CS-GY 6313: Web Search Engines

Assignment 2: Indexer

Vikram Sunil Bajaj (vsb259)

My code for this assignment is organized into 4 Python files:

- download_wet_files.py
- generate_page_table.py
- generate_postings.py
- index_construction.py

This document explains the internal working of these 4 Python files, as well as their expected outputs and run times.

Platform: Windows 10

---

download_wet_files.py

Goal

The aim of this program is to download the data for this assignment.

Working

The user must first download the WET files file list (CC-MAIN-2018-39/wet.paths.gz) from http://commoncrawl.org/2018/10/september-2018-crawl-archive-now-available/ and extract it.

This code will then download the first 50 WET files and store them in a new directory called **wet_files**.

---

generate_page_table.py

Goal

The aim of this program is to generate the page table (a.k.a URL table)

Working

The page table consists of the URLs for the pages in the WET files, along with the number of terms the page contains. It is indexed by the doc_id (0, 1, 2, 3, …).

Therefore, every entry in the page table has URL, # terms

(doc_id is assigned in the order in which the pages were crawled and are being parsed).

This program saves the page table in a text file **page_table.txt** in a folder called **page_table**.

Output File Size

page_table.txt is **148 MB**.

Time Taken

This program generates the page table for the pages in the 50 WET files in 10 minutes. There are about 2 million pages.

---

generate_postings.py

Goal

The aim of this program is to generate postings of the format *term, doc_id, freq*

It generates 50 **sorted** posting files (one per WET file), and then combines them into one large **final_postings.txt** file using **Unix sort**.

Working

For each WET file, read in compressed format (.gz), the pages in the WET file are parsed.

For each page, terms containing only ASCII characters (with ASCII codes between 33 and 126, since 0-31 are control characters, 32 is space and 127 is another non-printable character) are retained. Punctuation is also removed.

Each posting has term, doc_id and frequency of occurrence of the term in the corresponding doc. Postings are sorted by term and doc_id before writing them to a postings file (Python sort is used here). This results in 50 sorted postings files, one per WET file. These are stored in a folder called **sorted_postings**.

Finally, Unix sort is used through Python (using the new bash shell in Windows 10 via an os.system() call) to merge these 50 files. This generates a large **final_postings.txt** file in a folder called **final_postings** within sorted_postings.

Sort Command: sort -m -k1,1 -k2n,2 --buffer-size=1024 *.txt > final_postings.txt
-m: perform only merge (since the files are already sorted)
-k1,1 -k2n,2: sort first by term and then numerically by doc_id
--buffer-size: limiting the amount of RAM that can be used

Output File Size

final_postings.txt is **10.1 GB**. It contains around 0.5 billion postings.

Time Taken

The program takes about **3 hours** to generate the 50 sorted intermediate postings files. This is an average of 185 documents per second.

It then takes **25 minutes** to merge these files into a single postings file (with a buffer size of 1024 MB).

---

index_construction.py

Goal

The aim of this program is to build the inverted index (as well as the lexicon).

Working

The code generates an inverted index (compressed using varbyte encoding) and the lexicon, from the final_postings.txt file.

Every entry in the lexicon consists:
term, start_location_of_inverted_list_in_inverted_index, num_of_docs_containing_term

It is implemented using a Python dictionary (with terms as keys) and is stored in a text file called **lexicon.txt**.

For each term, the inverted index contains a set of varbyte-encoded *doc_id gaps* followed by a set of varbyte-encoded *frequencies*. This forms the inverted list for the term. It's start location in the inverted index is obtained by a tell() call before it is written to the inverted index. This is stored in the lexicon.

The compressed inverted index is stored in **binary** format in a file called **inverted_index.dat**.

Output File Sizes

lexicon.txt is **378 MB**.
inverted_index.dat is **1.22 GB**.

Time Taken

This program generates the compressed inverted index and the lexicon in **45 minutes**.

Summarized Design Decisions

- 50 WET files considered
- Intermediate posting files are **individually sorted** to speed up merge
- The intermediate sorted posting files are not in binary format
- They are merged using the **Unix sort** command (with -m)
- The Page Table is implemented as an array indexed by doc_id, and stored in **page_table.txt**
- The Lexicon is implemented using a Python dictionary, and stored in **lexicon.txt**
- The Page Table and Lexicon are not in binary format

- The Inverted List (for each term) contains all the doc_id gaps (**varbyte-encoded**) followed by all the frequencies (also varbyte-encoded)
- The Inverted Index is stored in **binary** format in a file called **inverted_index.dat**

Output File Sizes Summary

Combined Postings File

final_postings.txt: 10.1 GB

Page Table

page_table.txt: 148 MB

Lexicon

lexicon.txt: 378 MB

Inverted Index

inverted_index.dat: 1.22 GB

Time Taken Summary

- **Time to generate page table**: 10 min
- **Time to generate intermediate sorted postings files**: 3 hours
- **Time taken to merge intermediate files into a combined postings file**: 25 min (with buffer size: 1024 MB)
- **Time taken to generate lexicon and compressed inverted index**: 45 min

---

Known Limitations

- Intermediate posting files are not in binary format
- Overall speed can be improved

Note: Developer mode needs to be enabled in Windows 10 (to use the new bash shell)