Median and Standard Deviation

Instrucciones: Compilar proyecto con Eclipse, o bien ejecutar "ant build" sobre el directorio Summarization. El jar quedará en el directorio padre, con nombre summarization.jar. Las clases a ejecutar con hadoop son: BrutalRunner, SmartRunner.

El objetivo es obtener la mediana y desviación estándar de los largos de comentarios en sitios de Stack Exchange, según la hora del día en que estos se ingresan.

Descripción y Datos de Entrada

Para este Challenge se siguió bastante de cerca el código del libro *Map Reduce Design Patterns*. Las principales diferencias son:

- Para la lectura de los archivos, en vez de simplemente dividir las líneas usando las comillas como separadores (como se detalla en la página *xiii* del prefacio), se usó un InputFormat especializado para lectura de XML (XmlInputFormat) cuya versión original es parte del proyecto *Apache Mahout* (https://github.com/apache/mahout) y adaptado al framework mapreduce según las instrucciones de http://xmlandhadoop.blogspot.com.
- Se definió una clase CommentParser para procesar cada elemento XML de tipo Comment usando las clases del package org.w3c.dom de java.
- Los mappers, a diferencia de los del libro, usan clases strong-typed para procesar los comentarios.

Como datos de entrada, se usaron los archivos de comentarios de sitios de StackExchange (*Stack Exchange Data Dump March 2013*, http://www.clearbits.net/creators/146-stack-exchange-data-dump/contents). De entre ellos, se seleccionaron tres archivos Comments.xml de diferentes órdenes de magnitud en cuanto a tamaño:

- meta.scifi.stackexchange.com (1.8 MB, 6,227 comentarios).
- android.stackexchange.com (9.8 MB, 38,857 comentarios).
- english.stackexchange.com (47 MB, 168,054 comentarios).
- superuser.stackexchange.com (131MB, 531,604 comentarios).

Nota: Por su tamaño, estos archivos no se adjuntan en el paquete entregado.

Implementación "Brutal"

En esta implementación:

- El output del Mapper es de tuplas *<hora del día, largo de comentario>* (una por cada comentario en el archivo de entrada).
- No se utiliza un combiner.
- El Reducer calcula la mediana ordenando todos los largos para una determinada hora del día y buscando el valor central (o el promedio de los valores centrales) y luego el promedio y desviación estándar de la forma habitual. El promedio se calcula al mismo tiempo que se calcula la mediana para evitar una tercera recorrida por los valores.

Comparación de ejecuciones

	meta.scifi	android	english	superuser
Map input / output records	6,227	38,857	168,054	531,604
Map output bytes	49,816	310,856	1,344,432	4,252,832
Reduce input groups	24	24	24	24
Tiempo insumido	**8s**	**10s**	**28s**	**1m 13s**

La relación entre *map output records* y *map output bytes* es 1/8, esperable puesto que la salida del mapper es un par <IntWritable, IntWritable> (y por lo tanto 4 + 4 bytes).

¿Por qué no es posible utilizar un Combiner?

En primer lugar, es evidente que no se puede usar el reducer como combiner ya que sus tipos de datos de salida son diferentes. Suponiendo que no se permite modificar el reducer, no es posible implementar un combiner por el mismo motivo del ejercicio de promedios del challenge 2: el cálculo de la mediana y desviación estándar no son operaciones asociativas, y no pueden realizarse sobre la totalidad del conjunto de datos si únicamente se cuenta con dichos valores sobre subconjuntos del mismo. Por ejemplo:

$$mediana([1,1,1,2,10,11,12]) = 2$$

$$mediana([mediana(1,1,1,2), mediana(10,11,12)]) = mediana([1,11]) = \frac{1+11}{2} = 6$$

Sin embargo, es posible utilizar un combiner si se modifica el funcionamiento del map/reduce.

Implementación "Memory-conscious"

Básicamente optimiza el uso de memoria del reducer, teniendo un diccionario ordenador de *<valor, cantidad>* en vez de la lista de todos los valores. Por ejemplo, si hay n_i comentarios de largo k_i, entonces en vez de tener una lista:

$$\dots, k_{i-1}, \underbrace{k_i, \dots k_i}_{n_i}, k_{i+1}, \dots$$

simplemente se tiene un par *<k_i, n>*. Esto permite reducir considerablemente el tamaño de esta estructura, especialmente si los valores de *n* son altos.

En esta implementación, el mapper produce como valor de su output "mini diccionarios" con el valor del largo del comentario y el número 1. Luego el reducer acumula estos diccionarios de entrada en uno con todos los valores para una determinada hora, y halla la mediana recorriendo esta estructura. Para el cálculo de promedio y desviación estándar las modificaciones son similares, simplemente hay que tener en cuenta que se procesan todos los valores iguales "a la vez", sabiendo la cantidad que hay de cada uno.

Además, se puede utilizar un combiner para disminuir el número de reduce input records, sumando los contadores de los outputs de un mismo mapper (podría haberse hecho también con *in-mapper combining* sin la necesidad de un combiner separado).

Los resultados son iguales a los de la implementación anterior, modulo efectos de redondeo por cálculos con numéricos de punto flotante.

El código de mapper, combiner y reducer es casi idéntico al del libro, con la salvedad un cambio necesario para esquivar un aparente bug de Hadoop. La clase que se usa para representar los diccionarios es la SortedMapWritable (de org.apache.hadoop.io), que tiene la particularidad que en su implementación de readFields() no limpia su contenido previo. Puesto que la iteración por los valores de input de un reducer (o combiner) reutiliza el objeto iterado, esto provoca que *se acumulen los valores de maps anteriores*. Para evitar este error, basta con asegurarse de ejecutar clear() sobre la instancia de SortedMapWritable como última instrucción del cuerpo del loop.

Comparación de ejecuciones (caso "superuser")

En primer lugar, resultó que no era estrictamente necesario utilizar esta implementación, ya que la anterior no falló por límite de memoria. Esto era esperable, en cierto modo, ya que:

- El conjunto de datos más grande utilizado (531,604 registros) ciertamente no es excesivo.
- El tipo de datos de los valores en este problema es básicamente enteros (4 bytes) por lo que la reducción de memoria obtenida gracias a evitar la duplicación de los mismos no es tan considerable.

Ciertamente se justificaría, en cambio, si tuviésemos muchos más registros o, especialmente, valores de tipos de datos que requieran más memoria (tales como strings u otros objetos complejos).

	"Brutal"	**"Memory-conscious"**
Map output records	531,604	531,604
Map output bytes	4,252,832	12,226,892
Map output materialized bytes	5,316,070	791,246
Reduce input records	531,604	120

La cantidad de *map output records*, como se esperaba, es el mismo. Sin embargo, el número de *reduce input records* se ve notoriamente disminuido gracias al combiner.

El número de *bytes* de resultado del mapper, en cambio, es mayor en esta segunda implementación. Esto es lógico puesto que el tipo de datos de output del primer mapper era simplemente int (4 bytes), mientras que en el segundo caso se trata de un diccionario (con un valor) lo cual agrega cierto overhead por la estructura del mismo.

Por otro lado, el número de *map output materialized bytes* es considerablemente menor. Seguramente se deba al agregado del combiner, pero la documentación no es clara en este punto.