# Document On Web Report

10406141 Wenchang Liu

## Functionality

### Implemented Features:

1. Inverted Index
   It is often convenient to index large data sets on keywords, we build inverted indexes in advance, so that searches can trace terms back to records that contain specific values quickly.
2. Stemming
   We remove suffixes and possibly prefixes before indexing, collapsing similar forms to a canonical form.
   For grammatical reasons, documents are going to use different forms of a word, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.
3. Stopword
   We remove some extremely common words (stopwords) before indexing to lower the dimensional space without affect the analysis.
4. In-mapper Aggregation
   Since reducer has heavier workload, we do some summarisation inside mapper, so that we can reduce the workload of summarisation and reduce which results in performance enhancement.
5. Case Folding
   I think to get all terms to be lower case is a practical solution. Although we can try to get capitalization as correct as possible, it might not be helpful because users may usually use lowercase regardless of the right case of words.
6. Positional Index
   We can still use single-word index terms for tokenization, and trying to solve the multi-word queries by using positional index.
   Users may pose a query which is a multi-word phrase that would like to be treated as a whole phrase. To be able to support such queries, we may propose a positional index solution in which we record the position a term appear in the corresponding document instead of just recording the name of the document so that terms that are closer together score higher during retrieval procedures.
7. Tf-idf Scores
   Tf-idf scores combine term frequency and document frequency, representing how well a term can represent a specific document. With the scores, we can do document ranking for a query and present the most related document to the users.

### Limitations and Problems:

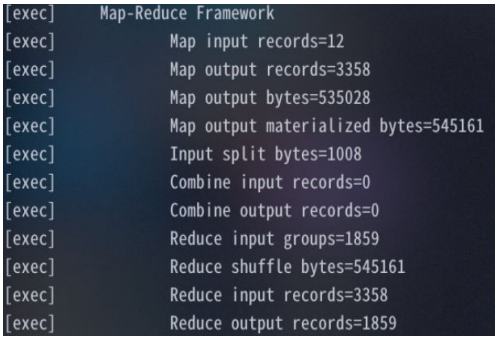
1. Tf-idf:
    a. Tf-idf can be slow to compute regarding large vocabulary.
    b. Tf-idf assumes that counts for each words contribute independently to similarity which may not be true.
    c. Tf-idf makes no use of semantic similarities among words.
2. Filtering terms: I filtered out all non-alphabetical terms like exercise 1.1, this can be harmful if the query contains those terms.
3. Tokenization: I use space to split the document, which ignore the impact of phrases. Using positional index can alleviate this problem.
4. Positional indexing: if documents were bigger than a file split, the position may be incorrect since we cannot know the exact order of the file splits. However, this problem does not occur in this exercise and can be solved by using a global hashmap to store the positions of each word, but it can be inefficient.
5. Case folding: ignoring proper nouns or names may have negative effects on retrieval results, because they have different meanings but it is hard to come up with a perfect solution on this one.
6. In-mapper aggregation: usage of hashmaps increases space complexity.
7. Stopwords: the stopwords list is important, inappropriate stopwords removal may have negative effects on results. (e.g. if we remove the word ‘first’, we are not able to find such query like first episode.)
8. Stemming: there might be stemming errors including understemming fails to conflate related forms (e.g. divide -> divid, division -> divis) and overstemming conflates unrelated forms (e.g. neutron, neutral -> neutr)

## Performance

In our case, we cannot see significant time improvements or limitations due to the small scale of our documents data, so we analyse in theory or use other evidence (e.g. written bytes).

### Design Patterns and Efficiencies:

1. Inverted Index Summarizations Pattern: allowing faster searches or data enrichment capabilities, including using in-mapper aggregation.
   In-mapper aggregation can decrease the workload of the reducer part, in our case, we do see mapper produces less which can enhance performance in a large project.



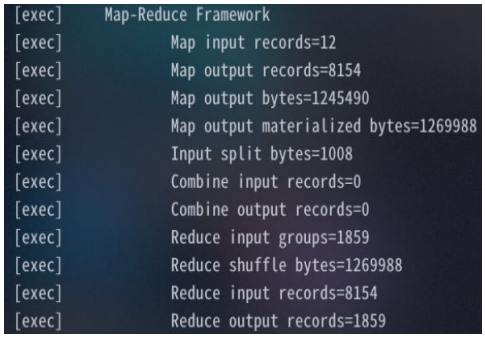



Figure 1, 2: with(left) vs. without(right) in-mapper aggregation

2. Filtering Pattern: we remove the records that we are not interested in, for example, we filtered out stopwords because they are not helpful in our case, we can see that the program writes less with stopwords which can simplify the task and improve the performance especially for large documents.

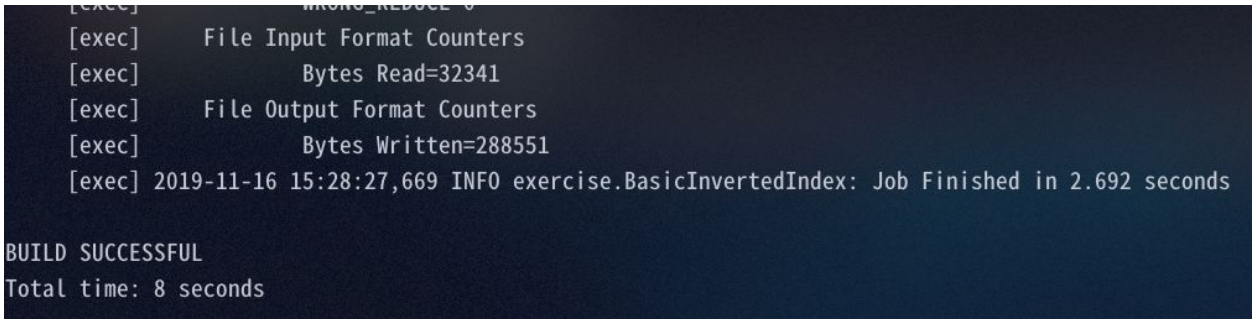



Figure 3: filtered stopwords

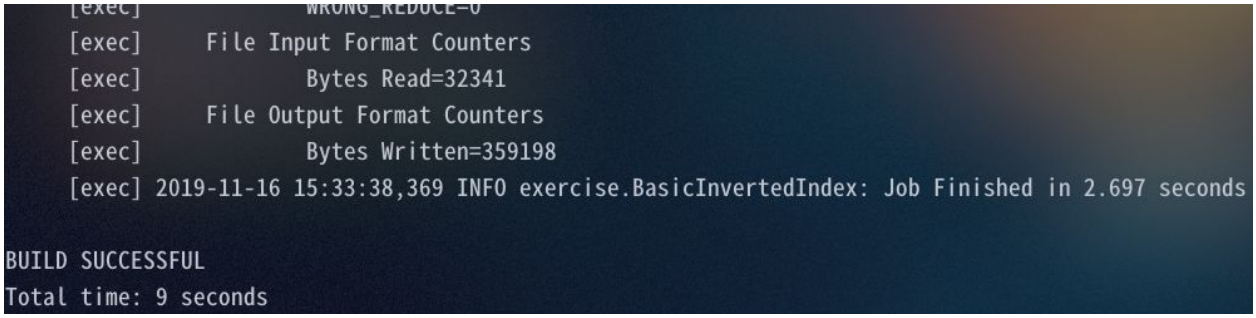



Figure 4: without filtering stopwords

3. Structured to Hierarchical Pattern: in our case, our output including nested hashmap and lists, which are kind of hierarchical. This allows us to have a reducer output containing a lot of information.

## Performance Limitations:

1. Tf-idf requires a lot more computations which may limit our performance, and also increases the burden of I/O. We can compare Figure 5 with Figure 3:

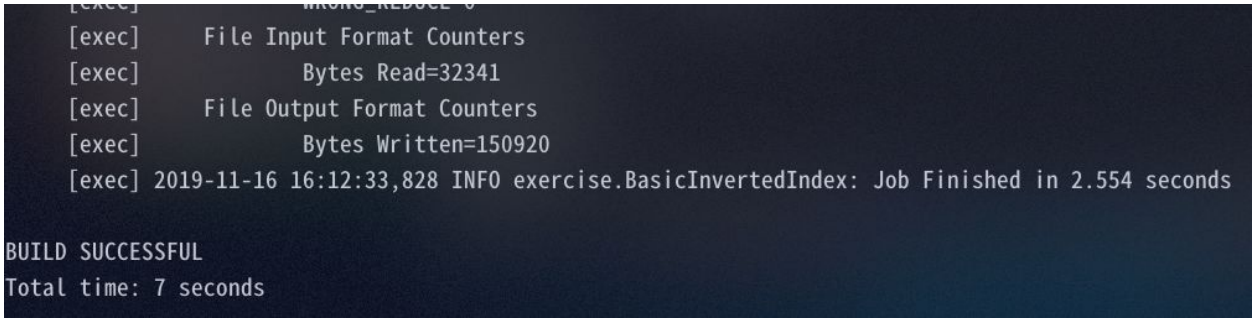



Figure 5: without calculating tf-idf

2. Positional indexing increases the workload, for example, we need to build a new hashmap structure for each term with filename, position list pairs inside.

# Results and Evaluation

## Output Fragment:

As the fragment screenshot from the output file showing below, each line in the output file is an index, including:

- ❏ The term in the front of the line.
- ❏ A hashmap corresponding to the term, each pair in the hashmap is the filename and a list of statistics about the appearance of the term in that document.
- ❏ The list of statistics has 5 entries, term frequency, document frequency, inverted document frequency, tf-idf, position index list respectively.

```
9  accept  {Bart_the_Lover.txt.gz=[1, 2, 0.47712125471966244, 0.47712125471966244, [1392]], Bart_the_Genius.txt.gz=[1, 2, 0.47712125471966244, 0.47712125471966244, [308]]}
10 accident    {Bart_the_Fink.txt.gz=[1, 3, 0.3010299956639812, 0.3010299956639812, [266]], Bart_the_Mother.txt.gz=[3, 3, 0.3010299956639812, 0.44465780490343443, [350, 574, 1163]],
   Bart_the_Murderer.txt.gz=[1, 3, 0.3010299956639812, 0.3010299956639812, [305]]}
11 accord  {Bart_the_Lover.txt.gz=[1, 1, 0.7781512503836436, 0.7781512503836436, [1095]]}
12 account {Bart_the_Fink.txt.gz=[3, 1, 0.7781512503836436, 1.1494237513283618, [466, 479, 1290]]}
```

Figure 6: fragment of output

## Simple Evaluation on Tf-idf:

I also write a simple python script to read in the output file and output the top N scores terms of each document to check whether the tf-idf scores are making sense.
For example, we take the output of Bart_the_Mother as an example:

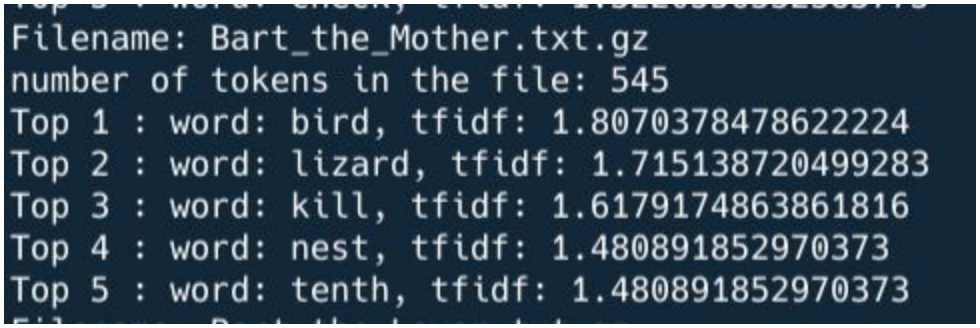



Figure 7: fragment of evaluation output

According to the definition of tf-idf, the results show the top 5 words that can best represent the document “Bart the Mother”. And we can search in Google to see whether the terms can represent “Bart the Mother”to a certain extent:

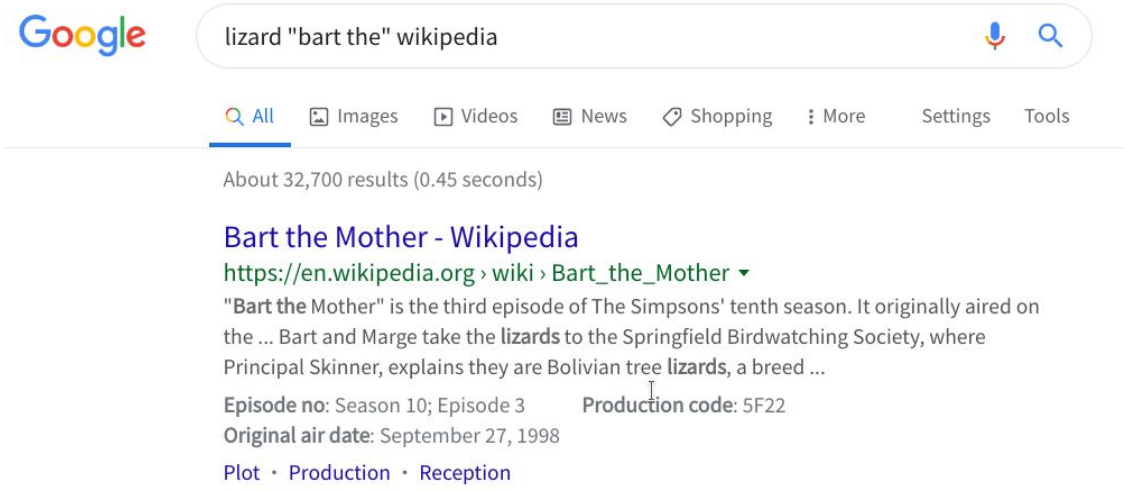



Figure 8: result from Google