en 9, se observan en el cuadro 12.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	568.9 MiB	668.55s	-	335.3s	290.75s	55.05s
CodeCountFixed	262.9 MiB	606.25s	-9.3 %	302s	264.8s	48.8s
CodeCountFixed4B	350.6 MiB	590s	-11.7 %	289.7s	257.8s	49.1s
CodeCountVariable	157.1 MiB	778.3s	+16.4 %	384.8s	341.95s	49.95s

Cuadro 12: Resultados del dataset *EnWiki 5GiB* en las cuatro aplicaciones MapReduce normalizadas a 9 bloques.

Con esta configuración se mantiene la tendencia. La aplicación CodeCountFixed4B es más rápida que WordCount en aproximadamente 11.7 %. CodeCountFixed también es más rápida que WordCount en un 9.3 %. CodeCountVariable es más lenta que WordCount en un 16.4 %.

Los resultados de aplicar la configuración de 18 bloques por cada dataset se muestra en el cuadro 13

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	284.4 MiB	560.4s	-	173.3s	321.8s	41.25s
CodeCountFixed	131.5 MiB	521.2s	-7 %	155.35s	294.4s	56.2s
CodeCountFixed4B	175.3 MiB	502.45s	-10.3 %	150.7s	281.35s	55.8s
CodeCountVariable	78.6 MiB	642.8s	+14.7 %	195.65s	369.15s	60.05s

Cuadro 13: Resultados del dataset *EnWiki 5GiB* en las cuatro aplicaciones MapReduce normalizadas a 18 bloques.

En estos resultados ocurre algo similar a los obtenidos con el dataset *English Limpio*. Con los datasets divididos en 18 bloques se obtienen los mejores resultados de WordCount y CodeCountVariable. Sin embargo, los mejores resultados entregados por CodeCountFixed y CodeCountFixed4B se obtienen con la configuración de bloque por defecto (128 MiB). En estas dos aplicaciones se utiliza la clase FixedLengthInputFormat para leer los códigos de largo fijo, ¿es la razón de que tengan mejor rendimiento al utilizar el tamaño de bloque por

defecto?. No es posible responder con certeza a esta pregunta. Para intentar responderla se requiere el diseño y ejecución de otros experimentos.

Los resultados de los experimentos realizados con las configuraciones de cantidades de 27 y 36 bloques se observan en los cuadros 14 y 15.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	189.6 MiB	564.85s	-	122.45s	347.7s	45.65s
CodeCountFixed	87.6 MiB	527.65s	-7 %	110.35s	283.45s	62.75s
CodeCountFixed4B	116.9 MiB	509.05s	-10 %	106.2s	268.9s	61.8s
CodeCountVariable	52.4 MiB	637.1s	+13 %	136.5s	366.35s	68.1s

Cuadro 14: Resultados del dataset *EnWiki 5GiB* en las cuatro aplicaciones MapReduce normalizadas a 27 bloques.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	142.2 MiB	575.45s	-	97.6s	332.3s	50.75s
CodeCountFixed	65.7 MiB	532.05s	-7.5 %	86.75s	290.7s	66.6s
CodeCountFixed4B	87.7 MiB	518.2s	-10 %	84.1s	281.35s	68.35s
CodeCountVariable	39.3 MiB	636.1s	+10.5 %	106.85s	351.95s	72.4s

Cuadro 15: Resultados del dataset *EnWiki 5GiB* en las cuatro aplicaciones MapReduce normalizadas a 36 bloques.

En estos resultados se destaca que la aplicación CodeCountVariable tiene el mejor rendimiento con 36 bloques. Pero la diferencia entre los resultados con 18, 27 y 36 bloques es cercana al 1 %, lo cual no es significativo.

El gráfico comparativo de estos cinco conjuntos de experimentos se observa en la figura 38. Aquí se observa claramente cómo la configuración de 9 bloques por dataset es la que tiene los peores tiempos en todas las aplicaciones. También destaca cómo la configuración de tamaño de bloque por defecto y de 18, 27, 36 bloques por dataset dan resultados similares en todas las aplicaciones, excepto en CodeCountVariable, donde el tamaño de bloque por defecto tiene un menor rendimiento. Comparando con el gráfico del dataset *English Limpio* (figura 37) se observan las tendencias de mejores tiempos para la aplicación CodeCountFixed4B y de peores tiempos para CodeCountVariable.

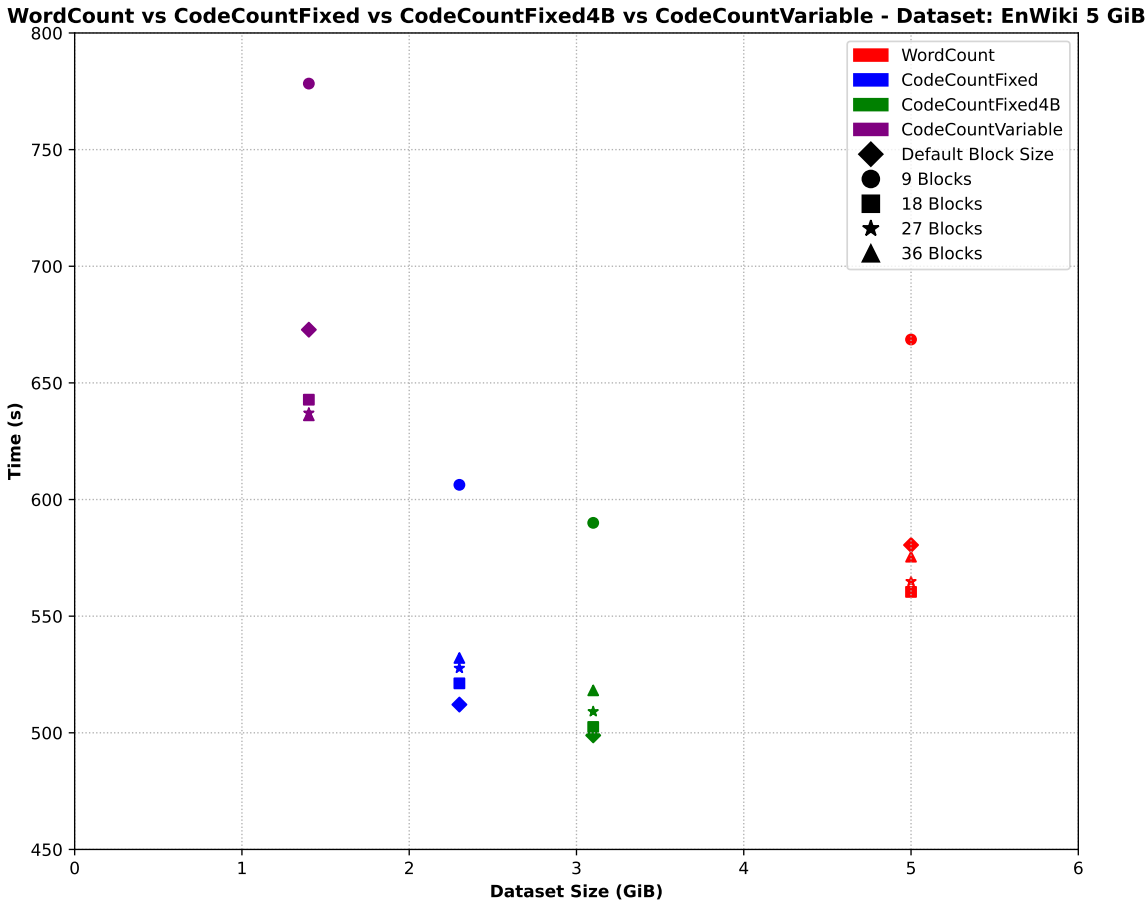


Figura 38: Gráfico de comparación de los tiempos de ejecución de las 4 aplicaciones MapReduce con el dataset *EnWiki 5GiB*.

4.5. Experimentos con el dataset *EnWiki 20GiB*

La última serie de experimentos se realizó con el dataset *EnWiki 20GiB* y sus versiones codificadas. No se utiliza la aplicación CodeCountFixed debido a que no es posible codificar la cantidad de palabras distintas de este dataset con 3 bytes. Los resultados de los tiempos de ejecución de los experimentos realizados con la configuración de tamaño de bloque por defecto se muestra en el cuadro 16.

En estos resultados se puede notar cómo, por primera vez, CodeCountVariable es más rápida que WordCount, aproximadamente en un 8 %. Siguiendo la tendencia de los experimentos anteriores, CodeCountFixed4B es la aplicación más rápida, superando a WordCount en un 18 %. En WordCount se tienen los promedios de la ejecución de las tareas de Map con menor tiempo y de las tareas de Shuffle con mayor tiempo, del mismo modo que se observa en los datasets anteriores, cuando la configuración del tamaño del bloque es por defecto. Esto se puede observar en los cuadros 6 y 11.

Aplicación	Tamaño Dataset	Tareas de Map	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	20.1 GiB	161	2253.75s	-	88.65s	1547.15s	193.9s
CodeCountFixed4B	12.4 GiB	100	1845s	-18 %	114.5s	1222.15s	176.2s
CodeCountVariable	5.6 GiB	45	2074s	-8 %	304.5s	1292.45s	134.15s

Cuadro 16: Resultados del dataset *EnWiki 20GiB* en las cuatro aplicaciones MapReduce con el tamaño de bloque por defecto.

También se realizó un experimento de normalización de bloques. Esta vez se configuraron 100 bloques por cada dataset. Los resultados se muestran en el cuadro 17.

Aplicación	Tamaño Bloque	Tiempo total	Diferencia porcentual relativa	Prom. Tareas de Map	Prom. Tareas de Shuffle	Tiempo Tarea de Reduce
WordCount	205.8 MiB	2125.85s	-	132.65s	1434.35s	191.8s
CodeCountFixed4B	128 MiB	1845s	-13.2 %	114.5s	1222.15s	176.2s
CodeCountVariable	56.9 MiB	2302.25s	+8.3 %	147.35s	1554.55s	176.15s

Cuadro 17: Resultados del dataset *EnWiki 20GiB* en las cuatro aplicaciones MapReduce normalizadas a 100 bloques.

Una vez más CodeCountFixed4B sigue siendo la aplicación más rápida, superando a WordCount en un 13.2 %. CodeCountVariable es más lenta que WordCount, aproximadamente en un 8.3 %. En la figura 39 se muestran los dos conjuntos de experimentos.

En el cuadro 18 se muestra un resumen de los mejores resultados para cada aplicación en cada uno de los tres datasets. La columna **Tiempo** indica el tiempo total de ejecución de la aplicación. La columna **Dif.** corresponde a la diferencia porcentual de tiempo respecto a la aplicación WordCount, donde los valores negativos indican menores tiempos de ejecución. La columna **Conf.** indica la configuración de bloques con la cuál se obtuvo el mejor tiempo para cada aplicación en cada dataset.

	Dataset								
	English Limpio			EnWiki 5GiB			EnWiki 20 GiB		
Aplicación	Tiempo	Dif.	Conf.	Tiempo	Dif.	Conf.	Tiempo	Dif.	Conf.
WordCount	251.65s	-	18 Bloques	560.4s	-	18 Bloques	2125.85s	-	100 Bloques
CodeCountFixed	228.2s	-9.3 %	Por defecto	512.05s	-8.6 %	Por defecto	-	-	-
CodeCountFixed4B	220.6s	-12.3 %	Por defecto	498.9s	-11 %	Por defecto	1845s	-13.2 %	Por defecto
CodeCountVariable	283.95s	+12.8 %	18 Bloques	636.1s	+13.5 %	36 Bloques	2074s	-2.4 %	Por defecto

Cuadro 18: Resumen de los resultados con los mejores tiempos de cada aplicación en cada dataset.

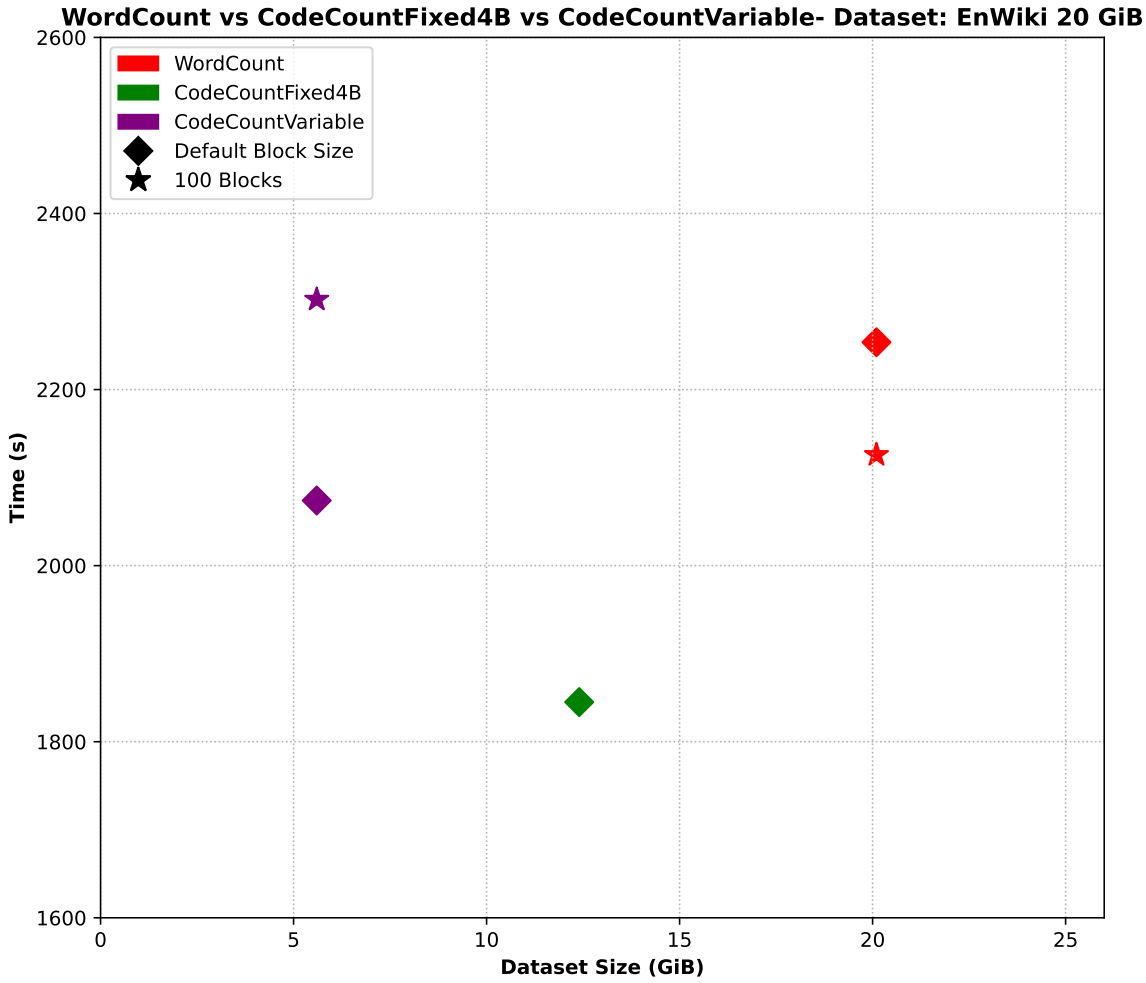


Figura 39: Gráfico de comparación de los tiempos de ejecución de las 3 aplicaciones MapReduce con el dataset *EnWiki 20GiB*.

5. Conclusiones y trabajo futuro

La utilización de los códigos de largo fijo permitió la reducción de los datasets originales de texto plano y también un menor tiempo de ejecución al contar la cantidad de palabras codificadas distintas, ofreciendo un equilibrio entre el uso de almacenamiento y tiempo de ejecución. En todos los casos en que se utilizaron las aplicaciones CodeCountFixed y CodeCountFixed4B, se obtuvieron mejores resultados con esta última. Esto podría ocurrir por la conversión que se realiza en la clase Mapper de CodeCountFixed para pasar de un BytesWritable de 3 bytes a un IntWritable de 4 bytes. Aunque esta diferencia no es demasiado significativa en los resultados de los datasets *English Limpio* y *EnWiki 5GiB*.

La aplicación CodeCountVariable presentó el menor rendimiento en comparación con las otras aplicaciones. Aunque se utilizaron datasets de menor tamaño, el uso de códigos de longitud variable afectó su lectura en la clase RecordReader. Es importante recordar que la

clase VByteRecordReader, encargada de leer los datos de entrada codificados con longitud variable, lee cada registro que contiene un único código. A diferencia de las otras aplicaciones, donde WordCount lee una línea de texto como registro (la cual generalmente contiene varias palabras) y las aplicaciones CodeCountFixed y CodeCountFixed4B leen 20 códigos por registro.

En el experimento con el dataset más grande, se observó que CodeCountVariable superó a WordCount por primera y única vez. Se puede asumir que CodeCountVariable aprovechó la ventaja de utilizar menos tareas de Map que WordCount, que utilizó 2.6 veces más de estas tareas para este dataset.

Con estos experimentos se pudo evidenciar el comportamiento de las aplicaciones que utilizan palabras codificadas, ya sea con códigos de largo fijo o variable. Se obtuvo un buen resultado al usar códigos de largo fijo, superando los tiempos de ejecución y almacenamiento requeridos por la aplicación base WordCount. El uso de códigos de largo variable permite una gran disminución del espacio de almacenamiento, pero con la aplicación desarrollada no fue posible conseguir mejores tiempos de ejecución con los dos datasets más pequeños.

Hay varios aspectos en los que este trabajo puede seguir desarrollándose, ya sea modificando las aplicaciones existentes o creando nuevas aplicaciones. Por ejemplo:

- Se requieren más experimentos para determinar de manera certera si la cantidad de palabras por línea de texto tiene influencia en el rendimiento de la aplicación WordCount. Para esto, se pueden utilizar datasets diversos y de mayor tamaño a los cuales se les modifique la cantidad de palabras por línea. Con los resultados actuales no parece que la cantidad de palabras por línea sea un factor demasiado importante. Sin embargo, estos resultados se obtuvieron con un solo dataset de tamaño modesto.

- Usando las aplicaciones que se desarrollaron es necesario realizar más experimentos para comprender la razón de que al usar códigos de largo fijo con 4 bytes se obtuvo mejor rendimiento que usando códigos de largo fijo de 3 bytes. Habría que determinar si realmente la conversión de BytesWritable de 3 bytes a IntWritable de 4 bytes, que se realiza en la aplicación CodeCountFixed, provoca un menor rendimiento. Esto se podría hacer modificando la aplicación CodeCountFixed4B para que realice el mismo procedimiento de conversión. Luego se realizarían los experimentos pertinentes para obtener las conclusiones.

- Los registros de la aplicación CodeCountVariable contienen un solo código, a diferencia de las otras aplicaciones donde cada registro contiene 20 códigos. Es posible modificar las aplicaciones que utilizan códigos de longitud fija para que contengan un solo código en cada registro. Además, es posible modificar los conjuntos de datos utilizados por WordCount para que cada palabra esté en una línea separada. De esta manera, se pueden realizar experimentos con las cuatro aplicaciones que tendrán normalizada la cantidad de códigos por registro.

- El bajo rendimiento de CodeCountVariable se podría solucionar modificando la clase de lectura de los archivos codificados, VByteRecordReader, para permitir que un registro contenga varios códigos de largo variable. Se podría modificar el programa que realiza la codificación para colocar un marcador cada cierta cantidad de códigos. Este marcador puede ser un byte especial que sea identificado por VByteRecordReader. Si se logran generar registros que contengan varios códigos es posible que los tiempos de ejecución de CodeCountVariable mejoren considerablemente.

- La realización de experimentos modificando los recursos disponibles en el clúster. Se puede utilizar la aplicación *yarn-utils* [26] para modificar la cantidad de recursos de hardware disponible en el clúster y así simular máquinas con menos memoria o menos núcleos de CPU disponibles. También se pueden modificar la cantidad de nodos disponibles en el clúster. Además se puede limitar la velocidad de la red para observar cuanto influye este aspecto en la ejecución de las aplicaciones MapReduce.

- En este trabajo se usaron dos datasets de textos en el idioma inglés y que fueron modificados para solamente conservar letras y números, además de convertir todas las letras a minúsculas. Se podría buscar la manera de superar el problema de codificar aquellas palabras que contienen caracteres Unicode. De esta forma se pueden utilizar los datasets sin modificar y también permitiría el uso de datasets de textos en otros idiomas.

- La decodificación de los resultados entregados por las aplicaciones CodeCountFixed, CodeCountFixed4B y CodeCountVariable no formaron parte de estas aplicaciones. Se podrían modificar estas aplicaciones para que, una vez obtenido el resultado de la cantidad de palabras codificadas, se puedan decodificar usando los diccionarios de decodificación. De esta forma se tendrían resultados idénticos a los entregados por WordCount. También se podría determinar si la decodificación agrega una sobrecarga importante a las aplicaciones originales, considerando que el archivo de salida es muy pequeño en comparación al archivo de entrada. La implementación de este aspecto sería un gran avance en acercar este estudio a un nivel más práctico y usable en aplicaciones de producción.

- Se pueden crear nuevas aplicaciones para leer otros tipos de datos, por ejemplo, datos tabulares. Apache Parquet [31] ofrece un formato de almacenamiento para datos estructurados como columnas y forma parte del ecosistema de Hadoop. Se necesitaría estudiar el funcionamiento de este componente para su implementación en las aplicaciones MapReduce y tratar de desarrollar una manera compacta de trabajar con los tipos de datos tabulares.

El uso de datos codificados en el ambiente de procesamiento distribuido Hadoop puede ser una alternativa viable para reducir el espacio de almacenamiento y también los tiempos de ejecución de las aplicaciones MapReduce. Para lograr esto se requiere seguir investigando y trabajando en los puntos mencionados anteriormente y así, en un futuro no muy lejano, se usen los recursos computacionales de manera más eficiente.

Referencias

[1] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 20–26, 2016.

[2] N. Mor, "Research for practice: Edge computing," *Commun. ACM*, vol. 62, p. 95, mar 2019.

[3] G. Navarro, *Compact Data Structures – A practical approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 536 pages.

[4] Z. Li, D. Seco, and J. Fuentes-Sepúlveda, "When edge computing meets compact data structures," in *2021 IEEE Cloud Summit (Cloud Summit)*, pp. 29–34, 2021.

[5] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[6] J. Plaisance, N. Kurz, and D. Lemire, "Vectorized vbyte decoding," *CoRR*, vol. abs/1503.07387, 2015.

[7] C. D. Manning, P. Raghavan, and S. Hinrich, *Index compression*. Cambridge University Press, 2019.

[8] Apache, "Apache hadoop." https://hadoop.apache.org/.

[9] Apache, "Apache hbase." https://hbase.apache.org/.

[10] Apache, "Apache hive." https://hive.apache.org/.

[11] Apache, "Apache spark." https://spark.apache.org/.

[12] Apache-Hadoop, "Hdfs architecture." https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[13] Apache-Hadoop, "Yarn architecture." https://hadoop.apache.org/docs/r3.3.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[14] Apache-Hadoop, "Writable interface." https://hadoop.apache.org/docs/r3.3.1/api/org/apache/hadoop/io/Writable.html.

[15] A. Holmes, *Hadoop in practice*. Manning Publications Co., 2015.

[16] Apache-Hadoop, "Fileinputformat." https://hadoop.apache.org/docs/r3.3.1/api/org/apache/hadoop/mapred/FileInputFormat.html.

[17] Apache-Hadoop, "Mapreduce tutorial." https://hadoop.apache.org/docs/r3.3.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[18] L. Aravena, "Wordcount with codes." https://github.com/ldaravena/wordcount-with-codes.

[19] Oracle, "Stringtokenizer class." https://docs.oracle.com/javase/8/docs/api/java/util/StringTokenizer.html/.

[20] Raspberry-Pi, "Raspberry pi 4 model b." https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[21] Ubuntu, "Ubuntu server arm." https://ubuntu.com/download/server/arm.

[22] Raspberry-Pi, "Raspberry pi imager." https://www.raspberrypi.com/news/raspberry-pi-imager-imaging-utility/.

[23] Oracle, "Openjdk 8 arm64." https://packages.ubuntu.com/source/jammy/arm64/openjdk-8.

[24] Apache-Hadoop, "Apache hadoop 3.3.1." https://archive.apache.org/dist/hadoop/common/hadoop-3.3.1/.

[25] Cloudera, "Determine yarn and mapreduce memory configuration settings." https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.0.9.0/bk_installing_manually_book/content/rpm-chap1-11.html.

[26] M. Konar, "Ambari yarn utils." https://github.com/mahadevkonar/ambari-yarn-utils.

[27] V. Bochkarev, A. Shevlyakova, and V. Solovyev, "Average word length dynamics as indicator of cultural changes in society," *Social Evolution and History*, vol. 14, pp. 153–175, 08 2012.

[28] M. Arias, J. Fernandez, M. Martinez-Prado, and A. Andres, "Java implementation of variable-byte encoding for integers." https://github.com/rdfhdt/hdt-java/blob/master/hdt-java-core/src/main/java/org/rdfhdt/hdt/compact/integer/VByte.java.

[29] P. Ferragina and G. Navarro, "Pizza & chili." http://pizzachili.dcc.uchile.cl/.

[30] D. Shapiro, "Plain text wikipedia 2020-11." https://www.kaggle.com/datasets/ad5e75c9f4643f587ab945243c5b46381eb1698d628b7d1815b75010615c5c0f.

[31] Apache, "Apache parquet." https://parquet.apache.org/.

A. Glosario

- **Aplicación:** Ver **MapReduce**.
- **Bloque:** División básica de los archivos en Hadoop. Cada archivo se divide en bloques y se almacenan en diferentes nodos del clúster. El tamaño por defecto del bloque es de 128 MiB.
- **Clúster:** Conjunto de computadores interconectados que tienen características de hardware similares. Trabajan juntos como si fueran una sola entidad computacional.
- **Container:** Ver **Contenedor**.
- **Contenedor:** Unidad de recursos computacionales (CPU, RAM, red y disco) que se asigna a un trabajo específico dentro de Hadoop. Cada contenedor trabaja de manera independiente.
- **Context:** Ver **Contexto**.
- **Contexto:** Objeto proporcionado por MapReduce que permite la comunicación de los datos entre las distintas etapas de la ejecución de una aplicación.
- **Framework:** Conjunto de herramientas, bibliotecas, convenciones y patrones de diseño que se utilizan para desarrollar aplicaciones de software de manera más eficiente y efectiva.
- **Hadoop:** Framework de software de código abierto utilizado para almacenar y procesar grandes conjuntos de datos en clústeres de computadores distribuidos.
- **HDFS:** *Hadoop Distributed File System* es el sistema de archivos distribuidos de Hadoop. Administra los archivos (divididos en bloques) ubicados en distintas máquinas que componen el clúster.
- **Job:** Ver **Trabajo**.
- **Map:** Etapa de una aplicación MapReduce que procesa los datos de entrada (registros) en paralelo y produce una lista de pares clave-valor intermedios.
- **MapReduce:** Aplicación MapReduce es un modelo de programación utilizado para procesar grandes conjuntos de datos de manera distribuida.
- **Nodo:** Computador individual que forma parte del clúster y que está conectado a otros computadores similares mediante una red.
- **Partición:** Ver **Split**.
- **Record:** Ver **Registro**.

- **Reduce:** Etapa de una aplicación MapReduce que toma la lista de pares clave-valor intermedios producidos por la fase de *Shuffle and Sort* y los combina en un conjunto menor de pares clave-valor que representa el resultado final de la aplicación.
- **Registro:** Entrada de datos individual que se procesa en la fase de Map. Cada registro está representado como una clave-valor y se procesa de forma independiente.
- **Shuffle and Sort:** Etapa intermedia en una aplicación MapReduce que ocurre después de la fase de Map y antes de la fase de Reduce. En esta etapa, los datos de salida del Map (pares clave-valor) se agrupan y ordenan para ser enviados a las tareas de Reduce.
- **Split:** Porción de datos de tamaño fijo o variable que se extrae de uno o más bloques y se procesa de forma independiente en la fase de Map.
- **Trabajo:** Instancia de ejecución de una aplicación MapReduce.
- **YARN:** *Yet Another Resource Negotiator* es el sistema de gestión de recursos de Hadoop que administra la asignación de recursos para ejecutar aplicaciones MapReduce.