

# Cloud Programming HW2

## Inverted Index

### Instructions

---

```
sh compile.sh          // compiles Java codes
sh execute.sh          // generates inverted index table through MapReduce, and waits for queries
```

### Query Format

---

A white space specifies OR operation. (lowest priority)

A '+' character specifies AND operation.

A '~' character specifies NOT operation. (highest priority)

For example, **A B+~C** is parsed as **A or (B and (not C))**.

\*\*\* 除了 OR 用空白表示，其他部分不能有空白

### Algorithm Design

---

#### • Build Inverted Index Table - Phase 1

1. 定義 WordFilePair class，將單字與其檔案存成一對
2. 定義 WordEntry class，包含單字出現的檔案、offsets、還有 term frequency

<b>WordFilePair</b>	<b>WordEntry</b>
<ul style="list-style-type: none"> <li>- String fileName</li> <li>- String word</li> </ul>	<ul style="list-style-type: none"> <li>- String fileName</li> <li>- ArrayList&lt;Long&gt; offsets</li> <li>- double termFrequency</li> </ul>

3. Mapper (Input: LongWritable, Text ; Output: WordFilePair, WordEntry)
  - 利用 reporter 的 getInputSplit() 取得檔案名稱
  - 利用 regular expression 將讀進來每一行的單字取出，並存成一個新的 WordFilePair，此 WordFilePair 即為 Mapper 的 output key
  - 用 regular expression 取出單字時可藉由 matcher.start() 知道這個單字在這一行的 offset，所以將這個 offset 加上 Mapper 的 input key 就可以得到此單字在整份檔案裡的 offset，把算好的 offset 存入一個新的 WordEntry，此 WordEntry 即為 Mapper 的 output value
  - 利用 reporter.incrCounter() 為每個檔案新增一個 counter，並在每次讀到一個單字時去遞增，這樣就可以記錄每個檔案的總字數
4. Partitioner (Input: WordFilePair, WordEntry)
  - 依照每個 WordFilePair 單字的第一個字母去分配，Aa~Gg 開頭的 WordFilePair 分給第一個 combiner，其餘分給第二個 combiner

5. Combiner (Input: WordFilePair, WordEntry; Output: WordFilePair, WordEntry)
  - 將出現在同份檔案中，相同單字的所有 offset 收集到一個新的 WordEntry
6. Reducer (Input: WordFilePair, WordEntry; Output: WordFilePair, WordEntry)
  - 將出現在同份檔案中，相同單字的所有 offset 收集到一個新的 WordEntry
  - 從 Mapper 的 reporter 抓出每個檔案的總字數分別計算 term frequency，即 offset 的總數除以檔案總字數
  - 將 offset 由小到大排序
7. Phase 1 Result
  - Phase 1 結束後會產生以下格式的檔案，即對於每個 WordFilePair 輸出其 term frequency 以及 offsets

```

Although 1kinghenryvi 1.2619358095318218E-4 [124097,128680,130942]
Although 2kinghenryvi 1.8040772145047808E-4 [23462,37913,58636,79858,86219]
Although 3kinghenryvi 3.719546215361726E-5 [47753]
Although allswellthatendswell 3.977408320738207E-5 [52380]
Although asyoulikeit 4.274965800273598E-5 [52058]
Although coriolanus 3.3155399356785255E-5 [135782]
Although cymbeline 6.659563132658497E-5 [2102,164049]
Although loveslabourslost 4.2468254979402894E-5 [28844]
Although periclesprinceof tyre 9.841551028442082E-5 [20496,110943]
Although sonnets 2.743785326236075E-4 [22172,24998,49982,81132,85434]
Although various 2.77623542476402E-4 [255]
Although winterstale 3.7250884708511824E-5 [43550]
```

• Build Inverted Index Table - Phase 2

1. 定義 TableEntry class，記錄對於某單字的所有 WordEntry 並存成陣列，即最終 inverted index table 的格式

<b>TableEntry</b>
- ArrayList<WordEntry> entries

2. Mapper (Input: LongWritable, Text; Output: Text, TableEntry)
  - 讀取 Phase 1 的輸出結果，並轉換成 TableEntry 的形式，此時每個 TableEntry 裡的陣列都只會有一個 WordEntry
  - 將單字當作 output key、TableEntry 當作 output value
3. Partitioner (Input: Text, TableEntry)
  - 依照每個 WordFilePair 單字的第一個字母去分配，Aa~Gg 開頭的 TableEntry 分給第一個 combiner，其餘分給第二個 combiner
4. Combiner (Input: Text, TableEntry; Output: Text, TableEntry)
  - 收集屬於同一個單字的 WordEntry，然後將其串成 TableEntry
5. Reducer (Input: Text, TableEntry; Output: Text, TableEntry)
  - 收集屬於同一個單字的 WordEntry，然後將其串成 TableEntry

- Retrieval

1. 定義 FileRankEntry class，記錄每個檔案的名稱、分數，以及一個 HashMap 去存查了什麼單字，還有這個單字在這個檔案裡的所有 offsets

<b><i>FileRankEntry</i></b>
<ul style="list-style-type: none"> <li>- String fileName</li> <li>- double weight</li> <li>- HashMap&lt;String, ArrayList&lt;Long&gt; map</li> </ul>

2. Load Table

- 先將 inverted index table 讀進 HashMap<String, ArrayList<WordEntry>> table，這樣就可以很快找到一個單字所在的檔案及其他資訊
- 在讀 table 的同時，將所有讀到的檔案名稱存入一個 HashSet<String> fileSet，可以得到計算TF.IDF時需要的檔案總數

3. Retrieval Algorithm

- 將每個 query 先依照空白字元拆解，即可得到需要做聯集的單字或單字群
- 再將需要聯集的單字或單字群依照 '+' 字元拆解即可得到需取交集的單字
- 對於每個開頭為 '~' 字元的單字做 negation()，會回傳一個 FileRankEntry 的集合
- 對於每個開頭沒有 '~' 字元的單字的所有檔案，分別新增 FileRankEntry，並計算 TF.IDF 然後將 offsets 存入，把此 FileRankEntry 加入一個暫時的集合
- 將這個暫時的集合跟 maintain 好的 FileRankEntry 集合再取交集，即可得到許多子交集，最後再將這些子交集做聯集，得到最終的輸出集合

<b><i>Retrieval Algorithm</i></b>
<pre> <b>for</b> each query {     <b>if</b> query.equals("q") <b>terminate</b>     split query into components to do union     <b>for</b> every union component U {         split U into components to do intersection         <b>for</b> each intersection components I {             <b>if</b> I needs negation                 tmpSet = negation(I)             <b>else</b>{                 <b>for</b> every file of I {                     <b>create</b> new FileRankEntry entry                     calculate TD.IDF                     update weight and offsets of entry                     tmpSet.put(entry)                 }             }             iSet = intersection(iSet,tmpSet,I)         }     }     uSet = union(uSet,iSet) } </pre>

## 4. Negation Algorithm

- 先新增一個包含所有檔案 FileRankEntry 的集合，再將需要 negation 的單字的所有檔案從這個集合中刪除，回傳剩下的集合

```

Negation Algorithm

negation(String word) {
    for each file in fileSet {
        create new FileRankEntry entry
        nSet.put(entry)
    }
    for each file of word
        nSet.remove(file)
    return nSet
}

```

## 5. Intersection Algorithm

- 對於兩個集合中的所有 FileRankEntry 找屬於相同檔案的 entries，然後將其中一個 entry 的 TF.IDF 以及 offsets 加到另一個 entry 上，然後放進要回傳的集合中

```

Intersection Algorithm

intersection(Set set1, Set set2) {
    for each FileRankEntry e1 in set1 and e2 in set2 {
        if e1 and e2 are the same file {
            add weight of e2 to e1
            append offsets of e2 to e1
            iSet.put(e1)
        }
    }
    return iSet
}

```

## 6. Union Algorithm

- 對於每個存在於 set1 的 FileRankEntry，如果也存在於 set2，則選擇 TF.IDF 較大的 FileRankEntry 並把它加入回傳的集合；如果不存在於 set2，則直接加入回傳的集合
- 對於每個存在於 set2 的 FileRankEntry，如果不存在於 set1，則加入回傳的集合

```

Union Algorithm

union(Set set1, Set set2) {
    for each FileRankEntry e1 in set1 {
        if e1 also exists in set2
            choose the entry with greater weight and put it into uSet
        else uSet.put(e1)
    }
    for each FileRankEntry e2 in set2
        if e2 does not exist in set1
            uSet.put(e2)
    return uSet
}

```

## 7. Retrieval Result Format

- 輸出前十個最高分的檔案
- 對每個單字，輸出前五個 fragment，依照 offsets 遞增排列
- 對每個 fragment，從其 offset 前五個字元開始總共 25 個字元，且將其中的換行字元取代成空白

```

query      → About+Above+about+above
number of total files retrieved → 6 files retrieved
*****
rank  file name  score → Rank 1  juliuscaesar  score = 2.684516620069106E-4
-----
word in query →
fragments {
    about :
        fragment 1: men about the streets?
        fragment 2: I'll about,          And drive aw
        fragment 3: peep about To find ourse
        fragment 4: men about me that are fa
        fragment 5: lk'd about the streets,
    above :
        fragment 1: soar above the view of me
        fragment 2: ting above.          [A crowd o
        fragment 3: mell above the earth      Wit
    Above :
        fragment 1: RUS [Above]  O my lord!
        fragment 2: RUS [Above]  Titinius is
    About :
        fragment 1:      About the ninth hour
        fragment 2: all      About his funeral: a
        fragment 3: nge! About! Seek! Burn! F

```

## Questions

Q1:

我用了兩個 phase 去實作 MapReduce，第一個 phase 是根據每個 WordFilePair 當作 key 去處理，主要是在計算每個單字在不同檔案裡的 term frequency，而最後會輸出一個檔案，其內容也是依照 WordFilePair 排列，所以同個單字會有很多筆屬於不同檔案的資料。第二個 phase 則是去讀 phase 1 的輸出結果，只用單字當作 key 去做整理，所以最後的結果就會是每個單字只有一筆包含其所有檔案的資料，也可以從每筆資料的長度得知各個單字的 document frequency。

如果只要用一個 phase 就完成的話，就要在 Reducer 裡面把屬於同個單字的所有 WordFilePair 收集起來然後最後以單字當 key 輸出。但這樣就必須要在一個 phase 裡面算完 term frequency 跟 document frequency，process 之間的 dependency 就可能會變高，平行的效能也會降低，因為一個 term frequency 與其他的 WordFilePair 之間沒有關係，所以一對 WordFilePair 處理完就可準備輸出。但是如果同時計算 document frequency，就必須等其他有相同單字的 WordFilePair 處理完才能計算 document frequency，然後才可以輸出。

因為這次作業的 input 量很小，所以可能看不出實際的影響，但是當檔案數量及單字數量大到某個程度的時候，Reducer 也不一定可以存這麼大量的資料，只用一個 phase 的方法其效能可能就會不及兩個 phase 的方法。雖然只有一個 phase 的方法可以少做一次檔案 I/O，但此瓶頸最終仍會被無法完全平行的問題所影響。

Q2:

我實作的 extension 包括 AND, NOT，還有這些運算的組合，較困難的部分就是把 union、intersection、negation 的運算以及回傳值分離清楚。我原本讓所有運算都共用一個 FileRankEntry 的集合，所以所有運算都會修改同一個集合，這樣在計算  $A+B$  的時候由於所有 A 跟 B 的檔案都會算出 TF.IDF，雖然輸出的時候只取交集，可是如果搜尋  $A+B \sim B$  且當  $A+B$  的檔案數不足十個的時候，就會把其他包含 A 的檔案輸出。另外當搜尋  $A+B \ B+C$  時，B 的檔案的 TF.IDF 會被重複計算，所以分數排名會出錯。這些問題都是因為原本的寫法讓這些運算之間產生依存性，例如在 negation 裡面又做了 intersection 或 union，而且都使用同一個集合，運算的架構很混亂，最終結果也就容易出錯。後來將這些運算完全分離，在過程中也使用暫時的集合去處理計算出來的檔案，才解決這個問題。