HW2 - PageRank

102062111 林致民

Instruction

MapReduce

```
# Compile all map-reduce code
cd HW2_102062111_MR
./compile.sh

# Build map for SIZE testcase
./build_graph.sh ${SIZE}

# Start iteration
./execute ${SIZE}

# Get result of page-rank
hdfs dfs -cat HW2_${SIZE}/result/* | less
```

Spark

```
cd HW2_102062111_Spark
./compile

# Start to build map and iterate
./run.sh ${SIZE}
```

Implementation

不管是Map-reduce 還是 Spark，對於演算法的實作是一樣的，我把它簡單分成幾個步驟：

1. 建圖：把raw data parese 完，先對連到的點的title capitalize，之後把無效邊拔掉，可以得到所有點和邊，並且可以知道點邊的數量，對於每個點做初始化 $\frac{1}{N}$
2. 先把常數項（對於其他點來說是常數，但是不同iteration可能不是常數）算出來，常數項的公式是：

$$(1-\alpha)\left(\frac{1}{N}\right)+\alpha\sum_{j=1}^{m}\frac{PR(d_j)}{N}$$

這兩項可以先做，右邊那一項是總和「出度＝0」的所有node的PageRank並對他normalize到N，配給他的比例是 $\alpha = 0.85$，目的是考慮邊緣人的程度，而左邊是增加自己這一點獨立個體的權重，配給他的比例是$(1-\alpha)$

3. 統計每個node的predecessor到當前點的值，公式的中間項：

$$\alpha \sum_{i=1}^{n} \frac{PR^{(k-1)}(t_i)}{C(t_i)}$$

在這個步驟，我把這一項拆掉，並且使用key-value的方式來存。由於我們建表是以有向圖來存，並且可以知道當前點連到其他哪些點（設其他點為B），這個點為Predecessor，設當前點為P，因此要把自己的「值」給其他人。這個點要給其他人的value為：

$$\alpha \frac{PR^{(k-1)}(P)}{C(t_i)}$$

至於 Key的話，由於是從predecessor把直給連到的點（B），因此key要設為B，所以reduce 的時候只要把相同key的值sum在一起就可以了

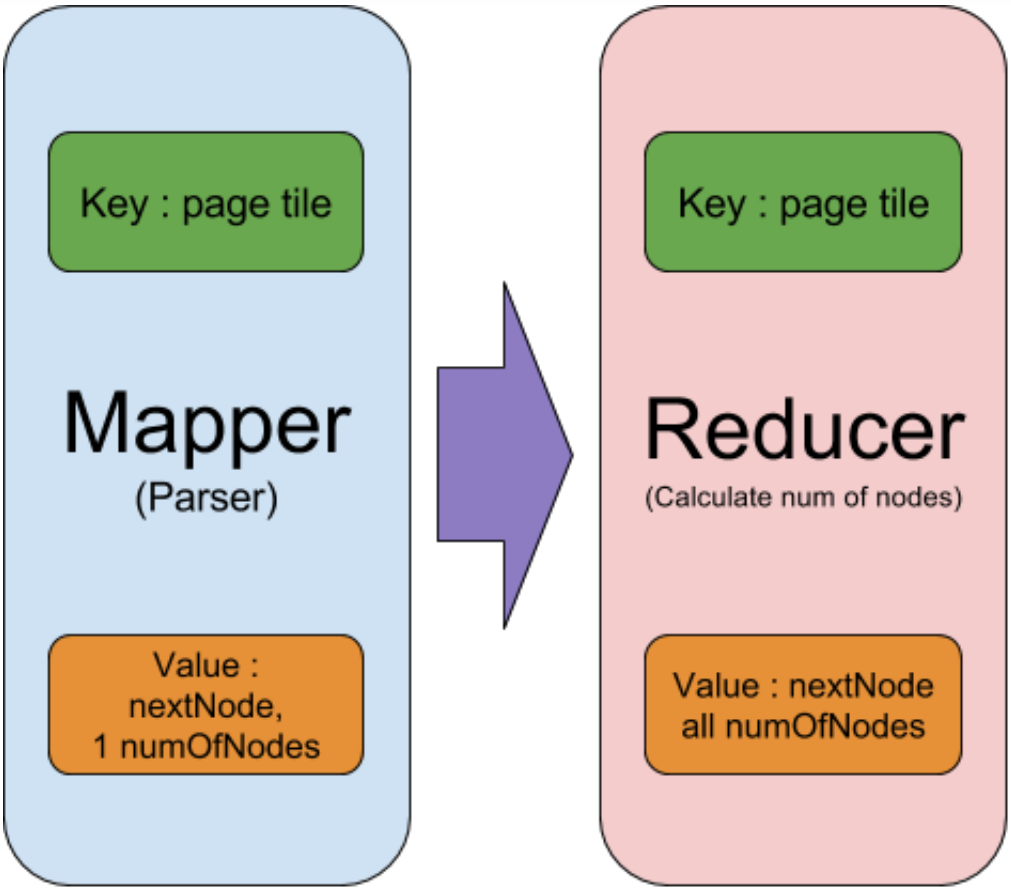
4. 既然都把各項的值算完了，就把 2 步驟與 3 步驟的值總和起來，就是當前點新的Page Rank，此時去判斷誤差值總和是否比0.001還要小，如果比0.001還要小就再繼續做地 2 ～ 3步驟
5. 排序好把答案輸出

Map-reduce ＆Spark Build Graph

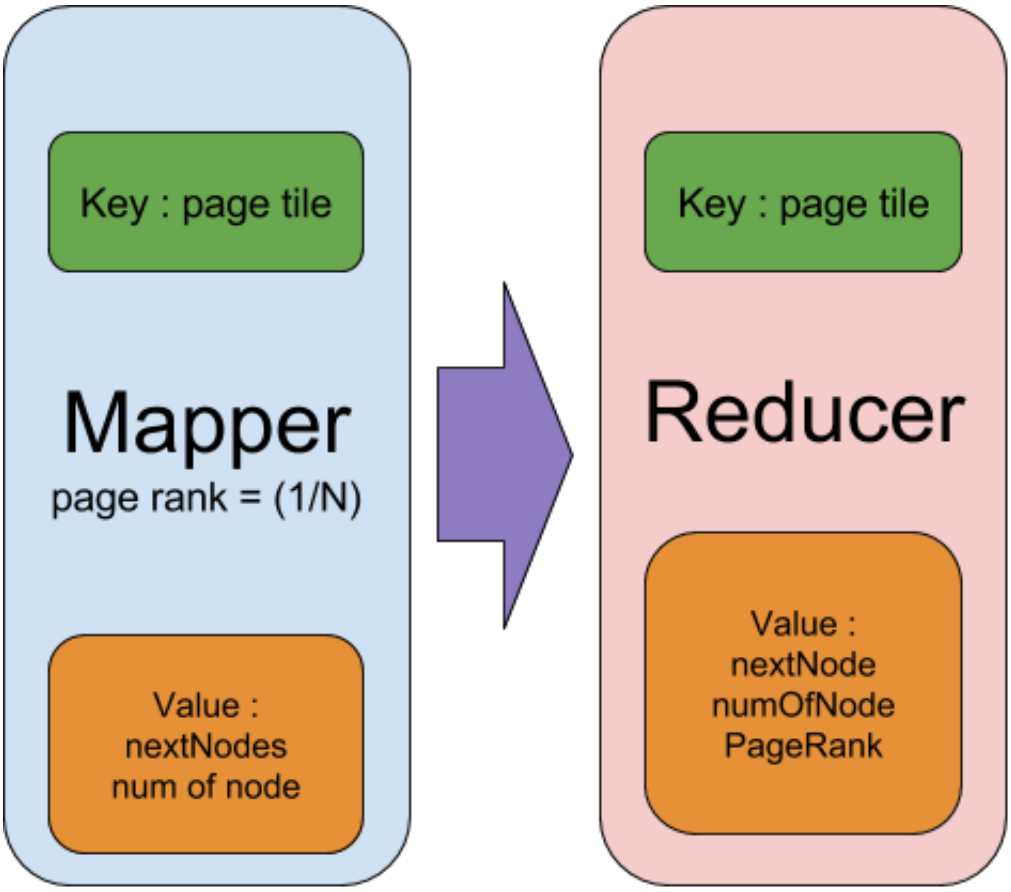
建圖Spark & Hadoop 的概念實作上沒太大的差異，如下圖

1. Parse document & build graph & 去除無用邊

   Parse 節點 & 計算總共有多少的page(node)



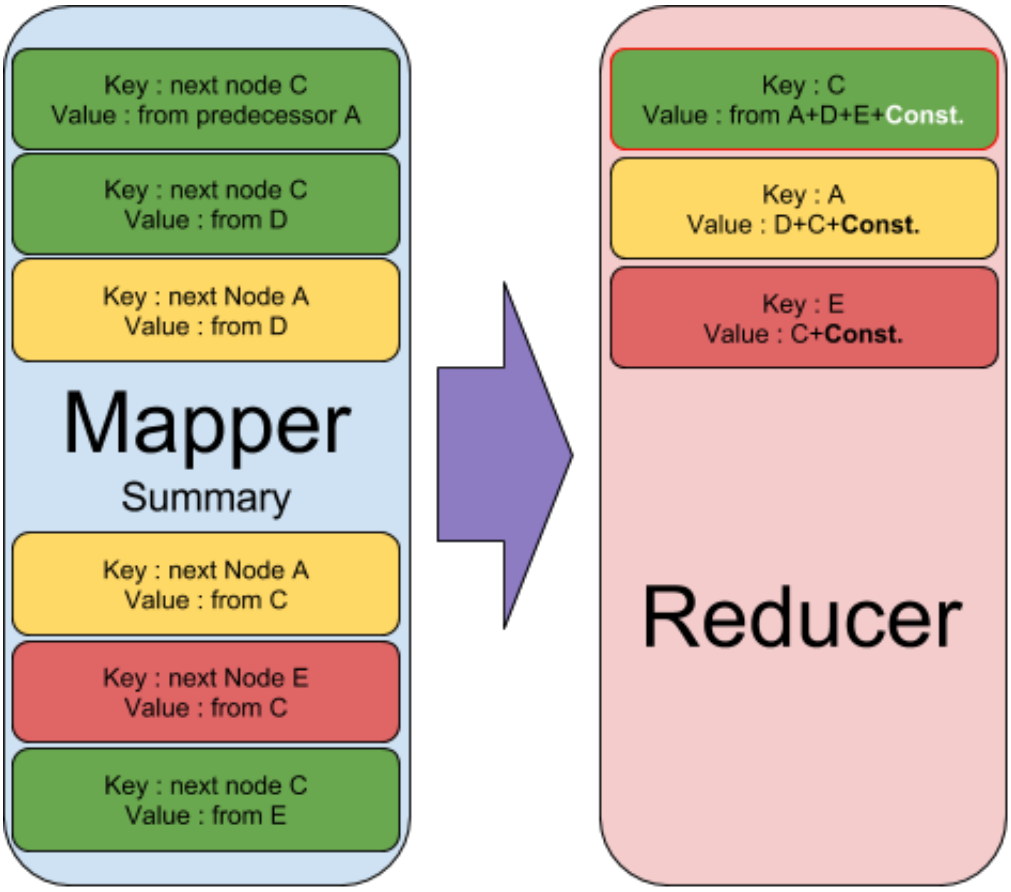
2. Prepare graph for iteration : 初始化每個node的pagerank



Iteration

1. 統計Zero Degree 的node的Page Rank 的總和

2. 計算predecessor連到其他點的值：



3. 計算與上一回合的誤差值，如果 >= 0.001 就跳到 1. 繼續iteration，直到收斂為

Experiment

Hadoop Job ID Success

由於是自動化同時跑 100M 1G 10G 50G，所以大概是這一大群Job ID

job*14642291280232038* ~ job*14642291280232268*

Spark Application ID

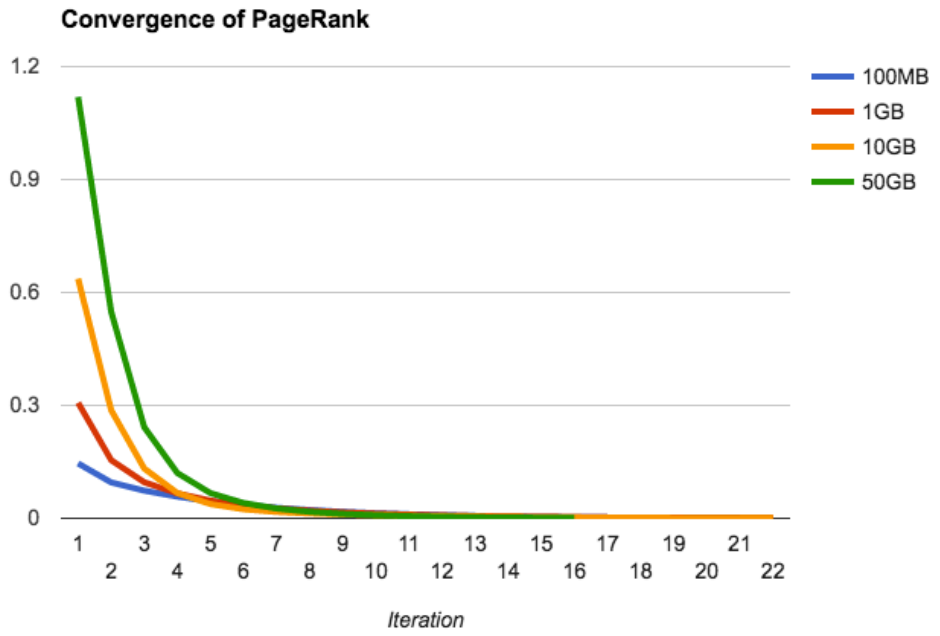
application*14644009507920773* 50G

application*14644009507920629* 10G

application*14644009507920619* 1G

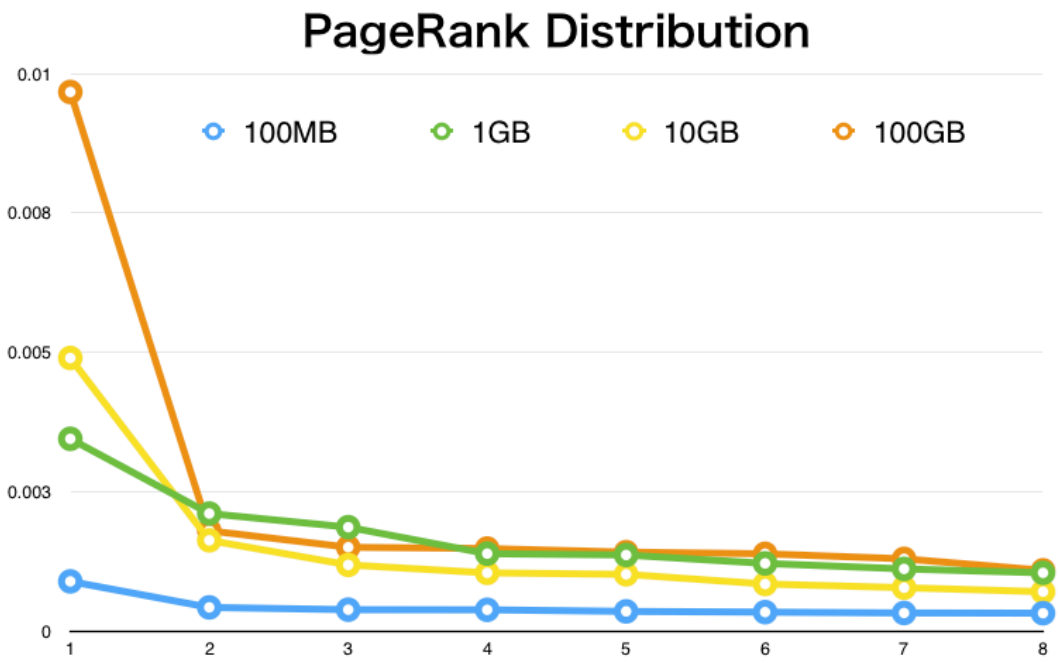
application*14644009507920291* 100M

1. **Analyze the converge rate**



	100MB	1GB	10GB	50GB
1	0.1458633566	0.3069644075	0.6366619038	1.119517657
2	0.09530126653	0.1545868618	0.2871390139	0.5487877109
3	0.07348334313	0.09584771824	0.1322033096	0.2419811313
4	0.05756479622	0.06622825351	0.06712513673	0.1203677387
5	0.04520508464	0.04704252766	0.03792473208	0.06696989999
6	0.03552303274	0.03496016949	0.02406126164	0.04068519597
7	0.02791710373	0.02641466546	0.01698640336	0.02606594352
8	0.02194354166	0.0203756572	0.01297036629	0.01736117746
9	0.01724775647	0.01589447251	0.0103555026	0.01186513279
10	0.0135623284	0.01251161708	0.008414047088	0.008218617521
11	0.0106610205	0.009889029606	0.006885731823	0.005717301663
12	0.008383679143	0.007849658453	0.00566890332	0.003985673118
13	0.006590802774	0.006235960936	0.004671214169	0.002776480631
14	0.00518397451	0.004967032162	0.003853124988	0.001942451966
15	0.004076279917	0.003955843756	0.003178841372	0.001365892018
16	0.003206730803	0.003157596744	0.002622730616	9.63E-04
17	0.002521844499	0.00251984886	0.002164064102	
18	0.001984152456	0.002016204052	0.00178569751	
19	0.001560607003	0.001611451824	0.001473577575	
20	0.001228053636	0.001290864896	0.001216042686	
21	9.66E-04	0.001032751477	0.001003554933	
22		8.28E-04	8.28E-04	

2. **Page Rank distribution (First 8 ranks)**



	100MB	1GB	10GB	100GB
1	8.888280354967823E-4	0.003447238130466023	0.004896891311214256	0.009670143160526364
2	4.223437734715658E-4	0.002106393188633645 4	0.001625665860348684 5	0.001798016926295063
3	3.815694476124404E-4	0.001861802004941410 6	0.001184945026717184 2	0.001501417928853476 9
4	3.789874350553408 3E-4	0.001386337386295 91	0.001042068983944 2218	0.001478768407235 1517
5	3.508066149914622E-4	0.001362979982843 5844	0.001015744995951 4338	0.001414288730832 9422
6	3.376296078883774 3E-4	0.001211756040331 1636	8.42089441628161E-4	0.001384272833395 43
7	3.225017659355928 3E-4	0.001113152255539 5976	7.759957357570838E-4	0.001294008066456722
8	3.18662309597462E-4	0.001042568886959 6746	7.036529078622806E-4	0.001087927480424 2404

3. **Compare two implementations**

   演算法基本上在這兩個不同的實作方式是一樣的，但是在實作上有很大的不同。Map-redeuce我要自己處理過度資料，比如說先Parse資料完後才會去把圖建出來，這裡做了兩次Map-reduce，實際上處理的方式是第一次Map-reduece會先把檔案寫出去，然後第二次Map-redeuce把第一次輸出的結果吃進來。在沒辦法重新利用這些資料時，在Disk I/O上面花費很大量的成本。所以Spark解決一部分這個問題，可以把Parse完的資料先暫時放在Memory，用rdd可以直接交給建圖的，然後再交給iteration，在跑50G資料時明顯比Map-Redeuce的iteration快很多。50GB Data Size 速度差距大概是 $Spark : MR = 34Minutes : 900Minutes$

Experience and Conclusion

在寫程式的過程，先假設語言的熟悉度都是一樣的話，我覺得寫Spark比Mapreduce輕鬆的多，Mapreduce做iteration等於是要把檔案寫出去再讀回來，他的架構很難保存一次Map-reduce後的狀態，要重新利用這些資料等於是要先「自己」暫時寫出去，然後再讀回來，整個就很浪費時間。再者，map-reduce在處理圖問題就是超級麻煩，由於iteration一個stage的cost很大，等於是說我要減少一次iteration Map-reduce的次數，我要同時知道各種資訊（zero degree Page rank 或 誤差總和)在map-reduce上很難設計。而且Map-reduece做iteration我需要依賴shell script去執行不同單元的Map-reduce，把所有不同階段的Map-reduce寫在同一份Code裡面會讓整個設計變的更複雜。Spark解決掉這些問題，但是我對於scala 的熟悉度並不是那麼高，一些Functional programming 的用法不是很熟悉，所以在寫Spark時非常的痛苦，因此在寫Scala時我使用Java的Feature比較多，讓整個程式好寫一點。總而言之，Spark適合拿來處理圖計算以及iteration的計算，Map-reduece就暫時擱置。