MapReduce Fest

Lorena Etcheverry
Instituto de Computación,
Facultad de Ingeniería,
lorenae@fing.edu.uy

14 de mayo de 2013

Índice

1. Desafío 2

1.1. Introducción

Observe los valores de los contadores a través de la interfase web *JobTracker*.

1. ¿Qué representa el contador Reduce input groups? ¿Qué se puede hacer para que el valor de este contador varíe?

El contador *reduce input groups* representa el número de grupos a considerar en la fase de reduce, o lo que es equivalente a la cantidad de claves diferentes generadas durante la fase de map. Para cambiar este valor es necesario cambiar la política de asignación de claves durante la fase de map. En este caso se considera todo el texto de Los Miserables y por cada palabra se genera una pareja (clave, valor), donde clave es la palabra encontrada y valor es 1. Para reducir la cantidad de claves se podrían considerar las palabras ignorando mayúsculas y minúsculas.

2. ¿Qué representan los contadores map input records y map output records?

El contador *map input records* corresponde a la cantidad de lineas del archivo de entrada, mientras que el contador *map output records* corresponde a la cantidad de parejas (clave, valor) generadas durante la fase de map.

3. En su opinión, ¿cómo se relacionan los contadores map input records y map output records?

Tal como se mencionó en la pregunta 2, el contador *map output records* representa a la cantidad de parejas generadas a partir de una entrada de tamaño *map input records*, por lo tanto el primer valor será mayor o igual que el segundo.

1.2. Uso de *combiners*

Los *combiners* suelen considerarse como pequeños *reducers* y tienen como objetivo reducir la cantidad de registros que se consideran en la fase de *reduce*. El objetivo de esta etapa es agregar un *combiner* al ejemplo de Word Count y ejecutar nuevamente el código tomando como entrada los 5 tomos de Los Miserables, de Victor Hugo.

1. ¿Cuáles son los contadores que permiten verificar los efectos del combiner? Los contadores que permiten verificar los efectos de utilizar un *combiner* son: (a)*combine input records*, (b) *combine output records*, y (c) *spilled records*. El primero indica la cantidad de parejas (clave, valor) que son consumidas por el o los *combiners* mientras que el segundo indica la cantidad de parejas (clave, valor) luego de aplicar el *combiner*. Para comprender el significado del contador *spilled records* es necesario comprender el procesamiento y flujo de datos que se lleva a cabo durante las fases de map y reduce.

Durante la fase de map se generan, a partir de los datos de entrada, parejas (clave,valor) y, dado que una de las premisas de MapReduce es que la entrada de cada *reducer* está ordenada por clave estas parejas deberán ordenarse antes de ser distribuídas a los diferentes *reducers*. Este proceso de distribuir y ordenar se conoce como *shuffle and sort*[3]. Durante el mismo se utilizan estructuras en memoria (*buffers*) para ordenar las parejas o registros, y cuando estos *buffers* se llenan más allá de cierto umbral se realiza un volcado a disco o *spill*. La Figura 1 representa este proceso gráficamente. El contador *spilled records* indica la cantidad de parejas o registros que fueron volcados a disco, y por lo tanto puede utilzarse para evaluar el impacto del uso de un *combiner*, en lo que respecta a datos almacenados en disco.

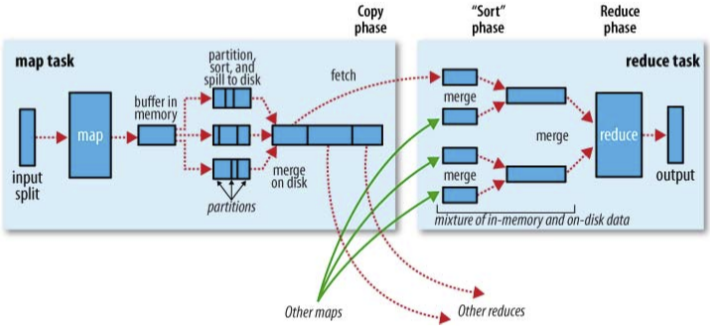

Figura 1: Shuffle y sort en MapReduce[3]

2. ¿Cómo podemos estimar las ventajas de utilizar un combiner? Justifique su respuesta, brindando valores al usar y no usar un combiner.

Para el caso de WordCount se utilizó como *combiner* el mismo código que se utiliza como *reducer*, realizando por lo tanto una pre-agrupación de registros por clave (palabra) y sumando la cantidad de valores. De esta forma la canti-

	Sin *combiner*	Con *combiner*
Map input records	52710	52710
Map output records	409359	409359
Combine input records	0	409359
Combine output records	0	67485
Reduce input records	409359	67485
Reduce output records	52554	52554
Spilled Records (map)	409359	67485
Spilled Records (reduce)	409359	67485
Tiempo de ejecución	36,248 s	37,365 s

Tabla 1: Valores obtenidos para el problema de conteo de palabras, con y sin *combiners*.

dad de registros que deben ser despachados a los *reducers* es menor (siempre y cuando las palabras se repitan en el texto, lo cual suele ocurrir). La tabla 1 presenta los valores de algunos contadores para el caso sin y con *combiner*. En la misma se puede apreciar que, al utilizar el combiner mencionado, se reduce en un 83 % la cantidad de registros a procesar en la fase de reduce y la cantidad de registros que son almacenados en disco (*spilled records*). El uso de *combiners* podría conducir a un aumento en el tiempo de ejecución. En este caso el proceso sin *combiner* tardó aprox. 1 segundo menos que la versión con *combiner*. También se puede apreciar, tal como se indica en[2], que el uso de *combiners* reduce el tamaño de resultados intermedios que son intercambiados mediante la red, pero no reduce la cantidad de parejas (clave,valor) que son generadas como salida de los *mappers*.

1.3. Aplicación del patrón *In-mapper combining*

In-mapper combining es un patrón de diseño para MapReduce que consiste en incorporar funcionalidades del *combiner* dentro de los *mappers*[2]. Este patrón se basa en cómo se ejecutan las tareas en Hadoop, en particular: (a) la invocación al método INITIALIZE antes de que los *mappers* procesen datos, (b) la capacidad de preservar el estado a través de diferentes llamadas al método MAP sobre la misma pareja (clave, valor), y (c) que luego de haberse aplicado el método MAP a todas las parejas de cierto bloque se realiza una llamada al método CLOSE. La Figura 2 presenta el pseudocódigo correspondiente a aplicar este patrón al problema de Word Count[2].

```
method INITIALIZE
    H ← new ASSOCIATIVEARRAY
method MAP (docid a, doc d)
    for all term t ∈ doc d do
        H{t} ← H{t}+1
    end for
method CLOSE
    for all term t ∈ H do
        EMIT(term t, count H{t}
    end for
```

Figura 2: Aplicación de *"in-mapper combining"* al problema de conteo de palabras[2].

En esta etapa se procedió a modificar el código provisto para adaptarlo al comportamiento descripto en el pseudocódigo de la Figura 2. La primer modificación consiste en migrar el código al API Java MapReduce disponible desde la versión 0.20.0 de Hadoop, el cual se implementa en el paquete `org.apache.hadoop.mapreduce` en lugar de `org.apache.hadoop.mapred`. Tal como se indica en la documentación técnica[1], cada trabajo o `job` MapReduce se descompone en tareas. A cada tarea se le asigna una porción de los datos de entrada representada por un objeto del tipo `InputSplit` y se genera una instancia de la clase `Mapper` para procesar esta porción de la entrada. El método `run` de la clase `Mapper` describe el comportamiento de la misma. Se comienza por realizar una invocación al método `setup`, luego se invoca al método `map` sobre cada una de las entradas en el `InputSplit` y por último se invoca al método `cleanup`. En la Figura 3 se presenta el código del método `run`.

```
1 public void run(Context context) throws IOException,InterruptedException {
    setup(context);
3   while (context.nextKeyValue()) {
      map(context.getCurrentKey(), context.getCurrentValue(), context);
5     }
      cleanup(context);
7 }
```

Figura 3: Método `run` de la clase org.apache.hadoop.mapreduce.Mapper [1]

Se puede apreciar una correspondencia entre los métodos `initialize` y `close` del pseudocódigo de la Figura 2 y los métodos `setup` y `cleanup` de la Figura 3, respectivamente. Es así que en la implementación se define una estructura auxiliar (en este caso un `HashMap`) el cual se inicializa en el método `setup`. En el método `map` se mantiene un contador por cada palabra encontrada y por último, en el método `cleanup` se procede a volcar el contenido de la estructura auxiliar al contexto, para que sea procesado por los *reducers*. En la Figura 4 se presenta el código desarrollado y en la Tabla 2 se presentan los valores de contadores y tiempo de ejecución. Comparándolos con los valores de la Tabla 1 se puede apreciar una reducción en la cantidad de registros que salen de la etapa de map (*Map output records*) lo cual coincide con lo esperado dado que en lugar de generarse un registro por cada ocurrencia de cada palabra, en cada tarea se genera un registro por palabra (independiete de la cantidad de instancias de la misma). En cuanto al tiempo de ejecución esta estrategia parece insumir más tiempo, aunque deberían realizarse pruebas más exhaustivas para tener resultados concluyentes.

1.4. Análisis de datos públicos

En esta sección se presentan las soluciones de las preguntas 4 a 8, donde se analizan datos correspondientes a pagos por categoría realizados por la ciudad inglesa de Bedford-Borrough en el período abril 2010 a julio 2010. Los datos de entrada se encuentran en archivos en formato CSV (un archivo por mes) en los cuales la columna 1 corresponde a la categoría del pago, la columna 2 corresponde al proveedor al cual se le pagó, la columna 3 al identificador de transacción y por último, la columna 4 corresponde al monto del pago. La Figura 5 presenta una muestra de los datos de entrada correspondientes a abril

Map input records	52710
Map output records	52538
Combine input records	0
Combine output records	0
Reduce input records	52538
Reduce output records	52538
Spilled Records (map)	52538
Spilled Records (reduce)	52538
Tiempo de ejecución	45,257 s

Tabla 2: Valores obtenidos para el problema de conteo de palabras aplicado in-map combinning.

```
1  public class WordCountMapper
   extends  Mapper<LongWritable, Text, Text, IntWritable>
3  {
     private Text word = new Text();
5    Map<Text, Integer> local;

7    public void setup(Context context)throws IOException,
     InterruptedException {
9      local = new HashMap<Text, Integer>();
     }
11
     public void map(LongWritable key, Text value, Context context)
13   throws IOException{
       //taking one line at a time and tokenizing the same
15     String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
17
       while (tokenizer.hasMoreTokens()){
19       word.set(tokenizer.nextToken());
         if (local.containsKey(word)) {
21         local.put(word, new Integer (local.get(word) + 1));
         }else{
23         local.put(word, new Integer(1));
         }
25     }
     }
27
     public void cleanup(Context context)
29   throws IOException, InterruptedException{
       Iterator<Entry<Text, Integer>> it = local.entrySet().iterator();
31     while (it.hasNext()) {
         Map.Entry pair = (Map.Entry)it.next();
33       context.write((Text)pair.getKey(),
           new IntWritable((Integer)pair.getValue()));
35       }
   }
37 }
```

Figura 4: Aplicación del patrón “in-mapper combining” al problema de conteo de palabras.

de 2010.

```
"Adult Services  ":"Central Bedfordshire Council":10064809:5279
"Adult Services  ":"Aragon Housing Association":10077807:11836.88
"Adult Services  ":"Peterhouse (Places for People)":10089836:1168.02
"Adult Services  ":"ITI (Bedford) Limited":10093739:1057.45
"Adult Services  ":"Beds & Luton Partnership NHS Trust":10095624:139187.83
```

Figura 5: Muestra de los datos de entrada utilizados en la Sección 1.4.

1.4.1. Pregunta 4: suma de pagos por categoría

Previo a computar la suma de pagos por categoría se unen los 4 archivos de entrada en uno solo. Luego se generan parejas (clave, valor), donde la clave corresponde a la categoría y el valor al monto pagado. Se observó que en los archivos de entrada existen categorías que parecen ser la misma, pero difieren en la cantidad de espacios en blanco al final de la misma (e.g. "Adult Services " y "Adult Services "). Dado que estas cadenas de caracteres parecen representar la misma categoría se decidió eliminar los espacios en blanco al comienzo y al final antes de generar las claves. Se optó por implementar el patrón *in-mapper combinning*, almacenando los resultados parciales en una estructura auxiliar al igual que en la sección 1.3. La figura 6 presenta un extracto del código desarrollado mientras que la Tabla 3 presenta los resultados obtenidos.

Adult Services	8769623.93000001
BT One Bill	37342.2
Carlisle Managed Solutions	445655.14
Chief Executive	3945091.96
Chief Executive & Corporate Services Directorate	1597416.62
Chief Executive Services	836409.38
Childrens Services	1.17E+007
Childrens Services, Schools & Families	8688694.63
Commercial Services	1219822.44
Commercial Services Department	2563104.79
Commercial Services Dept	2874665.35
Environmental & Sustainable Communities	6791296.5
Environmental Services	2558974.99
Environmental Services Directorate	3881692.97
Finance & Corporate Services	5348793.16

Tabla 3: Pagos por categoría.

1.4.2. Pregunta 5: suma de pagos por categoría y total de pagos

Para calcular el total de pagos, además de los pagos por categoría, se genera una categoría ficticia llamada Total donde se acumula la suma de todos los pagos. Esta solución es independiente de la cantidad de *reducers*, dado que se generan en la fase de *map* parejas (clave, valor) con clave "Total" que luego se procesan por un único *reducer*. Para probar que el comportamiento es correcto se indica en la clase principal que para este job deben usarse dos reducers. Esto

```
1       public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException
3       {
            //taking one line at a time and tokenizing the same
5           String line = value.toString();
            String[] parsed =line.split(":");
7
          //category is the key, amount is the value
9               String category = parsed[0].replace("\"", "").trim();
                Text word= new Text(category);
11          if (local.containsKey(word)) {
                    local.put(word, new Double (local.get(word) +
13                      Double.parseDouble(parsed[3])));
            }else{
15                  local.put(word, new Double(parsed[3]));
            }
17      }

19      public void reduce(Text key, Iterator<DoubleWritable> values, Context context)
          throws IOException, InterruptedException
21      {
            int sum = 0;
23          /*iterates through all the values available with a key and add them
              together and give the final result as the key and sum of its values*/
25          while (values.hasNext()){
                sum += values.next().get();
27          }
            context.write(key, new DoubleWritable(sum));
29      }
```

Figura 6: Métodos `map` y `reduce` implementados en la Sección 1.4.1

se realiza usando la directiva `job.setNumReduceTasks(2)`. La Tabla 4 presenta los resultados obtenidos para cada uno de los *reducers*.

1.4.3. Pregunta 6: promedio de pagos por categoría

El promedio de pagos por categoría es el cociente entre la suma de pagos y la cantidad de pagos por cada categoría. El enfoque más simple para calcular estos promedios usando *map reduce* consiste en generar parejas con clave la categoría y valor el monto y realizar el cómputo de la suma y la cantidad en la fase de *reduce*. Con este enfoque se obtienen los resultados que se presentan en la Tabla 5. Si se utiliza este *reducer* simple como *combiner* se obtendrían valores incorrectos, dado que se realizarían promedios de promedios en la fase de *reduce*. Si se desea implementar un *combiner* para reducir la cantidad de registros que pasan a la fase de reduce es necesario que el campo valor de las parejas (clave,valor) contenga la suma de pagos para cierta categoría y la cantidad de pagos que componen esa suma. De esta manera, en la fase de reduce, se cuenta con los datos necesarios para computar el promedio correctamente.

Dado que Hadoop no garantiza la cantidad de veces que invoca al *combiner* es preferible utilizar el patrón "in-mapper combining", siempre y cuando las restricciones en el uso de memoria lo permitan[2]. En este caso se utiliza este patrón y se implementa además la clase `ValuesPair`, que implementa la interfaz `Writable`, la cual se utiliza para modelar las parejas de valores (cantidad, suma).

part-r-00000	
BT One Bill	37342.2
Chief Executive	3945091.96
Chief Executive Services	836409.38
Childrens Services, Schools & Families	8688694.63
Commercial Services Dept	2874665.35
Finance & Corporate Services	5348793.16
part-r-00001	
Adult Services	8769623.93000001
Carlisle Managed Solutions	445655.14
Chief Executive & Corporate Services Directorate	1597416.62
Childrens Services	1.17E+007
Commercial Services	1219822.44
Commercial Services Department	2563104.79
Environmental & Sustainable Communities	6791296.5
Environmental Services	2558974.99
Environmental Services Directorate	3881692.97
Total	6.1217285870000005E7

Tabla 4: Pagos por categoría y total.

1.4.4. Pregunta 7: suma de pagos por mes

Para calcular la suma de pago por mes se obtiene el mes del nombre del archivo, y se generan parejas (clave,valor) con clave el mes y valor el pago realizado. Se obtienen los valores que se presentan en la Tabla 6

1.4.5. Pregunta 8: suma de pagos por categoría y por mes

Para calcular la suma de pagos por categoría y por mes se utiliza una clave compuesta que representa parejas (mes, categoría). La clase que representa esta clave debe implementar la interfaz `WritableComparable`. La Figura 7 presenta un extracto del código de dicha clase y la Tabla 7 presenta los resultados obtenidos. Cabe destacar que el mes se extrae directamente desde el nombre del archivo, no siendo necesario modificar los datos de entrada.

Referencias

[1] Documentación técnica - clase org. apache. hadoop. mapreduce. Mapper. `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Mapper.html`, 2013.

[2] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[3] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

```
1  public class CompoundKeyWritableComparable implements
     WritableComparable<CompoundKeyWritableComparable> {
3
          private String month;
5         private String category;

7         public CompoundKeyWritableComparable(){}
          public CompoundKeyWritableComparable(String month, String category) {
9                 this.month=month;
                  this.category=category;
11        }

13        @Override
          public void readFields(DataInput arg0) throws IOException {
15                this.month = Text.readString(arg0);
                  this.category = Text.readString(arg0);
17        }

19        @Override
          public void write(DataOutput arg0) throws IOException {
21                Text.writeString(arg0, this.month);
                  Text.writeString(arg0, this.category);
23        }

25        @Override
            public int hashCode() {
27            return month.hashCode() * 163 + category.hashCode();
            }
29
            @Override
31          public boolean equals(Object o) {
              if (o instanceof CompoundKeyWritableComparable) {
33                CompoundKeyWritableComparable tp = (CompoundKeyWritableComparable) o;
                return month.equals(tp.month) && category.equals(tp.category);
35            }
              return false;
37          }

39          @Override
            public String toString() {
41            return month + "\t" + category;
            }
43
          @Override
45        public int compareTo(CompoundKeyWritableComparable o) {
                  int cmp = month.compareTo(o.month);
47                if (cmp != 0) {
                          return cmp;
49                }
                  return category.compareTo(o.category);
51        }
          }
```

Figura 7: Extracto de la clase que implementa la clave compuesta (mes,categoría) usada en la Sección 1.4.5

Adult Services	11615.3959337748
BT One Bill	37342.2
Carlisle Managed Solutions	148551.713333333
Chief Executive	19433.9505418719
Chief Executive & Corporate Services Directorate	15359.7751923077
Chief Executive Services	10200.1143902439
Childrens Services	20489.8098594025
Childrens Services, Schools & Families	14409.1121558872
Commercial Services	5700.1048598131
Commercial Services Department	4967.2573449612
Commercial Services Dept	7389.885218509
Environmental & Sustainable Communities	11977.5952380952
Environmental Services	13190.5927319588
Environmental Services Directorate	14115.2471636364
Finance & Corporate Services	10805.6427474747

Tabla 5: Promedio de pagos por categoría.

April2010	13641324.7
July2010	15251311.16
June2010	16506297.35
May2010	15818352.66

Tabla 6: Total de pagos por mes.

April2010	Adult Services	1192923.29
April2010	Chief Executive Services	836409.38
April2010	Childrens Services	6662523.09
April2010	Commercial Services	1219822.44
April2010	Environmental Services	2558974.99
April2010	Finance & Corporate Services	1170671.51
July2010	Adult Services	3457976.23
July2010	Chief Executive	1648251.5
July2010	Childrens Services, Schools & Families	4001635.28
July2010	Commercial Services Dept	2874665.35
July2010	Environmental & Sustainable Communities	2555680.25
July2010	Finance & Corporate Services	713102.55
June2010	Adult Services	2079470.37
June2010	BT One Bill	37342.2
June2010	Carlisle Managed Solutions	445655.14
June2010	Chief Executive	2296840.46
June2010	Childrens Services, Schools & Families	4687059.35
June2010	Commercial Services Department	1583782.33
June2010	Environmental & Sustainable Communities	4235616.25
June2010	Finance & Corporate Services	1140531.25
May2010	Adult Services	2039254.04
May2010	Chief Executive & Corporate Services Directorate	1597416.62
May2010	Childrens Services	4996178.72
May2010	Commercial Services Department	979322.46
May2010	Environmental Services Directorate	3881692.97
May2010	Finance & Corporate Services	2324487.85

Tabla 7: Total de pagos por mes y categoría.