Tarea 1: Modelos de comunicación y Middleware

Sistemas Distribuidos

Maite Bernaus Gimeno y David Fernández Márquez

FECHA DE ENTREGA: 23/04/2019

Índice

1. Diseño general

En esta práctica se nos ha pedido que diseñemos un **sistema paralelo de tipo Map-Reduce simplificado**, el cual debería procesar de forma paralela un fichero de texto previamente subido al Cloud Object Storage de IBM y contar las palabras en éste de dos posibles maneras: número total de palabras, o recuento de incidencias de cada palabra.

Los ficheros de esta práctica se encuentran en el siguiente repositorio de git: https://github.com/Hilnur/SDPract

La práctica tiene **tres secciones** diferenciadas: un distribuidor local que se ejecuta desde el propio ordenador (orchestrator.py + fichero de apoyo ibm_cf_connector.py), una serie de funciones cloud almacenadas en el sistema CloudFunctions de IBM (Map, Reduce, GetHead, GetObject, Delete, ListObjects), y un punto de almacenaje en cloud para el cual se utiliza IBM COS.

Las funciones y el orchestrator se analizarán con más detalle en los apartados subsiguientes, pero consideramos importante especificar aquí la estructura del almacenaje cloud.

Para la gestión de los distintos ficheros utilizados se ha creado 2 depósitos en el Cloud Object Storage de IBM llamados *originals* y *temps1*. En el depósito *originals* se habrán de poner, como su nombre indica, los originales que se desean analizar (en este caso, los ficheros de: big.txt, pg10.txt y pg2000.txt). *temps1* es un depósito de archivos temporales que se utiliza para guardar el resultado de los distintos maps, para permitir que el reduce pueda obtener posteriormente dichos resultados y unirlos todos en un fichero *resultados* final que también se guardará en temps1.

2. Análisis detallado

2.1 Orchestrator.py

Parámetros de entrada: (nombre del fichero, numero de chunks a paralelizar, operación):

> Nombre del fichero: El nombre con el que el fichero figura en COS/*originals*
>
> Numero de chunks: El número de procesos independientes en los cuales se desea partir el trabajo.
>
> Operación: dos posibilidades: 'count' para contaje normal de palabras, 'diffcount' para contar las incidencias de cada palabra.

Funciones:

Orchestrator.py realiza toda la función de manejo de la aplicación, incluyendo los siguientes puntos:

- Comprueba la corrección de los parámetros nombre de fichero, número de particiones y operación a realizar son correctos.
- Lee los parámetros de configuración para el sistema de IBM (functions y COS) del fichero *ibm-cloud_config* (que ha de estar situado en la carpeta raíz del usuario).
- Prepara los diccionarios *argsOriginal* y *argsTemp* que contendrán todos los argumentos necesarios para invocar a las funciones cloud.
- Obtiene los metadatos del fichero introducido y calcula el tamaño de los chunks dividiendo el tamaño de archivo entre nchunks.

- Mediante un bucle, va asignando puntos de inicio y fin (*startByte*, *endByte*) de chunk a cada proceso, e invocando asíncronamente la función Cloud Map para cada chunk.
- Espera a que finalicen todos los Maps (comprobando los resultados mediante la función ListObjects) e invoca a la función Reduce.
- Termina de procesar el resultado del Reduce y elimina ficheros temporales para evitar problemas de superposición.

De forma general, orchestrator.py también realiza control de errores (timeouts, ficheros inexistentes, etc), así como muestra del tiempo elidido para facilitar el análisis.

Notas de diseño adicionales: Debido a problemas con la codificación de diferentes tipos de ficheros, nos vimos obligados a añadir un control de error de timeout, ya que ficheros con codificaciones complejas (de múltiples bytes por carácter) "bloquean" el análisis del Map, produciendo un bucle de espera eterno en el orchestrator.

2.2 Map

Parámetros de entrada: Diccionario conteniendo los siguientes campos:

configCOS: credenciales para configurar el backend

source_bucket: el bucket del cual leerá el archivo a mapear

source_file: el archivo a mapear

target_bucket: el bucket de archivos temporales para almacenar los resultados parciales

target_file: el nombre bajo el cual se almacenarán los resultados parciales (importante para mantener el orden, ver abajo)

bytestart: byte del fichero donde empieza el chunk asignado

byteend: byte del fichero donde termina el chunk asignado

op: operación a realizar (count o diffcount)

Función:

Map utiliza la librería COSBackend proporcionada por el profesor para leer un chunk del fichero objetivo, lo convierte en un string, y lo procesa mediante las funciones wordCount() o countingWords() (ver debajo), obteniendo un diccionario que se convierta a json y se guarda en el COS según los parámetros de entrada.

Importantemente, **el Map que hemos diseñado es capaz de unir palabras partidas por el chunking** y añadirlas al recuento, utilizando un sistema de "tanteo". Para esto, cada Map analiza los puntos inicial y final de su chunk. En caso de encontrar un carácter alfanumérico en alguno de los extremos, el Map almacena en sus resultados el "pedazo" al que pertenece este carácter como flag que avisará al Reducer de que es posible que se encuentre ante una palabra cortada, y por tanto habrá de comprobar los resultados del elemento inmediatamente subsiguiente/anterior. Por esto es importante mantener el orden de los ficheros.

Este sistema, como hemos descubierto durante las pruebas, tiene problemas con caracteres complejos de 3 o más bytes, ya que éstos pueden producir que el carácter quede partido y el Map no sepa cómo actuar ante un semi-carácter que no es puntuación, whitespace, o alfanumérico, resultando en un bucle infinito.

2.2.1.countingWords (text)

Función: Cuenta la cantidad de palabras en el string suministrado. Recibe por paramento los datos a analizar como un string y divide el texto en una lista de palabras utilizando split().

Antes de proceder al recuento, se realiza el análisis de primer y último carácter previamente mencionado, aislando los pedazos primero y último en caso de ser necesario.

Para obtener el número de palabras se devuelve la longitud de la lista de palabras.

2.2.2.wordCount (text)

Función: Cuenta el número de ocurrencias de cada palabra individual. Recibe por paramento los datos a analizar como un string y divide el texto en una lista de palabras utilizando split().

Antes de proceder al recuento, se realiza el análisis de primer y último carácter previamente mencionado, aislando los pedazos primero y último en caso de ser necesario.

Mediante un bucle que recorre la lista, genera un diccionario con las palabras como claves y el número de apariciones de cada palabra como valor. Una vez se completa la lista se devuelve el diccionario creado.

2.2. Reducer

Parámetros de entrada:

configCOS: credenciales para configurar el backend

bucket: el bucket en el cual se encuentran los resultados parciales

op: operación que se ha realizado (count o diffcount)

prefix: prefijo etiqueta de los archivos de resultado parcial a leer (utilizado para mantener el orden)

Función: Combina todos los resultados parciales almacenados en bucket y guarda un archivo final *resultados* en la carpeta temporal con el resultado combinado.

Para realizar esto, Reducer va leyendo los ficheros *prefix*+(num) en orden, obteniendo los diccionarios y uniéndolos mediante la función joinDictionary (que obtuvimos de una librería pública de Python) en un diccionario final finalresult. Una vez ha añadido un fichero parcial, la función lo borra de la carpeta de archivos temporales, para simplificar el mantenimiento y evitar que los archivos se superpongan entre múltiples ejecuciones.

Reducer es además la segunda parte de la lógica que permite al sistema compensar las palabras partidas. Cada vez que Reducer lee un resultado parcial que contenga un flag de "chunk terminado en carácter", se lo guarda como carry, y en caso de que el siguiente resultado parcial tenga activo el flag de "chunk comienza por carácter", añade los dos pedazos para formar una palabra completa y la añade al resultado final.

Un problema que encontramos es que las funciones de Cloud Function parecen tener un límite en el número de entradas de un diccionario que pueden devolver, así que para que funcionase con archivos grandes, nos vimos obligados a hacer que en lugar de simplemente devolver *finalresult*, Reducer almacenase su resultado final finalresult en un json y lo guardase en la carpeta de archivos temporales para que la pudiese leer el orchestrator.

3. Funciones auxiliares

Dada su simpleza, nos limitaremos a enumerar lo que hacen, sin entrar en detalles:

GetHead ({*configCOS, filename, bucket*}): Obtiene los datos del header del fichero *filename* ubicado en *bucket*, y los devuelve como un diccionario.

Delete({*configCOS, filename, bucket*}): Borra el fichero *filename* ubicado en *bucket*.

ListObjects ({*configCOS, bucket*}): Devuelve un listado con todos los nombres de ficheros contenidos en el depósito *bucket*.

GetObject({*configCOS, filename, bucket*}): Obtiene el fichero filename del depósito bucket del COS, devolviendo su contenido en la forma {'content': [contenido del fichero]}

4. Juego de pruebas

4.1.Parámetros incorrectos

Parámetros / Modificaciones	Salida Esperada	Salida Real	Verificación
El fichero no está en el cloud	File does not exist in target COS. Exiting program	File does not exist in target COS. Exiting program	OK
La operación no es 'count' ni 'diffcount'	Valid operators are count (total number of words) or diffcount (count amounts of each individual word)	Valid operators are count (total number of words) or diffcount (count amounts of each individual word)	OK
Numero de particiones= 0 Numero de particiones= -6	number of parallel processes needs to be a positive integer!	number of parallel processes needs to be a positive integer!	OK
El fichero de configuración ibm-coud_config no se encuentra en la raíz del usuario.	User file ibm-coud_config does not exist.	User file ibm-coud_config does not exist.	OK

4.2.Parámetros correctos CountWords

Para validar el correcto funcionamiento se considera que el comportamiento es correcto si en todas las ejecuciones con distintos numero de particiones se obtiene el mismo resultado ya que se ha analizado ficheros muy grandes.

Parámetros / Modificaciones	Salida	Verificación
big.txt 1 'count'	1121410	OK. Los resultados son iguales en todas las ejecuciones.
big.txt 4 'count'		
big.txt 8 'count'		
big.txt 16 'count'		
big.txt 32 'count'		
big.txt 50 'count'		
pg10.txt 1 'count'	824429	Se puede observar que en 16 y 50 chunks se obtiene una palabra más
pg10.txt 4 'count'		
pg10.txt 8 'count'		
pg10.txt 16 'count'	824430	
pg10.txt 32 'count'	824429	
pg10.txt 50 'count'	824430	
pg2000.txt 1 'count'	384931	OK. Los resultados son iguales en todas las ejecuciones.
pg2000.txt 4 'count'		
pg2000.txt 8 'count'		
pg2000.txt 16 'count'		
pg2000.txt 32 'count'	*	El programa no termina, posiblemente sea debido a que está en español y hay caracteres no estándares
pg2000.txt 50 'count'	*	

		codificados por más de 8 bits.

4.3.Parámetros correctos WordCounting

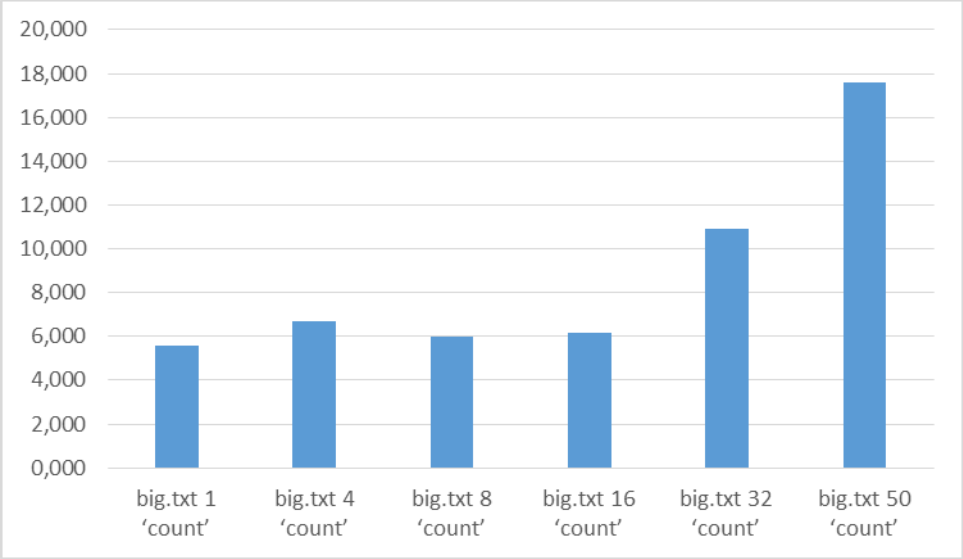
Parámetros / Modificaciones	Salida Real	Verificación
big.txt 1 ‘diffcount’	{"the": 79178, "project": 264, "gutenberg": 238, "ebook": 79, "of": 39976, "adventures": 17, "sherlock": 100, "holmes": 434, "by": 6701, "sir": 168, "arthur": 27, (…)	OK. Los resultados son iguales en todas las ejecuciones.
big.txt 4 ‘diffcount’		
big.txt 8 ‘diffcount’		
big.txt 16 ‘diffcount’		
big.txt 32 ‘diffcount’		
big.txt 50 ‘diffcount’		
pg10.txt 1 ‘diffcount’	{"the": 64195, "project": 83, "gutenberg": 87, "ebook": 10, "of": 34779, "king": 2202, "james": 48, "bible": 8, "this": 2809, "is": 6996, "for": 8907, "use": 47, "anyone": 5, "anywhere": 2, "at": 1583, "no": 1398, (…)	OK. Los resultados son iguales en todas las ejecuciones.
pg10.txt 4 ‘diffcount’		
pg10.txt 8 ‘diffcount’		
pg10.txt 16 ‘diffcount’		
pg10.txt 32 ‘diffcount’		
pg10.txt 50 ‘diffcount’		
pg2000.txt 1 ‘diffcount’	{"the": 174, "project": 84, "gutenberg": 87, "ebook": 9, "of": 118, "don": 2651, "quijote": 2088, "by": 24, "miguel": 21, "de": 18193, "cervantes": 17, "saavedra": 11, "this": 47, "is": 25 (…)	OK. Los resultados son iguales en todas las ejecuciones.
pg2000.txt 4 ‘diffcount’		
pg2000.txt 8 ‘diffcount’		
pg2000.txt 16 ‘diffcount’		
pg2000.txt 32 ‘diffcount’	*	El programa no termina, posiblemente sea debido a que el texto está en español y hay caracteres multi-byte que rompen el algoritmo.
pg2000.txt 50 ‘diffcount’	*	

5. SpeedUp

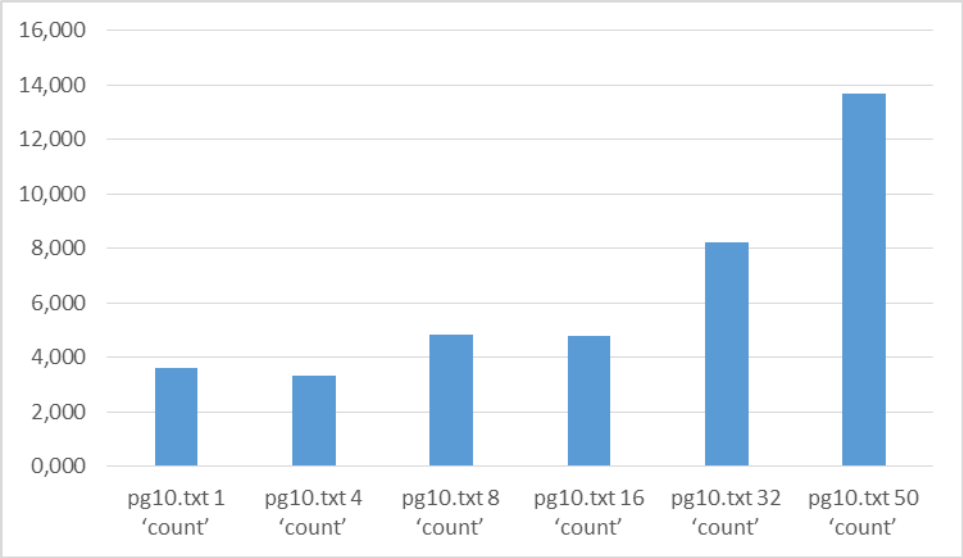
Para calcular el speed up se ha realizado un análisis del tiempo de ejecución de las dos operaciones en los 3 ficheros de texto de prueba (big.txt, pg10.txt y pg2000.txt). Para realizar dicho análisis se ha medido por triplicado el tiempo de ejecución de cada uno de ellos en 1, 4, 8, 16, 32 y 50 chunks para poder observar la evolución del tiempo de ejecución respecto el número de chunks y respecto un solo chunk.

CountWords

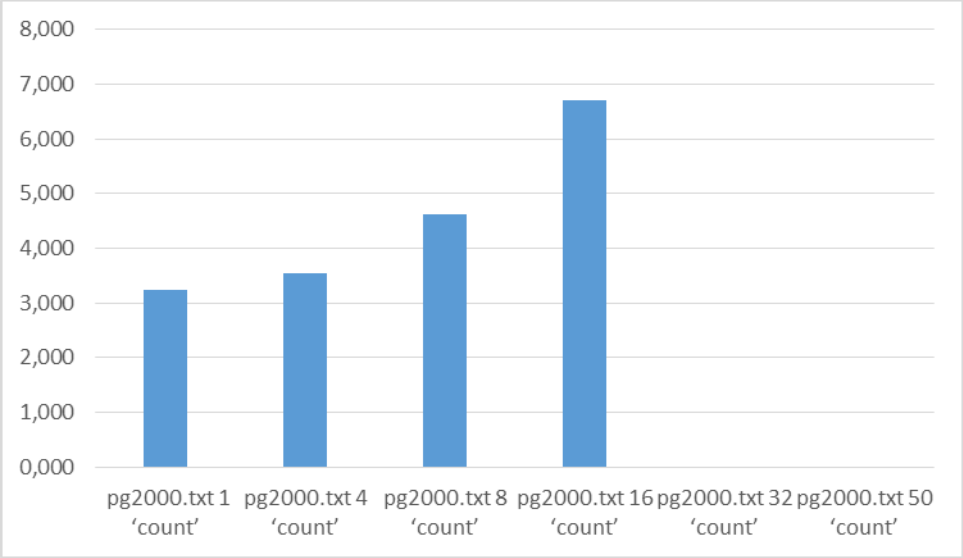
Parámetros / Modificaciones	**Tiempo (s)**	Tiempo1 (s)	Tiempo2 (s)	Tiempo3 (s)
big.txt 1 'count'	5,560	5,963224	4,873338	5,843474
big.txt 4 'count'	6,683	6,475924	7,136852	6,436971
big.txt 8 'count'	6,012	5,315795	7,220214	5,500561
big.txt 16 'count'	6,162	4,582304	7,033606	6,870909
big.txt 32 'count'	10,934	10,833477	9,930476	12,036622
big.txt 50 'count'	17,570	18,468072	18,866403	15,374200



pg10.txt 1 'count'	3,628	4,255168	2,868264	3,760436
pg10.txt 4 'count'	3,322	3,391775	2,951420	3,623039
pg10.txt 8 'count'	4,810	5,919925	4,841889	3,667196
pg10.txt 16 'count'	4,790	4,289844	5,539111	4,542323
pg10.txt 32 'count'	8,232	8,052288	7,711426	8,933443
pg10.txt 50 'count'	13,668	14,988628	14,519068	11,497555

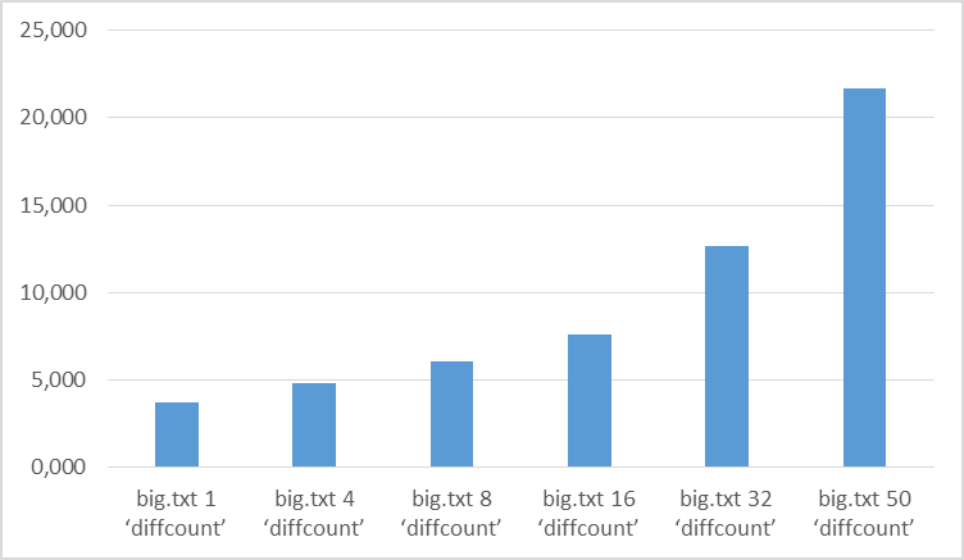


pg2000.txt 1 'count'	3,237	3,847035	2,635826	3,226656
pg2000.txt 4 'count'	3,551	3,921054	3,424445	3,306249
pg2000.txt 8 'count'	4,633	4,529277	5,173830	4,196907
pg2000.txt 16 'count'	6,720	5,635819	6,986782	7,536276
pg2000.txt 32 'count'	*	*	*	*
pg2000.txt 50 'count'	*	*	*	*

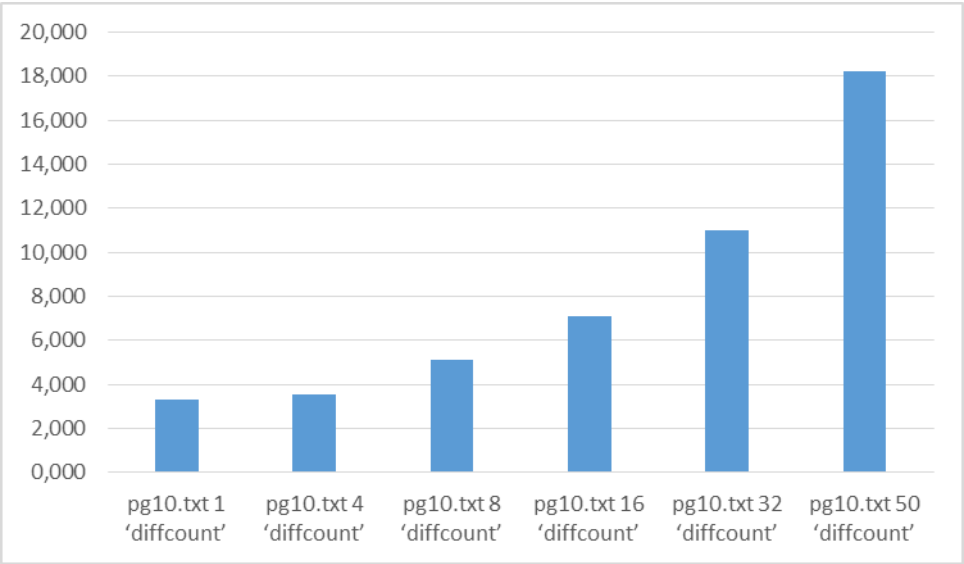


WordCounting

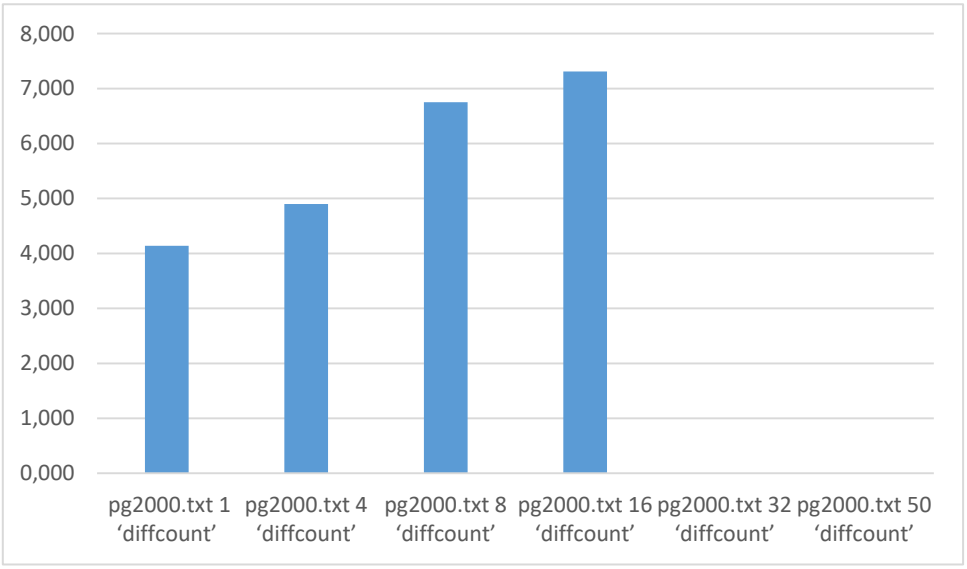
Parámetros / Modificaciones	Tiempo (s)	Tiempo1 (s)	Tiempo2 (s)	Tiempo3 (s)
big.txt 4 'diffcount'	4,849	4,894091	5,183828	4,468384
big.txt 8 'diffcount'	6,057	5,833006	6,219848	6,116741
big.txt 16 'diffcount'	7,627	7,704300	7,553456	7,622234
big.txt 32 'diffcount'	12,692	12,164547	13,670013	12,240240
big.txt 50 'diffcount'	21,712	21,384127	21,532607	22,218981



pg10.txt 1 'diffcount'	3,305	3,235159	3,088749	3,590037
pg10.txt 4 'diffcount'	3,548	3,172252	3,346849	4,125330
pg10.txt 8 'diffcount'	5,107	4,911308	5,641488	4,768545
pg10.txt 16 'diffcount'	7,118	6,567609	6,480253	8,306345
pg10.txt 32 'diffcount'	11,025	9,694687	11,473813	11,905964
pg10.txt 50 'diffcount'	18,249	18.230842	19.524272	16.990695



pg2000.txt 1 ‘diffcount’	4,138	4,350501	3,642922	4,421221
pg2000.txt 4 ‘diffcount’	4,899	4,668181	5,490995	4,537021
pg2000.txt 8 ‘diffcount’	6,751	6,618613	6,745522	6,889628
pg2000.txt 16 ‘diffcount’	7,309	7,396572	7,968871	6,562469
pg2000.txt 32 ‘count’	*	*	*	*
pg2000.txt 50 ‘count’	*	*	*	*



Se puede observar que en las dos operaciones y los 3 ficheros de texto analizados el tiempo de ejecución aumenta considerablemente respecto el número de chunks generados, aunque las operaciones del map se ejecuten de forma paralela. Esto es debido a distintos factores entre ellos el tiempo de leer y realizar el join de los diccionarios en el reduce que se realiza de forma secuencial el cual aumenta en aumentar el numero de chunks y el tiempo de invocación de las distintas funciones maps.