Memoria Explicativa Patrones de Diseño Tema 6: Hadoop Java API

En este documento se explica muy brevemente el desarrollo realizado en el ejercicio:

- Idea en la que se basa el desarrollo.
- Diseño técnico explicativo de desarrollo de la solución.

Idea de la solución

Idea:
1 Job:
Mapper: Mapearemos las líneas de cada fichero obteniendo de las líneas:

Clave[Fecha-hora]Evento Valor: Evento:Repeticiones
Usaremos el **patrón de diseño visto en clase mapper-cobiner**, mediante el cual realizaremos la tarea que realizaría el combiner de emitir por cada evento una suma de cada una de sus repeticiones en todas líneas.
No emitiremos aquellas líneas donde el evento obtenido comience por "vmet".

Sorter: **Clase vista en clase** mediante la cual ordenaremos las líneas emitidas en el mapper de mayor a menor basándonos en la hora de los eventos.
Particioner: **Clase vista en clase** mediante la cual y fijándonos en uno de los campos de la key que emitimos en mapper, redireccionaremos a 2 reducer que tendremos declarados en el driver, 1 para los eventos kernel y otro para el resto.
Reducer: Shuffle and sort y emisión de cada de uno de los eventos en el fichero que corresponda en función de la hora y la fecha y el evento en sí mismo:
1 fichero para los evento kernel y cada día
1 fichero para el resto de eventos y cada día
sumatorio de cada una de las entrada que vienen generando el par:
[29/11/2014-07] ntpd:4,avahi-daemon:5,ntpdate:6
Además cada una de los eventos de cada línea irán ordenados de menor a mayor en función de la cantidad de repeticiones.
Para los eventos kernel tendremos otro fichero con otro nombre.
[*29/11/2014-14] kernel:1*

Diseño Técnico Explicativo

La clase principal que implementa el driver es AnalisisLogsDriver.java la cual consta de las siguientes clases y responden a peticiones del ejercicio :

1º Un Mapper optimizado:
LogsMapper.java
En esta clase hacemos uso del patrón de diseño mapper-combinig explicado en clase. La ventaja del mapper combining sobre el Combiner tradicional, es que el primero puede que se ejecute o puede que no, no tenemos control sobre ello. Mientras que el mapper combining podemos hacer que se ejecute siempre.
La segunda ventaja de este patrón de diseño es controlar cómo se lleva a cabo exactamente. Otra ventaja es que con el Combiner se reduce el número de datos que llegan al Shuffle and Sort para luego enviarlos al Reducer, pero realmente no reduce el número de pares key/value emitidos por el Mapper.

Con este patrón sumamos todas las repeticiones de cada evento emitiendo una suma de los mismos, y con ello emitiendo menos pares-valor y mejorando el envío de información por red.

En este mapper además detectamos si el evento que vamos a tratar comienza por "vmet" en cuyo caso ni siquiera será tratado para su mapeo.

En esta clase hacemos uso de 2 writables para la salida:

LogServiceWritable.java: El cual utilizamos para emitir la key de manera compuesta el día , el evento y la hora del mismo para su ordenación en el shuffle and sort. Además aprovechamos el evento en el partitioner para enrutar al reducer correspondiente y la hora en la clase de sort para ordenar de mayor a menor cada una de las fechas iguales. Su método toString nos devolverá la composición de clave deseada en el ejercicio:
[29/11/2014-14] Teniendo en cuenta y dando por asumido que siempre tratamos días de noviembre del 2014.

LogValueEventWritable.java El cual utilizamos para emitir el value de manera compuesta el evento y número de repeticiones del mismo para su ordenación de menor a mayor en la salida hacia los ficheros, y el recuento de número de repeticiones de cada evento en el mismo mapper y a posteriori en el reducer.

2º Una clase que seteamos en job.setSortComparatorClass(LogsSortHourKey.class). LogsSortHourKey.java la cual se encarga de ordenar las key de los pares emitidos por el mapper basándose primero en si la fecha es igual y después en caso de ser verdadera esta condición en la hora de mayor a menor con su método toCompare.

3º Una clase Partitioner job.setPartitionerClass(LogsPartitioner.class);
LogsPartitioner.java la cual se encarga de redireccionar a un reducer u otro en función del valor que posea la key que emitimos, 0 para aquellos que posean el evento kernel y 1 para el resto.

4º Hago uso de MultiOutPuts:

```
job.setInputFormatClass(TextInputFormat.class);
LazyOutputFormat.setOutputFormatClass(job, TextOutputFormat.class);
```

De manera que la salida se repartirá en diferentes ficheros en función del evento que estemos tratando y la hora del mismo. Creando un fichero por cada día con todos los logs del kernel y otro para cada día para el resto de eventos. Esta repartición a cada fichero se realizara en cada uno de los reducers(2 reducers tal y como se setea en esta clase driver).

5º Una clase Reducer `job.setReducerClass(LogsReducer.class)` :

LogsReducer.java : En esta clase llegara cada una de las keys ya ordenadas por hora y cada uno de los eventos asociados con su número de repeticiones correspondientes. A pesar de haber hecho este recuento en la fase mapper tenemos que volver hacerlo y guardamos el recuento de cada uno de los eventos en un hashmap.

Para el resto de eventos hará los mismo con la diferencia que cada línea por hora dentro de su fichero correspondiente tendrá que ser un conglomerado de cada uno de los eventos de ese día y hora y sus totales ordenados de menor a mayor, para ello con la ayuda de un Treemap de java y otra clase que implementamos para hacer esta ordenación ValueComparator.java ordenamos cada uno de los eventos de menor a mayor en función de su valor y lo emitimos hacia su fichero correspondiente en función del día que tenga en la key.

6º Contadores Generales

Serán emitidos en los reducers, ya que es allí donde conocemos el total de cada uno de los eventos que vamos a emitir y los recuperamos en esta clase Driver:

```
//Contadores
Counters counters = job.getCounters();
```

No estoy seguro del todo si lo que me devuelve cg.iterator(); podría ser un hashmap, lo intento pero me da errores en ejecución por lo que primero lo meto en un hashmap, aprovecho el método de ordenación que utilizamos en los reducers pero de manera descendente en este caso e implemento que la clase ValueComparator.java tenga la opción de ordenación ASC y DES para este caso.
Finalmente los muestro por pantalla con sysout.

Adjunto diagrama de clases y comentarios varios dentro del código.