# TI2736-B: Assignment 2 Big Data Processing

Wing Nguyen, 4287118

November 27, 2015

## 1. Data streams

(a)
- i Sample (employer, department)
- ii Sample (employer, employeeID)
- iii Sample (employer, department)
- iv Sample (employer)
- v Sample (employer, employeeID)

(b) Because of the way $h_1$ and $h_3$ work, we will get estimations of 1 unique item ($2^0$) if we choose the tail to be the number of 0's or 1's. I've defined the hash tails here to be the number of 1's because that leads to better results (for $h_1$ and $h_2$) than using 0's. But generally, the probability of finding a tail should be the same whether we've defined the tail to be 1's or 0's.

- $h_1$: max tail = 3, $2^3 = 8$, estimation: 8 unique items.
- $h_2$: max tail = 2, $2^2 = 4$, estimation = 4 unique items.
- $h_3$: max tail = 0, $h^0 = 1$, estimation = 1 unique item.

Table 1: Results of hash functions

| $h_1(x) = 2x + 1 \, mod \, 32$ |  | $h_2(x) = 3x + 7 \, mod \, 32$ |  | $h_3(x) = 4x \, mod \, 32$ |  |
|---|---|---|---|---|---|
| 7 | 00**111** | 16 | 10000 | 12 | 01100 |
| 3 | 00011 | 10 | 01010 | 4 | 00100 |
| 9 | 01001 | 19 | 100**11** | 16 | 10000 |
| 3 | 00011 | 10 | 01010 | 4 | 00100 |
| 11 | 01011 | 22 | 10110 | 20 | 10100 |
| 19 | 10011 | 2 | 00010 | 4 | 00100 |
| 5 | 00101 | 13 | 01101 | 8 | 01000 |
| 13 | 01101 | 25 | 11001 | 24 | 11000 |
| 11 | 01011 | 22 | 10110 | 20 | 10100 |

Table 2: The data stream from slide 28, lecture 3

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | b | d | a | c | d | a | b | d | c | a | a | b |

| At **Position 3:** | At **Position 8:** | At **Position 13:** |
|---|---|---|
| $X_1$ = (c, 1) | $X_1$ = (c, 2) | $X_1$ = (c, 3) |
| | $X_2$ = (d, 1) | $X_2$ = (d, 2) |
| | | $X_3$ = (a, 1) |

(c) i The formula for calculating the $k^{th}$ order moments: $n \times (v^k - (v-1)^k)$.
So, for the $3^{rd}$ order moment, we apply formula: $n \times (v^3 - (v-1)^3)$.

- estimate from $X_1$: $15 \times (3^3 - (3-1)^3) = 285$
- estimate from $X_2$: $15 \times (2^3 - (2-1)^3) = 105$
- estimate from $X_3$: $15 \times (1^3 - (1-1)^3) = 15$

$AVG(X_1, X_2, X_3) = 135$
True value: $m_a^3 + m_c^3 + m_d^3 = 5^3 + 3^3 + 3^3 = 179$

Table 3: The data stream from slide 28, lecture 3

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | b | d | a | c | d | a | b | d | c | a | a | b |

| At **Position 2:** | At **Position 3:** | At **Position 8:** | At **Position 9:** | At **Position 10:** |
|---|---|---|---|---|
| $X_1$ = (b, 1) | $X_1$ = (b, 1) | $X_1$ = (b, 2) | $X_1$ = (b, 2) | $X_1$ = (b, 3) |
| | $X_2$ = (c, 1) | $X_2$ = (c, 2) | $X_2$ = (c, 2) | $X_2$ = (c, 2) |
| | | $X_3$ = (d, 1) | $X_3$ = (d, 1) | $X_3$ = (d, 1) |
| | | | $X_4$ = (a, 1) | $X_4$ = (a, 1) |

ii
- estimate from $X_1$: $15 \times (3^3 - (3-1)^3) = 285$
- estimate from $X_2$: $15 \times (2^3 - (2-1)^3) = 105$
- estimate from $X_3$: $15 \times (1^3 - (1-1)^3) = 15$
- estimate from $X_4$: $15 \times (1^3 - (1-1)^3) = 15$

$AVG(X_1, X_2, X_3, X_4) = 105$
True value: $m_a^3 + m_b^3 + m_c^3 + m_d^3 = 5^3 + 4^3 + 3^3 + 3^3 = 243$

(d) i Because bucket sizes are powers of 2, and $N = 22$, the largest possible bucket size would be $2^4 = 16$.

ii - Bucket b is the bucket with the earliest time stamp that includes some of the k most recent bits: **left most bucket in this case**, size(b) = 4.

- The sum of 1's in the buckets to the right of b: 8.
- Estimation: $8 + size(b)/2 = 8 + 2 = 10$

iii After adding the first 1 bit to the window, because it is a 1, a new bucket is created. Now there are 3 buckets so we merge the 2 existing size-1 buckets into 1 size-2 bucket. Now there are 2 size-2 buckets. The most recent size-4 bucket is still unchanged, but the oldest size-4 bucket is dropped because there aren't 4 1's in the bucket any more. See table 5.

By repeating this process for the rest of the bits from 1 0 1 1 1 0 0 1, we get the bucket configuration as shown in Table 6.

Table 4: Initial window and bucket configuration

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |

Table 5: After adding a 1...

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |

Table 6: After adding the whole sequence 1 0 1 1 1 0 0 1.

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

iv - Bucket b is the bucket with the earliest time stamp that includes some of the k most recent bits: in this case, it is the **4th bucket from the right**, size(b) = 4.

- The sum of 1's in the buckets to the right of b: 5.
- Estimation: $5 + size(b)/2 = 5 + 2 = 7$

v See table 7. The numbers are converted into binary integers and placed in a table with the most recent integers on top. The colored cells are integers within k = 3.

Table 7: Extension to DGIM: positive integers, k = 3

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **56** | - | 1 | 1 | 1 | 0 | 0 | 0 |
| **99** | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| **9** | - | - | - | 1 | 0 | 0 | 1 |
| **13** | - | - | - | 1 | 1 | 0 | 1 |
| **5** | - | - | - | - | 1 | 0 | 1 |
| **77** | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| **2** | - | - | - | - | - | 1 | 0 |
| **125** | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| | $c_6$ | $c_5$ | $c_4$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ |

We apply: $\sum_{i=0}^{m-1} c_i 2^i =$
$2 \times 2^0 + 2^1 + 2 \times 2^3 + 2^4 + 2 \times 2^5 + 2^6 = 164$

## 2. Hadoop

**The repo can be found at:** `https://github.com/codesalad/hahadoop`
The source codes are in the **/src** folder. The output of the filtered list and the reversed index are in the **/hadoopout** folder.

(d) The combiner doesn't always run, that's for Hadoop to decide (in this case, I've commented/uncommented a line of code suggesting Hadoop to use the combiner). Below is a comparison of having the combiner off and on:

**Combiner off**

```
// job.setCombinerClass(Reduce.class);
...
Combine input records=0
Combine output records=0
Reduce input records=88674
Reduce output records=10530
```

**Combiner on**

```
job.setCombinerClass(Reduce.class);
...
Combine input records=88674
Combine output records=14940
Reduce input records=14940
Reduce output records=10530
```

We can see that with the combiner on, the amount of Reduce input records decreased significantly (83% decrease). This will save network usage between machines on a large cluster, so a combiner makes the job much more efficient.

(e) The amount of unique terms are all the terms processed by the WordCount class, the number of lines in part-r-00000 are the number of unique terms: 10971. However, the output file does not really contain 10971 unique terms since the wordcounter also counts numbers, special signs and such. So the number of unique terms should be lower. Hamlet occurred 115 times.

(f) The three changes made to the tokenizer were: filtering out numbers [0-9], filtering out "all-caps" words, filtering out "words" consisting of special characters such as [, . / ! @ # $ % & ...] using regular expressions. The pattern created in Java can be seen below, the word will only be counted and written to file if it's not in all-caps, don't contain or aren't special signs, don't contain or aren't numbers. The words with capital letters are also converted into lowercase letters so 'Hamlet' and 'hamlet' will be the same word.

```
private static final Pattern FILTER = Pattern.compile("([A-Z][a-z]+)|([a-z]\\w+)");
...
Matcher nospecials = FILTER.matcher(word);
   if (nospecials.find()) {
      currentWord = new Text(word);
      context.write(currentWord,one);
   }
```

The amount of unique items after filtering out unwanted terms is: 9118. This makes a difference of 1853 unique words, which is quite a big difference. The word 'hamlet' occurred 119 times in total.

(g) I've created a duplicate of the WordCount.java class of which the map() reduce() functions were modified to implement 'reverted index'. In the map() method, the words parsed are used as keys and the file names are used as values. In the reduce() method, I'm using a string builder to create a string containing all the file names the words are in. The only problem I see with this implementation is that the stringbuilder may cause trouble when using a very large corpus because we might get a lot of file names. **See the /hadoopout folder in the repository for the reversed index output**. Example of 1 output line: *above (pg1524)(pg1112)(pg2267).*