CS-GY 6313: Web Search Engines

Assignment 2 and 3: Indexing and Query Processing

Vikram Sunil Bajaj (vsb259)

Introduction

This report deals with the process of **inverted index generation** and **query processing**.

The data used in these assignments was obtained from the CommonCrawl data set. **150 WET files** were processed. Each WET file contains several pages/docs. This resulted in the use of **5,906,499 documents/pages**.

Before an inverted index can be created, **postings** must be generated. A posting is of the format: *term, doc_id, frequency*

The following are the Python files for the assignments:

- **download_wet_files.py**
  This downloads the 150 WET files and stores them in a directory called wet_files
- **generate_page_url_table.py**
  This generates the page table/URL table (url_table.txt) from the 150 WET files
- **generate_postings.py**
  This generates postings from each of the 150 WET files. It generates 150 sorted postings files and then combines them into a final_postings.txt file using Unix merge
- **index_construction.py**
  This generates the inverted index and the lexicon, from final_postings.txt
- **generate_docs_table.py**
  This creates an SQL table called docs that stores the page text for each URL, extracted from the WET files
- **query_processing.py**
  This performs query processing (conjunctive and disjunctive) in a Flask-based web app

These files are discussed in detail in the upcoming sections.

There are 4 main files:

- **Inverted Index (inverted_index.dat)**
  It is a large file **(3.74GB)** that contains **varbyte** encoded doc_id gaps and frequencies for each term, in binary format. These are called inverted lists. The inverted index is generated by index_construction.py.
- **Lexicon (lexicon.txt)**
  This file contains one entry per term, in the format:
  term, start_pos, num_docs

where start_pos is the start location of the inverted list for the term in the inverted index, and num_docs is the number of docs that contain the term.
This file is **848MB** and is generated during inverted index construction, by index_construction.py

- **URL Table/Page Table (url_table.txt)**
  This file contains one entry for each doc/page, in the format:
  doc_id, URL, page_size
  where page_size is the number of terms contained by the page.
  It is **494MB** and is generated using generate_page_url_table.py
- **web_search_engine.db**
  This is a database that contains a table called **docs**. This table stores the page text for each doc, extracted from the WET files, and is used for snippet generation.
  It is **27.2GB** and is generated by generate_docs_table.py

---

Python Scripts Explained

This section explains the working of the main Python files listed earlier, as well as their expected outputs and runtimes.

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 150 WET files and store them in a new directory called **wet_files**.

---

generate_page_url_table.py

Goal

The aim of this program is to generate the page table (i.e. the 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 **url_table.txt**.

Output File Size

url_table.txt is **494 MB**.

Time Taken

This program generates the page table for the pages in the 150 WET files in **40 minutes**. There are about 6 million pages.

---

generate_postings.py

Goal

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

It generates 150 **sorted** posting files (one per WET file), and then combines them into one large **final_postings.txt** file using **Unix merge**.

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 150 sorted postings files, one per WET file. These are stored in a folder called **sorted_postings**.

Finally, Unix sort (technically Unix merge) is used through Python (using the new bash shell in Windows 10 via an os.system() call) to merge these 150 files. This generates a large **final_postings.txt** file in a folder called **final_postings** within sorted_postings.

Unix 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 **30.9 GB**. It contains around 1.6 billion postings.

Time Taken

The program takes about **7hr 20min** to generate the 150 sorted intermediate postings files. This is an average of 224 documents per second.

It then takes **1hr 20 min** 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 of:
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 **848 MB**.
inverted_index.dat is **3.74 GB**.

Time Taken

This program generates the compressed inverted index and the lexicon in **2hr 20min**.

---

generate_docs_table.py

Goal

The aim of this program is to create a SQL database table called **docs** that stores the page text for each URL/doc, extracted from the WET files. This is needed for snippet generation.

Working

The **sqlite3** package is used in Python. First, a database called **web_search_engine.db** is created and a table called **docs** is created in this database.

CREATE TABLE docs (doc_id integer primary key, url text, page_text text);

The doc_id is the primary key. Every record contains a doc_id, the corresponding URL and the corresponding page text. This table is generated upon parsing the 150 WET files.

Page text is filtered in the same way as it was while generating postings, before inserting it in the table: 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, and punctuation is removed.

Output File Size

The database **web_search_engine.db** is **27.2 GB**

Time Taken

This program generates the database table in **4hr 25min**.

---

query_processing.py

Goal

The goal of this program is to process a query and return the top 10 results.

Working

The following is a fairly detailed overview of the query processing process:

1. **User Input**
   First, an interface was created using HTML and CSS. This interface elicits the query from the user, and allows the user to choose between conjunctive and disjunctive query processing modes
   Conjunctive: all the query terms must be present in the results (AND semantics)
   Disjunctive: at least one of the query terms must be present in the results (OR semantics)
2. **Query Preprocessing**
   When a query is input, it is converted to lowercase. It is then split into tokens, and these tokens/terms are processed the same way as they were while generating postings: 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, and punctuation is removed. Then, only terms that are present in the lexicon are retained (an error is shown if the query contained unknown terms, and only the known terms are processed)
3. **Query Processing**
   Once the preprocessed query terms are obtained, their inverted lists are opened using the start positions stored in the lexicon, and doc ids and frequencies for the terms are obtained (decoded). Then, based on user choice, one of the below two modes is performed:
   - Conjunctive Query Processing: Here, the aim is to return docs that contain all the

query terms. This is done using the nextGEQ() interface. Starting with the first doc_id in the shortest list, this technique aims to determine doc_ids that contain all the query terms. nextGEQ(lp, k) returns the next doc_id in the list lp that is greater than or equal to k.
Once the doc_ids are obtained, the corresponding frequencies are obtained as well, and the BM25 scores for the docs are computed.
A **top-10 heap** is used that always contains the docs corresponding to the top-10 BM25 scores. This is implemented using the nlargest() method of the **heapq** module in Python.

- Disjunctive Query Processing: Here, the aim is to return docs that contain any of the query terms. For every term, the BM25 score of the docs containing it is computed, and the score of the doc is updated if it is present in the doc_id list of another term. Finally, once the final BM25 scores are obtained, the docs corresponding to the top-10 BM25 scores are returned.

BM25 Score Formula

$$\boldsymbol{BM25(q,d)} = \sum_{\boldsymbol{t \in q}} \log\left(\frac{\boldsymbol{N - f_t + 0.5}}{\boldsymbol{f_t + 0.5}}\right) \times \frac{\boldsymbol{(k_1 + 1) f_{d,t}}}{\boldsymbol{K + f_{d,t}}}$$

where $K = k_1 \times ((1 - b) + b \times \frac{|d|}{|d|_{avg}})$

k_1= 1.2, b = 0.75
N = total number of documents
f_t = number of documents that contain t
$f_{d,t}$ = number of times t occurs in d i.e. frequency of t in d
|d| = length of document d
$|d|_{avg}$ = average document length
t = term
q = query

4. **Snippet Generation**
   Snippet generation can be **query-dependent** or **query-independent**.
   In query-independent snippet generation, the snippet may be a summary of the doc text, or the first few lines of the doc. This will not change based on the query terms. However, in query-dependent snippet generation, the snippet changes dynamically based on the query terms. I have implemented query-dependent snippet generation.

   Once the top-10 doc_ids are obtained, the database is queried to retrieve the page texts corresponding to these doc_ids. Then, a snippet is generated for each doc, with 5 terms before and 5 terms after the position (of the first occurrence) of each query term in the doc. This is a simple, but effective, query-dependent snippet generation technique.

Varbyte Encoding – Overview

This section discusses varbyte encoding, the compression technique used to compress the doc_id gaps and the frequencies (i.e. the inverted lists) for each term in the inverted index.

It is a simple and fast encoding technique. It works as follows:
- if the number is < 128, use one byte with the highest bit set to 0
- if the number is >=128 but < 128*128, use two bytes, the first one with its highest bit set to 1, and the other with its highest bit set to 0
- if the number is >= 128*128 but < 128*128*128, use three bytes, and so on

For example, 100 is encoded as **0**1100100 i.e. 128^0*100
150 is encoded as **1**0000001 **0**0010110 i.e. 128^1*1 + 128^0*22
32906 is encoded as **1**0000010 **1**0000001 **0**0001010 i.e. 128^2*2 + 128^1*1 + 128^0*10
and so on.

doc_id gaps and frequencies are encoded using varbyte encoding and are stored in the inverted index. Using varbyte encoding shrinks the otherwise large index to 3.74 GB.

Summary of Design Decisions

- **150** WET files considered
- About **6 million** docs processed
- 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 **url_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**
- The Lexicon and URL Table are loaded entirely into memory
- SQL Database used to store page texts for snippet generation, since it is impractical to use an in-memory data structure for doing so (it would run out of memory)
- Query terms are preprocessed in the same way as during posting generation, and only terms that are in the lexicon are processed
- Both conjunctive and disjunctive query processing are implemented
- Conjunctive query processing is implemented using DAAT (Document at a Time i.e. nextGEQ()) and disjunctive query processing is implemented using TAAT (Term at a Time) processing

- Entire inverted list is decoded, and inverted list caching is not implemented
- **Flask** is used to create a web app

Flask Web App

A fully-functional front-end was built to demonstrate the working of the search engine.

The user is asked to input a query and to choose whether conjunctive or disjunctive query processing is to be performed.

The query is preprocessed (as discussed earlier) and then, based on the user's choice, either conjunctive or disjunctive query processing is performed.

The time taken to generate the top-10 results is displayed, along with the top-10 results. If there were unknown terms (terms not in the lexicon), those terms are displayed as well. Query processing is done using the known terms. If no results were retrieved, an appropriate error message is shown.

Each result contains the page URL, the doc_id, the BM25 score, the query term frequencies and the query-dependent snippet. Results are shown in decreasing order of the BM25 scores.

Screenshots

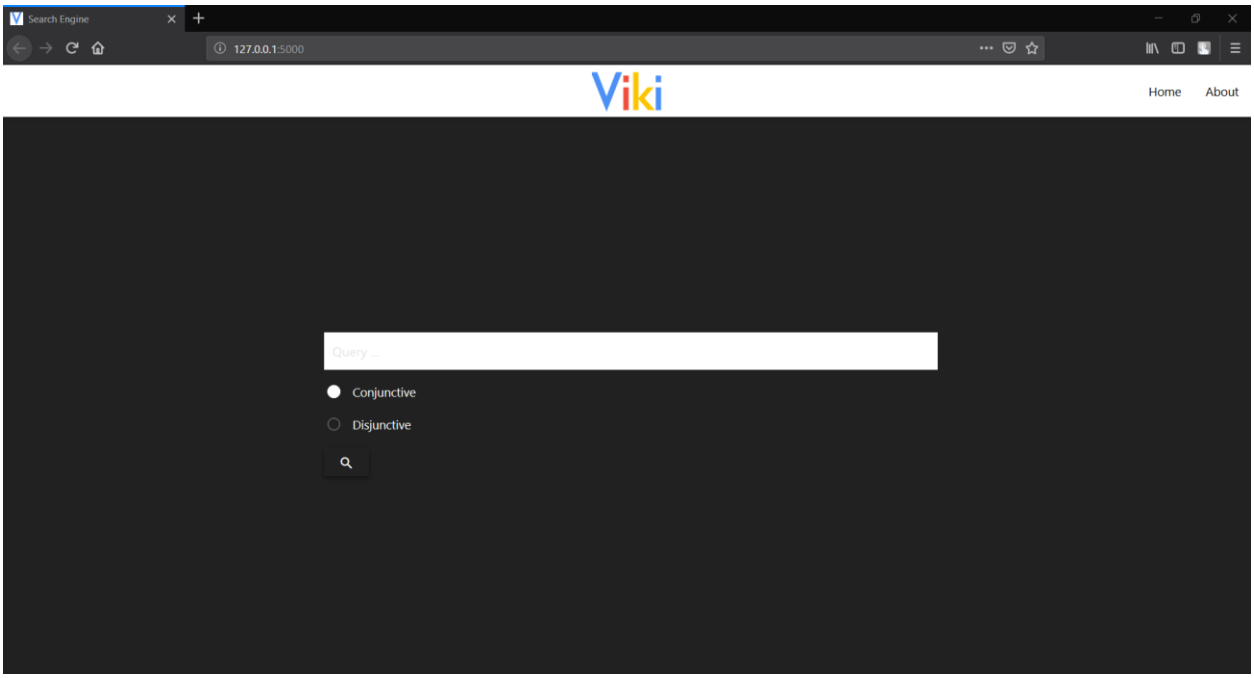

Fig. 1: The Home Page

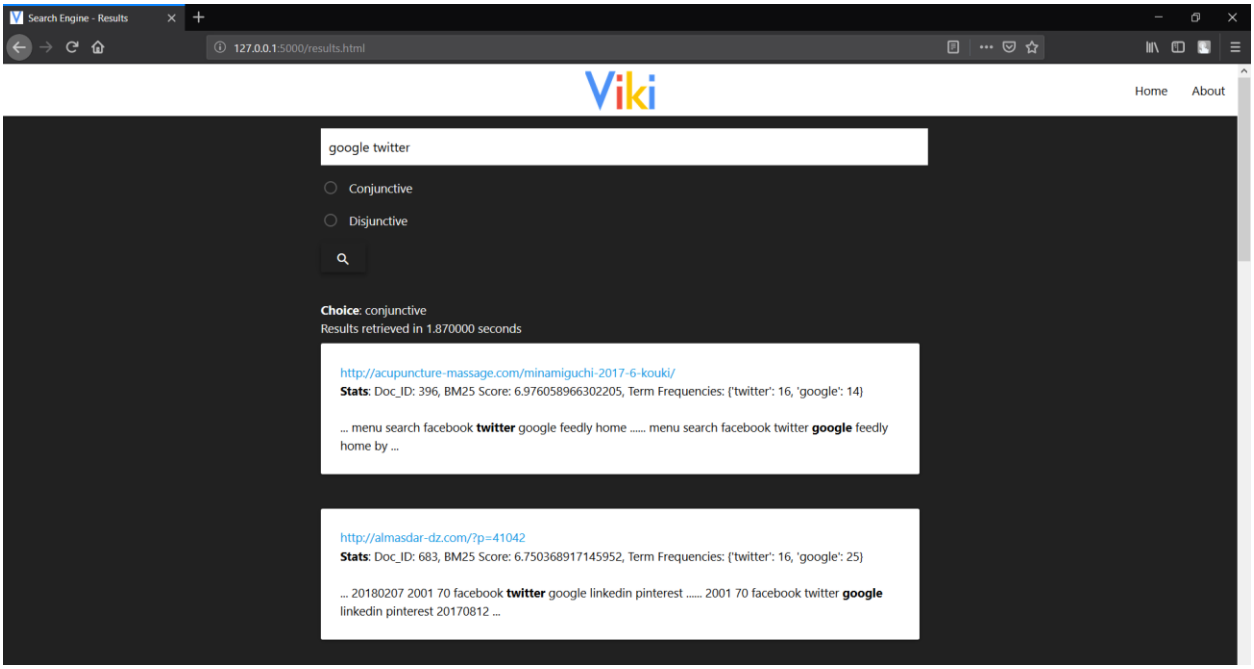

Fig. 2: The Results Page

Stats - Summarized

Sizes

Note: 150 WET files were processed, resulting in 5,906,499 documents/pages. This amounted to a total of 1,611,630,894 postings.

File	Description	Size
final_postings.txt	Merged Postings File	30.9GB
inverted_index.dat	Inverted Index	3.74GB
lexicon.txt	Lexicon	848MB
url_table.txt	URL Table/Page Table	494MB
web_search_engine.db	Database for page text	27.2GB

Time

Process	Time
Downloading 150 WET files	Depends on Internet Speed
Generating Sorted Posting Files	7hr 20min (approx. 224 docs/sec)
Merging 150 Posting Files	1hr 20min (with buffer size 1024)
Inverted Index and Lexicon Generation	2hr 20min
URL/Page Table Generation	40min
Database Creation (for Page Texts)	4hr 25min
Time to Load the Lexicon into Memory	About 1 minute

Time to Load the Page Table into Memory	Less than half a minute
Query Processing – Conjunctive	A couple of seconds on average
Query Processing – Disjunctive	A couple of seconds on average

---

Known Limitations and Future Scope

- Intermediate postings are not in binary format
- Since Windows 10 was used to develop this search engine, Developer Mode must be enabled in order to use the new bash shell to run the Unix sort (Unix merge) command
- Inverted List is decoded entirely
- To achieve the required speed, not all the doc_ids are considered (first 1500 per term)
- Speed can be improved using chunk-wise compression and Inverted List caching
- Better ranking functions, such as Page Rank, can be used
- Smarter snippet generation can be employed, using NLP

Required Environment

- **Python**: 3.6+
- **Platform**: Windows 10 (Developer Mode enabled to use the new bash shell)
- **Browser**: Mozilla Firefox, Google Chrome
