Introduction to Massive Data Analysis

Homework 1 Report

Student ID: 105062635
Name: 吳浩寧

I. IMPLEMENTATION

在我 MapReduce 程式的 jobs 皆採用預設的設定，不過為了達到作業輸出的要求，將 mapreduce.output.textoutputformat.separator 從預設的 tab 改為”,”；此外 mapper 的輸出，reducer 的輸入輸出格式皆設定為(Text, Text)，由與 key 和 value 皆不只一個值，必須先將每個值串起來並用”,”分隔，處理起來較一致，老師上課教的兩種矩陣相乘的方式都有實作，分別為 *MM1S.java* 與 *MM2S.java*，假設 M 為 a×b 大小的矩陣，而 N 為 b×c，兩者相乘得到 a×c 的矩陣 P，$p_{ik} = \sum_{j=1}^{b} m_{ij} n_{jk}$。

1.) 1-Step

須先在程式中指定兩個矩陣的大小，HEIGHT=a; COMMON=b; WIDTH=c

Map Function:

由矩陣相乘的式子可看出，m_{ij} 會與 n_{j1}~n_{jc} 相乘，因此須產生 c 個 key-value pairs，同理每個 n_{jk} 須產生 a 個 pairs，格式為((i, k), (M, j, m_{ij}))或((i, k), (N, j, n_{jk}))。

```
public void map(Object key, Text value, Context context
    ) throws IOException, InterruptedException {
    String[] input = value.toString().split(",");
    if(input[0].equals("M")){
        for(int i = 0; i < WIDTH; i++){
            ik.set(input[1] + "," + Integer.toString(i));
            Mjm.set(input[0] + "," + input[2] + "," + input[3]);
            context.write(ik, Mjm);
        }
    }else{
        for(int i = 0; i < HEIGHT; i++){
            ik.set(Integer.toString(i) + "," + input[2]);
            Mjm.set(input[0] + "," + input[1] + "," + input[3]);
            context.write(ik, Mjm);
        }
    }
}
```

Reduce Function:

每次執行 reduce 會負責 P 的其中一個元素 p_{ik}，先將收集到所有(i, k)相同的值區分為來自於 M 矩陣，或來自於 N 矩陣，由於來自這兩個矩陣 pair 數量皆為 b 個，可以直接用 j 作為 index 分別存到兩個陣列中，最後遍歷兩個陣列，將每個元素分別相乘在計算總合即為最終結果。

```
public void reduce(Text key, Iterable<Text> values,
    Context context) throws IOException, InterruptedException {
    for (Text val : values) {
        String[] input = val.toString().split(",");
        if(input[0].equals("M"))M[Integer.parseInt(input[1])] = Integer.parseInt(input[2]);
        else N[Integer.parseInt(input[1])] = Integer.parseInt(input[2]);
    }
    for(int i = 0; i < COMMON; i++)sum += M[i] * N[i];
    result.set(Integer.toString(sum));
    context.write(key, result);
}
```

2.) 2-Step

須在主程式中宣告兩個 job，以執行兩組 mapper, reducer，且在 job1 輸出得先存到一個暫時的資料夾，供 job2 使用，最後將資料夾刪除。

Map Function 1:

產生格式為(j, (M, i, m_{ij}))或(j, (N, k, n_{jk}))的 pairs，與 1-Step 的差異在對於輸入的每一行只會產生一組 pair。

```
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    String[] input = value.toString().split(",");
    if(input[0].equals("M")){
        j.set(input[2]);
        m.set(input[0] + "," + input[1] + "," + input[3]);
    }else{
        j.set(input[1]);
        m.set(input[0] + "," + input[2] + "," + input[3]);
    }
    context.write(j, m);
}
```

Reduce Function 1:

將收集到所有 j 相同的值區分為來自於 M 矩陣，或來自於 N 矩陣，分別存到長度為 a 與 c 的陣列中，再用一個兩層迴圈計算所有陣列元素相乘的組合，並將格式為((i, k), m_{ij}* n_{jk}) 的 pairs 傳給下個 mapper。

```
public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
    for (Text val : values) {
        String[] input = val.toString().split(",");
        index = Integer.parseInt(input[1]);
        scalar = Integer.parseInt(input[2]);
        if(input[0].equals("M")){
            M[index] = scalar;
            if(height < index)height = index;
        }else{
            N[index] = scalar;
            if(width < index)width = index;
        }
    }
    for(int i = 0; i < height + 1; i++){
        for(int j = 0; j < width + 1; j++){
            newkey.set(Integer.toString(i) + "," + Integer.toString(j));
            result.set(Integer.toString(M[i]*N[j]));
            context.write(newkey, result);
        }
    }
}
```

Map Function 2:

直接將輸入傳給下個 reducer。

```
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    String[] input = value.toString().split("\\t");
    j.set(input[0]);
    m.set(input[1]);
    context.write(j, m);
}
```

Reduce Function 2:

將所有(i, k)相同的值進行加總，得到最後 p_{ik} 的值。

```
public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
        for (Text val : values) {
            sum += Integer.parseInt(val.toString());
        }
    result.set(Integer.toString(sum));
    context.write(key, result);
}
```

使用 1-Step 的好處在，只需進行一次 MapReduce，缺點是沒有有效的方法，靠讀取輸入檔來得知整個矩陣的大小，必須寫死在程式裡或靠參數傳入；使用 2-Step 必須進行兩次 MapReduce，因此須更多的 communication 時間，讀寫檔案也必須進行兩次，優點是可以不用預先知道矩陣的大小。

2. RESULT

自己寫了 M(4x2)與 N(2x5)相乘的測資，輸出結果和自己計算後的結果一樣

input		output	
M,0,0,87	N,0,0,78	0,0,17006	2,0,16458
M,0,1,268	N,0,1,455	0,1,25025	2,1,96005
M,0,2,542	N,0,2,10	0,2,2046	2,2,2110
M,1,0,95	N,0,3,478	0,3,31594	2,3,100858
M,1,1,0	N,0,4,73	0,4,21423	2,4,15403
M,1,2,222	N,1,0,187	1,0,75792	3,0,11517
M,2,0,11	N,1,1,0	1,1,55965	3,1,5005
M,2,1,57	N,1,2,22	1,2,9018	3,2,1364
M,2,2,306	N,1,3,78	1,3,86406	3,3,9704
	N,1,4,256	1,4,99603	3,4,15395

至於助教提供的 500input.txt，我使用兩種方式跑出來的結果為一模一樣的，跑完成程式 Hadoop counters 的 metadata 整理如下：
(除了 MM2S Job2 的 Launched map tasks=13，其他 Task 皆為 1)

	MM1S Job1	MM2S Job1	MM2S Job2
map time (ms)	1935009	18495	660178
reduce time (ms)	876712	125802	611199
CPU time (ms)	1262230	124730	811360
physical mem (b)	319623168	309682176	2370768894
virtual mem (b)	3890950144	3885580288	27094822912
heap usage (b)	165150720	164526432	1522532352
JobHistory (s)	2540	152	1524

※JobHistory 的執行時間是從 web UI 上查看的

使用 1-Step 的 mapper-reducer communication=2*a*b*c；2-Step 則為 a*b+b*c+a*b*c 不過額外需要檔案讀寫 a*b*c 組 pairs，job setup 所需要的時間也會變為兩倍。從上表來看，2-Step 的方式稍快一點，且它的第一部分主要花在 reducing，第二部分花在 mapping，雖然我不太確定上面數據有沒有包括 IO 讀檔的時間；而兩種方法使用到的記憶體量是差不多的。

3. EXPERIENCE

這次作業讓我了解如何在 Hadoop 架構上進行 MapReduce 程式的開發，遇到比較大的問題包括以下幾點：了解 mapper 與 reducer 之間資料傳送的格式；忘了把 WordCount 範例中的 combiner 刪除；如何使用複數組 mapper、reducer 等等。由於只能在自己電腦的 VM 上運行 Hadoop 平台，因此計算時間非常之久，嘗試設定 mapreduce.job.reduces 參數，執行時卻會產生一些 IOExceptions；有時 2-Step 的版本也會產生 failures，雖然覺得可能是 timeout 限制問題，但把該值加大，或嘗試改變一些參數配置仍無法解決；也覺得可能是記憶體不足，嘗試把垃圾桶清空，最後終於有跑完，但還是無法確定產生此現象的原因。